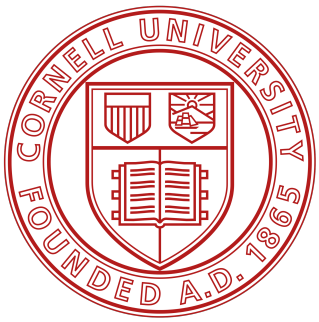
MAPPING NEURAL NETWORK INFERENCE ONTO HETEROGENEOUS HARDWARE PLATFORMS

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Science



by
Yassine Ghannane
August 2023

ABSTRACT

Datacenters are evolving towards heterogeneity, incorporating specialized hardware for tasks such as networking, video processing, and particularly deep learning. To effectively harness the compute capabilities of modern heterogeneous datacenters, this thesis proposes an approach for compiler-level partitioning of deep neural networks (DNNs) across interconnected hardware devices.

We present a comprehensive framework for heterogeneous DNN compilation, offering automatic partitioning and device mapping. Our scheduler integrates an exact solver, utilizing a mixed integer linear programming (MILP) formulation, and a modularity-based heuristic for scalability. Additionally, we introduce a theoretical lower bound formula to assess the quality of heuristic solutions, enabling the evaluation of optimal solutions.

We evaluate the proposed scheduler by optimizing both traditional DNNs and randomly-wired neural networks, while considering latency and throughput constraints. Our experiments are conducted on a heterogeneous system consisting of a CPU and two distinct GPUs. Compared to simply running DNNs on the fastest GPU, our framework achieves latency reductions of over 3× and throughput improvements of up to 2.9× by automatically leveraging both data and model parallelism.

Furthermore, our modularity-based "splitting" heuristic significantly enhances solution runtime by up to 395×, without compromising solution quality compared to the exact MILP approach. Additionally, it outperforms alternative heuristic baselines by 30-60% in terms of solution quality.

Lastly, we present two case studies to demonstrate the capabilities of our sched-

uler. The first case study investigates performance in memory-constrained environments, while the second explores the extension of our framework for scheduling large language models across multiple heterogeneous servers by leveraging symmetry in the hardware setup.

Overall, this research contributes to the efficient deployment of DNNs in heterogeneous datacenters through compiler-level partitioning, showcasing improved latency, throughput, and solution scalability.

BIOGRAPHICAL SKETCH

Yassine Ghannane is currently pursuing a Master's degree in Electrical and Computer Engineering at Cornell university. Prior to that, he earned his undergraduate engineering degree at Ecole Polytechnique in France majoring in computer science after two years of higher school preparatory classes in Morocco. Yassine's fascination with combinatorial optimization stems from the profound impact it can have on addressing complex problems across various domains. Eager to delve deeper into this field, he sought to combine his theoretical knowledge with practical applications to develop innovative solutions that could address real-world challenges. Recognizing the potential of his Master's program as an incubator for advanced research, Yassine seized the opportunity to contribute to the existing body of knowledge and make meaningful strides in his chosen field.

To my family and friends for their unwavering support.

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to all those who have contributed in any form to the completion of my master's degree.

First, I am deeply indebted to my supervisor, Pr. Mohamed Abdelfattah, for his invaluable guidance and continuous support throughout this research endeavor. His insightful feedback and encouragement in my abilities have been instrumental in shaping this thesis and my approach to research work in general. I am sincerely grateful for entrusting me with such an ambitious project and providing me with the opportunity to immerse myself in this academic journey.

I am also grateful to my colleagues who offered their assistance and listening ear during throughout my time at Cornell Tech. Our heated brainstorming discussions and their willingness to share patiently their knowledge on topics that I have been far from familiar have been truly invaluable and are something that I will look back at fondly.

Finally, I am forever grateful to my beloved family for their profound understanding and unwavering encouragement throughout my studies. Their consistent motivation has served as the primary driving force behind any of my accomplishments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Deep neural networks (DNNs) have emerged as an important computing paradigm making significant breakthroughs in many fields. However, DNNs are both computationally-intensive and memory-hungry, leading to a major hardware restructuring of modern datacenters to keep up with this insatiable compute demand. GPUs are becoming commonplace, FPGAs have been included by companies like Microsoft [22], and custom DNN accelerators such as Google's TPU [49] are continuously being developed. DNNs themselves are composed of a growing list of diverse building blocks such as convolutions, matrix-multiplications, element-wise operations, non-linear functions and shape transformations. Each of those primitives exhibits different vectorization patterns, sparsity and quantization tolerance and so may be suitable for implementation on different hardware accelerators [2, 63].

In addition to hardware heterogeneity, DNN topologies are becoming evermore irregular and complex thanks to their automated design through neural architecture search (NAS) [93]. NAS has demonstrated considerable success in creating DNN architectures that are highly efficient in terms of computational resource usage [18, 35, 83]. However, the irregular topologies it generates can be challenging to efficiently schedule on heterogeneous systems. In fact, in its most simple form, with no resource usage constraints or batching, the problem of mapping and scheduling a set of tasks with dependence is a classical NP-Hard problem [69]. Finding scalable and efficient methods for mapping such complex DNN computational graphs on heterogeneous systems is becoming more and more important to meet latency and throughput requirements imposed by modern DNNs and hardware platforms

during inference.

Even though this scheduling problem has been previously explored in the context of traditional computing [42, 61], few works investigate the challenges associated with neural network models. In this paper, we investigate the scheduling of irregular DNN topologies onto heterogeneous hardware platforms with different latency and throughput requirements, under different batching conditions, and leveraging the *module-based* nature of DNNs to significantly improve the speed and quality of our automatic scheduler. Many have used randomly-wired neural networks (RWNNs) [4] to represent NAS-designed DNNs in the context of scheduling [6], and we follow suit. Our scheduler operates on a coarse-grained computational graph of DNNs that is available through domain-specific frameworks such as PyTorch [67] or TVM [25]. Our goal is to create a fast heterogeneous scheduling plugin that can be easily integrated into these DNN frameworks to leverage heterogeneous computing platforms.

To achieve this goal, we curate a set of DNNs from the vision domain, both manually-designed ones such as ResNet [45], and NAS-found DNNs represented by an assortment of RWNNs. We investigate the scheduling of these DNNs on a sample heterogeneous computing platform with two GPUs and a CPU, and we demonstrate a considerable improvement compared to many past heuristic baselines. Our key algorithmic contribution is a fast DNN splitting heuristic, MILP-SPLIT, that detects and schedules each DNN module separately then combines the schedules in either an optimal or quasi-optimal fashion depending on the nature of the connection between modules. MILP-SPLIT also comes with a theoretical lower bound for the optimal solution, which facilitates the evaluation of the scheduling quality. Our contributions are enumerated below:

1. We formalize the problem of partitioning and scheduling a DNN onto interconnected hardware devices in a heterogeneous computing system. We leverage both model and data parallelism to handle two core optimization objectives; latency and throughput.

2. We propose a novel linear mathematical programming model which is the first, up to our knowledge, scheduling problem formulation capable of handling both model and data parallelism for batched DNN execution.

3. We introduce MILP-SPLIT: A splitting heuristic to schedule complex modular DNNs. Alongside, we perform a rigorous theoretical analysis on the implications of modularity and inter-module communication channels, on the performance of our heuristic, via the proposal of a lower bound formula.

4. We evaluate our algorithms on computer-vision DNN benchmarks, on both mainstream DNNs and randomly wired neural networks. Compared to a single device, we achieve more than 3× lower latency and 2.9× higher throughput. Compared to heursitics from prior work, we achieve 30–60% better solution quality, and up to 395× speedup compared to an exact solution.

CHAPTER 2

BACKGROUND

2.1 State-of-the-art

The scheduling of tasks and resources is a critical aspect of various computing domains, including general software systems orchestration and distributed computing. In this state-of-the-art section, we present an overview of the latest advancements in scheduling algorithms, focusing on two key areas: partitioning and scheduling of general software and DNN scheduling. In the realm of general software, scheduling algorithms aim to efficiently allocate computing resources and manage task execution to optimize performance, throughput, and resource utilization. We explore recent techniques that address challenges such as load balancing, task prioritization, and resource allocation in heterogeneous computing environments. Additionally, with the rapid growth of DNNs and their demanding computational requirements, specialized scheduling algorithms have emerged to maximize the utilization of hardware accelerators and enhance the training or inference efficiency. We delve into cutting-edge research that addresses the unique characteristics of DNNs, such as model parallelism, data locality, and communication overhead, to devise efficient scheduling strategies. By examining the state of the art in these areas, we aim to provide a comprehensive understanding of the recent advancements and give an extensive context to our approach for scheduling DNN inference on heterogeneous platforms.

2.1.1 Scheduling algorithms

Appearing in Karp's seminal compilation of 21 NP-complete problems [51], scheduling problems hold a fundamental position within the realm of combinatorial optimization. This class of problems has garnered significant attention across diverse domains, including parallel and distributed computing [14, 46], transportation and logistics [88, 91], and healthcare services [1, 7]. The computational complexity of scheduling problems varies considerably based on several factors, encompassing the constraints imposed by problem specifications, such as the characteristics of the machine environment (homogeneous, related [1], unrelated or heterogeneous), the existence of precedence relations (directed acyclic graphs, intrees, outrees ...), and the communication costs or optimization objectives (makespan, throughput, tardiness ...).

With no precedence constraints, ie. scheduling independent tasks on a platform, finding any optimal schedule is a strongly NP-Hard [39] problem (it remains NP-hard even if all numerical parameters of the problem are polynomially bound) but can be tackled efficiently in practice either through approximation schemes such as a $2-$approximation for unrelated machines [58] or an Epsilon Polynomial Time Approximation Scheme (EPTAS) in the form of a $(1+\epsilon)$-approximation algorithm in time $2^{O(1/\epsilon^2)log^3(1/\epsilon)} + O(nlogn)$ for every $\epsilon > 0$ for the homogeneous scheduling problem, or via the fixed-parameter paradigm such as the work of Mnich et al. [65] which presents an exact deterministic algorithm, for homogeneous platforms, running in time $f(p_{\max})n^{O(1)}$ with $p_{\max}$ the maximum task runtime and for some function f adding a double-exponential dependency in the parameter. In the same vein, an exact algorithm for heterogeneous machines was proposed in [54] which

[1] each machine j is assigned a speed s_j and each task i a cost c_i, and the processing time of i on j is $p_{i,j} = c_i/s_j$

runs in $\alpha^{O(\alpha^2)}n^{O(1)}$ for $\alpha = p_{\max}^K$ time, where K is the number of different kinds of machines. With the addition of precedency constraints, several instances of the problem on homogeneous platforms have been solved exactly and polynomially [29, 47, 55] on restricted graph structures, whereas a recent breakthrough by Davies et al. [31] offers a polynomial-time $O(log(c)log(m))$-approximation algorithm for this problem, where m is the number of machines and c is a bound on communication delays, based on a solution of the linear program (LP) relaxation and a randomized clustering routine.

On the heterogeneous side of the precedency constrained problem, and which is where our work is situated, little to no runtime results exist on exact solving or approximability guarantees. For related parallel machines heterogeneous systems, Chudak et al. [27] give a $O(log(m))$-approximation algorithm, whereas [11] links a super constant inapproximability result for the problem of scheduling precedence-constrained jobs with an open question on the structural hardness of k-partite graphs. Hence, the principal analysis of this problem has been tackled through heuristics and the empirical scope. We give now a quick overview of the recurring heuristics techniques that have been applied to the scheduling optimization problem. A very common problem specific method is critical path type scheduling approaches. Those are based on ordering tasks based on their critical path length and are often coupled with more general search heuristics [43, 57, 80]. Among the general-purpose search heuristics we can cite genetic algorithm-based scheduling utilizing evolutionary algorithms to optimize task allocation and scheduling and simulated annealing-based scheduling which has been adopted as a choice approach for various scheduling applications [23, 37, 44, 52, 66] in practice. We develop on these further in our evaluation section. Finally, Heterogeneous Earliest Finish Time (HEFT) [8, 34] assigns tasks to machines based on their expected execution

time and communication costs. It has gained much popularity due to its low cost and high-performance trade-off. The work conducted by Canon et al. [19] involved a comparison of 20 scheduling heuristics, and their findings indicated that, on average, the HEFT heuristic outperformed others in terms of both robustness and schedule length for random graphs.

2.1.2 General software partitioning

In the domain of general software partitioning, previous research has been conducted on the topic of heterogeneous compilation and scheduling. One notable work in this area is the study by Luk et al. [61]. The authors propose Harmonic, an approach that leverages tabu search [38, 41], a relaxed variant of local search, as the guiding heuristic for mapping and scheduling tasks on a heterogeneous processor system and which is illustrated in Figure [61]. Their work focuses on optimizing the performance of task execution by efficiently distributing and allocating computational resources.

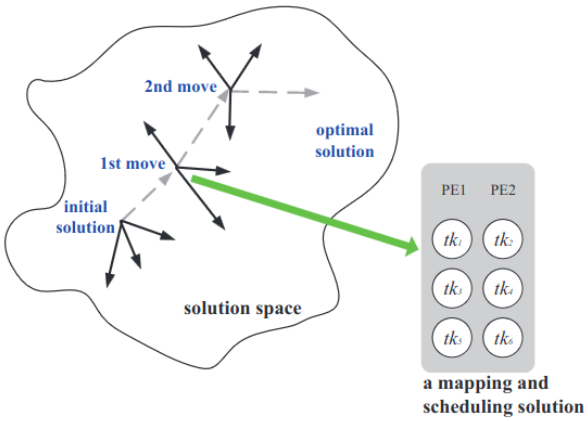


Figure 2.1: Harmonic : Tabu search process. The solid arrows designate considered movements by the neighbourhood functions. The dotted arrows denote the best move in each iteration of the search. PE: processing element, tk: task. Figure from [61].

Another relevant contribution to the field is the Polly-Acc compiler framework. Specifically designed for CPU-GPU systems, Polly-Acc offers an automatic heterogeneous compute compiler [42]. At the compiler intermediate representation (IR) level, the framework detects and extracts computationally intensive kernels. These kernels are then modeled and their execution strategy is described using a schedule tree over Presburger sets and relations, a powerful mathematical tool for polyhedral compilation [85]. Figure 2.2 shows an example of this transformation. By automatically identifying and optimizing compute-intensive portions of the code, Polly-Acc aims to enhance the performance and efficiency of CPU-GPU systems.

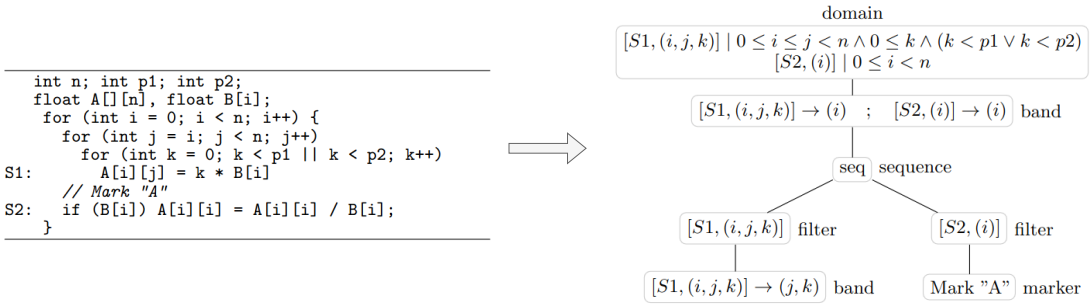


Figure 2.2: Polly-Acc : Schedule tree that models that models a simple program. Figure from [42].

AMAP (Adaptive Mapping Algorithm) is an online adaptive decision algorithm that aims to assess the benefits of executing a specific function in hardware compared to the overhead associated with transferring the corresponding parameters [79]. This algorithm provides a mechanism for dynamically determining whether offloading a computation to hardware accelerators is advantageous in terms of overall performance and efficiency. By considering the potential gains in computation speed and the associated costs of data transfer, AMAP enables intelligent decision-making regarding the optimal execution platform for each function. In

contrast, another approach proposed in [56] focuses on dynamic program scheduling based on the energy-delay product observed during tentative runs. The authors recognize that the energy and time required for executing a program on different platforms can vary significantly. Therefore, they introduce a scheduling strategy that leverages sampled energy-delay product measurements to guide the selection of the most appropriate execution platform. By dynamically considering the trade-off between energy consumption and execution time, this approach aims to achieve efficient program scheduling in heterogeneous computing environments.

Finally, HeteroGen [90] is proposed as a tool that automatically generates a High-Level Synthesis (HLS) version of C/C++ code with preserved behavior and improved performance. HeteroGen utilizes search-based program repair techniques (evolutionary program repair), overcoming challenges such as longer compilation and simulation times, through pattern-oriented program edits which identifies common sources of HLS compatibility issues (dynamic data structures, loop parallelization ...) and addresses them through fix patterns drawn from real-world HLS compatibility fixes.

Complementary to these endeavors, our approach, in contrast, is performed statically during compilation, is specifically tailored for deep learning architectures, and leverages coarse graph-level descriptions of DNNs.

2.1.3 DNN scheduling

Under the scope of DNN based partitioning, many existing research endeavors focus solely on training [59, 92]. Liu et al. [59] propose a profiling framework and runtime system to dynamically map and schedule neural network operations on

heterogeneous processing-in-memory (PIM) systems. For a static approach, Alpa [92] automates the search for pipeline-parallel schedules for DNN training on homogeneous multi-node GPU clusters. Additionally, ParDNN introduces a graph slicing heuristic which forms primary clusters, the first iterative critical paths of the graph, and secondary clusters, the single nodes or remaining paths, and optimizes for load balancing during training [72]. Alternatively, Chen at al. [26] propose heuristic methods to optimize latency based on HEFT and critical path consideration for mapping and scheduling DNNs on accelerators consisting of function units such as matrix multiplication or lookup tables. Unlike these approaches that were specific to DNN training, our scheduling algorithm is geared towards low-latency and high-throughput inference. On the inference side, Liu et al. [60] restrict their scope to the DenseNet architecture and give an exact and efficient algorithm for its scheduling on a heterogeneous system when the number of heterogeneous processors is constrained, by defining a notion of redundant edges. Intuitively, an edge from task u to task v is defined as redundant in their model, if its removal will not impact the starting processing time for v. Figure 2.3 gives a minimal example.

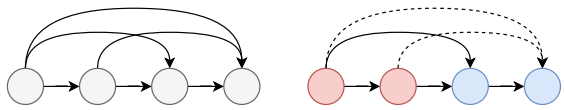


Figure 2.3: A DenseNet of 4 nodes and an example of redundant edges : blue and red designate two different processors. [60]

However, this approach is tailored for the particular topology of the DenseNet graph and is consequently difficult to generalize to broader model architectures. We propose a more general cut-based heuristic, which also takes advantage of the dynamic programming paradigm and can significantly speed up the mixed integer linear programming (MILP) solving. Compared to [60], Duan et al. [33] investigate

Table 2.1: DUET : Computation cost and device placement decisions (time in ms). Excerpt from results table from [89]

	Subgraph	**Comp. Time**		**Placement**	
		CPU	GPU	CPU	GPU
Wide-and- Deep	Wide	0.03	0.05		✓
	FF	0.07	0.07		✓
	RNN	2.4	6.4	✓	
	CNN	14.9	0.9		✓
	merge	0.03	0.06	✓	

a wider set of networks for inference jobs on mobile cloud computing, where the compute is executed on a single device and should be communicated afterward to a server. They propose an optimal (under reasonable assumptions) binary-search based approach for line structure DNNs such as VGG, which they extend to more complex networks with model parallelism as a heuristic, but acknowledge that it is not sufficient to leverage potential collaboration opportunity between two different paths in the DNN graph, by overlapping sub-optimal schedules of parallel paths and which may lead to the global optima. DUET [89] is a scheduler for heterogeneous DNN inference for CPU-GPU platforms which optimizes for latency. It partitions the graphs into very coarse-grained modules which will be processed on a single device and relies on a critical-path heuristic driven approach where it places critical path on fastest device, greedily place remaining operations and a post processing correction to mitigate communication cost. Once again, their approach does not explicitly exploit model parallelism but only heterogeneity of compute patterns (see Table 2.1), hence showing advantage against running the model either on CPU or GPU entirely, but do not compare their approach to other heterogeneous scheduling heuristics. Lalarand [50] follows a similar approach where subgraphs (DNN layers) are placed on either device, but does it in a real-time fashion, answering the allocation imbalance dynamically. Alternatively, several research endeavors [21, 36, 40, 86] have proposed Reinforcement Learning [10] (RL) as a scheduling

heuristic for DNN inference following the success of this approach on infamous hard problems [77, 78]. However, most of these works [21, 36, 86] propose RL for scheduling multiple inference requests, abstracting the entire DNN as a single query object. The work of Geng et al. [40] propose DRM-DQL, a layer-aware DNN inference task scheduler based a deep Q-learning (DQL) algorithm. In particular, DRM-DQL solves for a latency objective on homogeneous platforms (GPU clusters). Still, their approach may suffer from a lack generalization ability, for irregular models or different platform with complex communication topology, due to the RL component implemented which would need to be re-trained. This can be seen in the limited evaluation which focuses on classic and very regular networks such as VGG16 for which we show that classic search heuristic perform very well - even empirically reaching the optima - and which shows advantage only against an inferior DQL algorithm and a naive greedy algorithm. Additionally, Mirhosein et al. [64] propose a LSTM-based reinforcement learning approach to DNN mapping on CPU-GPU systems for both training and inference latency optimization. This again suffers from the same lack of generalization as other works relying on RL with a need to set manually and individually parameters depending on the input graph and train a model for each graph, with training time ranging between 12 to 27 hours. Hence, the evaluation is done on a small benchmark set (3 DNN models), and compares only to graph partitioning-based approaches (Scotch [68] and naive clustering). In comparison to our approach, for inference and on similar graphs, we achieve optimality for latency in significantly less time by leveraging modularity.

Finally, SERENITY achieves memory-aware scheduling of irregularly wired neural networks on a single device by resorting to graph rewriting and divide-and-conquer approaches [5] where the objective is one of optimizing the peak memory

footprint, putting a focus on platforms under strict memory capacity constraints, such as edge devices. They manage to achieve an optimal schedule by efficiently reducing and pruning the search space through divide-and-conquer paradigm and an adaptive budgeting technique which eliminates suboptimal solutions during the early stage of scheduling without affecting the optimality of the original algorithm.

A recent trend in DNN mapping and scheduling is the one addressing the deployment of Large Language Models (LLMs) on distributed hardware systems. In this case, the main difficulty reside in the consequent memory constraints where such models can no longer reside on a single processor or even a single compute node [17].

Deepspeed's ZeRO-inference [9] relies on hand-crafted solutions for the deployment of transformer model inference on GPU platforms using tensor and pipeline parallelism and the ability to rely on CPU or NVMe memory fetching. Additionally, ZeRO-Inference makes several case specific optimizations on transformer kernels steering away from a general-purpose scheduler and ignores opportunities for model or data parallelism. The work presented in the paper by Pope et al. [70] focuses on the PaLM Inference framework, which aims to map Large-scale Linear Models (LLMs) onto TPUv4 clusters. The authors propose a set of "best practices" for manual partitioning of LLMs, enabling efficient utilization of resources. Moreover, to address more challenging cases involving tensor-level partitioning, the framework incorporates MPI (Message Passing Interface) primitives, such as all-gather operations. The Orca framework, introduced by Ui et al. [87], focuses on enhancing the performance of autoregressive decoder models through the utilization of iteration-level scheduling and selective batching techniques. Notably, Orca incorporates an online serving scheduler that dynamically assigns and batches

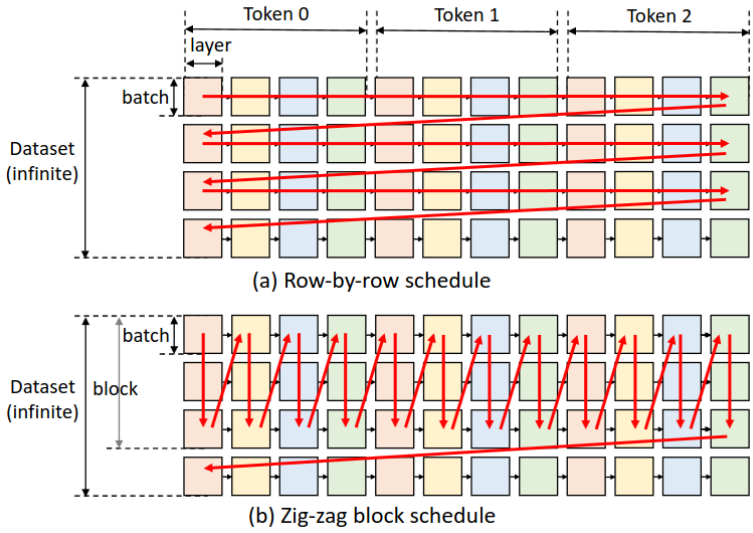


Figure 2.4: Flexgen : Row schedule and zig-zag schedule (implemented). Figure from [76]

requests in a distributed environment. However, it is important to note that the scheduler employed in Orca does not consider factors such as memory hierarchy, CPU characteristics, or the diversity of hardware configurations present in the system. Flexgen [76] takes into account the memory hierarchy between disk, CPU, and GPU and has a singular focus on fast execution on the GPU. It employs a linear programming formulation to decide on the ideal weight and activation placement for fixed block and batch sizes, and a fixed zig-zag scheduling strategy. However, Flexgen does not explore scheduling possibilities within each layer, putting focus on memory management instead of optimizing compute.

All of these proposals for large model mapping and scheduling rely mainly on hand crafted solutions and case-specific optimizations, and lack leverage of intra-layer parallelism or awareness of hardware platforms. We try to take a first step in that direction through a case study of GPT-3 on heterogeneous distributed nodes.

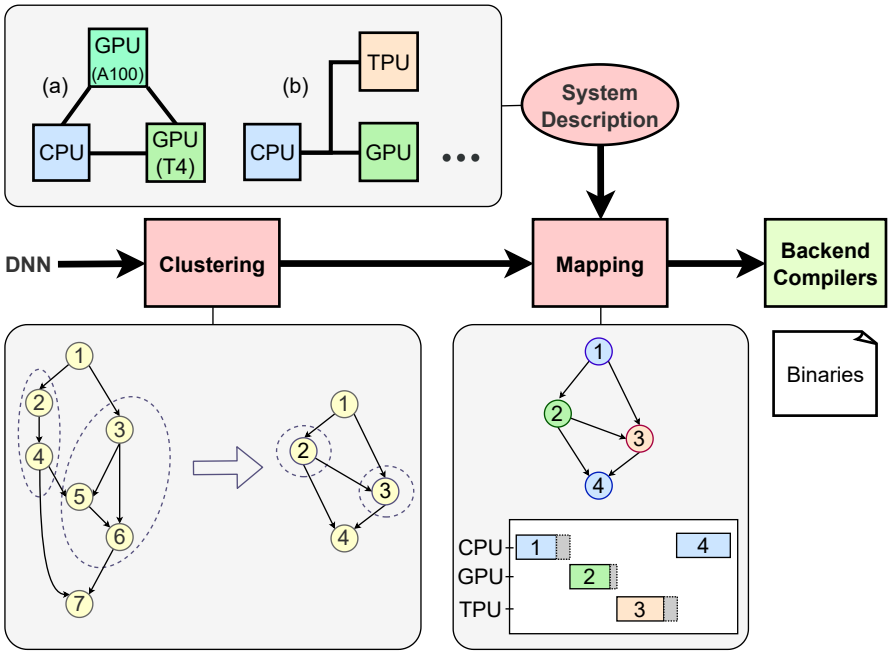


Figure 2.5: Our heterogeneous scheduling framework.

2.2 Problem statement and system description

We focus on latency and throughput optimization on multiple heterogeneous devices, taking into account each device's memory constraints and ability to perform batch processing. Our approach is based on a coarse-grained representation of computational graphs that is commonly used in deep learning compilers. We present a compile-time mapping and scheduling framework for DNNs on heterogeneous hardware systems. The scheduler's modeling is general and agnostic to back-ends, its only limitation being what is supported by different compilers' back-ends. Figure 2.5 illustrates how the partitioner is integrated in a DNN compilation pipeline. It is capable of reading an input consisting of a hardware system configuration and any intermediate representation (IR) of a DNN, and outputs the appropriate mapping on the system via existing compilation backends, and its corresponding schedule. An optional clustering step prepares the DNN graph for mapping by re-

ducing the number of task inputs to the mapping algorithms. A prime example is the fusion of convolution, batch normalization, and the ReLU activation function.

2.3 Problem Formulation

We represent DNNs as a weighted directed acyclic graph (DAG), with the edges denoting data dependencies and nodes representing a DNN task (e.g. a convolutional or linear operation). If two tasks with data dependencies are mapped onto the same processor, the communication between them is implemented through data sharing in device memory and no communication cost is incurred. Each processor may execute several tasks, but each task has to be assigned to exactly one processor, in which it is entirely executed without preemption. Formally, let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the DAG where $\mathcal{V}$ denotes the set of tasks and $\mathcal{E}$ represents the set of edges. Each edge $(i, j) \in \mathcal{E}$ defines a precedence relation between the tasks $i, j \in \mathcal{V}$, and is weighted by the size of the source task's output. A task cannot be executed unless all of its predecessors (parents) have been processed and all relevant data is available. Each task $i \in \mathcal{V}$ is assigned the following constants: (wm_i) the data size allocated for the DNN task weights, (im_i) the input tensor size and (om_i) the output tensor's size. As for our hardware system on which we map the DNN, we model it as a tuple of sets $\mathcal{H} = (\mathcal{K}, \mathcal{M}, \beta)$. $\mathcal{K}$ denotes the set of devices in our system. The two remaining sets are descriptors of the hardware system. $\mathcal{M} : \mathcal{K} \to \mathbb{R}^+$ is the memory capacity for each single processor and $\beta : \mathcal{K}^2 \to \mathbb{R}^+$ the communication bandwidth between linked chips—it is null if there is no link. If tasks i and j are executed on different compute nodes h, k ; $h \neq k$, and $(i, j) \in \mathcal{E}$, a communication time $om_i/\beta_{h,k}$ is incurred.

The objective of this task scheduling problem is to allocate and schedule the tasks onto the compute nodes such that the overall completion time (latency) is minimized. We link the dataflow graph and the hardware via a map $t : (\mathcal{V}, \mathcal{K}) \to \mathbb{R}^+$, which assigns to each task and device pair its corresponding latency. We finally add to our formulation the possibility of batching and throughput optimization. Hence we augment our problem description with a map $\mathcal{B} : \mathcal{K} \to 2^{\mathbb{N}}$ that assigns to each device the subset of batch sizes supported. t now describes the latency of each possible batch of similar tasks $i \in \mathcal{V}$ for each device and is redefined as $t : \mathcal{V} \times \mathcal{K} \times \mathcal{B}(\mathcal{K}) \to \mathbb{R}^+$. The objective is now to find for a set of $\mathcal{L}$ graph inputs the optimal mapping and scheduling of the tasks into different batches, while respecting the dependency within a single graph and the underlying resource constraints. Finally, we define the notion of a schedule. Let $\mathcal{S} : \mathcal{V} \times [1, \ldots, \mathcal{L}] \to \mathcal{K} \times \mathbb{R}^+$ be a map which assigns each task to a device and a starting time. $\mathcal{S}$ is a schedule if and only if $\mathcal{S}$ respects precedence and no overlap (no two distinct batches can overlap on the same device) criteria, i.e. for every $(i, j) \in \mathcal{E}$, $l \in [1, \ldots, \mathcal{L}]$:

$$\mathcal{S}(i, l)_2 + 1_{\mathcal{S}(i,l)_1 \neq \mathcal{S}(j,l)_1} \cdot m_i / \beta_{h,k} \leq \mathcal{S}(j, l)_2$$

The problem statement becomes:

Mapping and Scheduling problem

Input Objective function (latency/throughput) f, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathcal{S} = (\mathcal{K}, \mathcal{M}, \beta)$, t, $\mathcal{B}$, $\mathcal{L}$.

Output A schedule $\mathcal{S} : \mathcal{V} \times [1, \ldots, \mathcal{L}] \to \mathcal{K} \times \mathbb{R}^+$ which optimizes f

CHAPTER 3

ALGORITHMIC APPROACHES

In this section, we present the different solution schemes and algorithmic ideas that were investigated and in particular demonstrate our exact scheduling approach based on solving an MILP problem. Linear programming has been effective in solving communication constrained DAG scheduling problems for tractable instances [30]. Our contributions for the exact MILP formulation are twofold: First, we incorporate memory and batching constraints into our formulation, which are commonly encountered in deep learning workloads, and we integrate our scheduler into a graph partitioning routine that we rigorously analyze to ensure the quality of its results. However, the problem of scheduling DNNs is NP-Hard, making it intractable to find exact solutions for large graph sizes. Our second contribution addresses this issue. We take advantage of the inherent modularity in most DNNs to create fast solving schemes that are either optimal or provide strong approximation guarantees.

3.1 Branch-and-bound

A classical approach for combinatorial optimization is in the form of a branch-and-bound (B&B) paradigm. It is a fundamental and widely-used methodology for producing exact solutions to NP-hard optimization problems. It encapsulates a family of algorithms that all share a common core procedure; enumerating possible solutions to the problem under consideration, by storing partial solutions called subproblems in a tree structure. Unexplored nodes in the tree generate children by partitioning the solution space into smaller regions that can be solved recursively

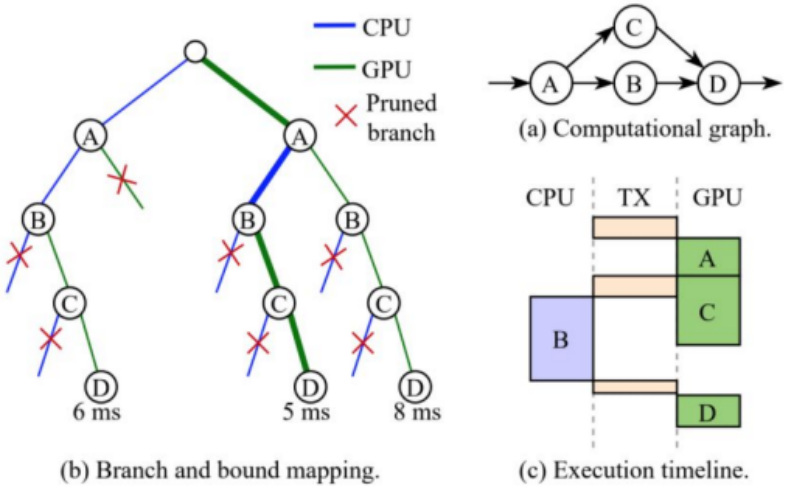


Figure 3.1: Example mapping of a computational graph onto a heterogeneous CPU-GPU system using branch and bound search.

(i.e., branching), and rules are used to prune off regions of the search space that are provably suboptimal (i.e., bounding). Once the entire tree has been explored, the best solution found in the search is returned. In the context of a scheduling problem, Figure 3.1 shows a simple computational graph, a B&B mapping tree, and an execution timeline for the best solution, highlighting both the data transfer overheads and the execution time of each operation. We traverse the computational graph in topological order to construct the B&B tree in which each branch indicates mapping a single operation on a specific device; in our example we have two devices, a CPU and a GPU, making our B&B tree binary. At each vertex in the B&B tree we maintain the total runtime so far and the location of all live tensors. The latter allows us to (a) estimate the data transfer overhead if a tensor needs to be moved from one device to the other, and (b) perform legality checks on whether the inputs to an operation fit in a device's memory. We will transfer constant values between devices whenever the communication link is available, allowing us to overlap computation with communication. For example DNN parameters during

inference are constant so we do not need to transfer them to a device just before they are used in an operation; rather, we can perform this transfer beforehand.

Limitations

Despite the theoretical optimality provided by the exact approach and the advantageous formulation flexibility it offers, we encountered limitations when applying branch and bound (BB) techniques to graphs of approximately 100 nodes. One plausible explanation is that the available branching strategies, which require a topological order of selection to adhere to data precedence rules, lack the necessary adaptability. This stands in contrast to recent significant advancements in BB techniques observed in applications such as the Traveling Salesman Problem (TSP) [74] and special ordered sets [12, 13].

Furthermore, the absence of substantial general symmetries within the solution space presents another challenge. The exploitation of symmetry, as demonstrated in addressing other NP-Hard problems (vertex coloring problems [62]), enables the establishment of equivalence classes across instances and enhances the effectiveness of the Branch and Bound scheme. Consequently, we need to employ a more robust approach that combines heuristic reasoning with an optimality guarantee. In what follows, we introduce our Mixed Integer Linear Programming (MILP) formulation for addressing the problem at hand. The appendix proposes a formal definition of MILP and a general overview the solving procedure, which implicitly relies on branch-and-bound.

3.2 MILP Problem Representation

We introduce a novel formulation of the problem as an MILP model that explicitly considers the option of batching, where a device can process multiple inputs simultaneously. By incorporating batching, our formulation is better suited to capture the characteristics of modern deep learning workloads, which often involve a large numbers of inputs that can be efficiently processed in batches. Our approach enables us to find optimal solutions that balance the trade-offs between computation and communication costs while respecting batching and memory constraints. We add to the notation introduced earlier the following binary decision variables: $x_{i,j,l}$ which encodes if the DNN task i corresponding to the l-th input is mapped to a device k. Meanwhile, $b_{i,j,l}$ describes if tasks of kind i running on j form a batch of size l, and $d_{i_1,i_2,l_1,l_2} = 1$ iff task i_1 from input l_1 is scheduled before i_2 from input l_2. We also consider the continuous variables: $s_{i,j}$ the starting time for processing the batch of i tasks on j, and C the total latency. The objective function f is equal to C in the latency optimization scenario or $\mathcal{L}/C$ when optimizing for throughput. Now, we can write the mixed integer linear program, with objective `minimize C`, and whose constraints are as follows:

Condition 3.1 asserts that each task is assigned to a single machine:

$$\sum_{u \in \mathcal{K}} x_{i,u,l} = 1; \;\; i \in \mathcal{V}, \;\; l = 1, \ldots, \mathcal{L} \tag{3.1}$$

Condition 3.2 ensures that each task finishes within the reported latency :

$$s_{i,u} + \sum_{l \in \mathcal{B}_u} b_{i,u,l} \cdot t_{i,u,l} \leq C; \;\; i \in \mathcal{V}, \;\; u \in \mathcal{K} \tag{3.2}$$

Condition 3.3 is the condition expressing the dependency and communication con-

straint:

$$\begin{aligned} s_{i,u} + \sum_{p \in \mathcal{B}_u} b_{i,u,p} \cdot t_{i,u,p} + (om_i/\beta_{u,v}) \cdot (x_{j,v,l} + x_{i,u,l} - 1) \\ \leq s_{j,v}; \;\; j \in \mathcal{V}, \;\; i \in par(i), \;\; u, v \in \mathcal{K}, \;\; l = 1, \ldots, \mathcal{L} \end{aligned} \tag{3.3}$$

Condition 3.4 ensures that the batch decomposition adds up correctly to the total number of items in the batch:

$$\sum_{u \in \mathcal{K}} \sum_{l \in \mathcal{B}_u} l \cdot b_{i,u,l} = \mathcal{L}; \;\; i \in \mathcal{V} \tag{3.4}$$

The following condition 3.5 ensures that only supported batch sizes are chosen:

$$\begin{aligned} b_{i,u,l} = 1 \quad \text{iff} \sum_{l' \in [1 \ldots \mathcal{L}]} x_{i,u,l'} = l; \\ i \in \mathcal{V}, \;\; u \in \mathcal{K}, \;\; l = 1, \ldots, \mathcal{L} \end{aligned} \tag{3.5}$$

In its form above, it is not a linear equation but we can linearize it via the BIG M METHOD [28] into[1]:

$$\begin{cases} 2(\sum\limits_{l' \in [1 \ldots \mathcal{L}]} x_{i,u,l'} - l) - 1 \leq M(1 - b_{i,u,l}) \\ -2(\sum\limits_{l' \in [1 \ldots \mathcal{L}]} x_{i,u,l'} - l) - 1 \leq M(1 - b_{i,u,l}) \\ 2(\sum\limits_{l' \in [1 \ldots \mathcal{L}]} x_{i,u,l'} - l) - 1 \geq -Mb_{i,u,l} - M'\delta_{i,u,l} \\ 2(\sum\limits_{l' \in [1 \ldots \mathcal{L}]} x_{i,u,l'} - l) + 1 \leq Mb_{i,u,l} + M'(1 - \delta_{i,u,l})) \end{cases} \tag{3.6}$$

where M and M' are two sufficiently large constants with $M << M'$ and $\delta_{i,j,l}$ are binary variables.

Condition 3.7 holds the memory constraint under the supposition that all data should be preemptively moved:

$$\begin{aligned} \sum_{i \in \mathcal{V}} ((im_i + om_i) \sum_{l \in [1 \ldots \mathcal{L}]} x_{i,u,l} + wm_i \sum_{l \in \mathcal{B}_u} b_{i,u,l}) \\ \leq \mathcal{M}_u; u \in \mathcal{K} \end{aligned} \tag{3.7}$$

[1]See appendix for more details

Conditions 3.8 ensures no overlap of device usage between different batches. We linearize it similarly to condition 3.5:

$$\begin{cases} s_{i,u} + \sum\limits_{p \in \mathcal{B}_u} b_{i,u,p} \cdot t_{i,u,p} - s_{j,u} \leq 0 \\ \text{or} \\ s_{j,u} + \sum\limits_{p \in \mathcal{B}_u} b_{i,u,p} \cdot t_{i,u,p} - s_{i,u} \leq 0 \end{cases} \tag{3.8}$$

$$\text{if } \; x_{i,u,l_1} = x_{j,v,l_2} = 1;$$

$$i,\ j \in \mathcal{V},\ \ u \in \mathcal{K},\ \ i \neq j, l_1,\ l_2 = 1, \ldots, \mathcal{L}$$

An optimization of the formulation of the MILP is to restrict constraint 3.8 to pairs of tasks (i, l_1) and (j, l_2) which do not belong to the same batch graph or are not part of a path in the DAG. The system remains equivalent to the original as the other constraints from 3.8 are enforced by the dependency constraint 3.3. Eliminating these redundant constraints is done by computing the transitive closure of the graph and which can be obtained efficiently with Purdom's algorithm [71].

3.3 MILP-SPLIT: Leveraging Graph Modularity

3.3.1 Single-channel modularity

The presence of highly connected clusters is a prevalent feature in many DNN graph structures. An example is shown in Figure 3.2a This characteristic can be leveraged by the scheduler to partition the global problem into independent sub-problems

consisting of weakly communicating modules. This approach is particularly useful when dealing with graphs that consist of modules linked to one another, such as ResNets [45], Inception [81], or especially RWNNs [4] that are composed of several instances of sequentially linked random graph modules.

A straightforward method to identify these modules involves detecting articulation points or bridges in the graph, which correspond to vertices or edges whose removal disconnects the undirected graph, grouping tasks between them into the same module, and solving each subproblem independently. However, this approach can lead to suboptimal solutions as it does not account for communication costs through bridges and may result in inconsistent assignments of articulation points across modules. Fortunately, a dynamic programming solution exists to address these issues.

To obtain an optimal global solution for the whole graph, we compute the optimal schedule for each module for every possible input-device and output-device pairings, and we combine the resulting building blocks into the best configuration. As a preprocessing step, we transform articulation points that are not endpoints of bridges into bridge edges by introducing a dummy node and a zero-cost edge between them. We also add an additional constraint that mandates the mapping of these two vertices to the same device in the global solution as is illustrated in Figure 3.2b. From now on, we refer to bridges as "communication channels".

Formally, Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a DAG with single input and output. We denote by $\mathcal{I}(\mathcal{Q}, \mathcal{F})$ the graph obtained by reducing every module into a single vertex, where $\mathcal{Q}$ is a partition of $\mathcal{V}$ into a set of disjoint modules and $\mathcal{F} := \{(u, v) \in \mathcal{Q}^2 |\ \exists x \in u\ \exists y \in v\ \ (x, y) \in \mathcal{E}\}$. In particular, if $\mathcal{Q}$ is defined as the set of vertex modules, then $\mathcal{I}$ is a path, and we can enumerate $\mathcal{Q}$ as the set $[1, \ldots, |\mathcal{Q}|]$, and through this

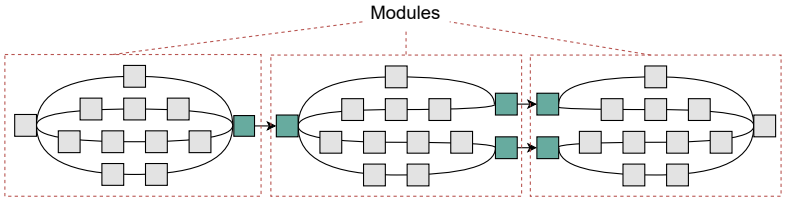
ordering we can obtain a dynamic programming problem formulation. For a given module $M_i \in \mathcal{Q}$ and a pair of devices $u, v \in \mathcal{K}$ onto which the input and output of M_i are mapped, and if we denote by *opt* the solution of a module subproblem, the recursion can be written as:

$$dp(M_i, u, v) = min_{u',v' \in \mathcal{K}} \Big(dp(M_{i-1}, u', v') + com(i, v', u) \Big) + OPT(M_i, u, v)$$

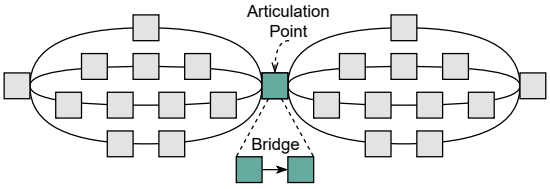
The effectiveness of the proposed splitting method is influenced by the number and size balance of the extracted modules. The complexity of the approach can be expressed as $O(|\mathcal{K}|^2|\mathcal{Q}|\mathbb{T})$, where $\mathbb{T}$ represents a runtime bound for each module. This complexity analysis assumes a specific cutting strategy, but can be generalized to arbitrary cuts, where $\mathcal{I}$ becomes a multigraph.

3.3.2 Multi-channel modularity

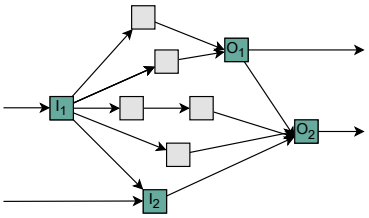
Modularity is an important property of graphs that enables exact solving for the scheduling problem on large graphs using a divide-and-conquer approach. However, many graphs can not be split into distinct modules of comparable size that communicate through a *single* input-output channel. In such cases, it may still be possible to decompose the graph into balanced modules that communicate through *multiple* edges, and solve for each subgraph independently. Figure 3.2a shows an example with 1 and 2 channels. Identifying the modules boils down to computing the $k-$edge connected components [24] where $k-1$ is the number of channels. Although this approach may result in a loss of optimality, it can significantly improve runtime without a significant reduction in quality. In the case of partitioning a large graph into multichannel communicating modules, it is desirable to compute a lower bound on the optimal solution to evaluate the quality of the MILP-SPLIT



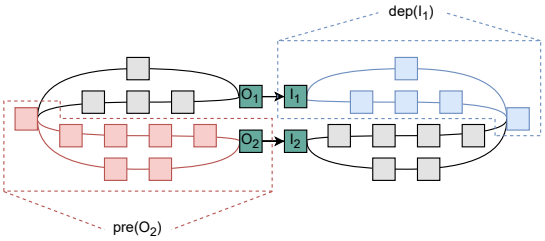
(a) Partitioning scheme on modular graph



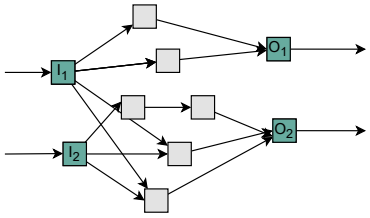
(b) Articulation point and bridge between Inception modules



(c) Example of 2-channel Erdos-Renyi sd module



(d) Dependency and predecessor subgraphs on channel endpoints



(e) Example of 2-channel Erdos-Renyi wd module

Figure 3.2: Modularity in DNN graphs. **sd**: all paths within a module stem from (converge toward) at least one input (output). **wd**: module inputs and outputs are randomly sampled for their dependencies.

(or other) heuristic, especially when solving for the entire graph is not tractable.

In order to express the lower bound for a DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$ that can be split into multichannel communicating modules, we first define for a fixed $T \subseteq \mathcal{V}$ and for every node u in $\mathcal{G}$ the set of nodes $dep(u)_T = \{v \in T \mid \text{there is a path from } u \text{ to } v\}$, which we will refer to as the dependency set of u, and the set of nodes $pre(u)_T = \{v \in T \mid \text{there is a path from } v \text{ to } u\}$, and which we will refer to as the predecessor set of u (as shown in Figure 3.2d). Let $M_1, \ldots, M_{|\mathcal{Q}|}$ be a decomposition of $\mathcal{G}$ into

such modules, where $\bigcup_{1\leq i\leq|\mathcal{Q}|} M_i = \mathcal{V}$. We denote by $\mathcal{G}_s = \bigcup_{s\leq i\leq|\mathcal{Q}|} M_i$. Our approach is to proceed inductively by computing the lower bound in a recursive manner, and using the following remark:

Remark. *Let c denote the number of channels, and $(I_t)_{1\leq t\leq c}$ and $(O_t)_{1\leq t\leq c}$ denote respectively the set of vertices in the communication channels between M_1 and $\mathcal{G}_2$ for which the edges are in-going and out-going, i.e., the inputs of $\mathcal{G}_2$ and the outputs of M_1. For any valid scheduling of the whole graph, there exists a t' such that the subgraph induced on $dep(I_{t'})_{\mathcal{G}_2}$ is completely scheduled after M_1, and there exists a t" such that $pre(O_{t"})_{M_1}$ is completely scheduled before $\mathcal{G}_2$.*

Hence, if we denote by OPT the function mapping subgraphs of $\mathcal{G}$ onto their optimal schedule, then we obtain the pair of inequalities:

$$OPT(\mathcal{V}) \geq OPT(M_1) + min_{u\in\text{inputs}}(OPT(dep(I_u)_{\mathcal{G}_2}))$$

and

$$OPT(\mathcal{V}) \geq OPT(\mathcal{G}_2) + min_{v\in\text{outputs}}(OPT(pre(O_v)_{M_1}))$$

The lower bound of the problem is obtained as the maximum value among the right-hand sides of the inequalities. This lower bound can be immediately extended to the batched throughput scenario by observing that the partial ordering defined earlier for dependency, predecessor, and module subgraphs applies to scheduling the minimal batch size that can be executed on each device. Specifically, it is necessary to schedule a minimum portion of the input to maintain the specified constraints via the communication channels outlined in the remark. However, we can do better; let M_1 and $dep(I_{t'})_{\mathcal{G}_2}$ be defined as in the remark; then if $\mathcal{L}$ is the total input batch to be processed and b any batch size supported on every device, then there is at least a batch of $\mathcal{L} - b + 1$ that needs to be processed

through $dep(I_{t'})_{\mathcal{G}_2}$ after scheduling a load b of M_1. The same reasoning holds between $OPT(pre(O_v)_{M_1}$ and $\mathcal{G}_2$, and recursively throughout the graph. These bound computations can be accomplished efficiently using the presented recursive formula, which lends itself well to parallelization due to the independent nature of the subproblems considered.

3.3.3 Complete analysis of the 2-channel case

As discussed in Section 3.3.2, the concept of modularity in a graph enables finding the exact solution of scheduling problems on large graphs using a divide and conquer approach. However, it is often challenging, and sometimes impossible, to partition graphs into separate modules of comparable sizes that communicate solely through a single input-output channel. In what follows, we provide a detailed and exhaustive analysis of the effectiveness of our lowerbound for approximate graph scheduling by dividing the graph into modules that communicate through two input and output channels.

We recall some of the previously introduced notations :
$opt(A)$: optimal scheduling time of A.
$dep(u)_T = \{v \in T \mid \text{there is path from u to v}\}$
$pre(u)_T = \{v \in T \mid \text{there is path from v to u}\}$

Let $\mathcal{G}$ be a DAG composed of 2 modules A and B connected through 2 channels and we denote $G = A + B$. In order to comprehensively analyze the scheduling requirements between components A and B, it is essential to consider not only the tasks that need to be processed in A before scheduling B but also the tasks in B that are dependent on the completion of scheduling A. By examining all possi-

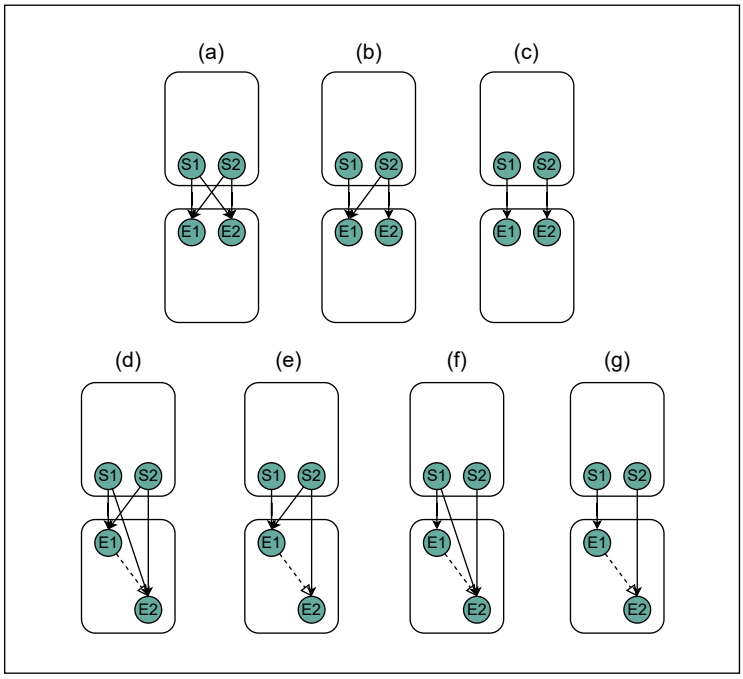


Figure 3.3: All possible communication configurations between two blocks (modulo symmetry between upper and bottom block and symmetry between channels). Solid arrows mean an edge, dotted arrows mean a path.

ble scenarios, we can derive a complete understanding of the various sequencing possibilities and their implications.

This complete derivation involves systematically exploring and evaluating different combinations and orders of tasks within A and B. It allows us to capture the entire range of potential scheduling scenarios and provides valuable insights into the dependencies and constraints that must be considered. We provide below the bounds with their corresponding topology in Figure 3.3 :

- (a) : Both input channels of B are depending on the outputs of A, hence B can only be scheduled after A has been completely processed :

$$opt(A + B) = opt(A) + opt(B)$$

- (b) : Since $E1$ depends on all outputs of A, then $opt(A + B) \geq opt(A) +$

Present dependencies	guarantee factor
2-to-2 : (S1,E1), (S1,E2),(S2,E1), (S2,E2)	1
1-to-2 : (Si,E1), (Si,E2)	$2 - \beta_i$
2-to-1 : (S1,Ei), (S2,Ei)	$2 - \alpha_i$
1-to-1 (S1,E1), (S2,E2)	$2 - max(min(\alpha_1, \alpha_2), min(\beta_1, \beta_2))$

Table 3.1: Optimality guarantees for the 2-channel case

$opt(dep(E1)_B) = opt(A) + \alpha_1 opt(B)$ where $\boldsymbol{\alpha_1 = opt(dep(E1)_B)/opt(B)}$. Since $dep(E1)_B \subseteq B$ then $0 \leq \alpha_1 \leq 1$ and thus $(1 - \alpha_1)opt(B) \leq (1 - \alpha_1)opt(A + B)$. Hence $opt(A) + opt(B) \leq (2 - \alpha_1)opt(A + B)$.

Similarly, since both inputs of B depends on $S2$, $opt(A + B) \geq opt(B) + opt(pre(S2)_A) = opt(B) + \beta_2 opt(A)$ where $\boldsymbol{\beta_2 = opt(pre(S2)_A)/opt(A)}$. Consequently, $opt(A) + opt(B) \leq (2 - \beta_2)opt(A + B)$. Finally,

$$opt(A) + opt(B) \leq (2 - \max(\beta_2, \alpha_1))opt(A + B)$$

- (c) In every schedule of G, either $dep(E1)_B$ or $dep(E2)_B$ is scheduled completely after A. This means $opt(A) + opt(B) \leq (2 - min(\alpha_1, \alpha_2))opt(A + B)$. In the same fashion, either $pre(S1)_A$ or $pre(S2)_A$ is scheduled completely before B. We can then write

$$opt(A) + opt(B) \leq (2 - max(min(\alpha_1, \alpha_2), min(\beta_1, \beta_2))opt(A + B)$$

- (d) and (e) : equivalent to (a)
- (f) and (g) : equivalent to (b)

These bounds can be summarized in the compact form of Table 3.1 Each pair (S,E) in the first column designates the existence of a path for S to E. Also, note that these bounds are cumulative; case (b) corresponds to the second and third rows of Table 3.1.

CHAPTER 4

EVALUATION

We evaluate our mapping and scheduling framework on mainstream DNN models, a set of computer vision neural networks popular in the field of image classification, from the *Torchvision* model library, and on randomly wired neural networks (RWNNs) also performing image classification tasks [4]. We focus more on the latter because the topological irregularity of RWNNs makes it more difficult to have a good intuition on what a good mapping and scheduling should look like thus necessitating automated algorithms. We choose representatives from three random graph models (Erdos-Renyi, Watts-Strogatz and Barbasi-Albert), with parameters chosen corresponding to the seeds which achieved the best accuracy in prior work [4]: we sample 6 models generated with parameters WS(4, 0.75), ER(0.2) and BA(5), and with module size $N \in \{10, 32\}$. We consider systems comprised of a CPU (Intel Xeon (skylake) CPU 2.00GHz) and two different GPUs (Nvidia Tesla T4 and A100 GPUs) to represent a typical heterogeneous system—relative speeds are shown in Table 4.2.

Our experiments perform a thorough comparison of our exact MILP solution, our modularity-based splitting heuristic (MILP-SPLIT), and a large number of established baselines from prior work, introduced in Section 4.2. We present our findings when optimizing solely for latency (Section 4.3) using model parallelism, and when optimizing for throughput (Section 4.4) using both data and model parallelism. In both cases, we evaluate the solution quality and cost for Torchvision models, for single-module RWNNs, and for multi-module RWNNs. Our findings demonstrate the superiority and practicality of MILP-SPLIT compared to existing baseline algorithms, and the fidelity of our estimated lower bound.

4.1 Experimental setup

The complete pipeline of our scheduler's evaluation setup for the aforementioned networks starts with a Pytorch model object. To convert it into a coarse grain DAG, while performing our (optional) preprocessing clustering step and benchmarking each operation on the hardware platform, we employ the profiling toolkit torch.fx[73]. torch.fx consists of three main components: a symbolic tracer, an intermediate representation, and Python code generation. What we are interested in is the symbolic tracer and in particular its Interpreter class. The Interpreter class is responsible for executing an FX graph, which represents the dataflow graph of the inference process of a neural network, on a node-by-node basis. This execution pattern offers various benefits, such as facilitating code transformations and analysis passes. The behavior of execution can be customized by overriding methods within the Interpreter class. By overriding the node run method, we can individually measure the performance of executing each computational node on different backends by invoking the appropriate routine on the respective node, thus creating our DAG while simultaneously benchmarking each operation on every device. This constitutes the initial input of our scheduling framework (Figure 2.5).

4.2 Baselines and Heuristics

We compare our MILP solver and MILP-SPLIT against popular scheduling algorithms and general purpose optimization heuristics which have shown success in DAG scheduling contexts or graph problems more generally.

- MET: the Minimum Execution Time algorithm is a list-based scheduling

algorithm that schedules tasks based on their minimum execution time to minimize the latency of a DAG. We extend the MET algorithm to the batched throughput optimization by selecting the best batch-device combination for each task.

- Greedy: is a greedy heuristic that considers the overall latency for scheduled tasks so far when scheduling the current task.
- HEFT: the Heterogeneous Earliest Finish Time [34] algorithm is an effective approach for scheduling tasks in a heterogeneous computing environment. It assigns tasks to processing nodes with different processing speeds to minimize overall execution time, using two phases to prioritize tasks based on estimated finish times.
- Simulated Annealing (SA) [53]: is a stochastic optimization heuristic algorithm that draws inspiration from statistical mechanics concepts and has been widely used in various optimization problems, including scheduling, for example, to minimize latency [23, 37, 52].
- Biased (1+1) EA: We implement a biased version of the (1+1) EA [32] as an additional approximation heuristic. Also known as the random hill climbing algorithm, it is one of the most basic evolutionary algorithms but has been surprisingly efficient in practice [16, 20]. We qualify as biased the (1+1) EA when the initialisation is not randomly sampled but chosen in a greedy manner, by assigning each task to the device on which it runs fastest.

Fitness function: Here we give a succinct formulation of our problem as an objective function and an integer-string search space, which are adopted by two of our search heuristics: (1+1) EA and SA. We encode the mapping solution as

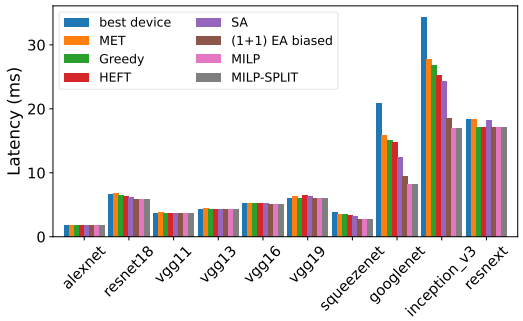


Figure 4.1: Inference latency for Torchvision DNNs deployed on a heterogeneous platform with different schedulers.

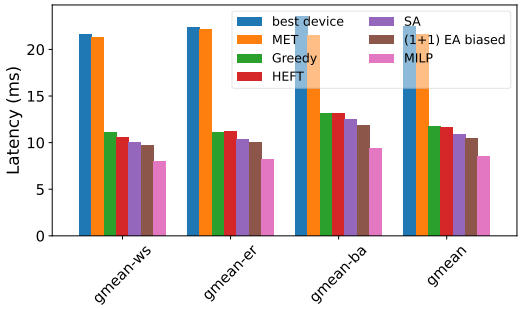


Figure 4.2: Inference latency for single RWNN modules on a heterogeneous platform with different schedulers.

Table 4.1: Latency of RWNNs consisting of 10 modules. Results reported in milliseconds (ms).
Best and second best results are highlighted in red (bold) and blue respectively.

Model	BD	MET	Greedy	HEFT	EA	SA	MILP	SPLIT	Bound
1-ch	211.7	209.9	104.4	99.9	97.5	98.9	**80.1**	**80.1**	80.1
sd, 2-ch	234.9	219.3	111.8	109.3	103.8	106.2	**78.9**	79.1	73.7
sd, 3-ch	236.5	235.4	114.2	108.5	104.7	106.1	**79.9**	80.3	68.1
sd, 4-ch	250.5	249.1	116.3	111.3	107.6	109.5	**79.1**	79.4	61.3
wd, 2-ch	225.2	223.8	103.7	101.7	97.3	98.1	**74.3**	77.6	73.3
wd, 3-ch	229.9	229.3	107.4	104.3	100.9	103.1	**76.6**	78.7	71.8
wd, 4-ch	242.9	240.7	106.8	104.4	102.0	103.6	**71.6**	77.0	62.1

a string of integers, wherein each integer in the string signifies a distinct identifier of the device to which a node is mapped. The position of each integer in the string corresponds to the layers of the DNN, arranged in a breadth-first topological ordering. Finally, the fitness function adopted for the latency (throughput) optimization problem corresponds to the latency (throughput) of a sampled mapping with a breadth-first topological ordering.

4.3 Latency Optimization

Figure 4.1 evaluates our scheduler to optimize latency for mainstream Torchvision models. There are no real improvements for DNNs with little to no parallelism, such as AlexNet or ResNet or VGG, the optimal schedule is usually the one where all tasks are mapped to the best performing device (A100 GPU). However, for models with higher parallelism, the improvement from MILP and MILP-SPLIT are significantly higher—more than 100% and 150% for Inception v3 and GoogLeNet respectively. Both MILP and MILP-SPLIT converge to the optimal solution for all Torchvision models without a substantial increase in runtime, thanks to the simplicity and regularity of these DNNs.

Next, we evaluate RWNNs which we expect to be a significantly more challenging workload. In our first experiment in Figure 4.2, we schedule a *single* module on our heterogeneous system, optimized for latency. Compared to simply running the RWNN module on the best device, there is a major $\sim 2\times$ improvement in overall latency from fully-utilizing our heterogeneous system with a CPU and 2 GPUs. When comparing across different scheduling algorithms, MILP converges to the optimal solution and is 22%-26% better than the best available heuristic on equivalent runtimes. However, with RWNNs featuring multiple modules, ten in our experiment, solving MILP on the whole model is more difficult for the solver and is exponentially slower. This motivates the use of MILP-SPLIT for those more realistic multi-module RWNNs that are representative of DNNs created by NAS.

To evaluate MILP-SPLIT, we stack multiple RWNN modules to represent realistic NAS-discovered models. In this case, each module is generated using the ER(0.2) model and may include multiple communication channels to connect to the next module. As indicated by our lower bounds formulation (Section 3.3.1),

Table 4.2: Relative speed in milliseconds (ms) on experiment devices, averaged over our evaluated DNNs.

	CPU	GPU (T4)	GPU (A100)
Torchvision	223.10 (29×)	12.16 (1.6×)	7.80 (1×)
RWNNs	183.39 (7.10×)	32.58 (1.26×)	25.84 (1×)

Table 4.3: Speedup of the splitting heuristic for the latency optimization of RWNN models with [5, 10, 20] modules.

	sd			**wd**		
Modules	MILP	SPLIT	factor	MILP	SPLIT	factor
5	82.69s	2.26s	37x	129.08s	2.45s	53x
10	232.24s	4.83s	48x	271.66s	5.00s	54x
20	1907.12s	13.49s	141x	5850.37s	14.81s	395x

the density of nodes and edges that are accessible from the endpoints of communication edges can significantly impact the quality of the splitting heuristic and the accuracy of the corresponding lower bound. Therefore, we evaluate our splitting heuristic using two different scenarios for the topology of communication edges. In the first scenario, module inputs and outputs are randomly sampled for their dependencies, while in the second scenario, all paths within a module stem from (converge toward) at least one input (output). We refer to these scenarios as the "weakly dependent" scenario (**wd**) and the "strongly dependent" scenario (**sd**), respectively, and examples are shown in Figures 3.2e and 3.2c.

Based on the results presented in Table 4.1, it can be observed that our splitting heuristic (MILP-SPLIT) exhibits a solution that is in close proximity to the optimal solution. Additionally, this heuristic outperforms all other scheduling methods considered in this study by a significant margin, as it is ∼30% better compared to the best heuristic baseline. Table 4.3 highlights that the MILP-SPLIT heuristic provides a substantial improvement (37×–395×) in runtime compared to MILP

Throughput Optimization

(2 hours timeout for MILP and 600 seconds timeout for MILP-SPLIT.)

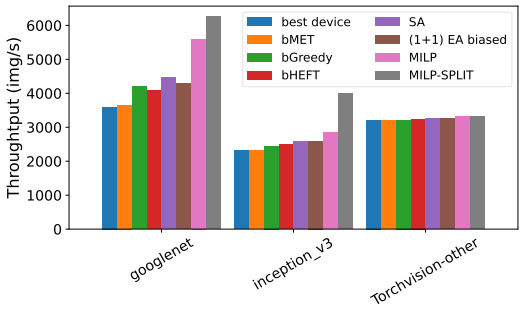


Figure 4.3: Inference throughput for a batch (B=128) inputs on Torchvision models on a heterogeneous platform.

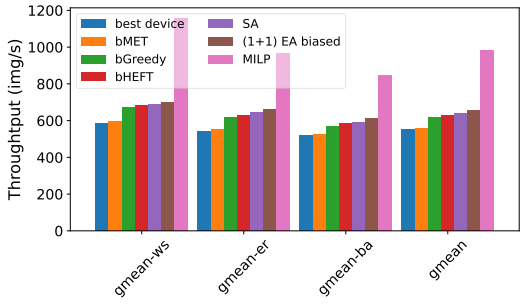


Figure 4.4: Inference throughput for a batch (B=16) inputs on single RWNN modules on a heterogeneous platform.

Table 4.4: Throughput for RWNNs consisting of 10 modules. Results reported in images-per-second (imgs/s).
Best and second best results are highlighted in red (bold) and blue respectively.

Model	BD	MET	Greedy	HEFT	EA	SA	MILP	SPLIT	Bound
1-ch	54	56	74	75	84	87	114	**135**	164
sd, 2-ch	48	50	67	66	75	78	95	**119**	180
sd, 3-ch	49	51	68	70	78	81	116	**129**	196
sd, 4-ch	47	48	65	67	76	79	73	**126**	209
wd, 2-ch	51	53	76	75	86	87	89	**137**	182
wd, 3-ch	49	52	73	73	82	85	72	**137**	181
wd, 4-ch	47	50	72	74	82	84	65	**138**	207

when both scheduling algorithms reach their best solution. Also shown in Table 4.1 is our lower bound (LBound), which offers a convenient means of obtaining a quick performance guarantee for the splitting heuristic. Our observations indicate that for the **wd** models, the LBound is closer to the true optimum than for the **sd** models, where it tends to be more pessimistic. This difference is attributed to the lower bound computation which considers complete overlap in scheduling separate paths originating from each module output. This is more likely to hold in an optimal schedule for the **wd** scenario, where the distribution of these paths is more evenly spread compared to the **sd** scenario, where a specific endpoint's emanating

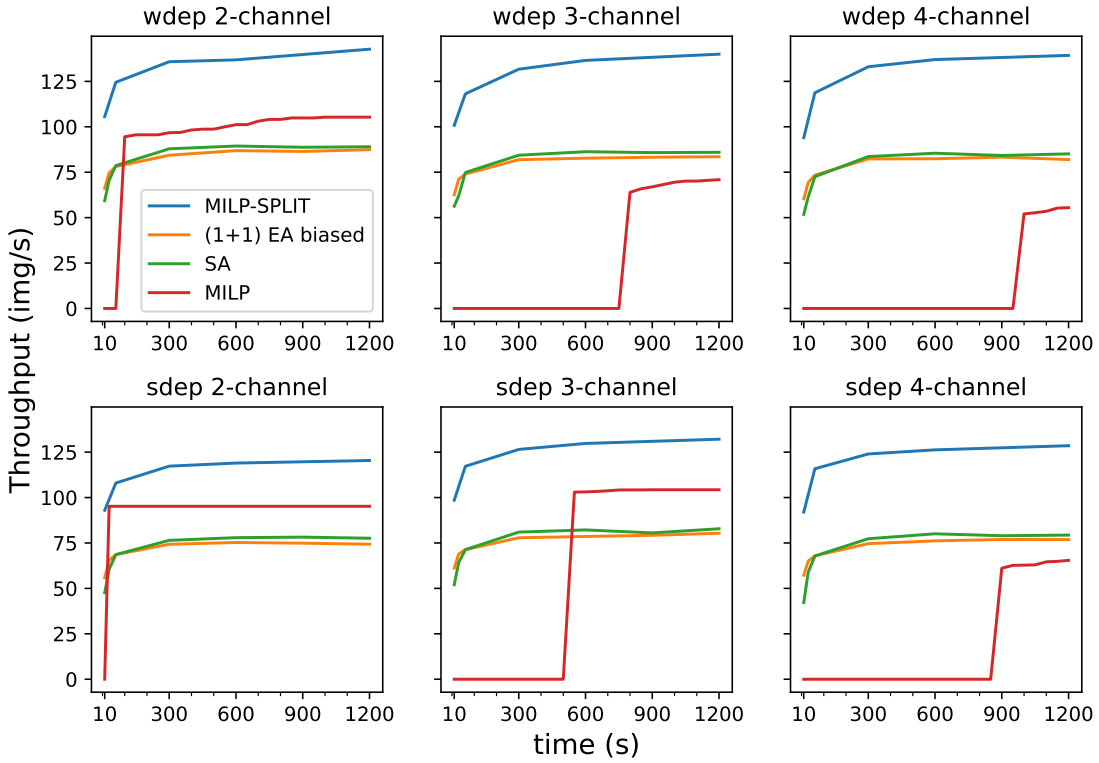


Figure 4.5: Solution quality (throughput) over time for MILP, MILP-SPLIT and heuristics on 10 modules RWNNs.

paths cover all the predecessor or dependency subgraphs—this phenomenon is also the reason why MILP-SPLIT is closer to the optimum on **sd** graphs. Our results show that MILP-SPLIT is a viable and high-quality heuristic that offers lower-bound guarantees on quality.

4.4 Throughput Optimization

We consider throughput optimization in the low-latency inference regime, where we batch B inputs (e.g. 128) and we find the fastest way to run that batch using the available devices. Successive inputs are queued together in groups of B before going to the hardware system for execution. This realistically mimics how inference

is done in datacenters where low latency is critical to respond to user requests promptly.

Figures 4.3, 4.4, and Table 4.4 show our throughput optimization results attained with our framework via batching. bMET, bGreedy and bHEFT are the batched equivalent of the corresponding heuristics. In this case, we have a batch of inputs B queued for processing, and our scheduler can further decompose this batch into $B/4$, $B/2$, and $3B/4$ when allocating inputs to different devices. This enables our scheduler to leverage both model and data parallelism when mapping the DNN workload onto the hardware system. Unlike the latency objective, the MILP solving on the whole graph does not terminate within a 2 hours deadline, even for single RWNN modules or for regular networks with high model parallelism such as inception-based DNNs. Consequently, MILP-SPLIT outperforms naïve MILP solving both in terms of scheduling quality and runtime. It is worth noting that since MILP cannot reach the optimum solution for a single RWNN module, MILP-SPLIT provides only an approximate solution for each of its module schedules. However, our splitting heuristic achieves up to $\sim$60% better performance than the best-performing heuristic baseline with equivalent running times. Results reported in Table 4.4 are based on 600s deadlines for MILP-SPLIT and for other search heuristics, EA and SA. Moreover, Figure 4.5 provides a more detailed view of the solution quality over time, illustrating the challenge of solving the scheduling problem on the entire graph using MILP with numerous communication channels.

CHAPTER 5

CASE STUDIES

Subsequently, we present two case studies that serve as illustrations of the adaptability and versatility of our scheduling framework. The first case study involves mapping Inception v3 onto a platform with limited memory resources. The objective is to demonstrate how our framework efficiently handles the memory constraints during the scheduling process. In the second case study, we explore the scheduling of a Large-scale Linear Model (LLM) across a distributed infrastructure comprising heterogeneous compute nodes. This case study showcases the capability of our framework to effectively allocate and utilize resources in a diverse computing environment.

5.1 Case Study: Inception v3 on Memory Constrained Platform

The fastest accelerator in our platform now has a limited memory thus making it more challenging to accelerate a DNN batch of 128 inputs. We handle this case in our scheduler by limiting the maximum batch size available to the A100 GPU to 64. Table 5.1 presents 5 different scenarios that illustrate how we can obtain significantly better throughput compared to naïve scheduling when the entire batch does not fit in the fastest accelerator's memory. First, as shown in 5.1, we can simply run two batches of 64 serially on A100. However using both GPUs (T4 and A100) in parallel, running batches of 64 on each, increases performance by taking advantage of the obvious data parallelism. Combining both accelerators allows better performance trivially by saturating the faster A100 GPU and mapping the rest of

Table 5.1: When the fastest device (A100) doesn't have enough memory to store the entire batch (B=128), we limit its maximum batch size to 64. We report the latency of processing the whole batch (B=128), and its corresponding throughput for Inception v3.

Scenario	**Latency**	**Throughput**
T4 only, batch = 64	258.8 ms	494 imgs/s
T4 only, batch = 128	235.7 ms	543 imgs/s
A100 only, batch = 64	181.3 ms	706 imgs/s
Naïve, T4 + A100	129.4 ms	990 imgs/s
EA + MILP (Ours)	71.8 ms	1782 imgs/s

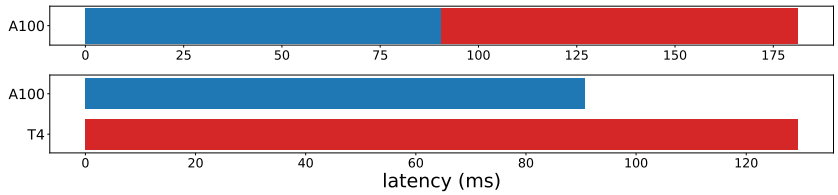


Figure 5.1: From top to bottom: A100 only (batch 64). Naïve (uniform batch 64)

the batch on the T4 GPU, compared to running the workload on a single device. However, by further leveraging model parallelism, our scheduler outperforms this naïve solution by 1.8× under the same constraints. Figure 5.2 shows an example of a single inception module mapped onto the considered heterogeneous system. We are able to automatically split a DNN batch onto different hardware devices to leverage data parallelism (as shown in Figure 5.2 when the same operation runs on two different devices), combined with the model parallelism of selecting different devices for different operations. In Figure 5.3, we plot throughput at different memory capacities, showing that our schedule can achieve a significantly higher throughput compared to naïve data parallelism that can be manually performed with deep learning frameworks today. This highlights the need for our automatic data and model parallel partitioning for DNNs on heterogeneous hardware platforms that are already pervasive in today's computing systems.

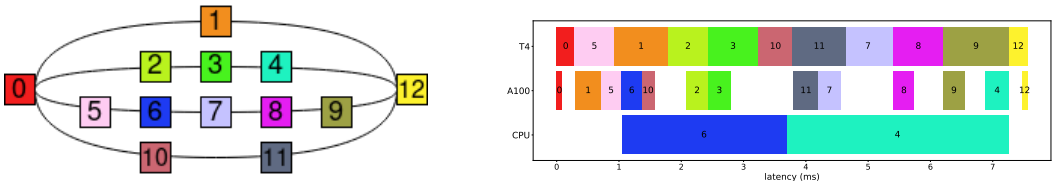


Figure 5.2: One inception module mapping and scheduling (batch = {128/64, 128/64, 64}, EA + MILP)

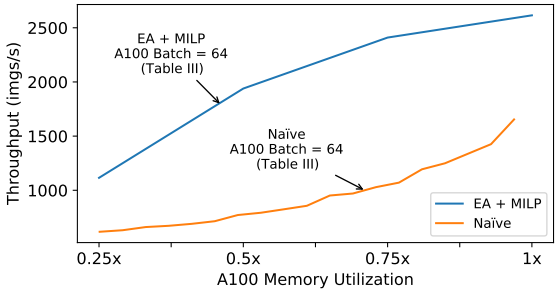


Figure 5.3: Throughput for a batch of 128 inputs for Inception v3 with a memory constrained accelerator (A100). Memory shown on a relative scale.

5.2 Case Study: GPT-3 Inference on Distributed Heterogeneous Compute Nodes

As DNN models continue to grow in size, it has become necessary to extend our focus beyond single-node servers to extend our formulation to more complex setups. In this case study, we investigate the use of our scheduler for a large language model (LLM), GPT-3 [17], on a distributed heterogeneous platform as shown in Figure 5.4. This model belongs to a category of deep neural networks that exhibits notable modularity, as it is predominantly constructed by stacking transformer modules [84]. In contrast to our earlier analysis of RWNNs, GPT-3 modules exhibit high regularity but the complexity of the problem stems from the larger search space of hardware device options. To counterbalance that, a key aspect of our analysis revolves around the exploitation of **symmetry** in the hardware platform to restrict the search space size without sacrificing solution quality.

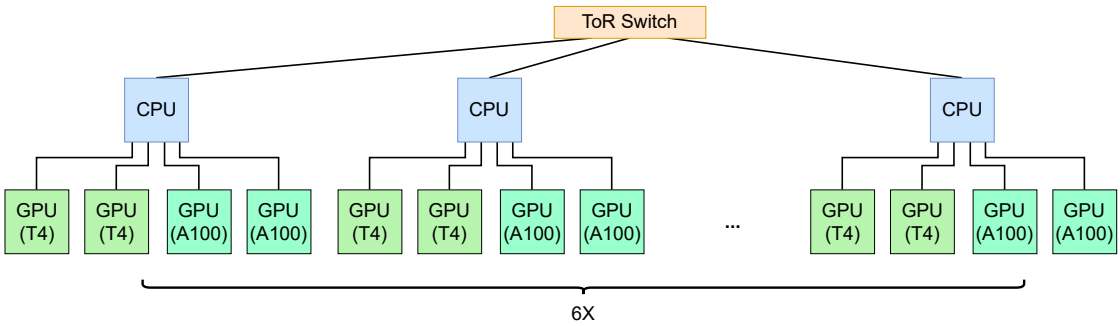


Figure 5.4: Multi-node heterogeneous system for GPT-3 inference.

As reported in Table 5.2 we consider two ways to schedule our GPT-3 graph. "Single node" utilizes our MILP solvers to schedule one-sixth of the GPT-3 graph on a single node, then replicates that schedule for the rest of GPT-3 on the remaining 5 nodes. We consider this a competitive baseline because it automates the common practice of manually partitioning LLM inference to fit a single compute node then scaling up the number nodes. "Multi node" exposes the entire compute system (all 6 nodes) to our MILP-SPLIT solver, but we employ *symmetry-breaking* techniques to compress the solution space of the explored schedules, allowing us to find a high-quality schedule in reasonable time. Symmetries arise from the fact that each schedule S represents a set of equivalent solutions E_S, where any element within this set can be derived from S by permuting device mappings while maintaining the same overall latency. In our approach, we introduce additional constraints to our MILP formulation, enforcing a partial ordering of certain variables (e.g. #batches, #tasks, time utilization) between identical devices within a node or between nodes. For example, we can ensure that the number of tasks assigned to node i is always less than or equal node j for $0 \leq i < j < 6$ in our example system (Fig. 5.4). This retains all non-isomorphic solutions in our search space whilst compressing it by $\sim 4^6 6! = 2.9 \times 10^6$, where the 6! and 4^6 factors represent inter- and intra-node symmetry respectively.

Furthermore, our experimental results demonstrate that the choice of symmetry-breaking criterion can significantly impact the quality of the solution. This can be attributed to the phenomenon of premature convergence. If the symmetry-breaking constraints overly restrict the problem or generates a compressed space whose topology is not regular enough, the solver may settle for a locally optimal solution instead of exploring other potentially superior regions of the solution space either located outside of the compressed space or harder to access with the solver's intrinsic optimization heuristics due to the irregularity of the new space. We hypothesize that utilizing #batches as the symmetry-breaking criterion tends to be overly restrictive, discouraging the solver from performing batch rearrangements that would contradict the ordering constraints, thus resulting in relatively smaller improvements over MILP-SPLIT without symmetries. On the other hand, despite the discrete nature of task variables and the continuous nature of utilization time variables, both variables are coarser grain than #batches thus yielding comparable performance and surpassing the baseline schedule by $\sim$31% and the single node MILP-SPLIT by $\sim$10%. Furthermore, the higher abundance of intra-node symmetries compared to inter-node symmetries leads to a greater compression of the search space ($\sim 4^6 = 4096\times$ vs. $\sim 6! = 720\times$), resulting in better performance when considered independently. Our results lay the foundations towards multi-node heterogeneous scheduling leveraging MILP-SPLIT, and we aim to further explore this topic in future work.

Table 5.2: GPT-3 throughput (inputs/s) on our distributed system. We use a 600s timeout and each input is 20 tokens.

Heuristic	**Throughput**
Single node - MILP (baseline)	4.8
Single node - MILP-SPLIT	5.7
Multi node - MILP-SPLIT no symmetries	5.0
Multi node - MILP-SPLIT all symmetries – batch	5.6
Multi node - MILP-SPLIT all symmetries – task	**6.3**
Multi node - MILP-SPLIT all symmetries – time	**6.3**
Multi node - MILP-SPLIT inter-node symmetry - time	5.7
Multi node - MILP-SPLIT intra-node symmetry - time	6.1
UBound	9.9

CHAPTER 6

CONCLUSION

In conclusion, this master's thesis presented a comprehensive framework that effectively utilizes both data and model parallelism to schedule Deep Neural Networks (DNNs) on heterogeneous hardware systems. The proposed algorithmic approaches, including an exact Mixed-Integer Linear Programming (MILP) solution and a splitting heuristic called MILP-SPLIT, demonstrated significant advancements in optimizing throughput and latency for both conventional and randomly-wired DNNs.

The experimental results showcased remarkable improvements, with throughput and latency optimizations exceeding 30–60% compared to the best and most widely-used heuristics. Moreover, MILP-SPLIT proved to be up to$\sim$395 times faster than a full MILP solution. These findings highlight the efficacy and efficiency of the proposed framework in scheduling DNNs on heterogeneous hardware systems.

Furthermore, the usage of the scheduler to larger multi-node heterogeneous server deployments was successfully demonstrated by improving the scheduling of GPT-3 through the exploitation of symmetries in the hardware system. This expansion opens up possibilities for deploying the framework in real-world scenarios involving distributed systems.

Looking forward, the future research agenda aims to expand the framework's capabilities by exploring more efficient methods for scheduling large DNNs on distributed systems. This will involve addressing the challenges associated with DNN training, as well as incorporating pre- and post-processing components of

deep learning workloads. By addressing these aspects, the framework can further enhance the performance and scalability of scheduling deep learning tasks in diverse computational environments.

In summary, this thesis contributes to the field of DNN scheduling by presenting a robust framework that harnesses the power of both data and model parallelism. The proposed algorithmic approaches, supported by empirical evaluations and theoretical guaantees, demonstrate significant advantage in optimizing throughput, latency, and overall scheduling efficiency. The future research directions outlined will enable the framework to tackle additional challenges and pave the way for even more efficient scheduling of large DNNs in distributed systems.

BIBLIOGRAPHY

[1] Zahraa A Abdalkareem, Amiza Amir, Mohammed Azmi Al-Betar, Phaklen Ekhan, and Abdelaziz I Hammouri. Healthcare scheduling in optimization context: a review. *Health Technol (Berl)*, 11(3):445–469, April 2021.

[2] M. S. Abdelfattah et al. DLA: Compiler and fpga overlay for neural network inference acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 411–4117, 2018.

[3] Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007. Mixed Integer Programming.

[4] Byung Hoon Ahn et al. Exploring randomly wired neural networks for image recognition. In *ICCV*, pages 44–57, 2019.

[5] Byung Hoon Ahn et al. Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, pages 44–57, 2020.

[6] Byung Hoon Ahn et al. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, volume 2, 2020.

[7] Ali Ala and Feng Chen. Appointment scheduling problem in complexity systems of the healthcare services: A comprehensive review. *J Healthc Eng*, 2022:5819813, March 2022.

[8] Ilkin Aliyev, Joshua Mack, Nirmal Kumbhare, Ali Akoglu, and H. Fatih Ugurdag. Fpga-based minimal latency heft scheduler for heterogeneous computing. In *2021 6th International Conference on Computer Science and Engineering (UBMK)*, pages 1–5, 2021.

[9] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

[10] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, nov 2017.

[11] Abbas Bazzi and Ashkan Norouzi-Fard. Towards tight lower bounds for scheduling problems. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms – ESA 2015*, pages 118–129. Springer, sep 2015.

[12] E. M. Beale et al. Global optimization using special ordered sets. *Math. Program.*, 10(1):52–69, dec 1976.

[13] EML Beale and JA Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. j. lawrence, ed., or 69: Proceedings of the fifth international conference on operational research, 1970.

[14] Timo Berthold. *Heuristic algorithms in global MINLP solvers*. PhD thesis, 2014.

[15] Timo Berthold and Domenico Salvagnin. Cloud branching. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 28–43, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] Pavel A. Borisovsky et al. A study on performance of the (1+1)-evolutionary algorithm. In *FOGA*, 2002.

[17] Tom Brown et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[18] Han Cai et al. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations (ICLR)*, 2019.

[19] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. *Comparative Evaluation Of The Robustness Of DAG Scheduling Heuristics*, pages 73–84. Springer US, Boston, MA, 2008.

[20] Ted Carson et al. Hill-climbing finds random planted bisections. In *SODA, ACM press, 2001*, pages 903–909, 2001.

[21] Gabriele Castellano, Juan-José Nieto, Jordi Luque, Ferrán Diego, Carlos Segura, Diego Perino, Flavio Esposito, Fulvio Risso, and Aravindh Raman. Scheduling inference workloads on distributed edge clusters with reinforcement learning, 2023.

[22] Adrian M. Caulfield et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[23] Esra Celik et al. A novel simulated annealing-based optimization approach for cluster-based task scheduling. *Cluster Computing*, 24(4):2927–2956, Dec 2021.

[24] Lijun Chang et al. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '13, page 205–216, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Tianqi Chen et al. TVM: An automated End-to-End optimizing compiler for deep learning. In *OSDI*, pages 578–594, 2018.

[26] Xiaobing Chen et al. Partition and scheduling algorithms for neural network accelerators. In *APPT*, pages 55–67, 2019.

[27] Fabián A Chudak and David B Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms*, 30(2):323–343, 1999.

[28] Marco Cococcioni et al. The big-M method with the numerical infinite M. *Optimization Letters*, 15(7):2455–2468, Oct 2021.

[29] G.I. Davida and D.J. Linton. A new algorithm for the scheduling of tree structured tasks. In *Proc. Conf. Inform. Sci. and Syst.*, pages 543–548, Baltimore, MD, 1976.

[30] Tatjana Davidović et al. Mathematical programming-based approach to scheduling of communicating tasks. 01 2005.

[31] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. Scheduling with Communication Delays via LP Hierarchies and Clustering. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 822–833, Durham, NC, USA, nov 2020. IEEE.

[32] Stefan Droste et al. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1):51–81, 2002.

[33] Yubin Duan and Jie Wu. Joint optimization of dnn partition and scheduling for mobile cloud computing. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Kalka Dubey et al. Modified heft algorithm for task scheduling in cloud environment. *Procedia Computer Science*, 125:725–732, 2018. The 6th International Conference on Smart Computing and Communications.

[35] Lukasz Dudziak et al. BRP-NAS: Prediction-based nas using gcns. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[36] Zhou Fang, Tong Yu, Ole J. Mengshoel, and Rajesh K. Gupta. Qos-aware scheduling of heterogeneous servers for inference in deep neural networks. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, page 2067–2070, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Crescenzio Gallo et al. A simulated annealing algorithm for scheduling problems. *Journal of Applied Mathematics and Physics*, 7, 10 2019.

[38] Dorabela Gamboa, César Rego, and Fred Glover. Data structures and ejection chains for solving large-scale traveling salesman problems. *European Journal of Operational Research*, 160(1):154–171, 2005. Applications of Mathematical Programming Models.

[39] M. R. Garey and D. S. Johnson. " strong " np-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, jul 1978.

[40] Hongmin Geng, Deze Zeng, and Yuepeng Li. Performance efficient layer-aware

dnn inference task scheduling in gpu cluster. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 2242–2247, 2022.

[41] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers Operations Research*, 13(5):533–549, 1986. Applications of Integer Programming.

[42] Tobias Grosser et al. Polly-acc transparent compilation to heterogeneous hardware. In *ICS*, 2016.

[43] Xiaokang Han, Wenzhou Yan, and Mei Lu. Intelligent critical path computation algorithm utilising ant colony optimisation for complex project scheduling. *Complexity*, 2021:9930113, Jul 2021.

[44] Mohammed-Albarra Hassan Abdel-Jabbar, Imed Kacem, and Sébastien Martin. Unrelated parallel machines with precedence constraints: application to cloud computing. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 438–442, 2014.

[45] Kaiming He et al. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[46] Biao Hu, Zhengcai Cao, and MengChu Zhou. Energy-minimized scheduling of real-time parallel workflows on heterogeneous distributed computing systems. *IEEE Transactions on Services Computing*, 15(5):2766–2779, 2022.

[47] T.C. Hu. Parallel sequencing and assembly line problems. *Oper. Res.*, 9:841–848, 1961.

[48] Zeren Huang, Kerong Wang, Furui Liu, Hui ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming, 2021.

[49] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, 2017.

[50] 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)*, pages 329–341, 2021.

[51] Richard Karp. Reducibility among combinatorial problems. volume 40, pages 85–103, 01 1972.

[52] Mostafa Haghi Kashani et al. Using simulated annealing for task scheduling in distributed systems. In *2009 International Conference on Computational Intelligence, Modelling and Simulation*, pages 265–269, 2009.

[53] S. Kirkpatrick et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[54] Dušan Knop and Martin Koutecký. Scheduling meets n-fold integer programming. *Journal of Scheduling*, 21(5):493–503, Oct 2018.

[55] Wieslaw Kubiak, Djamal Rebaine, and Chris Potts. Optimality of hlf for scheduling divide-and-conquer uet task graphs on identical parallel processors. *Discrete Optimization*, 6:79–91, 2009.

[56] R. Kumar et al. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *MICRO*, pages 81–92, 2003.

[57] Mohamed Kurdi. An effective genetic algorithm with a critical-path-guided giffler and thompson crossover operator for job shop scheduling problem. *International Journal of Intelligent Systems and Applications in Engineering*, 7(1):13–18, Mar. 2019.

[58] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1):259–271, January 1990.

[59] Jiawen Liu et al. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *MICRO*, pages 655–668, 2018.

[60] PANGFENG Liu et al. Theoretical study of densenet scheduling on heterogeneous system architecture. In *ACDA*, 2021.

[61] W. Luk et al. A high-level compilation toolchain for heterogeneous systems. In *SOCC*, pages 9–18, 2009.

[62] François Margot. Pruning by isomorphism in branch-and-cut. In *Proceedings of the 8th International IPCO Conference on Integer Programming and Combinatorial Optimization*, page 304–317, Berlin, Heidelberg, 2001. Springer-Verlag.

[63] Yuan Meng et al. Dynamap: Dynamic algorithm mapping framework for low latency cnn inference. In *ISFPGA*, page 183–193, 2021.

[64] Azalia Mirhoseini et al. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2430–2439. JMLR.org, 2017.

[65] Matthias Mnich and Andreas Wiese. Scheduling and fixed-parameter tractability. *Mathematical Programming*, 154(1):533–562, Dec 2015.

[66] Fatma A. Omara et al. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13–22, 2010.

[67] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[68] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot, editors, *High-Performance Computing and Networking*, pages 493–498, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[69] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[70] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[71] Paul Walton Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10:76–94, 1970.

[72] Fareed Qararyah et al. A computational-graph partitioning method for training memory-constrained dnns. *Parallel Comp.*, pages 104–105, 2021.

[73] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch.fx: Practical program capture and transformation for deep learning in python, 2021.

[74] Gerhard Reinelt. TSPLIB—A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3(4):376–384, November 1991.

[75] Edward E. Rothberg and Bruce Hendrickson. Sparse matrix ordering methods for interior point linear programming. *INFORMS J. Comput.*, 10:107–113, 1998.

[76] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single gpu, 2023.

[77] David Silver and other. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[78] David Silver and other. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[79] Vlad-Mihai Sima et al. Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform. In *ISPA*, pages 1–6, 2009.

[80] Zai-Xing Sun, Rong Hu, Bin Qian, Bo Liu, and Guo-Lin Che. Salp swarm algorithm based on blocks on critical path for reentrant job shop scheduling problems. In De-Shuang Huang, Vitoantonio Bevilacqua, Prashan Premaratne, and Phalguni Gupta, editors, *Intelligent Computing Theories and Application*, pages 638–648, Cham, 2018. Springer International Publishing.

[81] Christian Szegedy et al. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[82] T Berthold T K Ralphs, Y Shinano and T Koch. Assessing performance of parallel milp solvers how are we doing, really? : Siam conference on optimization, 2017.

[83] Mingxing Tan et al. Mnasnet: Platform-aware neural architecture search for mobile. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[84] Ashish Vaswani et al. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[85] Sven Verdoolaege. Presburger formulas and polyhedral compilation. 01 2016.

[86] Zhiyuan Xu, Dejun Yang, Chengxiang Yin, Jian Tang, Yanzhi Wang, and Guoliang Xue. A co-scheduling framework for dnn models on mobile and edge devices with heterogeneous hardware. *IEEE Transactions on Mobile Computing*, 22(3):1275–1288, 2023.

[87] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[88] James J. Q. Yu. Two-stage request scheduling for autonomous vehicle logistic system. *IEEE Transactions on Intelligent Transportation Systems*, 20(5):1917–1929, 2019.

[89] Minjia Zhang, Zehua Hu, and Mingqin Li. Duet: A compiler-runtime subgraph scheduling approach for tensor programs on a coupled cpu-gpu architecture. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 151–161, 2021.

[90] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. Heterogen: Transpiling c to heterogeneous hls code with automated test generation and

program repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1017–1029, New York, NY, USA, 2022. Association for Computing Machinery.

[91] Lu Zhen, Wenya Lv, Zheyi Tan, and Bin Dong. Container transportation scheduling between port yards and the hinterland in yunfeng. *INFORMS Journal on Applied Analytics*, 52(3):250–266, 2022.

[92] Lianmin Zheng et al. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578. USENIX Association, July 2022.

[93] Barret Zoph et al. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

APPENDIX A

PRIMER ON MILP SOLVING

In this section, we provide an overview of Mixed Integer Linear Programming solvers and their optimization loop, along with various linearization techniques employed in MILP problem formulations. MILP solvers play a crucial role in solving optimization problems that involve both discrete (integer) and continuous variables, allowing for the incorporation of combinatorial decision-making into linear programming frameworks.

We begin by presenting an analysis of the optimization loop within MILP solvers. This loop encompasses the iterative process of solving a MILP problem, where the solver explores the solution space by gradually improving the objective function value while respecting the constraints. We discuss the key components of the optimization loop, including the branching strategy, bound tightening, cutting planes, and heuristics. Understanding these components is essential for comprehending the inner workings and efficiency of MILP solvers.

Next, we delve into the various linearization techniques utilized in MILP problem formulations. Since MILP models involve both linear and nonlinear components, linearization techniques are employed to convert nonlinear constraints or objective functions into linear equivalents. We explore some techniques such as big-M formulations, and logic-based linearization approaches. These techniques not only facilitate the solution of MILP problems but also enhance computational efficiency by leveraging the linearity of the underlying mathematical models.

A.1 Optimization Loop

A Mixed Integer Linear Programming (MILP) problem is a mathematical optimization problem that can be formally expressed in its canonical form as:

Objective Function:

$$\text{maximize } \mathbf{c}^\top \mathbf{x} + \mathbf{d}^\top \mathbf{y}$$

Subject to:

$$\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} \leq \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}$$

$$\mathbf{x} \in \mathbb{R}^n$$

$$\mathbf{y} \in \mathbb{Z}^m$$

where:

- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ represents the vector of continuous decision variables,
- $\mathbf{y} = (y_1, y_2, \ldots, y_m)$ represents the vector of integer decision variables,
- $\mathbf{c}$ and $\mathbf{d}$ are coefficient vectors for the objective function,
- $\mathbf{A}$ and $\mathbf{B}$ are coefficient matrices for the constraints,
- $\mathbf{b}$ is a vector of constraint bounds,
- $\mathbf{0}$ represents the vector of zeros, and
- $\mathbb{R}$ and $\mathbb{Z}$ denote the sets of real numbers and integers, respectively.

The objective of solving an MILP problem is to find values for $\mathbf{x}$ and $\mathbf{y}$ that maximize the objective function while satisfying all the given linear constraints. The set of values for $\mathbf{x}$ and $\mathbf{y}$ satisfying the constraints is called the feasible region.

In most solver frameworks, the optimization process of an MILP instance is essentially an augmented branch-and-bound algorithm. The problem is approached through a recursive process that involves dividing it into smaller subproblems. This recursive splitting creates a branching tree structure, allowing for the implicit enumeration of all potential solutions. At each subproblem, domain propagation techniques are employed to progressively reduce the feasible values for the variables. Additionally, a relaxation of the problem, assuming it is formulated as a minimization task, can be solved to obtain a local lower bound. To further enhance the relaxation, additional valid constraints can be incorporated, effectively tightening the relaxation and potentially improving the quality of the lower bound. In situations where a subproblem is determined to be infeasible, conflict analysis techniques are applied to learn new valid constraints that help identify and exclude infeasible regions of the solution space. Primal heuristics are utilized as supplementary methods aimed at improving the upper bound of the problem. These heuristics employ various strategies to generate feasible solutions quickly, offering a potential upper bound on the objective function value. Figure A.1 appearing in [82] illustrates the global optimization loop in a MILP solver. In what follows, we discuss in more details several of these components :

Presolving: During the presolving phase, a set of reformulations and simplifications are tried. Removing constraints and variables that are not necessary in the instance can reduce the amount of memory required by a computer to solve the instance and also speed the calculations. Rearranging the order of variables and constraints in the instance can speed several routines in the subsequent preprocess-

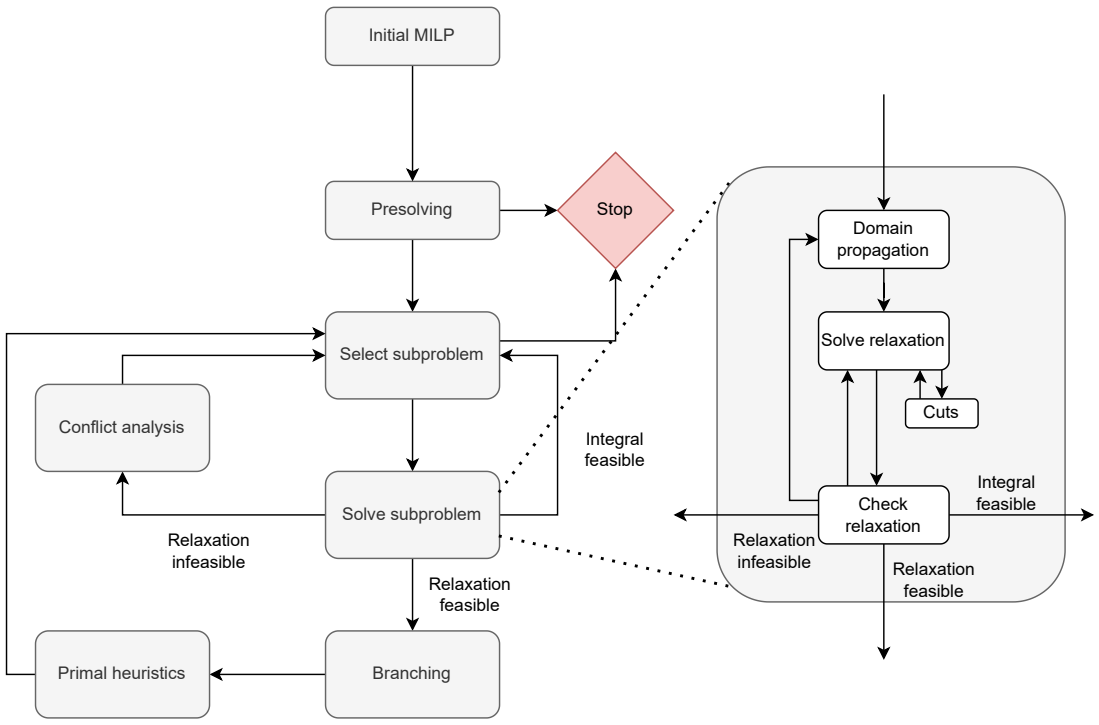


Figure A.1: Main solving loop of a generic MILP solver. Figure from [82]

ing and solve methods [75]. Constraint duplication and duplication, or granularity computation are also additional steps that can take place during the presolving phase.

Branching: The process of branching in optimization involves iteratively dividing the feasible region into smaller subproblems, allowing for a more focused exploration of potential solutions. This partitioning is accomplished through logical disjunctions that define regions within which the subproblems reside. The goal is to find the least costly solution among all solutions obtained by solving these subproblems, which ultimately leads to the determination of the global optimum. The branching strategy plays a crucial role in the optimization algorithm, as it determines the specific disjunction used to partition the current subproblem once the initial processing is completed. Typically, these disjunctions are designed to be violated by the solutions to the relaxation problem, which is solved to obtain

a lower bound on the objective function. In addition to disjunction selection, the task of branching involves establishing initial bounds for each of the generated subproblems. By default, the initial bound for a child subproblem is set equal to the final bound of its parent. However, more sophisticated branching strategies can calculate more accurate initial bounds for each child subproblem, enabling their use instead [15].

Primal heuristics: Algorithmic approaches designed to seek feasible solutions. Unlike exact methods, these heuristics do not provide a guarantee of finding a solution, let alone an optimal one. However, they have proven to be highly effective in practice and have become a crucial component of state-of-the-art solvers [14] (large neighborhood search, shift-and-propagate ...).

Cuts [48]: Cuts refer to supplementary constraints that are introduced to effectively reduce the feasible region of a linear programming problem. The primary objective of these cuts is to ensure that the points lying on the frontier of the feasible region closely approximate integer values located along the boundaries of the relaxed feasible region of the linear programming problem. It is imperative that the computed optimal solution to the LP relaxation violates these inequalities, while all solutions within the feasible region must conform to them, thereby yielding an improved upper bound in the optimization process.

Conflict analysis [3]: When a subproblem is found to be infeasible, this step is performed to analyze and track the logical implications of previous decisions taken. A conflict graph, which is continually updated whenever a subproblem is marked as infeasible, captures the underlying logical relationships that explain how the integration of constraints in the branching step resulted in the infeasibility. This graph enables the identification and prevention of similar combinations of con-

straints from emerging in other subproblems. By representing and analyzing the conflicts within the graph, it becomes possible to enhance the efficiency and effectiveness of subsequent subproblem resolutions by avoiding previously encountered infeasible constraint combinations.

A.2 Transformation and Linearization Techniques

In the process of formulating an optimization problem that reflects real-world scenarios, it is often essential to incorporate non-linear terms into the mathematical formulation to capture specific operational characteristics inherent to the decision problem at hand. However, the inclusion of non-linear terms typically leads to a significant increase in computational complexity of the optimization model. Linearization techniques in mixed-integer linear programming (MILP) refer to methods used to approximate nonlinear functions or constraints with linear equivalents. We present in this section a few techniques used in the formulation of our scheduling problem and a walk through the example of linearizing the batching constraint.

Let $(b_{i,u,l})_{i\in\mathcal{V},u\in\mathcal{K},l=1,\dots,\mathcal{L}}$, $(x_{i,u,l})_{i\in\mathcal{V},u\in\mathcal{K},l=1,\dots,\mathcal{L}}$ be sets of binary variables. We want to express :

$$b_{i,u,l} = 1 \quad \text{iff} \sum_{l'\in[1\dots\mathcal{L}]} x_{i,u,l'} = l; \tag{A.1}$$

Linearize equivalence: Let x be a binary variable and y an integer bounded variable constrained as follows :

$$(\mathrm{C}_1) \quad x = 1 \quad \text{iff} \quad y \leq 0$$

We introduce M a sufficiently large constant ($M > |x| + |y|$) and claim that C_1 is equivalent to:

$$(C'_1) \quad y \leq M(1-x) \quad \text{and} \quad -y < Mx$$

This can be seen as only one of the equations above is "active" depending on the sign of y: if $y \leq 0$, then $M(1-x) \geq y$ is always true, whereas $Mx > -y$ enforces $x = 1$. Similarly, if $y > 0$ then only $M(1-x) \geq y$ is active and sets $x = 0$.

Hence, by rewriting A.1 as :

$$b_{i,u,l} = 1 \quad \text{iff } 2|\sum_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l| - 1 \leq 0; \tag{A.2}$$

We transform the equivalence into [1] :

$$\begin{cases} 2|\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l| - 1 \leq M(1 - b_{i,u,l}) \\ 1 - 2|\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l| \leq M b_{i,u,l} \end{cases} \tag{A.3}$$

where M is a sufficiently large constant.

Linearize absolute value: Let x and y be integer bounded variable related by a constraint of the form $(C_2) \quad |x| \leq y$ can be easily rewritten by splitting the constraint into two

$$(C'_2) \quad x \leq y \quad \text{and} \quad -x \leq y$$

However, for its complementary form $(C_3) \quad y \leq |x|$, the feasible region for x is not convex anymore and thus cannot be unconditionally [2] transformed to linear equations. The constraint C_3 is equivalent to

$$(C'_3) \quad y \leq x \quad \text{or} \quad x \leq -y$$

[1] we can replace the strict inequality with a less than or equal since $2|\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l| - 1$ is never null

[2] without assuming that the variables are bounded

In a similar fashion to what we did for the linearization of equivalence, we introduce M a large enough constant and *a binary variable* d. Intuitively, d will encode the sign of x ($d = 1_{x>0}$). Then C'$_3$ is equivalent to

$$(\text{C"}_3) \quad x \geq y - Md \quad \text{and} \quad x \leq -y + M(1-d)$$

Depending on the value of d, it is either the first or second part of C'$_3$ that is active.

Composing with A.3, we obtain the final set of constraints:

$$\begin{cases} 2(\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l) - 1 \leq M(1 - b_{i,u,l}) \\ -2(\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l) - 1 \leq M(1 - b_{i,u,l}) \\ 2(\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l) - 1 \geq -Mb_{i,u,l} - M'\delta_{i,u,l} \\ 2(\sum\limits_{l' \in [1...\mathcal{L}]} x_{i,u,l'} - l) + 1 \leq Mb_{i,u,l} + M'(1 - \delta_{i,u,l}) \end{cases} \tag{A.4}$$

where M and M' are two sufficiently large constants with $M << M'$ and $\delta_{i,u,l}$ are binary variables.