# StarONNX: a Dynamic Scheduler for Low Latency and High Throughput Inference on Heterogeneous Resources

Olivier Beaumont, Jean-François David, Lionel Eyraud-Dubois, Samuel Thibault

# StarONNX: a Dynamic Scheduler for Low Latency and High Throughput Inference on Heterogeneous Resources

Olivier Beaumont[1], Jean-François David[1],
Lionel Eyraud-Dubois[1], and Samuel Thibault[2]

[1] Inria Center of the University of Bordeaux firstname.surname@inria.fr
[2] University of Bordeaux, France firstname.surname@u-bordeaux.fr

**Abstract.** Efficient inference of Deep Neural Network (DNN) models on heterogeneous processors is challenging, not only because of the heterogeneity between CPUs and hardware accelerators, but also because the problem is fundamentally bi-objective in many contexts, since both latency (time to perform an inference) and throughput (number of inferences per unit time) need to be optimized. We present StarONNX, a solution based on integrating ONNX Runtime in StarPU, which aims to optimize the distribution of inference tasks and resource management on heterogeneous architectures. This strategy relies on (i) the execution of deep learning models by ONNX Runtime to maximize individual resource utilization, and (ii) the orchestration of heterogeneous resources by StarPU to provide scheduling and overlapping strategies for computation and communication. An essential point of the framework is the ability to use a split DNN into two parts, one running on the GPU and the other on the CPU, thus increasing throughput by using all possible resources with a minimal degradation of worst case latency. We show that integrating ONNX Runtime into StarPU does not introduce significant overhead. We also evaluated our approach against Triton Inference Server and showed a significant improvement in resource utilization and reduced latency.

**Keywords:** Dynamic Scheduling · Resource Management · DNN inference · Throughput / Latency · Resource Heterogeneity · Graph Partitioning · ONNX Runtime · StarPU.

## 1 Introduction

The deployment of Deep Neural Networks (DNNs) in diverse environments such as data centers and edge devices has increased the need for dynamic management of heterogeneous computing resources, including CPUs and GPUs[1]. Due to their different processing capabilities and memory structures, these systems present challenges in optimizing latency and throughput.

To address these challenges, we propose StarONNX, an solution that integrates ONNX Runtime [2] with StarPU [3] for dynamic task distribution and

execution across heterogeneous resources. It optimizes DNN inference by minimizing latency and maximizing throughput through task scheduling and data management. This improves resource utilization and performance, especially in real-time processing environments such as autonomous vehicles [1] and real-time interactive systems [4].

StarONNX enables DNN components to be processed on heterogeneous hardware types by splitting DNN models into two parts. This approach not only improves resource utilization, but also maintains good performance. This paper introduces StarONNX and provides a detailed understanding of its architecture and advantages over the NVIDIA Triton solution.

The paper is organized as follows. Section 2 reviews recent advances in scheduling and pipelining optimization for deep neural network (DNN) inference. Section 3 describes the implementation of StarONNX and its integration with ONNX Runtime in StarPU, and discusses its advantages. Section 4 evaluates the overhead of the integration compared to that of the Triton inference server. Section 5 explores the use of heterogeneous resources and model partitioning to balance throughput and latency. Section 6 summarizes our contributions and suggests future research directions.

## 2 Related works

Optimizing the execution of DNN inference on heterogeneous architectures is a dynamic research area that addresses the challenges of balancing throughput, latency, and energy consumption. Several studies have proposed solutions to improve the DNN inference.

LaLaRAND [5] tackles real-time DNN task scheduling on CPU/GPU architectures through adaptive quantization to manage performance imbalances between processors. This layer-level scheduling approach enhances schedulability while preserving DNN task accuracy. Aghapour et al. [6] focus on improving CNN inference latency by alternating execution between CPU and GPU, an effective strategy for embedded devices where reducing latency is critical for real-time applications. Their work emphasizes fine-grained management of layered execution to minimize response times.

GSLICE [7] introduces a scalable inference platform employing spatial multiplexing and adaptive batching to optimize GPU utilization. This approach dynamically adjusts resources to balance throughput and latency, outperforming standard methods. REEF [8] enhances concurrent DNN inference on GPUs by incorporating a task preemption mechanism, facilitating better resource allocation for critical tasks. Wu et al. [9] propose a pipeline scheduling method to optimize CNN inference on heterogeneous multicore systems, using an iterative bi-partitioning approach to evenly distribute layers across CPU cores, thereby reducing latency and increasing throughput.

Yu et al.[10] present a scheduling framework for multitenant DNN inference on GPUs, focusing on optimized concurrency management to maximize resource utilization while ensuring task isolation. The Hierarchical Inter-Operator Sched-

uler (HIOS)[11] optimizes DNN model inference latency across multiple GPUs by combining intra- and inter-GPU parallelization, providing an adaptive workload distribution strategy.

To address the challenge of parallel processing multiple inference queries with latency constraints, a common approach is to employ a Round-Robin (RR) strategy when multiple computing resources (CPUs, GPUs, FPGAs) are available. Requests are stored in a FIFO queue and dynamically allocated to the first available resource. While simple and easy to implement, this strategy has several limitations [12], including a lack of scientific insight into the use of CPUs as GPU backups for neural network inference. Aghapour et al. [6] highlight the effectiveness of this solution due to the limited power of GPUs and the unified memory between CPUs and GPUs, which reduces transfer overhead.

StarONNX combines the precision of ONNX Runtime in neural network model tuning with StarPU's proficient management of heterogeneous computing resources. This approach enhances the execution of deep learning models on complex architectures by introducing dynamic scheduling and resource allocation facilitated by runtime systems like StarPU [3], OmpSs [13], or PaRSEC [14]. Unlike solutions that focus solely on resource load balancing or task partitioning, such as LaLaRAND [5] and AxoNN [15], or those optimizing concurrent GPU utilization like REEF [8] and GSLICE [7], StarONNX offers a unified, adaptive solution. It segments and distributes network layers across CPUs and GPUs, reducing inference latency and improving overall resource utilization through adaptive scheduling and task distribution.

## 3 Integration of ONNX Runtime into StarPU

### 3.1 StarPU

StarPU [3] is a runtime system that manages CPUs, GPUs and other types of processors. Its main goal is to handle tasks and data in a heterogeneous computing environment. It uses a task graph to represent dependencies between tasks. Dynamic scheduling is employed to optimize task allocation and data transfers, and the scheduler selects optimal computing resources for tasks by registering performance models. This includes strategies such as reserving accelerator cache to minimize unnecessary data allocations, allowing asynchronous data transfers to overlap with computations, prefetching data, and direct GPU-to-GPU transfers. StarPU also handles memory releases based on needs or resource constraints.

### 3.2 ONNX Runtime

ONNX Runtime [2] is an open source DNN inference optimization and execution platform that supports multiple hardware and software platforms and integrates with deep learning frameworks such as PyTorch and TensorFlow via ONNX

model conversion; it also includes model optimizations[1] such as node merging. An ONNX Runtime session is a data structure for running an ONNX model and manages the inference process and optimizations, starting with fetching the model, configuring parameters, allocating resources, and running the model to produce results. ONNX Runtime also provides customizable thread management to improve performance by optimizing CPU core utilization and controlling idle CPU usage.

### 3.3 Integration

The StarONNX solution, available in the repository[2] integrates ONNX Runtime within StarPU. It leverages the single resource/model optimization capabilities of ONNX Runtime and the management of multiple heterogeneous computing resources by StarPU.

In this section, we assume that the DNN model is divided into several parts. Each part can be optimized by ONNX Runtime for each type of available resource, e.g. with operator fusion, before inference starts. In StarONNX, each of these parts corresponds to a StarPU task, and StarPU is responsible for managing the dependencies and data transfers between tasks.

Figure 1 shows the process of inference in StarONNX for a neural network divided into two parts. StarONNX receives the implementations of both parts on both types of resources from ONNX. When the application submits an inference request to StarONNX, in Step 1, both tasks (corresponding to the first and second parts) are submitted to StarPU. However, only the first one is ready (due to the dependencies between the first and second tasks). In Step 2, StarPU's scheduling engine evaluates several criteria based on data locality, resource availability, and performance models to decide which resource (CPU or GPU, GPU in our example) should perform this task. In Steps 3, 4, and 5, the data manager asynchronously transfers the appropriate data to the GPU and the task is processed. Then, in Step 6, the task corresponding to the second part of the network becomes ready to be scheduled, and the scheduling engine selects the appropriate resource (CPU or GPU, in our example, CPU is selected). The data manager performs the data transfer, if necessary, and once the data is transferred, the second task can be processed. The application is notified asynchronously when the process finishes, indicating that the inference result is available.

When integrating ONNX Runtime with StarPU, we use strategy that automatically groups CPU cores on the same NUMA node to form multiple StarPU workers. This approach takes advantage of StarPU's worker, allowing the multithreaded capabilities of ONNX Runtime to execute tasks across multiple cores with thread pinning to improve CPU performance and reduce runtime interference due to reduced data transfer times between different CPU core caches [9]. In addition, StarPU uses statically pinned memory buffers for neural network tensors, allowing fast, asynchronous data transfers through the GPU driver. Memory

[1] `https://onnxruntime.ai/docs/performance/model-optimizations/graph-optimizations.html`

[2] `https://gitlab.inria.fr/topal/staronnx`

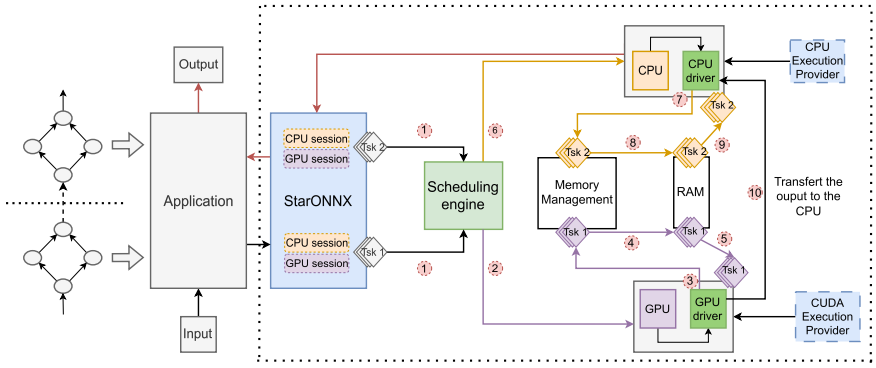



Fig. 1: An example of how StarPU works with ONNX Runtime on heterogeneous processors and a sliced model.

management is optimized by not specifying a memory node for intermediate tensors, giving StarPU the flexibility to dynamically allocate memory based on the executing resource. The architecture also includes a cache management system that recycles buffers, reducing overhead. By leveraging the asynchronous APIs of both ONNX Runtime and StarPU, StarONNX minimizes the need for synchronization, increasing the overlap between computation and communication, and improving the overall performance in complex computing environments, as shown in Section 4.

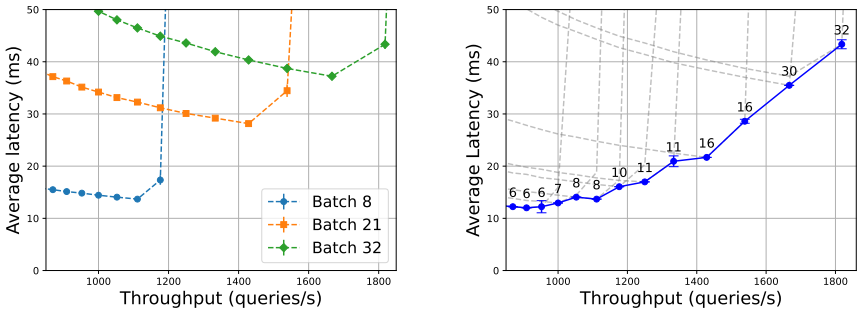
## 4 StarONNX Evaluation

### 4.1 Experimental Setup

All experiments were conducted on a system running CentOS Linux release 7.6.1810 (Core) with the GCC compiler version 11.2.0. The experiments utilized StarPU version 1.4, CUDA version 11.7, and ONNXRuntime version 1.6. The hardware configuration included a 16 GB NVIDIA P100 GPU and two 16-core Broadwell Intel Xeon E5-2683 v4 CPUs at 2.1 GHz, with a total of 256 GB RAM (8 GB per core).

### 4.2 Latency, Batch Size and Congestion

This section describes how we assess the performance of different inference servers. In practice, the parameter that describes the load on an inference system is the delay between two consecutive queries $\tau$. In the following, we use the throughput $\rho = \frac{1}{\tau}$ to characterize the load. For a given injection throughput of $\rho$, the system may or may not be able to process requests. We do not consider the possibility of dropping requests, so there is a maximum value of $\rho_{\max}$ beyond which the system cannot process requests. For a value of $\rho \leq \rho_{\max}$, a relevant




(a) Performance for different batch sizes.

(b) For each throughput, the batch size that minimizes maximal inference time for a query is chosen.

Fig. 2: Performance plots for StarONNX with the GoogLeNet network, showing latency as a function of throughput.

metric for comparing inference systems is their induced maximum latency, i.e., the maximum interval duration between the arrival of a query and the end of its processing. In practice, the maximum latency is obtained for the first data item of each batch. In the plots of Figure 2 (and all subsequent plots), each point corresponds to the average value, over 50 batches, of the (maximum) latency of the first data item in the batch, and the error bars are computed over the 50 batches.

To process inference queries and optimize the efficiency of computing resources (especially GPUs), queries are grouped into batches. In the case of a single GPU and an unpartitioned graph, only the batch size $b$ can be varied. The rationale is as follows: increasing the batch size increases the batch processing time and the average wait time for queries before batch processing. However, increasing the batch size increases the computational efficiency of the GPU, so the ratio of batch processing time to batch size decreases as $b$ increases.

Let us now demonstrate the evaluation of StarONNX's performance. We assume that queries arrive at a given throughput $\rho$ (queries per second), and we estimate the minimum batch size and the associated maximum query latency corresponding to that throughput. The experiments are performed on a 16 GB NVIDIA P100 GPU platform and two 16-core Broadwell Intel Xeon E5-2683 v4 at 2.1 GHz, 256 GB (8 GB/core). The results presented in this section have been obtained using GoogLeNet neural network [16].

Figure 2a shows latency measurements for a fixed batch size: latency decreases as throughput increases, up to a point where throughput is so high that the next query arrives before the previous one has finished. This leads to congestion, and the maximum latency diverges as the total number of queries increases. This highlights the need to choose a larger batch size to avoid overloading the GPU and preserve bounded latency.

In Figure 2b, we show how to minimize the maximum latency for a given throughput. We choose the batch size that minimizes the maximum latency among all possible batch sizes. The dotted plots in gray correspond to the results obtained with different batch sizes, and the blue plot shows the best choice of batch size (the optimal batch size is shown in the blue plot) as a function of injection throughput $\rho$.

### 4.3 Overheads comparison between StarONNX and Triton Inference Server.

Let us now compare the overhead of StarONNX and Triton Inference Server [3] for a single GPU and without model partitioning on the native ONNX runtime. NVIDIA's Triton Inference Server [17] is an open source platform for running neural network models that is compatible with frameworks such as TensorFlow, PyTorch, and ONNX Runtime. It supports deployment of models on multiple processors and provides ensemble models for combining multiple sequence models in a pipeline managed by a dedicated scheduler. Advanced scheduling policies, such as the dynamic batching policy, allow customized management of inference requests.

To compare the overhead of different inference systems, we use the same procedure as in Section 4.2: we specify an injection throughput for inference tasks and search for the batch size that (i) allows inference to be performed with the imposed throughput while (ii) minimizing latency.

The results for GoogLeNet are shown in Figure 3. For the native ONNX runtime results with a single GPU, for fair comparisons, we manually implement the same benefits as StarPU: we added pinned memory, task concurrency, and prefetching for overlapping communication and computation. Execution is asynchronous, copying data from the host to the driver on another stream. The results shown in Figure 3 illustrate that the integration of ONNX Runtime into StarPU does not introduce significant overhead, and trace analysis verifies that the utilization of computational and communication resources is indeed close to optimal.

However, it is important to clarify the value of integrating ONNX Runtime into StarPU in multi-processor environments. StarONNX becomes especially useful when multiple heterogeneous computing resources are involved. StarPU automatically and skillfully manages task scheduling and memory functions - including allocation, deallocation, and transfers - and optimizes operation overlap, CPU binding, and CUDA thread creation. It groups CPU cores and maintains a performance log for each type of task of each type of resource, enabling dynamic and task-resource matching. StarPU's automated task dependency management is critical to maximizing performance in multi-processor environments. By integrating ONNX Runtime, we leverage a mature and proven execution framework and eliminate the complexity of designing a new system from scratch for inference on multiple heterogeneous resources.

[3] We use the Docker version of Triton `nvcr.io/nvidia/tritonserver:24.01-py3`

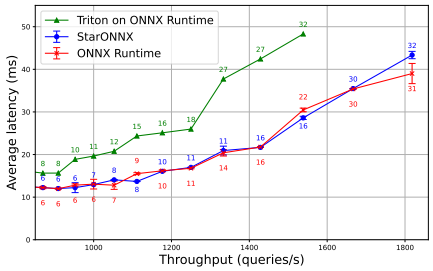



Fig. 3: StarPU overheads by integrating ONNX Runtime and comparison with Triton Inference Server.

Triton inference server introduces significant performance overhead compared to both the optimized native ONNX runtime and StarPU. First, Triton is associated with higher latency for a given injection throughput. Second, it handles a lower maximum input inference arrival rate, achieving only 1,550 queries per second instead of up to 1,800 queries per second with StarONNX. This performance difference is due to the differences in computation and communication management made possible by the use of StarPU.

Memory management and data transfer between the CPU and GPU are critical to performance optimization. Triton transfers data to unpinned GPU memory and performs data copies and computations in the same CUDA stream, preventing task overlap and limiting throughput. In contrast, StarONNX uses pinned memory and concurrent operations to allow overlapping data transfers and computations, along with prefetching. This increases processing speeds and reduces execution times, resulting in better performance when handling higher injection throughputs of inference queries.

## 5 Inference with Partitioned Neural Networks and Heterogeneous Resources

As discussed in Section 4.3, StarONNX makes good use of a GPU with negligible overhead with respect to native ONNX Runtime. In the case of multiple heterogeneous resources, StarONNX can handle inference requests with a near-optimal injection throughput, allowing overlap of computation and communication, and management of memory allocations. The runtime system can also make the best use of heterogeneous resource capacities by relying on simple but effective scheduling heuristics, such as assigning the highest priority ready task to the resource that can process it fastest, taking into account the processing time of each task on each resource and the placement of the task's predecessors to account for transfer times.

This section shows how CPU and GPU can be collaboratively used proficiently in this context. A first naive solution considers that each inference can

be performed either on the GPU or on the CPU. This does not require any code changes, making it easy to aggregate CPU and GPU processing computing capabilities. As long as the processing time of a batch on the GPU is less than the batch build time, the GPU is free when a new batch is built and this batch is sent to the GPU. If the input throughput becomes too high for all queries being processed on the GPU only, then some batches are processed by the CPU to increase throughput. However, in this case, the maximum latency increases significantly as soon as CPU is used, due to its relative low performance.

### 5.1 Model Partitioning Strategy

To mitigate this increase in latency, an effective strategy is to split the model into two parts. In the case of Convolutional Neural Networks (CNNs), the GPU typically handles the first part, which contains the computationally intensive early layers of a CNN. These early layers, which perform extensive data processing and feature extraction, benefit greatly from the parallel processing capabilities of the GPU. The latter part of the CNN, with smaller feature maps and less demanding computations, can be processed by the CPU or the GPU. This allocation is dynamically managed by StarPU's scheduler to improve load balancing and optimize resource utilization.

The decision made by StarPU's scheduler on where to allocate the first and second parts takes into account several critical factors, such as the size of the job in the queue, task locality, dependency chains, data transfer durations, and a performance history log maintained by StarPU that tracks execution times across processors. As the system approaches congestion, the scheduler can dynamically route the less complex second part to the CPU, reducing the load on the GPU and leveraging the CPU's efficiency for simpler tasks. This ensures a good management of computing resources.

While the CPU can technically process the first part, StarPU's scheduler avoids doing so to prevent significant latency increase and maintain system performance. Through dynamic scheduling and strategic model partitioning, the system ensures that the CPU can step in when needed without significantly impacting overall latency. As a result, when processing shifts from the GPU to the CPU under high load conditions, the observed increase in latency remains lower than if the entire model was processed on the CPU.

To partition a model, we aim to offload between 5% and 15% of the GPU's computation time to be allocated to the CPU. This strategy may increase the individual latency per batch, but frees up the GPU sooner to process new requests, thereby reducing the overall latency. To implement this partitioning, we first determine the computation time of each DNN node on the GPU. Based on these times, we can estimate where to partition the model, taking into account the transfer times of intermediate tensors from the GPU to the CPU.

### 5.2 Comparison and Results with and without Partitioning

The neural network models chosen in the evaluation-GoogLeNet, Nfnet (Normalized-Free Network) [18], Efficientnet-V2 [19], and Vit-face-expression [20]-exemplify advanced architectural strategies optimized for different computational environments. GoogLeNet uses a network-in-network design with inception modules that apply multiple convolutional filters. Nfnet eliminates the need for batch normalization through adaptive gradient clipping, providing better accuracy than models that normalize the batch. An Efficientnet-V2, which scales all network dimensions uniformly with a compound coefficient, sets the standard for scalability and performance, essential for evaluating inference over heterogeneous resources. Finally, the Vit-face-expression model uses the Vision Transformer architecture for facial expression recognition to take advantage of self-attention mechanisms that highlight relevant image regions. This is for evaluating transformer-based models in a partitioned environment involving GPUs and CPUs. Together, these models address different facets of neural network applications and serve as a testbed for the effectiveness of heterogeneous resource management in handling inference tasks.

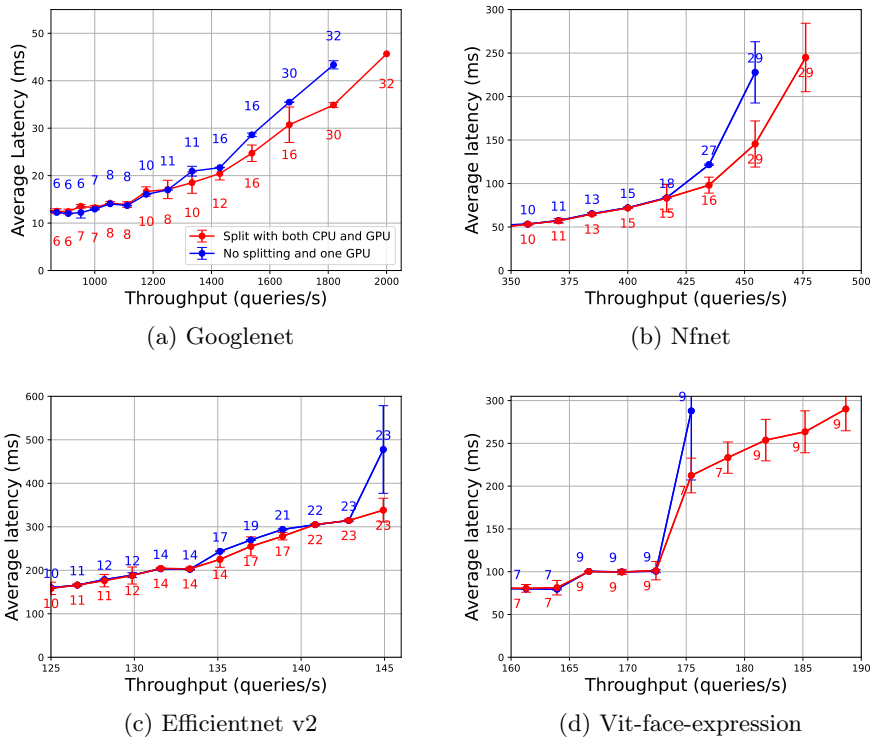
Figure 4 shows the performance of StarONNX on these partitioned models in two parts. We use the HEFT scheduling algorithm included in StarPU. As the arrival rate increases, StarPU's scheduling algorithm starts to dynamically allocate the second part of some queries on the CPU until all queries are individually processed in sequence on the GPU and then on the CPU, which is typically the case when the injection throughput is too high for the GPU to process all batch requests before a new one arrives. Figure 4 shows that model partitioning improves the maximum query processing throughput. For the Vit-face-expression model, by offloading 10% of the GPU computation, 11 of the 12 transformers of the model are placed in the first partition and the last in the second.

## 6 Conclusion and future work

This paper presents StarONNX, a dynamic scheduler designed to improve low-latency, high-throughput inference across heterogeneous computing resources by integrating ONNX Runtime with StarPU. This approach enables better utilization of resources for DNN inference across a set of CPU cores and a GPU, addressing the dual challenge of optimizing latency and throughput.

The strength of StarONNX lies in the sophisticated resource management and scheduling strategies inherited from StarPU that require no user intervention. Compared to the Triton Inference Server, this approach improves resource utilization and reduces latency.

Our future work including refining model partitioning strategies and further improving scheduler performance to accommodate the evolving landscape of DNN models and hardware architectures. As we continue to explore the integration of DNN inference with dynamic scheduling, StarONNX has potential to contribute to the progress of real-time DNN inference systems. This promises




(a) Googlenet (b) Nfnet

(c) Efficientnet v2 (d) Vit-face-expression

Fig. 4: Ability of StarONNX to handle heterogeneous resources for throughput.

a future where heterogeneous computing resources are fully leveraged to deliver faster, more AI-driven applications.

## References

1. Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil Dutt, and Jong-Chan Kim. R-tod: Real-time object detector with minimized end-to-end delay for autonomous driving. In 2020 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2020.
2. ONNX Runtime: Cross-platform, high performance scoring engine for ml models. `https://onnxruntime.ai/`. Accessed: [2024/03/20].
3. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In 15th International Euro-Par Conference, 2009.
4. Zhou Fang, Tong Yu, Ole J Mengshoel, and Rajesh K Gupta. Qos-aware scheduling of heterogeneous servers for inference in deep neural networks. In 2017 ACM on Conference on Information and Knowledge Management, 2017.
5. Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In 2021 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2021.

6. Ehsan Aghapour, Dolly Sapra, Andy Pimentel, and Anuj Pathania. Cpu-gpu layer-switched low latency cnn inference. In 2022 25th Euromicro Conference on Digital System Design (DSD). IEEE, 2022.
7. Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In ACM Symposium on Cloud Computing, pages 492–506, 2020.
8. Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022.
9. Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In 2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), 2020.
10. Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. Automated runtime-aware scheduling for multi-tenant dnn inference on gpu. In Int. Conference On Computer Aided Design (ICCAD), 2021.
11. Turja Kundu and Tong Shu. Hios: Hierarchical inter-operator scheduler for real-time inference of dag-structured deep learning models on multiple gpus. In IEEE International Conference on Cluster Computing (CLUSTER), 2023.
12. Babur Hayat Malik, Mehwashma Amir, Bilal Mazhar, Shehzad Ali, Rabiya Jalil, and Javaria Khalid. Comparison of task scheduling algorithms in cloud environment. Int. Journal of Advanced Computer Science and Applications, 2018.
13. Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters, 2011.
14. George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science & Engineering, 2013.
15. Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E Belviranli. Axonn: Energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In 59th ACM/IEEE Design Automation Conference, 2022.
16. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In IEEE conference on computer vision and pattern recognition, 2015.
17. Triton Inference Server: Open-source ai inference serving. `https://developer.nvidia.com/nvidia-triton-inference-server`. Accessed: [2024/06/20].
18. Andy Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. In International Conference on Machine Learning. PMLR, 2021.
19. M Tan and QV Le. Efficientnetv2: Smaller models and faster training. arxiv 2021. arXiv preprint arXiv:2104.00298.
20. Fuyan Ma, Bin Sun, and Shutao Li. Facial expression recognition with visual transformers and attentional selective fusion. IEEE Transactions on Affective Computing, 14(2), 2021.