Exploiting Processor Heterogeneity to Improve Throughput and Reduce Latency for Deep Neural Network Inference

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

Exploiting Processor Heterogeneity to Improve Throughput and Reduce Latency for Deep Neural Network Inference

Olivier Beaumont*, Jean-François David*, Lionel Eyraud-Dubois*, Samuel Thibault†
*Inria Center of the University of Bordeaux, Bordeaux, France
Email: {olivier.beaumont, jean-francois.david, lionel.eyraud-dubois}@inria.fr
†University of Bordeaux, Bordeaux, France
Email: samuel.thibault@u-bordeaux.fr

***Abstract*—The growing popularity of Deep Neural Networks (DNNs) in a variety of domains, including computer vision, natural language processing, and predictive analytics, has led to an increase in the demand for computing resources. Graphics Processing Units (GPUs) are widely used for training and inference of DNNs. However, this exclusive use can quickly lead to saturation of GPU resources while CPU resources remain underutilized. This paper proposes a performance evaluation of a solution that exploits processor heterogeneity by combining the computational power of GPUs and CPUs. A solution is proposed for distributing the computational load across the different processors to optimize their utilization and achieve better performance. A solution for partitioning a DNN model with different computational resources is proposed. This solution transfers part of the load from the GPUs to the CPUs when necessary to reduce latency and increase throughput. The partitioning of DNN models is performed using METIS to balance the computational load to be distributed among the different resources while minimizing communication. The experimental results show that latency and throughput are improved for a number of DNN models. Potential applications include real-time processing systems such as autonomous vehicles, drones, and video surveillance systems where minimizing latency and maximizing throughput are critical.**

***Index Terms*—dynamic scheduling, graph partitioning, heterogeneous computing, latency optimization, real-time systems**

I. Introduction

The use of deep neural networks (DNNs) in a variety of domains, including computer vision, natural language processing, and predictive analytics, has led to an increase in the demand for computational resources. This is evidenced by the findings of Thompson et al. [28], who observed a growing need for more powerful computing platforms to meet the demands of DNN training and inference. Traditionally, Graphics Processing Units (GPUs) have been the privileged choice for training and inference of DNNs due to their high computational power and efficiency. In this context, inference refers to the process of using a pre-trained DNN model to make predictions about new data. Using GPUs exclusively for inference tasks can lead to scenarios where GPU resources become saturated while CPU resources remain underutilized.

While several effective approaches have been proposed, they are typically limited in their ability to balance the computational load between GPUs and CPUs. This underutilization of resources has a detrimental effect on the overall system performance, especially in real-time processing environments where latency (time to compute an inference including data transfers) and throughput (number of inferences per unit time) are critical performance metrics.

In this context, we propose a solution to efficiently exploit processor heterogeneity by combining both GPUs and CPUs in order to leverage the performance of the system. To control latency when using both "slow" CPUs and "fast" GPUs, we rely on partitioning a DNN model into two segments with different computational loads. The methodology also relies on the use of the dynamic runtime scheduler StarPU [5] to optimize the utilization of computational resources and to efficiently overlap communication and computation. We rely on METIS [18] for graph partitioning, which allows partitioning with varying computational loads while minimizing data transfer costs between partitions. Finally, to optimize our partitioned DNN models for inference, we use ONNX Runtime [10].

The specific contributions of our work are as follows:

- Detailed analysis of CPU core grouping to optimize resource utilization and improve performance.
- Methodology for partitioning DNN models using METIS to balance the computational load between GPUs and CPUs.
- Demonstration of performance gains in terms of latency and throughput across various configurations and neural network models.

The applications of this approach include systems requiring real-time processing such as autonomous vehicles, drones, and video surveillance systems, where latency minimization and throughput maximization are crucial objectives.

The following is a description of the organization of this paper: Section II provides an overview of the motivation and related work, while Section III presents the StarONNX approach. SectionIV covers the Optimized Inference Request Batching strategy. Section V addresses the topic of CPU core grouping. The section VI discusses the partitioning strategy of DNN models in the context of heterogeneous resources. Section VII examines the impact of multiple CUDA streaming

on the performance. Section VIII proposes an experimental comparison of StarONXX against the Nvidia Inference Server Triton [1]. In conclusion, Section IX presents suggestions for future research.

II. Motivation and Related Work

A. Motivation

Our solution aims to be applicable to a wide range of models and scenarios where maximizing throughput and minimizing latency are both critical.

There are a number of challenges associated with the exclusive use of GPUs. First, as models become more complex and data resolution increases, GPU memory can become scarce. Conversely, CPUs are often ignored due to their lower computational capacity [4][19], resulting in a global underutilization of available resources. In [28] and [27], the authors emphasize that the computational load related to inference and training will soon become prohibitive in technical, economic, and environmental terms, requiring the community to explore novel concepts or adopt more energy-efficient methods.

These challenges underscore the need to leverage other well-known approaches in the HPC world, such as those proposed by [24], [25], [21], which exploit heterogeneous computing architectures to use both GPUs and CPUs. By exploiting the complementarity of these two architectures, we will show that it is possible to distribute the computational load in an appropriate manner with minimal impact on latency, relying on DNN models partitioning.

B. Related Work

Optimization of DNN inference has been studied extensively, and a variety of approaches have been employed with the goal of maximizing resource utilization and reducing latency. Non-academic solutions for inference servers include several robust options. TensorFlow Serving, designed for deploying TensorFlow models, is an example of a solution designed specifically for this purpose. Amazon SageMaker, which is designed for large-scale model training and deployment. Azure ML is proposed for deployment on the Azure cloud. In this paper, we focus our comparison on the NVIDIA Triton Inference Server https://developer.nvidia.com/triton-inference-server.

With respect to throughput optimization, several solutions have been proposed. InferLine [8] is a system that optimizes and manages pipelining to meet latency constraints. InferLine provides an automated parameter selection process and a dynamic resource adjustment mechanism to respond to variations in request throughput. [12] optimizes video streaming for DNN applications. It dynamically adjusts video quality based on inference requirements. To maximize throughput, [29] employs a dynamic GPU cache, hierarchical memory architecture, and asynchronous updates. The Proteus [3] system implements a DNN to adaptively adjust the throughput of inference requests. In addition, they propose an adaptive batching algorithm to handle variations in request arrival times. Some studies concentrate on energy efficiency and resource management. As shown by [15], the energy efficiency of a system can be improved by grouping requests into larger batches. The system we propose automatically adapts the batch size to the request input rate. The approach proposed in [13] improves the quality of service (QoS) by dynamically adjusting the batch size and DNN model selection, at the expense of accuracy. In this paper, we do not consider the possibility of modifying model accuracy to cope with high throughput, but rather optimize the use of resources. The combination of these approaches is left for future work.

The use of parallel GPUs to optimize throughput has been considered in several papers. AlpaServe [22] optimizes resource utilization by serving multiple models through model multiplexing and parallelism. In a serverless environment, AMPS-Inf [16] proposes a partitioning approach for inferring large models. [31] addresses the issue of optimizing multi-tenant inference on GPUs by considering the concurrency of operations from different models on the GPU. Other approaches include AxoNN [9], which schedules neural network inference across different accelerators to balance performance and energy consumption. GSLICE [11] is a GPU-based framework that optimizes the use of GPU resources for inference computations. It uses a number of strategies, including adaptive batching and spatial multiplexing. The REEF [14] layer-by-layer scheduling system for DNN inference on GPUs prioritizes critical tasks by preempting less important tasks in real time. Finally, HIOS [20] optimizes the real-time inference latency of deep learning models across multiple GPUs through a hierarchical inter-operator scheduler.

LaLaRAND [17] improves real-time scheduling of DNN tasks on CPUs and GPUs through CPU-adaptive quantization. In their study, the authors of [2] investigate the potential of alternating execution between CPU and GPU for embedded devices. Their results indicate that some DNN operations can be processed more efficiently on the CPU than on the GPU, depending on the size of the input data. In their study, Wu et al. [30] propose a pipeline scheduling method to optimize CNN inference on heterogeneous multicore systems. Their approach uses an iterative bi-partitioning strategy to evenly distribute layers across CPU cores, thereby reducing latency and increasing throughput. The present paper follows this line of research and proposes to optimize the use of heterogeneous resources based on model partitioning and efficient resource utilization (through CPU core grouping and the use of GPU multi-stream).

III. Overview of ONNX and StarPU Integration in StarONNX

This section introduces the main features of ONNX [10] and StarPU [5]. StarONNX, which was described in [6], integrates both runtimes to take advantage of both CPUs and GPUs for inference.

A. ONNX Runtime

ONNX Runtime [10] is an open source platform for the optimization and execution of DNN inference. It provides ex-

cellent interoperability through its compatibility with multiple frameworks, including PyTorch and TensorFlow, by converting a DNN model into the ONNX format. This enables the direct use of pre-trained models from PyTorch and TensorFlow, while taking advantage of the performance and optimization benefits of the ONNX runtime.

In terms of performance, the runtime has been optimized for use on a variety of hardware platforms, including CPUs, GPUs, and other hardware accelerators. ONNX Runtime's general-purpose and processor-specific optimizations for DNN models[1], such as operator fusion and compiled graphs, reduce computation time. As a result, the optimizations for deployment on CPUs are different from those for deployment on GPUs. These software and hardware optimizations ensure optimal performance for model deployment compared to PyTorch and TensorFlow, which prioritize development and experimentation.

However, there are some limitations when using ONNX Runtime. An ONNX Runtime session is an instance created to run an ONNX model and contains all the information necessary for that execution, including model parameters and configuration. An ONNX Runtime provider refers to the backend used for model execution, such as CUDA for NVIDIA GPUs, which improve performance based on the available hardware. For example, the initial step of initializing ONNX Runtime sessions and memory is particularly expensive when using the CUDA provider. This makes dynamic batch size changes challenging, as each size change requires a reset to reallocate memory, impacting latency.

B. StarPU

StarPU [5] is a runtime that enables the scheduling and execution of tasks on heterogeneous architectures, including CPUs, GPUs, and other hardware accelerators. StarPU provides great flexibility in managing tasks and data. It is unique in its ability to model task performance based on a performance history. This history allows prediction of execution times and data transfers, which in turn facilitates dynamic scheduling of tasks based on the current workload and other details about the overall system state. Another notable feature of StarPU is its support for automatic dependency management between tasks. The automatically generated task graph allows StarPU to identify dependencies between tasks. Furthermore, the runtime supports dynamic task scheduling. This capability is accompanied by the ability to perform asynchronous and overlapping data transfers, allowing data transfers to overlap with computations. As a result, StarPU can reduce idle resource time by ensuring that the necessary data for upcoming tasks is available in advance.

However, there are challenges to using StarPU. Implementing accurate performance models for each type of task can be complex, requiring a deep understanding of hardware characteristics and specific workloads. Furthermore, while StarPU is capable of automatically managing data transfers and dependencies, the initial configuration for complex applications may require development effort.

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

C. StarONNX

StarONNX [6] integrates the ONNX runtime with StarPU for dynamic management of heterogeneous computing resources, including CPUs and GPUs. This integration enables better resource utilization by dynamically scheduling inference tasks across different types of hardware, leveraging the strengths of both runtimes. StarONNX can be used with a partitioned model, where each part of the model is seen by StarPU as a task, for which StarONNX can produce an executable version on each resource type.

In this paper, we introduce the possibility to additionally partition the computing resources themselves and to partition the model accordingly. During deployment, an initialization phase is initiated, during which the CPU cores are grouped and the threads used by ONNX Runtime and StarPU are identified, including the binding of the CPU cores. A new ONNX Runtime session is created for each partition and processor, and the sessions are then connected to the same GPU streams. After allocating the requested memory and compiling the model, a second phase initializes the performance history. Once these steps are complete, the inference server can be launched.

Figure 1 shows the architectural framework of StarONNX, along with an illustrative example of inference on a model partitioned into two distinct parts.

When the application receives an inference request, the request generates two different StarPU tasks, one for each partition. In the initial phase of the process, the two tasks are submitted to the scheduler. StarPU is able to detect dependencies between tasks. In this case, the output data from the first partition is used as input for the second, indicating that the first task must be run before the second.

Suppose the scheduler selects the GPU to run the first task based on a number of criteria, including queue length, performance history, system state, and so on. In Steps 3, 4, and 5, the data for the initial task is prefetched in case the GPU is busy with other computations. When the GPU becomes available, it initiates processing of the initial task via the ONNX Runtime CUDA provider.

Meanwhile, assume that in step 6, the scheduler determines that the second task should be placed on the CPU. Then in steps 7, 8 and 9 the data is transferred to the CPU. In step 10, the GPU completes the computation of the first task and sends the results to the CPU. If the CPU is busy processing other tasks, the data is prefetched. In the absence of further instructions, the CPU initiates the processing of the second task, resulting in the transfer of the inference result to the application.

IV. Optimized Inference Request Batching

This section outlines a scenario in which StarONNX could potentially be employed in the processing of inference requests.

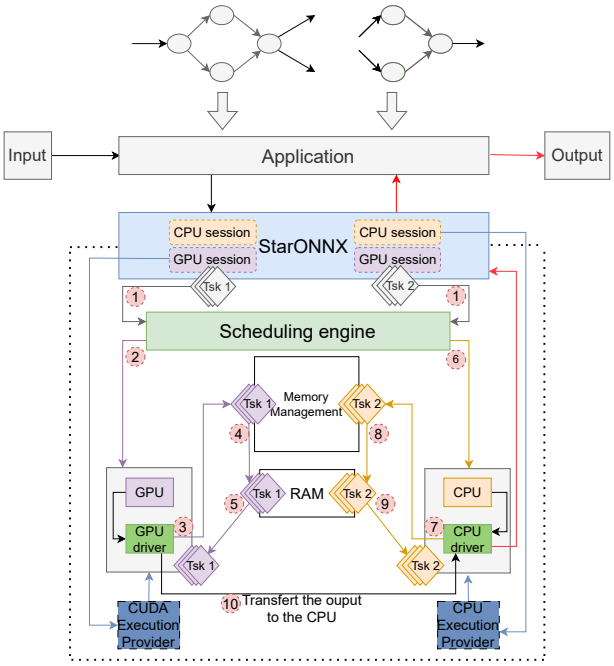
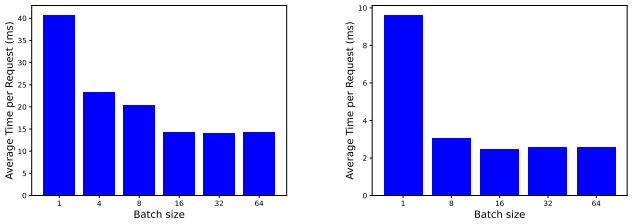


Fig. 1: StarONNX's architecture



(a) With 32 CPU's cores (b) On the GPU

Fig. 2: Nfnet batching efficiency

We consider a server that receives a constant stream of inference requests at a fixed rate. In order to optimize resource utilization through CPU and GPU parallelization and minimize the overall latency, these requests must be batched.

A. Grouping requests into batches

Inference requests are grouped together in order to take advantage of the parallelization on both the GPU and CPU. One limitation of batching is that it introduces latency when forming the batch. Consequently, the lower the request rate, the longer it takes to form a large batch. This is one reason why it is necessary to select the optimal batch size that minimizes latency at a fixed rate to achieve the best batch size. Batching improves processor utilization by leveraging their simultaneous processing capacity, as illustrated in Figure 2.

B. Batching time calculation

Assume that a batch has a size of N requests, and the requests arrive at a time interval T. The time required to form a batch of N requests is then given by:

$$T_{\text{batch}} = T \times (N - 1)$$

This time includes the interval between requests and the gradual formation of the batch until it reaches size N.

C. Total Latency Calculation

The total latency to process an inference request includes several components:

- **Batch formation time T_{batch}**: The time required to group the requests into a batch of size N.
- **Data transfer time**: The time taken to transfer data between the CPU and the GPU and vice versa.
- **Computation time on the GPU and CPU**: The time taken to process the data by the GPU and the CPU.

The total latency L_{total} is thus the sum of these different components:

$$L_{\text{total}} = T_{\text{batch}} + T_{\text{transfer}} + T_{\text{computation}}$$

This batching approach helps to reduce overall latency and increase throughput while optimizing CPU and GPU resource utilization through parallelization.

V. Optimizing Inference Workloads with CPU Core Clustering Strategies

This section introduces the benefits of clustering CPU cores. The work of [30] proposes a pipelined execution approach to optimize Convolutional Neural Networks (CNNs) on heterogeneous ARM multicore systems. The goal is to reduce the latency associated with data movement between CPU cores by clustering the cores into small compute clusters that form the pipeline stages. Pipelined execution circumvents the use of buses, which are inherently slower for intra-cluster transfers. Therefore, the goal is to avoid using the Cache Coherent Interconnect (CCI) bus or the L2 cache (second-level cache) as much as possible by no longer parallelizing the kernels across all CPU cores. The CCI bus is responsible for maintaining cache coherency between different processor cores, while the L2 cache is larger in capacity but also slower than the L1 cache. The results of [30] show that this method improves throughput by an average of 73% compared to a conventional multi-thread scheduler.

In order to understand the rationale behind the improved performance obtained by clustering CPU cores, it is necessary to describe the concept of cache locality and its importance in the context of Non-Uniform Memory Access (NUMA) architectures.

A. NUMA Node and CPU Package

A *Non-Uniform Memory Access* (NUMA) is a system architecture in which memory is directly accessible by certain cores of a processor but not by others. In a NUMA architecture, each processor or group of processors owns some local memory that it can access at a faster rate than non-local memory. The primary advantage of NUMA is the increased performance achieved by taking advantage of memory locality, which means minimizing the number of accesses to non-local memory. Accordingly, cores on the same NUMA node share the L2 cache.

A *CPU socket* (or package) is the physical unit containing one or more processor cores. A socket can contain multiple cores, a shared L3 cache, and other components such as memory controllers. Clustering cores into small groups within the same socket, a feature of NUMA architectures, reduces the latency associated with inter-core communication, thereby optimizing performance.

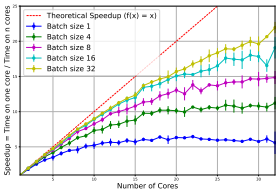
Multi-socket clustering refers to the use of multiple CPU packages (or sockets) within the same system. Each socket contains its own cores and local memory, forming multiple NUMA nodes. To optimize performance, it is necessary to minimize the amount of data exchanged between sockets. This can be achieved by assigning tasks in a way that maximizes data locality, ensuring that the data required for a task is primarily accessible within the same socket where the task is being executed.

B. Experimental Setup

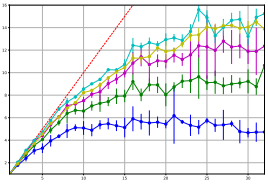
All experiments were performed on a platform consisting of a 16 GB NVIDIA P100 GPU and two Broadwell Intel Xeon E5-2683 v4 processors running at 2.1 GHz, with 256 GB of RAM (8 GB per core). A number of well-known DNN models were used, including GoogLeNet [26], Nfnet (Normalized-Free Network) [7], and Vit-face-expression [23], due to their prevalence in computer vision applications. Nfnet is characterized by its ability to operate without batch normalization, and Vit-face-expression uses transformers for facial expression recognition. The software environment used in this paper consists of ONNX Runtime 1.8.1, StarPU 1.3.4, METIS 5.1.0, Python 3.8, and NVIDIA CUDA 12.0 drivers.

C. Limitation of CPU Core Clustering using Default ONNX

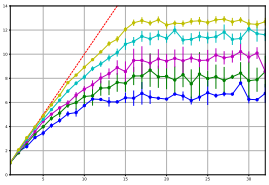
First, we evaluate the ability of ONNX Runtime to take advantage of CPU core grouping capabilities. The x-axis in Figure 3 represents the number of cores used to process the inference, with a maximum of 32 cores. The y-axis represents the speedup, which is calculated by dividing the computation time of a batch on a single core by the computation time on n cores. The different plots illustrate the speedup achieved for different batch sizes and different models. These results show that the speedup gains degrade as the number of CPU cores increases, reaching a plateau. This indicates that there are limits to parallelizing with ONNX Runtime on a large number of CPU cores. Consequently, it is necessary to cluster these



(a) GoogleNet

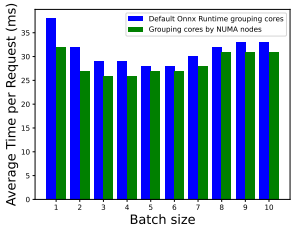


(b) Nfnet

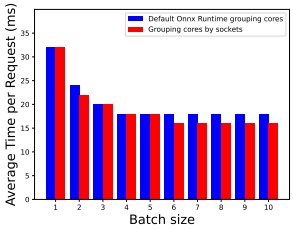


(c) Vit-Face-Expression

Fig. 3: A comparison of the speedups achieved by ONNX Runtime for different models, according to CPU core grouping.



(a) Grouping cores by NUMA



(b) Grouping cores by sockets

Fig. 4: ONNX Runtime grouping cores vs by socket, vs by NUMA on Nfnet

cores into smaller groups in order to achieve the theoretical speedup when performing inference.

It is also important to note that ONNX Runtime does not cluster CPU cores based on locality by default. Instead, StarONNX clusters CPU cores based on whether they are on the same NUMA node or within the same socket. Figure 4 illustrates the advantage of clustering CPU cores by NUMA node or by socket over having ONNX Runtime choose, in the worst case, which cores to group. For this experiment, we have B batches to process with different batch sizes. In Figure 4a, we have implemented two configurations for groupings, one where ONNX Runtime creates 4 groups of 8 cores, and one where cores that are on the same NUMA node are grouped together. In Figure 4b we perform the same study, but this time for cores on the same socket, 2 groups of 16 cores.

The observed results can be explained by a number of factors that are inherent to the architecture of NUMA systems. When performing inference on the same models, a performance degradation was observed when ONNX created groups.

For small batches, the group of cores on the same NUMA node shows superior performance. This is due to the proximity

of the cores to local memory, which reduces the latency associated with memory access. In this configuration, the cores benefit from faster access to local memory without having to traverse NUMA interconnects. In addition, lower memory bandwidth requirements and improved local cache utilization contribute to this performance.

The benefits of proximity to memory and low latency are most significant for small batches, favoring cores grouped on the same NUMA node. However, the issue of memory contention, which is minimal for small batches, becomes a limiting factor for larger batches.

In the scenario where the goal is to partition CPU cores to minimize latency for a given throughput, grouping cores based on their locality avoids the costs associated with transfers between different CPU caches. By grouping the cores into smaller clusters, it is possible to allow ONNX Runtime to better utilize these resources, thereby avoiding degradation from the theoretical speedup.

VI. Optimizing Performance with METIS-based Partitioning

The following section will present a methodology for optimizing performance through the use of METIS-based partitioning.

We assume that the minimization of latency and the maximization of throughput can be achieved through the implementation of dynamic partition scheduling on both GPU and CPU resources, thereby conferring a number of advantages. It is anticipated that the introduction of parallelism into the inference data processing will be achieved through the partitioning of neural networks. It is anticipated that the initial partition, which carries a significant portion of the computational load, is expected to be processed without congestion on the GPU. In contrast, the subsequent partition, which has a relatively lower computational load, is expected to be executed on either the GPU or the CPU, based on various parameters that are automatically defined by StarPU (queue size, processing efficiency on that processor, etc.).

The principal objective of this section is to improve the latency and the throughput of a DNN model through the partitioning of its components. The other objective of this optimization is to achieve a balance in the computational load and to minimize the costs associated with data transfer between the GPU and CPU, with the goal of improving the overall performance of the system. To achieve this, a two-step process is employed. At the outset of the process, the mean computation time on the GPU for each node is determined through the use of ONNX Runtime. Subsequently, the computational load is distributed using the METIS algorithm.

It is also important to note that each transfer introduces an additional latency, which can become significant, particularly for large batch sizes. To address this issue and reduce overall latency, it is essential to minimize data transfers and ensure a balance between data transfer times and computation times. The objective is to optimize the sequence of GPU transfers and computations in order to achieve complete overlap, thereby eliminating additional delays. The aforementioned points can be summarized as follows:

- **Minimization of Data Transfers:** By partitioning the DNN model in a way that minimizes data transfers between the GPU and CPU, the impact on latency is reduced.
- **Pipelining Transfers and Computations:** It is important to guarantee that the times required for data transfer are aligned with the times required for CPU computation. This guarantees that transfers and computations are conducted in an integrated and concurrent manner, thus eliminating any additional delays. This can be achieved by optimizing the sequence of GPU transfers and computations to ensure complete overlap.

A. Methodology

In order to transform the ONNX model into a graph, the model was converted into a graph structure. The graph-based approach allows for the partitioning of the model. In this representation, each node is associated with an operation within the model, while edges indicate the transfer of tensors between these operations.

In the subsequent phase, the dimensions of the output tensors for each node were determined through the application of the ONNX data types and tensor dimensions. Subsequently, the graph was encoded in METIS format, incorporating the weights of the nodes and edges. Subsequently, the partition weights were defined with the objective of achieving a desired distribution of computational load between the two partitions. Moreover, METIS was instructed to identify the partition that would result in the fewest data transfers.

1) Measuring Computation Time: In order to determine the precise computational requirements of each node in our DNN model, we conducted multiple inferences on a GPU. The process of conducting one hundred repetitions for each node was repeated until the requisite number of inferences had been completed. The constituent nodes of the model were executed in isolation during each inference, with the elapsed time for each computation recorded.

By averaging these times, an estimate of the computation time for each node was obtained. The provided average offers insight into the computational load imposed on the GPU for each node of the DNN model.

2) Partitioning with METIS: Once the necessary computation time for each node had been determined, the METIS software was employed to create partitions with varying computational loads. METIS is a graph partitioning software that provides partitions of subgraphs while minimizing the transfer costs between them. The software utilizes multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes [2] to achieve a balance in the computational loads and identify the partitioning that minimizes inter-partition communications. This is crucial for achieving an appropriate balance between communication and computation.

[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

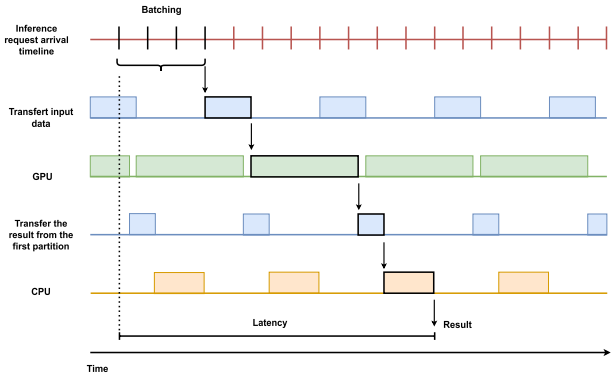


Fig. 5: Example illustrates the pipelining process, with the initial partition executed on the GPU and the subsequent partition executed on the CPU.

3) Implications of Data Transfer Costs on Overall Performance: The impact of partitioning is not solely contingent upon the distribution of computational load between the GPU and CPUs; it is also influenced by the data transfer costs between these computing units. If the DNN model is partitioned at points where data transfers are important, the resulting increase in transfer times will lead to an overall increase in latency. It is thus imperative to identify a partitioning strategy that minimizes data transfers while maintaining a close approximation to the desired computational load distribution.

Consequently, the effective control of data transfer costs is a prerequisite for heterogeneous computing. In order to achieve optimal performance within the constraints of a given application, a partitioning strategy must take into account the aforementioned factors.

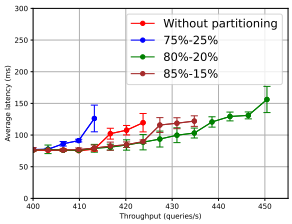
As illustrated in Figure 5, which depicts an exemplar of the pipelining process, the initial partition is executed on the GPU, while the subsequent partition is executed on the CPU. Subsequently, the data is transferred from the GPU to the CPU, where it is then subjected to further computations. The concurrent execution of transfers and computations serves to minimize the overall latency. Upon completion of processing a segment of data, it is transferred immediately to the CPU for the processing of the next partition, thus eliminating any unnecessary idle time.

To avoid an increase in latency resulting from partitioning, the transfer time of the results from the initial partition to the subsequent partition, added to the computation time of the latter, must not exceed the computation time of the former on the GPU.

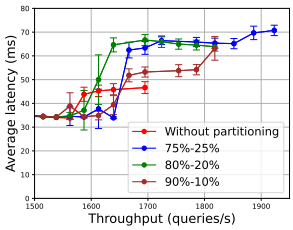
Consequently, by minimizing transfer costs and overlapping computations and transfers, latency can be reduced and throughput maximized, thereby improving the overall system performance.

B. Partitioning for a DNN Model

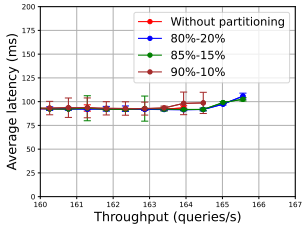
In this section, we present the results of an experiment with a fixed batch in which we performed several partitionings using METIS, varying the distribution of the GPU computational



(a) Nfnet, batch size of 16



(b) GoogleNet, batch size of 27



(c) Vit-Face-Expression, batch size 8

Fig. 6: StarONNX's ability to handle heterogeneous resources with different partitioning

load. Figure 6 illustrates the impact of partitioning on latency for different models with varying computational loads on the first and second partitions. The partitions were set to 75%-25%, 80%-20%, and 85%-15% or 90%-10%, respectively. For each partitioning, the latency and throughput for a fixed batch size were quantified.

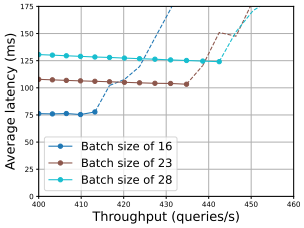
As illustrated in Figure 6a, the 80%-20% partitioning exhibits the lowest latency before congestion, while the 75%-25% partitioning demonstrates the highest throughput in accordance with the imposed rate.

It is important to note, however, that networks with higher bandwidth, as illustrated in Figures 6b and 6c, exhibit reduced sensitivity to throughput partitioning. In the case of nfnet, the 80%-20% partitioning is observed to be the most effective. In contrast, for Vit-Face-Expression, the two partitionings are evenly split between 85%-15% and 80%-20%. This is due to the fact that these networks have relatively low throughput, which restricts the window of opportunity for modifying the partitioning before congestion occurs.

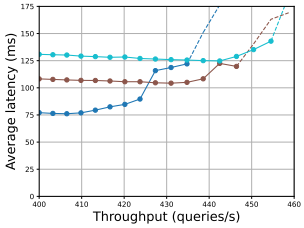
1) Selecting the Batch Size to Minimize Latency: The objective of this experiment is to identify the optimal batch size that minimizes latency while maintaining a specified throughput. The following explanations will focus on Nfnet.

Figure 7a shows the latency performance with a fixed batch for each plot on Nfnet, executed on a GPU. As detailed in Section IV-B, the latency measurements demonstrate a reduction in latency as throughput increases for a given batch size. This reduction is more pronounced with larger batches, as a higher throughput allows for the construction of a larger batch in a shorter time.

Nonetheless, in a subsequent phase, as illustrated by the dotted plots, a notable increase in latency is obvious. This

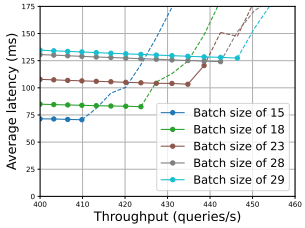


(a) Without partitioning

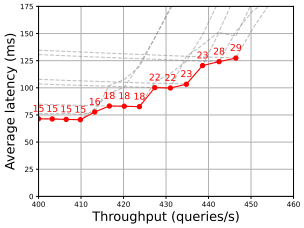


(b) 85%-15% partitioning

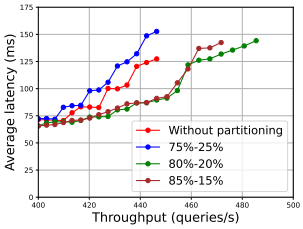
Fig. 7: Examine the impact of partitioning on latency with varying batch sizes on Nfnet



(a) Batch size latency without partitioning



(b) Optimize batch size to minimize latency without partitioning



(c) Minimize latency by selecting optimal batch size for differents partitions

Fig. 8: Selection of the optimal batch that minimizes latency for various partitioning schemes with Nfnet

suggests that the GPU is unable to maintain the necessary throughput due to congestion. In such instances, it is advisable to increase the batch size in order to ensure that the GPU is able to maintain the requisite throughput.

Figure 7b illustrates the latency performance with a fixed batch for each plot on Nfnet with an 85%-15% partition on a GPU and our 32 CPU cores.

It can be observed that partitioning and employing CPUs to assist the GPU results in an increased throughput, which delays the onset of system congestion. By deferring the onset of congestion to a higher throughput for each batch, it is possible to maintain a smaller batch size for a longer period of time, as opposed to switching to a larger batch size. This is particularly obvious in the case of a batch size of 16, which allows for a reduction in latency.

Figure 8 depicts an experiment wherein, for a fixed throughput, the optimal batch size that minimizes latency for each partition is determined. Figure 8a illustrates the latency of varying batch sizes under a constant throughput. Figure 8b presents the batch size that achieves the minimum latency for each fixed throughput.

The majority of the batches illustrated in Figure 8a are represented by the dotted gray lines. Finally, Figure 8c illustrates the selection of the latency-minimizing batch for each fixed throughput, considering each partition individually. As previously discussed in Section VI-B, there is only one partitioning scheme that simultaneously minimizes latency and maximizes throughput for the Nfnet model.

In instances where throughput is suboptimal, StarPU assigns both tasks associated with both partitions for inference processing to the GPU. As throughput increases, there is an opportunity to run the second partition on the CPU, which frees up the GPU to handle more computationally intensive tasks. In this way, the total latency is reduced, as the first, heavier partition is processed rapidly on the GPU, while the second partition can be processed on the CPU. This approach optimizes the utilization of the GPU for the most intensive tasks, prevents congestion on the GPU by distributing tasks to the CPU during periods of peak load, and reduces total latency by minimizing waiting times.

VII. Evaluation of CUDA Multistreaming in Neural Network Performance

The objective of this section is to conduct experiments with a variety of configurations with the aim of reducing latency and increasing throughput. The objective of this study is to demonstrate how different setups influence performance and provide insights into the underlying mechanisms.

It has been observed that certain tasks within the domain of neural networks do not fully exploit the whole set of cores of the GPU. CUDA multistreaming may prove an effective solution to this problem. The execution of multiple instruction streams in parallel allows to execute several of such tasks in parallel, thereby fully utilizing the GPU.

However, while multistreaming offers numerous performance benefits, it also increases the complexity of programming, which presents a challenge for developers. It is indeed imperative that developers exercise careful management of synchronization between streams to prevent potential races. In the absence of StarPU, the application would necessitate meticulous review to circumvent synchronization and concurrency issues. The dynamic schedulers of StarPU, however, already provide support for distributing tasks to available processors and streams. Enabling multistreaming in StarPU boils down to changing a StarPU parameter.

Figure 9 illustrates the relationship between throughput and latency for the Nfnet model with varying CUDA streams and partitioning schemes, with a fixed batch size.

As illustrated in Figure 9a, the results obtained in the experiment without partitioning and with exclusive GPU utilization align with expectations. The configuration with a single CUDA

stream reaches its congestion point first, followed by those with two, three, and four streams. It is obvious for this DNN model that an increase in the number of streams results in improved throughput while concurrently reducing latency.

As illustrated in Figures 9b, 9c, and 9d, the configuration with three streams demonstrates a better capacity to maintain higher throughput compared to that with four streams. Each CUDA stream strives to optimize the utilization of the GPU's available resources. However, an increase in the number of streams may result in heightened competition for the available resources, leading to contention and a subsequent reduction in overall performance.

The execution of several tasks in parallel on many CUDA streams results in the sharing of the same GPU resources, including compute units, shared memory, and caches. This common usage can result in resource contention. For example, if several tasks make extensive use of shared memory or frequently access the same cache, overall efficiency may be diminished. Furthermore, the management of streams and their synchronization can introduce additional processing overhead. While this overhead is typically minimal, it can become significant when multiple operations are executed in parallel. The necessity of certain synchronization operations to guarantee data integrity often results in an additional delay.

The results demonstrate that, for this DNN model and a fixed batch size, an increase in the number of streams facilitates the management of greater throughput and can reduce latency. It is noteworthy that multistreaming provides a substantial enhancement in performance when partitioning is not employed. However, the advantage of multistreaming is lost when partitioning is employed. In this example, four streams are necessary to achieve optimal performance in the absence of partitioning, whereas two streams are sufficient to attain a nearly identical result with partitioning. Moreover, the results demonstrate that partitioning yields greater performance improvements than multistreaming.

VIII. Comparative Analysis of Triton Inference Server and StarONNX Performance

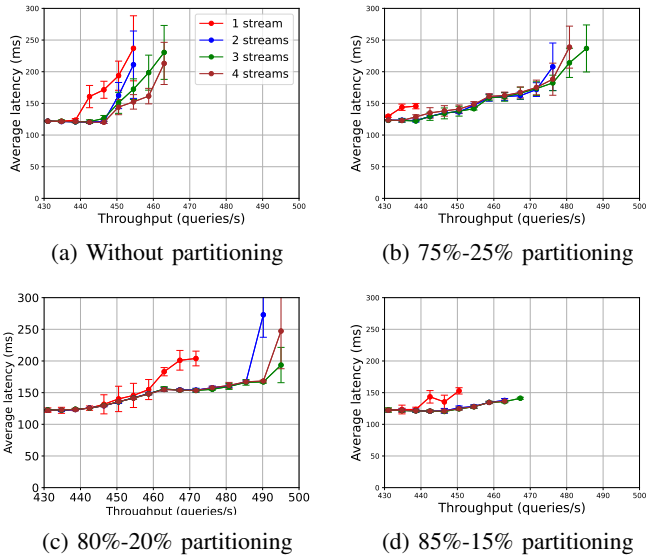
The objective of this section is to undertake a comparative analysis of the performance of the Triton Inference Server and StarONNX. This analysis has two objectives. The first is to evaluate both platforms in terms of latency and throughput. The second is to elucidate the underlying causes of any observed discrepancies in performance.

A. Triton Inference Server

Triton Inference Server [3], developed by NVIDIA, is a software platform designed for the deployment of DNN models. It is compatible with a wide variety of frameworks, including TensorFlow, PyTorch, ONNX Runtime, and TensorRT, thereby providing users with a high degree of flexibility.

Triton is capable of supporting numerous frameworks, and can optimize their performance through the use of techniques

[3]The Docker version of Triton nvcr.io/nvidia/tritonserver:24.01-py3 is employed



(a) Without partitioning (b) 75%-25% partitioning

(c) 80%-20% partitioning (d) 85%-15% partitioning

Fig. 9: Executing Nfnet with different partitioning loads and running on multiple streams with a batch size of 27

such as dynamic batching. It can be deployed in a cloud environment, on-premises, or at the edge. The platform is integrated with Prometheus for the purpose of monitoring performance, thereby enabling users to monitor and optimize models in real time. Moreover, Triton is capable of serving multiple models concurrently, which is advantageous for complex applications that necessitate multiple inferences.

B. Comparative Performance Analysis

We conducted a comparative analysis of the performance of StarONNX in the absence of partitioning and that of the Triton Inference Server, which was utilized with one and four instances of the DNN model on a GPU with GoogleNet. In this context, the term "instance" is used to describe a particular configuration of the inference server that is currently running. Each instance is capable of loading and serving a DNN model, managing inference queries, and optimizing performance in accordance with the specific requirements of the application. Instances may be deployed independently or in parallel in order to balance the load and enhance scalability.

Figure 10 shows the performance of an experiment performed on GoogleNet with a P100 GPU. It compares the latency of StarONNX in the absence of partitioning with Triton inference server, with one and four instances of the DNN model. Even with a single computation stream, StarONNX demonstrates better performance in terms of latency and throughput compared to Triton.

Figure 11 show the traces of four GoogleNet instances deployed on Triton. The green rectangles represent data transfers between the host and GPU memory, blue rectangles indicate computations performed by a CUDA kernel on the GPU, and the red rectangles (added post-hoc) show the complete execution of a batch on GoogleNet.

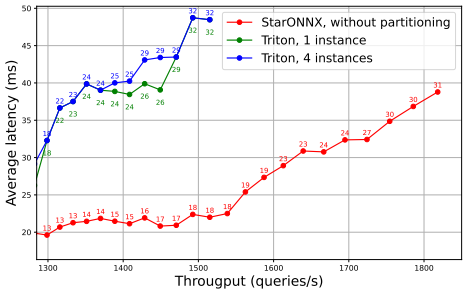


Fig. 10: Triton Inference Server vs StarONNX

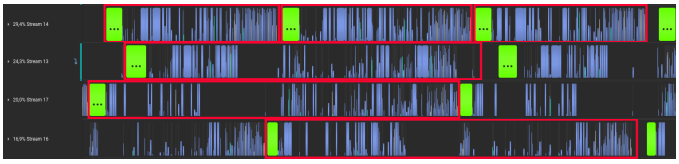


Fig. 11: Triton Inference Server trace with four GoogleNet model instances

It is noteworthy that the first Triton stream displays a performance discrepancy in comparison to the other Triton streams. Furthermore, it is obvious that there are considerable periods of GPU idleness observed on the other Triton streams, which suggests that the computational activity is minimal, thus increasing the latency of the corresponding batches. Consequently, a single instance of Triton, as shown in Figure 10, sometime demonstrates better performance.

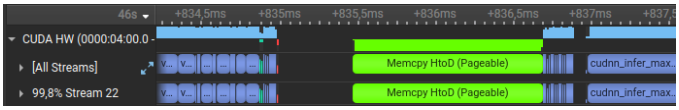
Figure 12a shows a trace of Triton Inference Server configured with a single instance, whereas Figure 12b shows a trace of StarONNX configured with one stream for computation and another for transfers.

It can be observed that, although Triton server typically utilizes pinned memory by default, the Triton Inference Server does not overlap transfers and computations; rather, all operations are conducted on a single stream.

In contrast, StarONNX performs overlapping on computation and communication, utilizing prefetching whereby data from the subsequent batch is copied while computations are performed on the current batch. These optimizations collectively improve the performance of StarONNX, both in terms of latency and throughput when compared to Triton Inference Server.

IX. Conclusion and Future Work

This paper presents an evaluation of a method designed to optimize deep neural network (DNN) inference by leveraging the capabilities of both GPUs and CPUs. The study concentrates on performance metrics that are of particular importance for real-time systems, such as latency and throughput. The methodology employs the METIS software for the partitioning of DNN models and the distribution of computational loads, with the objective of improving resource utilization.



(a) The traces of Triton Inference Server

(b) The traces of StarONNX

Fig. 12: Traces of Triton Inference Server and StarONNX

Through comprehensive experimentation, it was observed that notable improvements in performance were achieved, particularly in the reduction of latency and enhancement of throughput. By partitioning the models and employing the dynamic runtime scheduler StarPU, an overlap between communication and computation was achieved, thereby balancing the load between GPUs and CPUs. This approach ensures that the system's resources are fully utilized, addressing the issues of GPU saturation and CPU underutilization.

A comparative analysis with the NVIDIA Triton Inference Server indicates that the StarONNX solution demonstrates superior performance, particularly in high-throughput scenarios. The incorporation of CUDA multistreaming within StarONNX serves to improve performance, demonstrating the potential of parallel processing to optimize GPU utilization. In conclusion, this solution provides a framework for real-time processing applications, such as autonomous vehicles and video surveillance systems, where minimizing latency and maximizing throughput are of paramount importance.

Further research should investigate the integration of additional hardware accelerators, the refinement of partitioning strategies for complex network architectures, and the optimization of dynamic scheduling algorithms.

References

[1] Triton Inference Server: Open-source ai inference serving. https://developer.nvidia.com/nvidia-triton-inference-server. Accessed: [2024/03/20].

[2] 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.

[3] Sohaib Ahmad, Hui Guan, Brian D Friedman, Thomas Williams, Ramesh K Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. 2024.

[4] Shapna Akter, Hossain Shahriar, and Alfredo Cuzzocrea. Autism disease detection using transfer learning techniques: performance comparison between central processing unit vs graphics processing unit functions for neural networks. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2023.

[5] 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.

[6] Olivier Beaumont, Jean-François David, Lionel Eyraud-Dubois, and Samuel Thibault. StarONNX: a Dynamic Scheduler for Low Latency and High Throughput Inference on Heterogeneous Resources. In *HeteroPar 2024 - 22ND INTERNATIONAL WORKSHOP Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*.

[7] 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.

[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[9] 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.

[10] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: 1.8.1.

[11] 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*, 2020.

[12] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020.

[13] 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*.

[14] 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.

[15] Yoshiaki Inoue. Queueing analysis of gpu-based inference servers with dynamic batching: A closed-form characterization. *Performance Evaluation*, 2021.

[16] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Amps-inf: Automatic model partitioning for serverless inference with cost efficiency. In *Proceedings of the 50th International Conference on Parallel Processing*, 2021.

[17] 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)*.

[18] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 38:7–1, 1998.

[19] Luke Kljucaric and Alan D George. Deep learning inferencing with high-performance hardware accelerators. *ACM Transactions on Intelligent Systems and Technology*, 2023.

[20] 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.

[21] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[22] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*.

[23] 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.

[24] K Raju and Niranjan N Chiplunkar. Performance enhancement of cuda applications by overlapping data transfer and kernel execution. *Applied Computer Science*, 2021.

[25] Dominik Schäfer, Janick Edinger, Martin Breitbach, and Christian Becker. Workload partitioning and task migration to reduce response times in heterogeneous computing environments. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*.

[26] 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.

[27] Zhenheng Tang, Shaohuai Shi, Wei Wang, Bo Li, and Xiaowen Chu. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[28] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

[29] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. A gpu-specialized inference parameter server for large-scale deep recommendation models. In *Proceedings of the 16th ACM Conference on Recommender Systems*, 2022.

[30] 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.

[31] 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.