AxoNN: Energy-Aware Execution of Neural Network Inference on Multi-Accelerator Heterogeneous SoCs

Abstract

The energy and latency demands of critical workload execution, such as object detection, in embedded systems vary based on the physical system state and other external factors. Many recent mobile and autonomous system-on-chips embed a diverse range of accelerators with unique power and performance characteristics. The execution flow of the critical workloads can be adjusted to span multiple accelerators so that the trade-off between performance and energy fits to the dynamically changing physical factors.

In this study, we propose running neural network (NN) inference on multiple accelerators. Our goal is to provide an energy-performance trade-off by distributing layers in the NN between a performance- and a power-efficient accelerator. We first provide an empirical modeling methodology to characterize execution and inter-layer transition times. We then find an optimal layer-to-accelerator mapping, by representing the trade-off as a linear programming optimization constraint. We evaluate our approach on the NVIDIA Xavier AGX System-on-Chip with commonly used NN models. We use the Z3 SMT solver to find schedules for different energy consumption targets, with up to 98% prediction accuracy.

1 Introduction

Computing devices are becoming highly heterogeneous with the increased utilization of domain specific accelerators (DSAs), each of which is optimized to perform a specific type of operation. This trend is fueled by the need to run applications that span a diverse set of computations for emerging fields such as artificial intelligence, machine learning, autonomous systems, and smart/connected communities. The latest generation of system-on-chips (SoC)—such as NVIDIA's Xavier architecture, Apple's M1 and A15 Bionic chip, and Qualcomm's Snapdragon 888 SoC—have dramatically increased the degree of architectural heterogeneity within the same die. In such systems, dozens of DSAs with diverse instruction set architectures (ISAs) work together to accelerate operations (*i.e.*, kernels or tasks in an application) that belong to emerging application domains.

In diversely heterogeneous SoCs, an operation (OP) can often be accelerated via different DSAs with varying performance, energy, and latency characteristics. For example, a convolution operation can be set to run on the CPU, GPU, programmable vision accelerator (PVA), or deep learning accelerator (DLA). The DSA that would provide the optimal execution time and/or energy efficiency for the operation depends both on the DSA capabilities, and on properties of the operation, such as matrix size and filter dimensions. Depending on the dynamic requirements of the system (*e.g.*, high throughput, low energy), runtime parameters of the operation (*e.g.*, number of objects, image size), and availability of DSAs, the programmer (or system scheduler) may choose to map different operations to different DSAs throughout execution of an application.

An emerging architectural feature of such heterogeneous SoCs is a *shared system memory* that all DSAs and the CPU can directly access and utilize. While this design choice is primarily motivated by the goal of reducing chip area and production costs, it also helps in eliminating additional data transfer management overhead between the CPU and private device memory [4]. Having shared memory directly accessible by every DSA in the system enables assigning OPs in a workload to the DSAs more flexibly. This flexibility also enables *collaborative execution* in shared-memory heterogeneous system architectures (SM-HSA), where OPs in a workload can be executed on different DSAs [9] to exploit the varying benefits (*i.e.*, energy, throughput, latency, etc.) that different types of DSAs optimally provide—*e.g.*, a convolution operation can be accelerated by a graphical processing unit (GPU) for high performance, or by a deep learning accelerator (DLA) for better energy efficiency. In such SM-HSAs, optimal utilization of the system resources heavily relies on carefully assigning the OPs to the available DSAs based on the target performance and power goals of a given scenario [17].

Collaborative execution of popular workloads, such as neural network inference, on different types of DSAs is a relatively new and unexplored scheme which has the potential to provide unique benefits for budgeted execution scenarios. To demonstrate the feasibility of collaborative execution for achieving different performance and energy goals on a heterogeneous platform, we conduct an exploratory experiment, which is shown in Fig. 1. In this experiment, we map the layers of the VGG-19 [23] network to the GPU and the DLA of NVIDIA's Xavier AGX SoC in three different ways. The left-most and right-most columns in the figure show where all layers are executed, either on the GPU or the DLA respectively. The middle column illustrates a *collaborative* execution where the first n layers are run on the GPU, and the remaining m layers on the DLA. The total execution time and energy consumed is given under each column. Experimental results show that running all layers on the GPU results in the fastest execution time, whereas running all layers on the DLA is the most energy-efficient. On the other hand, the collaborative execution scheme shown in the middle results in a trade-off between execution time and energy, as more layers of the network are executed on the DLA.

A hybrid (i.e., GPU+DLA) execution scheme could be more feasible in real-life scenarios, when there is an energy constraint in the system. For example, when an autonomous aerial drone is running low on battery, scheduling of the NN layers to DSAs can be adjusted at the expense of a higher execution time (i.e., latency), hence resulting in a lower images/second detected by the NN. In the Fig. 1 example, if the remaining energy budget per image detected is less than 175 Joules, (total energy/image needed for GPU-only execution), but more than 140 Joules, rather than running the entire NN on the DLA, choosing the GPU+DLA hybrid schedule in the middle will result in a more feasible operation. *The drone will still be able to complete its flight, but will be more responsive to the surrounding objects*, thanks to the lower latency (12.5 ms per image) achieved by a GPU+DLA hybrid execution against a DLA-only execution (18.5 ms per image).

A limited number of studies [5, 11] explore the benefits of using different types of DSAs collaboratively for the same application.

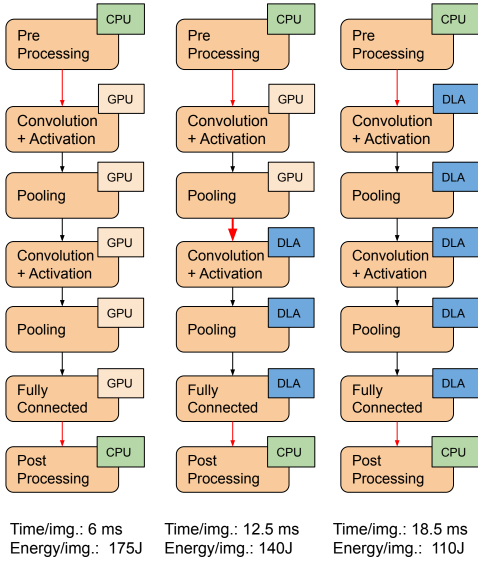


Figure 1: Simplified layer mappings for VGG-19 when executed with TensorRT on Xavier AGX: Leftmost and rightmost control flow graphs (CFG) show the traditional methods of executing the NN on a single type of DSA. The multi-accelerator execution scheme, shown in the middle, employs switches the execution flow from produces a diversity on latency and energy trade-offs by considering transition overhead.

While most [6, 13, 16, 21, 27, 28] focus on improving the total throughput by using multiple DSAs concurrently, only a few have investigated the performance-energy trade-off in limited aspects. To the best of our knowledge, none of the existing studies are able to address all of the following challenges altogether for multi-DSA collaborative execution:

- Holistic modeling of multi-accelerator execution that takes both the execution and data-transfer costs between different type of DSAs into consideration.
- Tunable objective for power consumption which can be targeted while finding schedules with optimal execution time.
- Generalized layer-wise characterization methodology for finding performance and energy costs of neural network inference on multi-DSA systems.

In this study, we propose an energy-aware multi-DSA execution scheme for NN inference on heterogeneous SoCs. Our proposed scheme, *AxoNN*, uniquely enables setting an energy consumption target (ECT) and finds a NN-layer-to-DSA mapping that minimizes the total execution time under a given ECT. *AxoNN* utilizes a novel inter-accelerator transition model to integrate the penalty of switching between DSAs into the cost function. Our scheme characterizes each layer in the network for each target DSA, and explores multiple transitions between layers to find schedules that satisfy the given ECT. We represent the scheduling problem as a constrained-objective optimization problem.

Our paper makes the following contributions:

- We present *AxoNN*, a multi-accelerator execution scheme for diversely heterogeneous SoCs, which finds schedules with near-minimal execution time for a given ECT.
- We propose a novel, empirical model-creation technique to represent the cost of inter-DSA transitions on a shared-memory heterogeneous SoC.
- We build cost models for estimating energy and execution times which uniquely take transition times and hardware-pipelined accelerator architectures into account.
- We evaluate *AxoNN* on the NVIDIA Xavier AGX SoC by using its embedded Volta GPU and DLA. Our results show that our methodology can find near-optimal schedules with one or two inter-DSA transitions within up to 98% accuracy while staying under the given ECT.

2 Multi-accelerator NN Inference on Diversely Heterogeneous SoCs

As demonstrated in Fig. 1, using a single type of DSA for the entire workload does not provide enough room to explore various latency/energy trade-offs. In NN inference, finding the desired trade-off requires a careful distribution of layers onto accelerators. However, using multiple DSAs to maximize the system's utilization while staying under resource constraints, such as ECT, introduces a number of challenges and considerations.

2.1 Challenges

Lack of flexibility in layer-to-DSA assignment: Each type of DSA has a different set of restrictions in terms of capabilities for running different OPs. For example, even though layer activation functions are considered as separate layers on TensorRT, NVIDIA GPUs do not allow the TensorRT scheduler to assign activation functions to a DSA other than the one that executed the preceding convolution OP. NVIDIA's DLA has additional restrictions on layer parameters and batch sizes. Moreover, TensorRT does not allow to transitions from DLA to GPU after certain layers (*e.g.*, Eltwise layer). Such limitations force some layers to fall back to the GPU, even though they are assigned to execute on the DLA. Therefore, the availability of potential inter-DSA transition points is restricted, and depends on the NN and the DSAs to which the layers before and after the transition are assigned.

Grouping layers: Operator fusion has become a commonly applied optimization by popular frameworks, such as TensorRT [20] and TVM [3], that minimizes the cache misses between OPs. Breaking potentially-fusible operations will increase execution time and, as a result, energy usage.

Profiling: Some highly-specialized accelerators such as DLAs run the consecutively-assigned layers as a single black box and do not allow internal profiling of execution times layer-by-layer. This limitation makes empirical modeling more challenging, since it presents an obstacle to fine-grained performance characterization.

2.2 Considerations

Inter-DSA transition overhead: On shared-memory SoCs, caches are often private to DSAs, due to complexity of cache coherency across diversely heterogeneous processing units. When the execution flow switches from one DSA to another on a shared memory system, the transient data present in private caches or buffers of DSAs needs to be written back to the shared memory. Such additional memory read/write operations need to be considered as overhead, and added to the total execution time. The size of the

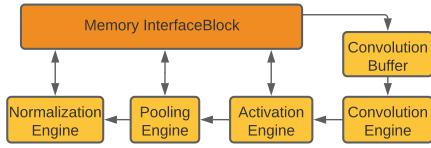


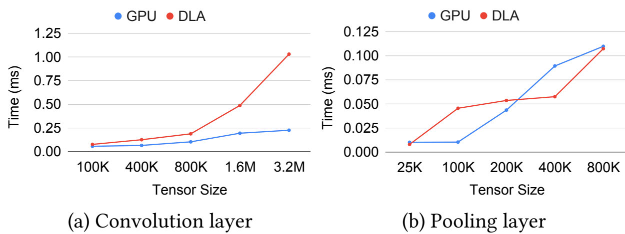
Figure 2: A simplified internal block diagram for NVDLA.

memory pages being written or read is the primary factor determining the magnitude of this transition overhead. In most NNs, the size of the input and output data that each layer consumes and produces often changes after each layer. Moreover, the internal memory hierarchy of each type of DSA affects the transition overhead differently, even though the amount of data being read or written by two DSAs is the same [11]. Therefore, modeling the cost of inter-DSA transitions requires careful consideration of the data size, layer type and the DSA where the transition is occurring.

Execution time characterization: While the execution time of a specific NN layer on a given DSA primarily depends on the layer type and the input data size, the location of the data before the layer is also an important factor. Therefore, we model layer execution time and inter-DSA transition time separately. The cold cache misses issued by DSAs as they begin executing a layer after a transition requires a warm-up period for layer-by-layer characterization. Another important factor affecting the execution time is the existence of internal hardware (HW) pipelines. As shown in Figure 2, NVIDIA's DLA architecture embeds a pipeline of internal engines for common layers, such as convolution, activation, and pooling, in the order that these layers often appear in NNs. The data between the engines are often forwarded with direct data buses and separate characterization of such layers may result in incorrectly estimated layer execution times by our empirical model. For example, since the pooling layer reduces the amount of data being passed to the next layer, measuring the execution time of the convolution and activation layers separately from the pooling layer will result in a longer execution time than the case where these three layers are profiled together. Therefore, layer-characterization needs to take such HW behavior into account for HW-pipelined DSAs.

3 Modeling Methodology

Taking into account the challenges and constraints, this section explains the methodology we utilized to build our empirical model.



(a) Convolution layer (b) Pooling layer

Figure 3: Empirical models for Convolution and Pooling layers' out-transition times after the layers are run on GPU and DLA for different tensor sizes.

3.1 Modeling inter-accelerator transition cost

The location of the transition point must be explored because of three major criteria: (1) the size of data needed to perform read/write operations; (2) DSA mechanisms causing different behaviors for transition operations; and (3) problematic compiler and hardware level optimizations. For the first criterion, the transition overhead will increase as the size of the data increases. The X-axes in Fig. 3.a and 3.b represents the tensor sizes, i.e. the size of the output produced after convolution/pooling layers. The Y-axis represents the transition cost in milliseconds if any transition is applied after the convolution/pooling layers. For example, performing the transition operation for the convolution layer with a data size of 3MB can result in 14x more time overhead compared to a data size of 100KB on the DLA. Thus, it is clear that the transition overhead decreases as the data movement decreases on both of the devices in our experiments. Another significant factor in determining DLA behavior for different sizes is that DLA has a second private buffer specifically for convolution operations. Since the buffer has a limited size, larger data necessitates data movement from afar rather than a private buffer.

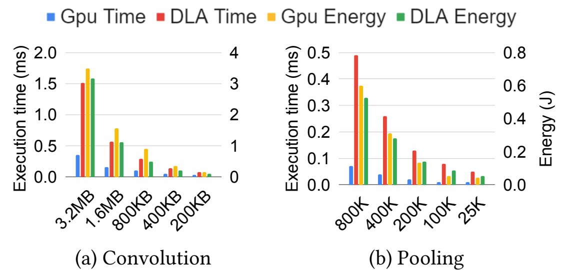
3.2 Energy and performance characterization

Based on architectural restrictions, we check whether a layer can run on all DSAs, or can be marked as a transition layer by using *canRunOnDLA* and *markOutput* TensorRT API calls. For example, a ReLU activation layer cannot run by itself on the DLA, but merging a ReLU activation layer with a convolution layer enables running both of them on the DLA.

Designing and measuring execution time and energy consumption is non-trivial because of the challenges explained in Section 2. We develop a macrobenchmark in order to analyze the behavioral differences of the DSAs for OPs, and measure the NN's resource consumption layer-by-layer. In Figure 4(a-b), we measure execution time and energy consumption of each layer on VGG19 by using GPU and DLA separately. The left vertical axis shows the execution time, whereas the right vertical axis represents the energy consumption. Convolution OPs on GPU are 3x to 4.5x faster than on DLA, whereas pooling on GPU is 3x to 7x faster than on DLA, depending on the data size.

4 Multi-accelerator Scheduling via Constraint-based Optimization

This section details how we build our cost functions and encode scheduling as a constraint-based optimization problem. First, we



(a) Convolution (b) Pooling

Figure 4: Empirical models for execution times of Convolution and Pooling layers on GPU and DLA for different tensor sizes. X axis indicates the tensor sizes.

Table 1: The notations used in this section.

Notation	Explanation
L_i	ith layer on a given layer set L, $0 \leq i \leq len(L)$
P_j	jth processor on a given processor set P, $0 \leq i \leq len(P)$
TR_i	Boolean variable set if a transition is occurred after L_i, 0<i<$len(L)$
$\tau(i, j, OUT)$	The transition cost of layer L_i assigned to processor P_j as output
$\tau(i, j, IN)$	Transition cost of layer L_i assigned to processor P_j as input
$e(i, j)$	Energy consumption of layer L_i on processor P_j
$t(i, j)$	Execution time of layer L_i on processor P_j
ECT	Energy consumption target
S(L_i)	Scheduling set of Layers, 0 < i < len(L)
$T(L, P, S, TR)$	Total time by a given layer set L, processor P, and schedule S
$E(L, P, S, TR)$	Energy consumption by a given set of layer L, processor P, and schedule S
$U(Li)$	The sub-unit executing the layer
$\gamma(L_i, s(L_i))$	Amount of time L_i saves by pipelining in its input
NumTransition	Maximum amount of transition allowed by user

formulate the total execution time using empirical values found for transition and layer-wise execution times in Section 3. Then we formulate an optimization problem that minimizes total execution time for a given energy constraint (i.e., ECT).

Table 1 lists the components needed to formulate our optimization problem. Layer set L is a specific parameter for a given network, as a combination of layers L_i with various sizes and activation functions. Processor set P includes available DSAs on the architecture. Each processor P_i has specialized capabilities to handle particular tasks with different time and energy results. Moreover, transition costs of layers $\tau(i, j, OUT)$ from processor P_i to processor P_j as outputs are observed when the transient data belonging to the previous layer are flushed back to the main system memory. The transition cost of layers $\tau(i, j, IN)$ as an input to the new processor P_j is observed since the cold cache misses caused by the initial memory instructions executed by the GPU result in an implicit warm-up period. All of these values are obtained via offline profiler *IProfiler*, using an API call on TensorRT, to obtain execution time $t(i, j)$ and energy consumption $e(i, j)$ of layer L_i on processor P_j.

Since each layer can be mapped into a different processor, the layer-to-processor schedule for a neural network is defined as $S(L_i)$, as shown in Equation 1:

$$S(L_i) = P_j \qquad where \quad 0 < i < m \quad \& \quad 0 < j < n \tag{1}$$

Since broken fusion operations and pipelined operations will cost extra overhead, we consider another feature in our metholodogy, *pipeline( )* in Eq. 2. If the schedule does not prevent any OP from being pipelined, there will be no effect on the time and energy parameters. However, if the DSA used by the previous layer is not the same DSA on the current layer, it can severely affect execution time and energy results.

$$pipeline(L_i, S(L_i)) = \begin{cases} 0 & \text{if } U(L_i) = U(L_i - 1) \\ \gamma(L_i, s(L_i)) & \text{if } U(L_i) \neq U(L_i - 1) \end{cases} \tag{2}$$

After obtaining the related time parameters, the total execution time for a neural network $T(L, P, S(L \longrightarrow P, TR)$ can be calculated via four different parameters, as shown in Eq. 3. Each layer L_i has a characteristic execution time on processor P_j, represented by a scheduling parameter $s(L_i)$. The transition cost of each layer to a different processor as an input, $\tau(L_i, s(L_i), OUT)$, is added. The transition cost is added to the total execution only if the result of layer L_i transitions to a new processor, represented by TR_i. The transition cost of each layer as an input to a new processor is also added to the total execution time if the data for layer L_i is transferred from a different processor.

$$\begin{aligned} T(L, P, S(L \longrightarrow P), TR) = \sum_{i=0}^{len(L)} (t(L_i, s(L_i)) + \\ (TR_i \times \tau(L_i, s(L_i), OUT)) + (TR_i \times \tau(L_{i+1}, s(L_{i+1}), IN)) + \\ (TR_i \times pipeline(L_i, S(L_i)))) \end{aligned} \tag{3}$$

$$TR_i = \begin{cases} 1 & \text{if } S(i) \neq S(i+1) \\ 0 & \text{if } S(i) = S(i+1) \end{cases} \tag{4}$$

$$NumTransition = \sum_{i=0}^{len(L)} TR_i \tag{5}$$

The energy consumption during an execution of a network $E(L, P, S(L \longrightarrow P, TR)$ can be calculated via three different parameters, as shown in Eq. (6). The summation of energy consumption results of each layer L_i on a processor P_j with each possible schedule S_i represents the total energy consumption of a neural network inference execution, as follows:

$$\begin{aligned} E(L, P, S(L \longrightarrow P), TR) = \sum_{i=0}^{len(L)} e(L_i, s(L_i)) + \\ (TR_i \times e(L_i, s(L_i), OUT)) + (TR_i \times e(L_{i+1}, s(L_{i+1}), IN)) \end{aligned} \tag{6}$$

Total energy consumption is calculated on Eq. (6) via summation of each layer L_i executed on P_i. This function is used in Equation 1 to satisfy the energy constraint.

Our aim is to minimize the execution time of a NN inference for a given set of layers and processors, with each possible mapping of layers to the processors, and an energy constraint ECT. Thus, we define our objective function and the primary constraint as follows:

$$\begin{aligned} \min \quad & T(L, P, S(L \longrightarrow P)) \\ \text{s.t.} \quad & E(L, P, S(L \longrightarrow P)) < ECT \end{aligned} \tag{7}$$

5 Evaluation

The constraints described in Section 4 can be handled with an off-the-shelf constraint solver. In this section, we report results obtained from solutions to the Section 4 constraints.

5.1 Experimental setup

In this study, we use Nvidia's Jetson Xavier AGX SoC since it embeds one performance-efficient (*i.e.*, GPU) and energy-efficient (*i.e.*, DLA) DSAs, together with access to the same shared DRAM memory. The software versions utilized on our experimental platform are Ubuntu OS 18.04, Cuda 10.2, TensorRT 7.1.3, CuDNN 8.0.0, ONNX 1.6.0, and TensorFlow 2.3.1. We use the TensorRT engine to optimize the pre-trained models collected from several neural

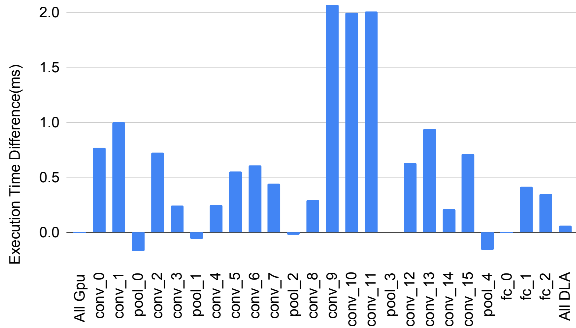


Figure 5: A transition is performed on the layer at the X-axis from GPU to DLA. As the number of layers increases on DLA, the execution time increases. Because the pipeline is broken, the transitions after pooling layers have negative values, as explained in Section 2.1

network models, VGG16/19 [23], Resnet18/50 [7], Alexnet [15], and GoogleNet[24]. The reason we focus on these networks is that all layers can be scheduled on the GPU and DLA. This allows us to flexibly explore all possible layer-to-DSA assignments, without TensorRT engine falling back to GPUs on DLA-assigned layers; hence we avoid obligatory layer transitions between the GPU and DLA. We also solve the equations shown in Section 4 with the Z3 SMT solver, which efficiently determines satisfiability of logical/numeric constraints.

5.2 Experimental Results

We designed an experiment to observe the effect of transitions after different types of layers on a NN. Since we mainly utilize two types of DSAs in our experiments (GPU and DLA), we first assigned all layers to GPU, and measure the execution time on the PU. Then, we repeated the experiment by applying a transition from GPU to DLA after the each layer on the neural network architecture and measured time on the DSAs. In order to analyze the effect of each transition, we subtracted the time for each transition point from the previous transition experiment, and plotted the results in Figure 5. In order words, for each L_i assigned to a different device, we calculated the execution time difference according to Equation 3. Since the layer is running on a slower device, the difference in terms of time is generally higher than 0 in Figure 5. The results of the execution time characterization for layers in Figure 4(a) supports the idea behind the results. However, there are some regions in Figure 5 where the results are less than 0. Those transition points always correspond to transitions after pooling layers. However, the effect of broken pipelining exceed the slowdown of layer operation on DLA, so the total execution time decreases.

5.3 Feasibility Analysis of Our Model

We test the equations on Section 4 by using Z3 with 6 different NN models. We define our objective functions and the main constraint on the solver as in Equation 7. The main idea here is to restrict the energy with an upper limit by minimizing the execution time of NN inference. We provide the profiling results of layer time, energy, and transition characterization as inputs to the solver, and obtain the optimal energy and time results from the solver. In Figure 6, our ECT constraint is set by a range from minimum amount of energy to maximum amount of energy by gradually decreasing the ECT constraint on the X-axis. Depending on the ECT value,

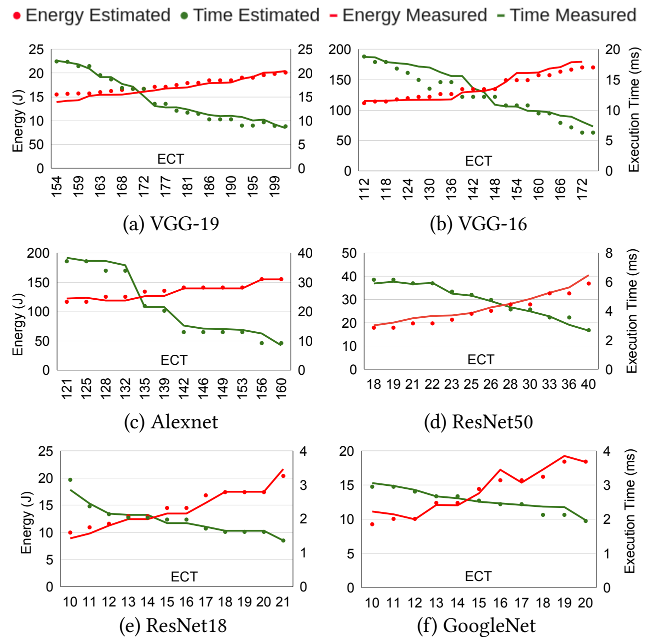


Figure 6: The comparison between the energy and execution times of the schedules estimated by *AxoNN* versus the actual energy and execution times measured for the corresponding actual runs. For each network we have tried varying ranges of ECT that is between the energy consumption of all-DLA and all-GPU execution.

Z3 finds the optimum transition point, in order to minimize the execution time of NN inference. The red dot values on each figure show the results of energy estimation we obtain from the solver, whereas the green dot values show the results of time estimation we obtain from the solver. On the left and right vertical axes, we represent energy consumption and execution time respectively, corresponding to the ECT. The red dot values on each figure show the results of energy estimation we obtain from the solver, whereas the green dot values show the results of execution time estimation we obtain from the solver. According to the plot results, we perform an experiment by applying the scheduling results that the solver finds. The experimental results are represented by the red line. The experiments show that our model provides up to 97.1% accuracy on time estimation, and up to 98.2% accuracy on energy estimation.

5.4 Multi-Transition and Scheduling Overhead

While each transition between DSAs costs extra time in the schedule, the most feasible execution may still include multiple transitions, i.e., going back and forth between DSAs. Therefore, we run our solver by allowing more than a single transition, by increasing the value of the *NumTransition* variable (Eq. 5) to 3. Table 2 lists the number of inter-DSA transitions that the near-optimal schedules include. The overhead of scheduling (i.e., solver execution time) is under 5 seconds when *NumTransition* is set to 1, and under 1 minute when *NumTransition* is set to 3.

6 Additional Related Work

GPipe [10] and PipeDream [18] split tasks between multi-DSAs by utilizing pipelines and considering transition time between accelerators for deep learning (DL) training. HetPipe [21] considers the

Table 2: The number of inter-DSA transitions that near-optimal schedules include when *NumTransition* variable is set to 3.

Model	1 Transition	2 Transitions	3 Transitions
GoogleNet	19	3	0
VGG-19	16	4	2
VGG-16	18	4	0
Alexnet	10	2	0
ResNet50	8	4	0
ResNet18	9	3	0

first step of heterogeneity, and sets up multi-GPU clusters to apply the pipeline parallelism idea by maximizing utilization. However, none of the aforementioned works takes energy into account.

HPC-DAG [8] explores the design space for latency-sensitive application modeling with a heuristic approach. Narayanan et al. [19] propose round-robin scheduling for a target-latency deadline for DL workloads. Shamsa et al. [22] prioritize resource management over goals by considering dynamic changes. HetSched [1] optimizes a CPS's mission time by exploiting the heterogeneity on SoCs and the application's characterization. These studies do not consider energy as a target or metric.

Pipelining in DL inference [26] is applied by distributing layers between CPU and GPU in order to maximize throughput of the system and synchronous data via cache-coherent interconnects. However, their methodology is not applicable to asynchronous data transfers. Kang et al. [12] optimize a single DL application's response time via the dynamic voltage and frequency scaling (DVFS) technique by finding the Pareto-optimal scheduling. Jeong et al. [11] offer a parallelization methodology for DL inference workloads to maximize throughput by leveraging TensorRT's GPU and DLA. However, none of these works considers using multiple accelerators for DL tasks by considering energy and minimum latency.

A similar methodology is presented by MEPHESTO [17], which first characterizes the workload and DSAs, then designs the energy-performance trade-off by collocating kernels and considering the memory contention on heterogeneous systems. However, this study does not take the dependencies between OPs into account, and is therefore not suitable for NN inference. NeuroPipe [14] also accelerates in an energy-efficient way using heterogeneous processing units, by modifying the energy restrictions on the system. Barik et al. [2] propose a mapping algorithm between CPU and GPU by characterizing the power consumption of data-parallel specific workloads. Tzilis et al. [25] propose an online profiling model for estimating power consumption and performance under DVFS configuration for a given application. The aforementioned works do not consider challenges on DL applications.

To the best of our knowledge, our work is the first to apply layer-wise mapping on heterogeneous accelerators by considering both latency and energy.

7 Conclusion

This study presents *AxoNN*, a multi-accelerator scheme for heterogeneous SoCs. We explore the factors affecting the energy-aware scheduling of NN workloads onto DSAs. We analyze the transition costs between DSAs in a shared-memory system and characterize the execution time and energy consumption of different NN workloads. We build a scheduling model in order to find the minimum execution time for different energy targets. We test our methodology with 6 different networks, and test our results with the Z3 SMT Solver, obtaining up to 98% prediction accuracy.

References

[1] Aporva Amarnath. 2021. Heterogeneity-Aware Scheduling on SoCs for Autonomous Vehicles. *IEEE Computer Architecture Letters* (2021).

[2] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In *CGO*.

[3] Tianqi Chen. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*.

[4] Marvin Damschen, Frank Mueller, and Jörg Henkel. 2018. Co-scheduling on fused CPU-GPU architectures with shared last level caches. *IEEE TCAD* (2018).

[5] Maria Angelica Davila Guzman. 2019. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *The Journal of Supercomputing* (2019).

[6] Li Han, Yiqin Gao, Jing Liu, Yves Robert, and Frederic Vivien. 2020. Energy-Aware Strategies for Reliability-Oriented Real-Time Task Allocation on Heterogeneous Platforms. In *ICPP 2020*.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[8] Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. 2021. The HPC-DAG Task Model for Heterogeneous Real-Time Systems. *IEEE Trans. Comput.* (2021).

[9] Sitao Huang. 2019. Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures. In *ICPE*.

[10] Yanping Huang. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NIPS* (2019).

[11] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. 2021. Deep Learning Inference Parallelization on Heterogeneous Processors with TensorRT. *IEEE Embedded Systems Letters* (2021).

[12] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. 2020. Scheduling of Deep Learning Applications Onto Heterogeneous Processors in an Embedded Device. *IEEE Access* (2020).

[13] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *ICCAD*.

[14] Bogil Kim, Sungjae Lee, Amit Ranjan Trivedi, and William J. Song. 2020. Energy-Efficient Acceleration of Deep Neural Networks on Realtime-Constrained Embedded Edge Devices. *IEEE Access* (2020).

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*.

[16] Svetlana Minakova and Erqian Tang. 2020. Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*.

[17] Mohammad Alaul Haque Monil, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. MEPHESTO: Modeling Energy-Performance in Heterogeneous SoCs and Their Trade-Offs. In *PACT*.

[18] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*.

[19] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*.

[20] NVIDIA. 2021. TensorRT. https://developer.nvidia.com/tensorrt

[21] Jay H. Park. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *USENIX ATC*.

[22] Elham Shamsa, Anil Kanduri, Amir M. Rahmani, Pasi Liljeberg, Axel Jantsch, and Nikil Dutt. 2019. Goal-Driven Autonomy for Efficient On-chip Resource Management: Transforming Objectives to Goals. In *DATE*.

[23] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, Yoshua Bengio and Yann LeCun (Eds.).

[24] Christian Szegedy. 2015. Going deeper with convolutions. In *CVPR*.

[25] Stavros Tzilis, Pedro Trancoso, and Ioannis Sourdis. 2019. Energy-Efficient Runtime Management of Heterogeneous Multicores Using Online Projection. *TACO* (2019).

[26] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. 2020. A Pipeline-Based Scheduler for Optimizing Latency of Convolution Neural Network Inference over Heterogeneous Multicore Systems. In *AICAS*.

[27] Hongzhi Xu, Renfa Li, Chen Pan, and Keqin Li. 2019. Minimizing energy consumption with reliability goal on heterogeneous embedded systems. *JPDC* (2019).

[28] Houssam-Eddine Zahaf, Abou El Hassen Benyamina, Richard Olejnik, and Giuseppe Lipari. 2017. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture* (2017).