When Placement Meets GPU: GPU-Accelerated VLSI Placement and Device Placement for GPUs

LIU, Lixin

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

August 2023

Abstract

Placement serves as a fundamental yet challenging combinatorial optimization problem in various research areas including but not limited to artificial intelligence, electronic design automation, resource scheduling, etc. With the rapid development of GPU's computational power, GPU acceleration become an important research direction to pursue to handle large-scale problems with parallelism. Motivated by this emerging trend, we exploit the potential of GPU in placement and study two essential problems when placement meets GPU: GPU-accelerated VLSI placement and device placement for GPUs.

VLSI placement is an extremely sophisticated problem in electronic design automation (EDA), that is highly correlated to the circuit's power, performance, and area (PPA). Meanwhile, the enormous number of transistors in modern chips hugely increases the computational complexity of VLSI placement. To address this rising demand for an effective yet scalable VLSI placer, we propose a GPU-accelerated VLSI placement framework that effectively generates high-quality solutions in an extremely fast manner. We also extend the proposed framework with neural network enhancement and illustrate the possibility of incorporating a neural network component into analytical placement. Besides wirelength, routability is another important metric in VLSI placement, which determines whether a solution is routable. A placement with low routability will cause flow failure and affect design closure. To this end, we develop a GPU-accelerated detailed-routability-driven placer, to achieve superior solution quality and remarkable runtime speedup.

As for device placement, it is an essential topic in machine learning system (MLSys) and aims at tackling the scalability problem of training a large-scale deep neural network (DNN) on a distributed system. Meanwhile, various network topologies emerge today for DNN training acceleration, which greatly challenges the existing device placement heuristics. We propose an efficient device placement framework to maximize the throughput of training a large DNN onto a multi-GPU system with general device topology. We show that our framework speeds up the pipeline training throughput on systems with different GPU topologies and is able to handle the placement problem for heterogeneously interconnected architectures.

摘要

佈局在人工智能、電子設計自動化、資源調度等各個研究領域中都是一個基礎且具有挑戰性的組合優化問題。隨著GPU計算能力的快速發展，GPU加速成為並行處理大規模問題的重要研究方向。受到這一新興趨勢的推動，我們發掘GPU在佈局方面的潛能，研究了佈局在與GPU相遇時的兩個關鍵問題：GPU加速的VLSI佈局和面向GPU的設備佈局。

VLSI佈局是電子設計自動化（EDA）中極其復雜的問題，其與電路的功耗、性能和面積（PPA）高度相關。與此同時，現代芯片中龐大的晶體管數量極大地增加了VLSI佈局的計算復雜度。為了滿足現如今對一個高效且可擴展的VLSI佈局器不斷增長的需求，我們提出了一個GPU加速的VLSI佈局框架，該框架可以以極快的速度生成高質量的佈局方案。我們還通過神經網絡增强擴展了所提出的框架，説明了將神經網絡組件用於解析性佈局的可能性。除了線長以外，可佈線性是VLSI佈局中另一個重要指標，它決定了佈局方案是否可以完成佈線。可佈線性較低的佈局會導致設計流程失敗且影響設計收斂。為此，我們開發了GPU加速的面向詳細佈線性的佈局器，其實現了卓越的佈局效果並且顯著提高了運行效率。

至於設備佈局，它是機器學習係統（MLSys）中的一個重要課題。它旨在解決在分佈式係統上訓練大規模深層神經網絡（DNN）時面臨的可擴展性問題。同時，如今出現了各種用於加速DNN訓練的網絡拓撲，這對現有的設備佈局啓發式算法提出了很大的挑戰。 對此，我們提出一個高效的設備佈局框架，以最大化在具有通用設備拓撲的多GPU係統上訓練大規模DNN時的吞吐量。我們的框架不僅可以提高在不同GPU拓撲下的流水線訓練吞吐量，而且能夠處理在異構互聯結構下的設備佈局問題。

Acknowledgments

First of all, I would like to express my sincere gratitude and appreciation to my supervisor, Prof. Evangeline F.Y. Young for her insightful guidance and kind support throughout my Ph.D. journey. I deeply appreciate every meaningful discussion we have had about research and life. I am forever grateful for her warm advice and continuous encouragement.

Besides my supervisor, I also want to express my thanks to the rest of my thesis committee members, Prof. Zili Shao, Prof. Bei Yu and Prof. Mark Po-Hung Lin, for their valuable comments and constructive suggestions.

My sincere thanks also go to all of my colleagues: Dr. Jordan Chak-Wa Pui, Dr. Haocheng Li, Dr. Yuzhe Ma, Dr. Haoyu Yang, Dr. Jingsong Chen, Dr. Bentian Jiang, Dr. Jinwei Liu, Dr. Xiaopeng Zhang, Dr. Dan Zheng, Fangzhou Wang, Wei Li, Zhuolun He, Shiju Lin, Xinshi Zang, Bangqi Fu, Tianji Liu, Qijing Wang, Wing Ho Lau, Qin Luo and Yang Sun. It was a great pleasure working with them at The Chinese University of Hong Kong.

Finally, this thesis would not exist without the support and sacrifice of my family. I am deeply grateful to my parents Feiwu Liu and Yixia Lin for their unconditional love. No matter what challenges I have encountered in life, they are always there for me, offering their best support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Graphics Processing Unit

The *graphics processing unit* (GPU) has a rich history. Originally conceived to enable the visualization of images and the simulation of real-world scenarios within virtual environments, GPUs have undergone remarkable advancements over the decades. This growth is depicted in Figure 1.1, showing the exponential increase in transistor count. As a result, the continuous evolution of GPUs has yielded numerous benefits, including enhanced gaming experiences, expanded range of desktop applications, and significant contributions to the progress of various industries. Today, GPUs have transformed into an essential component of the digital world, occupying an ever more vital role in modern life.

As computing power continues to advance, GPUs have emerged with a superior performance compared to central processing units (CPUs) in certain specialized computing domains. Modern GPUs are characterized by high parallelism, making them particularly well-suited for computationally intensive tasks, especially those that can be effectively divided into multiple independent subtasks. In scientific computing with tasks involving vector operations, matrix multiplication and fast Fourier transform, GPUs exhibit significantly faster computation speeds compared to CPUs. Besides the continuous improvement in computing power, the GPU software ecosystem has also flourished spontaneously. A wide

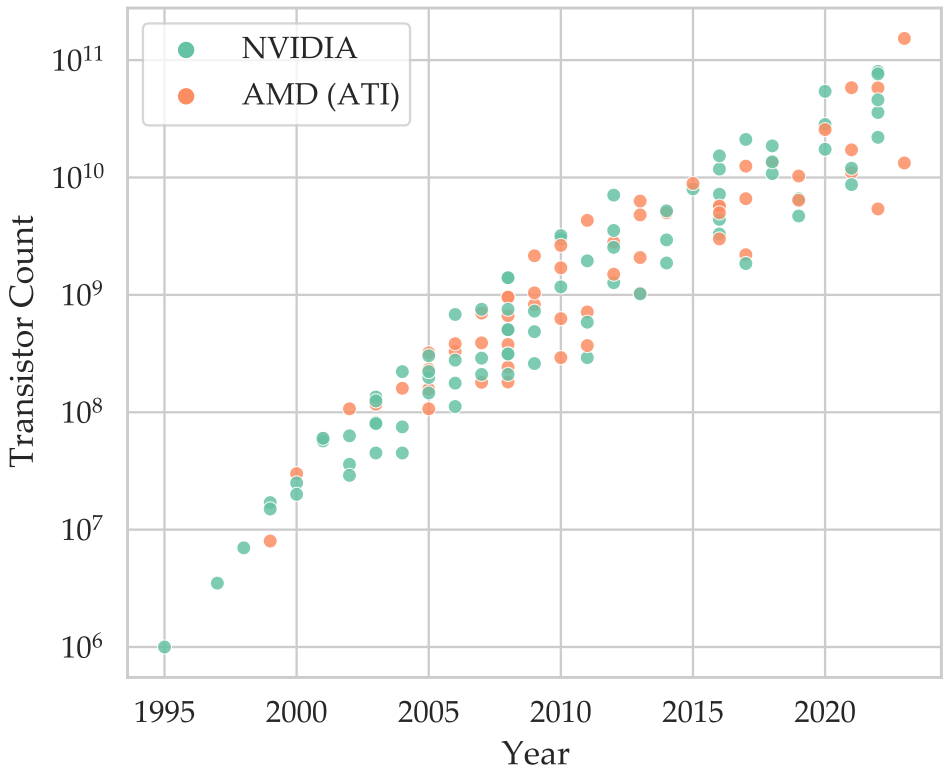


Figure 1.1: Transistor count on NVIDIA and AMD (ATI) GPUs. Data from Wikipedia [123].

range of GPU programming languages and libraries [117, 100, 104, 1] have been developed, offering optimized implementations of GPU-friendly algorithms and effectively leveraging the potential of GPUs.

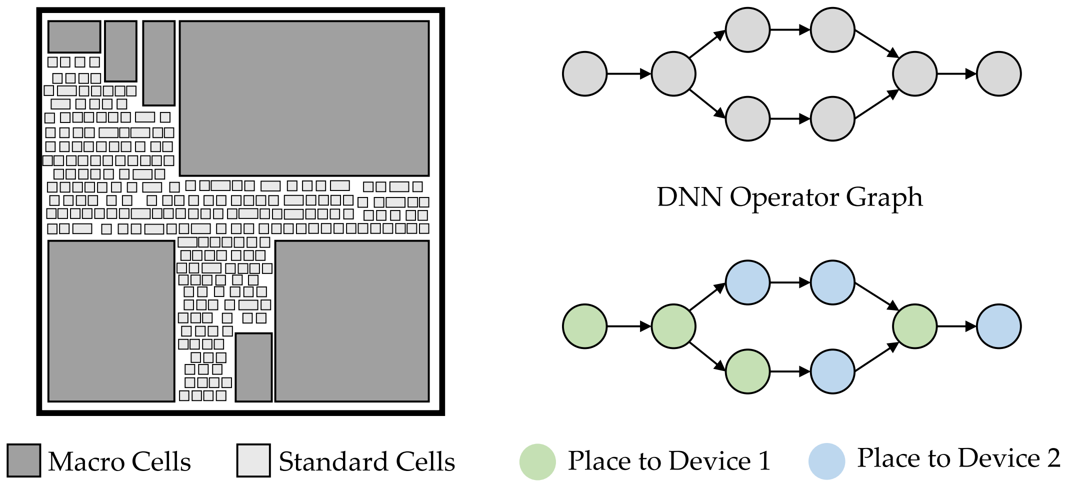
The continuous advancement of computing power and software ecosystems has facilitated the widespread adoption of GPUs across diverse fields. In addition to their traditional applications in graphics and gaming, modern GPUs have found utility in scientific computing, cryptocurrency, artificial intelligence (AI), and electronic design automation (EDA), among others. In artificial intelligence, a significant advancement of GPU was made by [106], which pioneered the adoption of GPU acceleration in deep learning. This pioneering effort resulted in a remarkable increase in the speed of training deep neural networks (DNNs). Subsequently, utilizing GPUs for DNN training acceleration became increasingly prevalent, greatly propelling the progress of AI [37, 21, 27]. With the expansion of data sizes and the increasing complexity of models, a growing number of researchers have turned their attention to optimizing machine learning systems (MLSys) for training large-scale DNNs, leading to the development of various algorithms that are specifically designed to address the challenges associated with multi-GPU training [107, 113, 51, 136]. Within the EDA domain, GPUs have also played a crucial role. Recently, GPU acceleration was introduced to tackle the very large-scale integration (VLSI) placement problem with impressive success in terms of runtime speedup [19, 80]. Besides this achievement, GPU acceleration quickly gained widespread adoption and found extensive application in other critical EDA problems. These problems encompass areas such as logic synthesis [75], placement [29, 34], timing analysis [33, 30, 31], routing [32, 77, 76] and design rule checking [41], where GPU acceleration shows high effectiveness and efficiency.

1.2 Placement Problem

Placement is a critical yet challenging combinatorial optimization problem. Its core objective is to effectively place various components within a given space, optimizing specific objectives while satisfying predefined constraints. Because of its direct correlation to solution quality, placement has been extensively studied in various research domains, including but not limited to VLSI placement in EDA and device placement in MLSys. Figure 1.2 illustrates these two critical placement problems.

1.2.1 VLSI Placement in EDA

The rapid growth in the number of transistors in VLSI design has led to a continuous increase in the computing power of modern chips. VLSI circuit, being integral to various aspects of life, such as mobile phones, personal computers, cars and more, significantly enhances the quality of modern living. However, the process of designing VLSI circuits is extremely complicated. Within the design flow, electronic design automation (EDA) stands out as one of the most challenging parts, which involves register-transfer level generation, logic synthesis, placement, routing and verification [54]. These intricate steps are vital for ensuring successful fabrication and manufacturing of VLSI circuits.



(a) VLSI Placement in EDA (b) Device Placement in MLSys

Figure 1.2: Illustration for VLSI placement and device placement.

Among the above steps in EDA flow, *VLSI Placement* holds an essential and foundational role due to the strong correlation between the placement solution quality and the circuit's PPA (power, performance and area). Meanwhile, modern circuits contain millions of standard cells, which highly increases the computational complexity of the placement problem and brings huge challenges to the leading-edge global placers.

To address the aforementioned challenge, VLSI placement is commonly decoupled into two consecutive phases: global placement and detailed placement. During the global placement (GP) phase, the primary goal is to minimize the interconnect wirelength between standard cells while guaranteeing an even distribution of cells. In addition, various objectives such as timing [34], routability [17] and power [74] are considered to further optimize the circuit's PPA. In the detailed placement (DP) phase, a global-placed solution is first legalized, satisfying the non-overlapping constraints while minimizing cell displacement. The DP phase then further optimizes the placement objectives with the solution legality maintained.

1.2.2 Device placement in MLSys

In recent years, the importance of machine learning systems (MLSys) has grown continuously, driven by the remarkable success of DNNs in artificial intelligence. The ever-increasing

DNN model size [37, 21, 85, 27] significantly stimulates the development of distributive learning scheme in pursuit of more efficient large-scale DNN training [51, 107, 7]. In response to such escalating demands, both researchers and industrial practitioners have been actively exploring dedicated model parallel schemes, with the objectives of improving (1) model throughput over the given device instances; (2) flow effectiveness in terms of network and device topology coverage; and (3) turnaround efficacy for searching a desirable parallelism strategy.

However, designing high-performance distribution strategies for different neural network architectures over different device topologies is non-trivial and challenging, because of the astronomically large search space brought by the growing model size and complex device interconnects. Various paradigms, including but not limited to data parallelism [26, 63, 139], model parallelism [18, 50, 113] and pipeline parallelism [49, 67, 68, 97, 23, 96], have been extensively studied to enable decent parallel execution.

As one of the most prevailing techniques, pipeline parallelism mainly covers two types of optimization problems, pipeline scheduling and device placement. Pipeline Scheduling schedules the pipelined stages for computation and communication considering other scheduling factors like pipeline flush, weight buffering and update mechanisms, etc., with the objective of finding the best tradeoff among device-utilization, memory footprint and training convergence (for asynchronous training).

Orthogonal to pipeline scheduling, *device placement* places thousands or even millions of DNN operators on a large-scale distributed system to maximize the training throughput with consideration of the interconnected bandwidth and peak memory constraint. Due to the extremely high computational complexity, device placement for a multi-GPU system is usually decoupled into two consecutive stages: (1) Model Partitioning: model parameters and associated activations are partitioned into a set of stages, where the workload of each stage is deployed on an individual device. This step aims to balance the workloads among all stages as well as to maximize the overall throughput. (2) Device Mapping: partitioned stages are physically placed on the devices with consideration of the hardware constraints, network topology and communication bandwidths. This step affects network traffic and computing resource utilization which turns out to be crucial for large-scale DNN training.

1.3 Overview of this Thesis

In this thesis, we study and investigate two important problems relating to placement and GPU: GPU-accelerated VLSI placement in EDA and device placement for GPUs in MLSys.

Chapter 2 conducts a literature review on the placement problem, including VLSI placement and device placement.

For GPU-accelerated VLSI placement, we first propose an extremely fast and extensible VLSI placement framework with GPU acceleration in Chapter 3. We also integrate a novel Fourier neural network into the proposed framework as an extension to enhance the solution quality. Chapter 4 further enables our framework to handle the detailed-routability-driven VLSI placement problem by effective GPU parallelization schemes and routability optimization techniques.

Regarding device placement for GPUs, Chapter 5 introduces an efficient design framework that effectively addresses the problem. We demonstrate the effectiveness of our proposed approach in maximizing DNN training throughput on a general device topology. Furthermore, we explore the potential for migrating the algorithm to a system with heterogeneous interconnection.

In summary, this thesis explores the potential of GPU in placement and proposes a set of algorithms to address two fundamental yet challenging combinatorial problems when placement meets GPU.

Chapter 2

Literature Review

In this chapter, we will review the literature on placement-related problems, including VLSI placement in EDA and device placement in MLSys.

2.1 VLSI Placement in EDA

VLSI Placement plays a crucial role in the EDA flow, as it has a significant impact on the overall quality of the circuit's PPA. Modern VLSI placers commonly employ analytical models to optimize the placement solution with consideration of various objectives such as wirelength, density and routability. Due to the extremely high computational complexity of the placement problem, GPU acceleration for VLSI placement has achieved notable success by leveraging the power of GPUs.

In Section 2.1.1, we will look into various analytical placers. In Section 2.1.2, we will review routability optimization techniques for placement. In Section 2.1.3. we will study recent advancement in GPU-accelerated placement.

2.1.1 VLSI Placement

The VLSI placement problem is a classical problem and has been vigorously studied. Various methods, such as cluster-growth [111], simulated annealing [112, 119, 94] and min-cut [5, 109, 120], have been proposed to tackle it. With the growing computational complexity of the placement problem, analytical approach has emerged as a widely adopted solution today. Over the past few decades, various kinds of CPU-based analytical placers have been introduced. These analytical placers aim to optimize the half-perimeter wirelength (HPWL) of the placement, which is essential for achieving efficient VLSI circuits. Based on their formulations of the objectives and constraints, these placers can be broadly classified into two types: quadratic placers and non-linear placers.

Quadratic VLSI Placement: Quadratic placers like GORDIAN [62] employed clique expansion to represent a net (a hypergraph), formulated the placement as a quadratic programming (QP) problem, and solved it using the well-known conjugate gradient method [42]. Eisenmann et al. [22] introduced the concept of cell spreading force, which was computed using Poisson's equation. FAR [47] inserted fixed points in the placement region to guide cells towards low-density areas. FDP [56] implemented a dynamic force weight to accelerate convergence.

To reduce the high computational complexity brought by the clique model, FastPlace [124] proposed a hybrid net model that combines clique models for two-pin and three-pin nets with star models for high-pin nets. This hybrid model effectively reduces the number of non-zero elements in the cell adjacency matrix, resulting in faster solving time. To evenly distribute cells and balance their distribution, FastPlace adopted cell shifting and introduced pseudo nets to connect on-boundary pseudo pins with real cells. Building upon FastPlace, RQL [125] supported on-die pseudo pins, offering greater flexibility. To provide a better approximation of half-perimeter wirelength (HPWL), Kraftwerk2 [116] studied the limitation of the clique-based net model and proposed Bound2Bound (B2B) net model, which considers only the boundary pins in the quadratic cost. For cell overlap control, Kraftwerk2 modeled cell density by an area supply and cell demand system, formulating it as Poisson's equation to achieve uniform cell distribution. Besides Poisson's equation-based method, DPlace [91] proposed a diffusion-based placement framework aimed at effectively distributing cells in a smooth manner.

SimPL [58, 60] developed an iterative lower-upper-bound strategy, with the lower-bound round solving the quadratic wirelength problem with pseudo-net-based spreading force and the upper-bound round incorporating look-ahead rough legalization (LAL) to address cell overlap while maintaining relative cell positions. The legalized positions serve as anchors in the next iteration, guiding the optimization process. ComPLx [59] generalized SimPL by primal-dual Lagrange relaxation and provided convergence analysis. Based on the lower-upper-bound framework, POLAR [78, 79] proposed an efficient rough legalization method based on region expansion and investigated the possibility of developing an effective CPU-parallel quadratic placer.

Non-Linear VLSI Placement: Although quadratic placers can converge quickly, their solution qualities are limited by the low modeling order of the wirelength. Non-linear placers approximate wirelength by a smooth version of the HPWL metric and relax the cell density constraint as a penalty term. Different from quadratic placers, non-linear placers produce higher solution quality while the running time overhead is huge.

The log-sum-exp (LSE) function is widely used as a non-linear approximation for the HPWL. The placer mPL6 [12, 13] employed the LSE function as its wirelength model and formulated the density control problem by a Helmholtz equation. The resulting unconstrained problem is solved by the explicit Euler method. Besides, mPL6 incorporated the multi-level optimization technique to tackle the scalability challenges. APlace [52, 53] utilized a differentiable bell-shaped function to model cell density and formulated a quadratic penalty term for this density function. The problem is then solved by the conjugate gradient (CG) method, which incorporates a line search. Similar to APlace, NTUplace3 [16] also applied a bell-shaped function to model cell density and solved the problem by CG. However, NTUplace3 incorporated the CG method with a dynamic step-size control for runtime speedup and introduced Gaussian smoothing for large macro. Furthermore, NTUplace3 proposed white-space allocation to address cell density overflow.

Compared to the LSE model, the weighted-average (WA) wirelength model [45, 44] offers several advantages. In theory, the WA model has a smaller estimation error of HPWL, and empirical studies have shown better solution quality. As a result, the WA wirelength model has gained considerable prominence in recent years.

Lu et al. [88, 89, 90] introduced a novel approach called ePlace, which utilizes an electrostatics-based method to solve the placement problem. This method had shown significant improvements in quality compared to previous analytical placers. Concretely, ePlace incorporated the WA wirelength cost and models the cell density problem as an electrostatic system (formulated as Poisson's equation). In this electrostatic model, cells are treated as charges and are evenly distributed using electric force. To effectively solve the whole non-linear problem, ePlace applied the Nesterov's method [99] and employed dynamic steplength. Additionally, ePlace utilized discrete cosine [6, 114] transformation to accelerate density computation. Building upon ePlace, Pplace [137] extended the computation of electric potential to an infinite series, providing a convergence proof. Another closely related placer, RePlAce [17] dynamically adjusted the density penalty function based on historical information, resulting in a further boost to the solution quality.

Besides the WA wirelength and electrostatics-based methods, recent research has explored various techniques to further enhance the quality of non-linear placers. GAML [138] utilized the augmented Lagrangian method to tackle the non-linear global placement problem, accompanied by a theoretical convergence guarantee. BiG [118] investigated the feasibility of a bivariate wirelength model, which has low approximation error and high numerical stability. The work [15] proposed a parameter-free quadratic-programming-based initialization method. This method focused on generating better initialization for non-linear placement, leading to improved optimization results and overall solution quality compared with using random initialization.

Machine-learning-based approaches have also emerged as valuable techniques in assisting the VLSI placement step. Mirhoseini et al. [92] developed a method based on graph neural network (GNN) for the macro placement problem and adopted reinforcement learning to optimize the circuit's PPA. The work [3] introduced reinforcement learning for parameter tuning in the placement process. PL-GNN [87] employed GNN to cluster cells and to guide the optimization of a commercial placer. AutoDMP [4] utilized Pareto optimization to explore the placement parameter space.

2.1.2 Routability Optimization in VLSI Placement

Besides the commonly used metric of HPWL, routability is another crucial metric to measure the quality of a placement solution. Effectively estimating and reducing routing congestion are key challenges in routability-driven placement.

Congestion estimation methods can be broadly categorized into probabilistic-based, machine-learning-based and router-based approaches. Probabilistic-based methods [86, 129, 9, 46, 39, 115] decompose multi-pin nets into two-pin nets and enumerate all possible detour-free routing solutions. By computing a probabilistic routing usage map, the placement congestion optimization can be guided. Machine-learning-based methods [130, 134, 83, 25, 14] prevail recently. They utilize deep neural networks to predict the routing congestion map and guide routability optimization. Compared to probabilistic-based and machine-learning-based methods, router-based methods provide higher congestion estimation accuracy and better solution quality. They invoke an internal or external global router to produce a real routing solution, offering valuable congestion information for the placers. Due to its high accuracy, router-based methods are widely applied in routability-driven placers to estimate routing congestion [102, 57, 38, 20, 48, 17].

To reduce routing congestion, there are different techniques including white-space allocation, local density adjustment, cell swapping and cell inflation. The works [133, 65] allocated white space to congested areas so that routing congestion can be alleviated. APlace [53] and NTUplace4h [46] adjusted the local density penalty to push cells away from congested areas. IPR [102] and RippleDP [66] applied cell swapping for congestion removal in the detailed placement stage while CROP [135] and Starfish [127] adopted this technique in the post-placement stage. Cell inflation [43] is a technique that inflates cells in congested areas using historical information, aiming to address the pin-accessibility problem. Due to its simplicity and effectiveness, this technique is widely adopted in routability-driven placers to reduce routing congestion [39, 57, 38, 20, 17, 48, 73, 72].

Modern routability-driven placers such as SimPLR [57], Ripple 2.0 [38], Eh?Place [20], NTUplace4dr [48] and RePlAce [17], invoked router-based congestion estimation methods and incorporated multiple congestion removal techniques mentioned above to effectively enhance routability. These placers demonstrated their effectiveness in achieving a co-optimization between HPWL and routability. It is worth noting that SimPLR, Ripple 2.0 and RePlAce only verified their solution with an academic global router NCTUgr [84] and lacked a detailed-routability evaluation. Eh?Place and NTUplace4dr handled detailed-routability by customized schemes that invoke a computationally intensive global router during global placement, largely affecting its runtime efficiency.

2.1.3 GPU-Accelerated VLSI Placement

With the rapid development of GPU's computational power, GPU acceleration becomes an important direction to pursue to handle large-scale problems with parallelism. In global placement, the work [19] accelerated multi-level analytical placer mPL6[12] with GPU by parallelizing the computation of the wirelength function and the spreading force, achieving around 15$\times$ speedup. The work [71] explored the idea of utilizing sparse matrix multiplications to compute wirelength and adopting a flattening technique for area computation. However, the maximum wirelength degradation in [19] is larger than 5% and the work [71] does not report their solution quality in details.

Recently, DREAMPlace [80] implemented the approach of ePlace [90] on GPU by casting the placement problem as a neural network training problem and demonstrated the superiority of GPU-accelerated global placers. DREAMPlace conducted a study on GPU-accelerated wirelength and density operators, employing various parallelization algorithms. In terms of wirelength computation acceleration, DREAMPlace explored both pin-level and net-level parallelization techniques and observed that net-level parallelization yielded a higher speedup. To parallelize the cell density operator, DREAMPlace employed multi-threading to compute the density contribution of each cell. Additionally, DREAMPlace utilized the fast Fourier transform (FFT) provided by PyTorch [104] to compute the electrostatics-based density force. In comparison to RePlAce [17], DREAMPlace not only produced more than 40x runtime speedup on average for large benchmarks but also provided an open-source analytical placement framework for researchers to further develop. As for routability optimizatin, DREAMPlace integrated NCTUgr for routability evaluation and employed cell inflation to mitigate routing congestion. Although DREAMPlace achieves the state-of-the-art solution quality and performance, it primarily focused on accelerating the wirelength and density operators on GPU while lacking a more general operator-level optimization.

Expanding upon DREAMPlace, the work [35] proposed an effective density accumulation method using parallel partial summation. Gu et al. [29] introduced a multi-electrostatics system to tackle the fence region constraint and implemented a quadratic density penalty to accelerate convergence. Liao et al. [70] developed a net weighting technique that employed a momentum method to tackle the timing-driven placement problem. Guo et al. [34] derived the gradient of the Elmore delay model and integrated it into the placement stage to make the timing-driven global placement objective differentiable. Yang et al. [132] explored the acceleration of the legalization step on CPU/GPU heterogeneous platforms through task scheduling. Additionally, ABCDPlace [81] conducted an investigation to boost the runtime efficiency of detailed placement by leveraging GPU parallelization. It explored batch-level parallelism for traditional sequential detailed placement operators such as independent set matching, global swapping and local reordering, achieving around 10x speedup through GPU acceleration. In the case of independent set matching, ABCDPlace split cells into smaller subsets and concurrently solved the matching problem for each subset. For global swapping, ABCDPlace parallelized the candidate collection and cost calculation, which led to faster runtimes. In terms of local reordering, ABCDPlace introduced parallel sliding windows to decompose the reordering problem into independent sub-problems and solved them in parallel.

2.2 Device Placement in MLSys

The impressive achievements of DNNs have led to a significant increase in the importance of machine learning systems (MLSys) over the past few years. In MLSys, DNN training parallelism has emerged as an important research direction, as it plays a critical role in optimizing training efficiency. Recently, various parallelism strategies and optimization techniques are proposed to achieve effective parallelization.

In Section 2.2.1, we will begin by reviewing two classical parallelism schemes, namely data parallelism and model parallelism. In Section 2.2.2, we will study the widely adopted pipeline parallelism. In Section 2.2.3, we will review the device placement problem.

2.2.1 Data and Model Parallelism

Data parallelism partitions training data at the batch level and enables model training to scale up to a distributed system [26, 63, 139]. Each device maintains a *replica* of the entire model and computes gradient synchronously by a technique called *allreduce* [105]. The computed gradient is then applied to update the model parameters. However, for a large model, pure data parallelism is infeasible because of the insufficient device memory to support model training or the large communication overhead caused by gradient synchronization.

Model parallelism is an alternative option to address the issue that the memory of a device is insufficient to maintain a DNN. By partitioning the DNN into multiple disjoint sets and distributing them across different devices, model parallelism enables the training of large models. However, traditional model parallelism approaches, as discussed in [18], suffered from limited resource utilization as they activate only one device at a time. Megatron-LM [113] explored tensor-level model parallelism to partition Transformer-based models to boost computational efficiency while requiring heavy-load collective communication for synchronization. FlexFlow [50] integrated tensor-level model parallelism with data parallelism and randomly searched the parallelization strategies according to the simulated throughput.

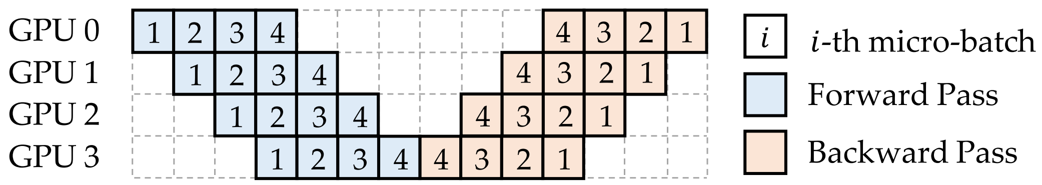


Figure 2.1: An example of GPipe's fashion pipeline parallelism with four partitioned stages and four micro-batches.

2.2.2 Pipeline Parallelism

Pipeline Parallelism aims at scheduling DNN training more elaborately to boost resource utilization. Extending from model parallelism, pipeline parallelism not only partitions a DNN to different devices but also divides a mini-batch of training data into several micro-batches. In pipeline parallelism, the split micro-batches and the partitioned DNN will be delicately scheduled, resulting in an increase in throughput.

One notable technique is GPipe [49] as shown in Figure 2.1. GPipe split the mini-batch into micro-batches and performs synchronous scheduling of forward and backward propagation during training. Although GPipe effectively increases training throughput, it requires additional device memory. DAPPLE [23] introduced a hybrid parallelism strategy (i.e., pipeline training combined with data parallelism) with an one-forward-one-backward (1F1B) synchronous pipeline. This technique reduces the peak memory consumption compared to the GPipe scheme. HetPipe [103] extended support for the hybrid parallelism strategy on heterogeneous GPU clusters. PipeDream [96] generalized the pipeline training in an asynchronous fashion and reduces the idling time (bubble). However, the asynchronous pipeline requires maintaining stale versions of weights for backward propagation, which can lead to accuracy degradation. PipeDream-2BW [97] proposed a double-buffered update technique to reduce the number of weight versions and save device memory usage. PipeMare [131] mitigated the accuracy loss through learning rate rescheduling and weight discrepancy correction. Chimera [67] further extended the synchronous pipeline to a bidirectional pipeline and achieves impressive throughput. TeraPipe [68] explored token-level pipeline parallelism specifically designed for Transformer-based language models and investigated a dynamic programming-based algorithm to search for the optimal execution flow. Additionally, the work in [98] extended Megatron-LM to support pipeline parallelism and introduced an interleaved 1F1B pipeline to boost utilization. Despite of the significant improvements in training efficiency achieved by these pipeline schedulers, they still lack a robust and effective device placement algorithm to achieve good DNN training throughput on general device topology.

2.2.3 Device Placement for GPUs

Orthogonal to pipeline scheduling, device placement is another important factor to maximize the training throughput. Its goal is to optimize the placement of DNN operators on a distributed GPU system. With modern DNNs consisting of millions of operators and distributed systems having diverse interconnected bandwidths, the problem complexity significantly increases.

One approach to solving this problem is the application of machine learning-based methods. The works [93, 24, 101] employed reinforcement learning to place DNNs on GPUs, using the resulting throughput as a training reward. However, these machine-learning-based methods often require time-consuming online throughput measurements, which can largely impact the overall runtime.

To tackle the growing problem complexity and achieve efficient yet practical device placement, recent advanced device placement approaches are usually divided into two consecutive stages: model partitioning and device mapping. These two-stage methods balance DNN partitions and properly map the partitions to devices.

Model Partitioning For model partitioning, PipeDream [96] employed a dynamic programming based (DP-based) method. The primary objective of the dynamic programming algorithm employed by PipeDream is to minimize the maximum stage time, aiming to achieve effective pipeline scheduling and maximize device utilization. This algorithm considers computation and communication time to balance the stage times. Due to its effectiveness, this DP-based approach has gained popularity. DAPPLE [23] extended the dynamic programming to consider the warmup and allreduce operations in pipeline training. Piper [122] supported a wider range of connection types in the computational graphs.

However, PipeDream, DAPPLE and Piper primarily focused on coarse-grained layer-level granularity, limiting their flexibility. In contrast, RaNNC [121] proposed a three-phase flow for operator-level graph partitioning. In the first phase, RaNNC atomically merges operators that are not descendants of the model input to reduce the size of the operator-level graph. In the second phase, computation blocks are constructed through k-way multi-level partitioning to balance the computation and communication costs. Finally, in the third phase, partitions are formed using dynamic programming, considering computation time, communication cost and memory usage. Alpa [136] further considered intra-operator parallelism through an integer linear programming approach. Different from RaNNC, Alpa clustered atomic operators using dynamic programming with consideration of the operator computation cost and connectivity. However, the model partitioning methods employed by both RaNNC and Alpa only focused on a specific topological order of a DNN, which may affect adversely the partitioning quality.

Device Mapping As for device mapping, RaNNC [121] assumed a flattened device topology of constant bandwidth, neglecting more complex topologies. However, the flattened topology is not practical in the real world since it is difficult to maintain constant bandwidth in a large-scale GPU cluster.

Instead of assuming constant bandwidth, DAPPLE [23] explored the compositions of three predefined policies (fresh, append and scatter) to generate final placement on a system with hierarchical interconnection while lacking a guarantee of solution quality. PipeDream [96] and Piper [122] addressed the hierarchical topology mapping problem by leveraging dynamic programming, where the planning at level L considers placement on K_L "superdevices" (each contains K_{L-1} "superdevices" from level $L-1$) according to the topology at level L. Alpa [136] generalized the dynamic programming algorithm to further consider all possible device combinations in hierarchical architectures. For example, given 2 nodes and 4 GPUs per node, Alpa considered them as 2×4, 1×8, 4×2 or 8×1. While PipeDream, Piper and Alpa integrated partitioning and mapping into a unified dynamic programming algorithm, they only tackled the hierarchical topology mapping problem and did not provide a general mapping algorithm for arbitrary device topologies.

Chapter 3

GPU-Accelerated VLSI Placement

In this chapter, we develop Xplace, an efficient yet extensible GPU-accelerated placement framework built on top of PyTorch [104], to consider factors at operator-level optimization. Xplace not only achieves better performance and quality than DREAMPlace but also shows high extensibility to incorporate neural network into analytical placer. The source code of Xplace is released on GitHub[1]. Our key contributions are summarized as follows.

- Efficiency: with operator combination, operator extraction, operator reduction and operator skipping, Xplace achieves around 3x speedup per GP iteration compared to the state-of-the-art global placer DREAMPlace. A placement-stage-aware parameter scheduling technique is also proposed to improve the solution quality. Experimental results show that Xplace achieves around 2x speedup with better solution quality compared to DREAMPlace.
- Extensiblity: we plug into Xplace a novel Fourier neural network as an extension. The neural network serves as a global guidance for placement. Experimental results not only show that the proposed framework can further improve the solution quality but also illustrates the possibility of adopting neural guidance in analytical global placement.

[1] `https://github.com/cuhk-eda/Xplace`

3.1 Preliminaries

Given a placement circuit $G = (V, E)$, V represents the set of cells and E denotes the set of nets. Let $p = \{(x_1, y_1), ..., (x_N, y_N)\} \in \mathbb{R}^{N\times 2}$ denote the 2D positions of the cells, where N is the number of cells. The placement region R is uniformly split into an $M_r \times M_c$ grid B. The objective of placement is to minimize the total HPWL of all the nets while satisfying the cell density constraint, which is formulated as,

$$\min_p HPWL(p) = \min_p \sum_{e\in E} HPWL_e(p) \tag{3.1a}$$

$$\text{s.t.} \quad D_b \le D_t, \forall b \in B \tag{3.1b}$$

where D_b and D_t denote bin b's cell density and the benchmark-given target density respectively. The HPWL of net e is defined as follows:

$$HPWL_e(p) = (\max_{i\in e} x_i - \min_{i\in e} x_i) + (\max_{i\in e} y_i - \min_{i\in e} y_i). \tag{3.2}$$

Because the HPWL in Equation (3.2) is not differentiable, analytical placement reformulates the objective with a smooth approximation of HPWL equipped with a cell density penalty to relax the cell density constraint:

$$\min_p \sum_{e\in E} WL_e(p) + \lambda D(p) \tag{3.3}$$

where the wirelength $WL_e(p) = WL_e(x) + WL_e(y)$ is modeled as the weighted-average (WA) [45] wirelength with a coefficient γ,

$$WL_e(x) = \frac{\sum_{i\in e} x_i e^{x_i/\gamma}}{\sum_{i\in e} e^{x_i/\gamma}} - \frac{\sum_{i\in e} x_i e^{-x_i/\gamma}}{\sum_{i\in e} e^{-x_i/\gamma}} \tag{3.4}$$

and similarly for $WL_e(y)$. A smaller γ leads to more accurate approximation of HPWL. The parameter λ (Lagrange multiplier) controls the weight of the cell density penalty $D(p)$. A typical placement flow starts with a small λ and gradually increase it to remove cell overlaps.

In ePlace [90] on which Xplace is based, each cell i is modeled as a charge, and the cell density is modeled as an electrostatic system denoted as:

$$\begin{cases} \nabla \cdot \nabla \psi(x,y) = -\rho(x,y), \\ \hat{\mathbf{n}} \cdot \nabla \psi(x,y) = \mathbf{0}, (x,y) \in \partial R, \\ \iint_R \rho(x,y) = \iint_R \psi(x,y) = 0, \end{cases} \tag{3.5}$$

where ∂R is the boundary of the placement region, $\rho(x,y)$ is the electron density map, $\psi(x,y)$ is the electric potential distribution, and $\xi(x,y) = -\nabla\psi(x,y)$ is the electric field distribution. The numerical solution of Poisson's equation in Equation (3.5) is derived by discrete cosine transformation (DCT) in [90].

3.2 Overview of Xplace

In this section, we will discuss the design of Xplace, a fast and extensible GPU-accelerated placer. Our Xplace framework is shown in Figure 3.1. Xplace is built on top of PyTorch and contains a placement core engine. Inside the core engine, the gradient engine takes cell position and placement parameters as input to compute the cell gradient. Next, the optimizer utilizes the computed gradient to update the cell position. The evaluator evaluates the placement solution, and the recorder records the placement metrics like HPWL and overflow. Finally, the scheduler decides how to modify the parameters and whether to stop the global placement. It is worth noting that all these parts are designed as independent modules in Xplace so that one can easily extend Xplace by applying new scheduling techniques, new gradient functions, new placement metrics and so on.

Section 3.3 will discuss several important technical details that enable Xplace running very efficiently on GPU. We propose operator-level optimization techniques to achieve effective parallelization. Section 3.4 will discuss a placement-stage-aware parameters scheduling to improve the solution quality. Section 3.5 will extend Xplace with a Fourier neural operator to show its high extensibility.

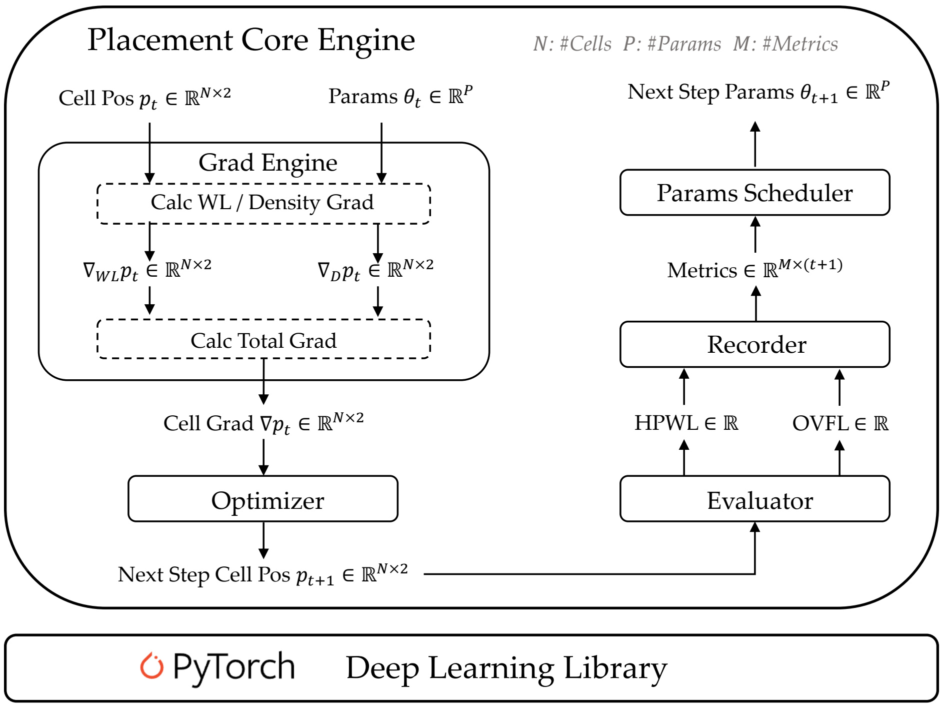

Placement Core Engine
N: #Cells P: #Params M: #Metrics
Cell Pos $p_t \in \mathbb{R}^{N\times 2}$
Params $\theta_t \in \mathbb{R}^P$
Grad Engine
Calc WL / Density Grad
$\nabla_{WL} p_t \in \mathbb{R}^{N\times 2}$
$\nabla_D p_t \in \mathbb{R}^{N\times 2}$
Calc Total Grad
Cell Grad $\nabla p_t \in \mathbb{R}^{N\times 2}$
Optimizer
Next Step Cell Pos $p_{t+1} \in \mathbb{R}^{N\times 2}$
Next Step Params $\theta_{t+1} \in \mathbb{R}^P$
Params Scheduler
Metrics $\in \mathbb{R}^{M\times(t+1)}$
Recorder
HPWL $\in \mathbb{R}$
OVFL $\in \mathbb{R}$
Evaluator
PyTorch
Deep Learning Library

Figure 3.1: Overview of Xplace

3.3 Operator-Level Optimization

3.3.1 Wirelength Operator Combination

Weighted average (WA) [45] wirelength is used as the wirelength objective in many analytical global placers. In Xplace, we adopt the WA wirelength in Equation (3.4) as our wirelength objective and update the cell position based on the guidance of the WA gradient. To avoid numerical overflow, a numerically stable version of the WA wirelength is given in Equation (3.6) which needs the minimum and the maximum position among all pins in a net.

$$WL_e(x) = \frac{\sum_{i\in e} x_i e^{\frac{x_i - \max_{j\in e} x_j}{\gamma}}}{\sum_{i\in e} e^{\frac{x_i - \max_{j\in e} x_j}{\gamma}}} - \frac{\sum_{i\in e} x_i e^{\frac{\min_{j\in e} x_j - x_i}{\gamma}}}{\sum_{i\in e} e^{\frac{\min_{j\in e} x_j - x_i}{\gamma}}} \tag{3.6}$$

A wirelength objective-and-gradient merging method, proposed in [80], computes both the WA wirelength and the WA gradient within a single GPU thread to mitigate memory bounded problems.

Since both the HPWL function in Equation (3.2) and the stable WA wirelength function in Equation (3.6) need the minimum and maximum cell positions in a net, we further modify this merging method by combining the three operators with heavy wirelength-related workload, WA wirelength, WA gradient and HPWL, into one operator to avoid redundant computation of the minimum and maximum function. The proposed operator combination technique can significantly reduce the total GPU execution time.

3.3.2 Density Operator Extraction

Density objective is one of the most computationally intensive operators in global placement. Similar to [80] and [71], we implemented a GPU-accelerated area accumulation operator to compute the cell density map and apply PyTorch built-in rfft2/irfft2 operators to derive the numerical gradient of the electrostatic system in Equation (3.5). We also implemented a GPU-accelerated version of the overflow ratio operator $OVFL$, whose CPU version is applied in NTUplace3 [16] and ePlace [88], to measure the evenness of cell distribution and guide the parameter update. The overflow ratio is described as,

$$OVFL = \frac{\sum_{b \in B} \max(D_b - D_t, 0) A_b}{\sum_{i \in V_{mov}} A_i} \tag{3.7}$$

where A_b and A_i denote the area for bin b and cell i; D_b is bin b's cell density; D_t is the target density, and V_{mov} is the set of movable cells. Concretely, each bin b's cell density D_b in the cell density map $D \in \mathbb{R}^{M_r \times M_c}$ is defined as,

$$D_b = \frac{\sum_{i \in V} A_i \cap A_b}{A_b}, \ \forall b \in B \tag{3.8}$$

where $A_i \cap A_b$ defines the overlap area between cell i and bin b. Similar to [88] and [2], we insert filler cells inside the electrostatic system to handle whitespace and prevent the density objective to overly spread the cells. The inserted filler density map $D_{fl} \in \mathbb{R}^{M_r \times M_c}$ is given as,

$$D_{fl,b} = \frac{\sum_{i \in V_{fl}} A_i \cap A_b}{A_b}, \ \forall b \in B, \tag{3.9}$$

where V_{fl} denotes the set of filler cells. Therefore the total density map $\tilde{D} \in \mathbb{R}^{M_r \times M_c}$ used for solving the electrostatic system is formulated as follows,

$$\begin{aligned} \tilde{D}_b &= \frac{\sum_{i \in V \cup V_{fl}} A_i \cap A_b}{A_b} && \text{(3.10a)} \\ &= \frac{\sum_{i \in V} A_i \cap A_b}{A_b} + \frac{\sum_{i \in V_{fl}} A_i \cap A_b}{A_b} && \text{(3.10b)} \\ &= D_b + D_{fl,b}, \ \forall b \in B && \text{(3.10c)} \end{aligned}$$

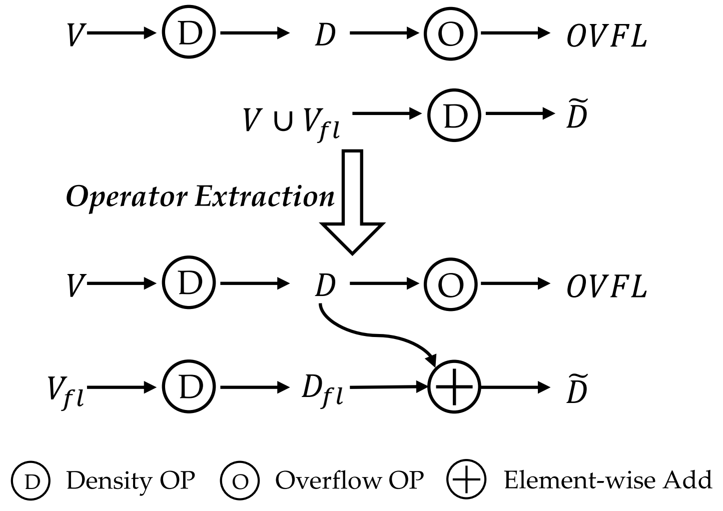
Then, the matrix form of Equation (3.10) is formulated as

$$\tilde{D} = D + D_{fl} \tag{3.11}$$

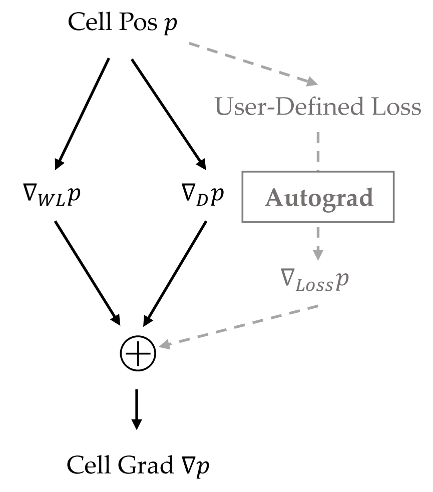
We observe that both Equation (3.8) and Equation (3.11) contain the computation of cell density map D. Due to the heavy load of the density map operator, performing a common sub-operator extraction in Equation (3.11) will naturally boost the performance. As shown in Figure 3.2(a), we first compute the cell density map D and the filler density map D_{fl} separately. We then adopt the element-wise add operator to compute the total density map $\tilde{D}$ and apply the overflow operator to calculate $OVFL$. Note that overflow ratio computation is needed for updating parameters in each GP iteration. The proposed sub-operator extraction technique will reduce the total computation time of the cell density map D and achieve a visible improvement in the total GPU execution time.

3.3.3 Operator Reduction

PyTorch [104] is a well-known deep learning library that provides a lot of built-in differentiable operators (e.g. element-wise addition, matrix multiplication, convolution, etc.). In forward propagation, users can apply the provided/user-defined operators to construct a neural network or a gradient-based optimization. In backward propagation, an automatic



(a) Operator Extraction



(b) Cell Gradient Computation

Figure 3.2: Illustration for the operator extraction technique and the cell gradient computation scheme.

differentiation (autograd) engine is invoked to compute derivatives automatically. Although PyTorch makes development convenient, there are technical details that need to be carefully considered when building a global placer using PyTorch.

In PyTorch, the execution of each operator will perform a kernel launching step on CPU before executing the core CUDA kernel on GPU. Not only that the forward propagation will execute operators but also the backward propagation, driven by the autograd engine, need to run gradient operators. However, the kernel launching overheads of these operators may even be much larger than their GPU execution overheads when their computation workloads are small. Except for the heavily loaded operators (e.g. for wirelength and density computations) that are related to the netlist size and the die area, the kernel launching overheads of most other placement operators are much larger than their GPU execution overheads. In this case, the more operators being executed, the larger the total kernel launching overhead there will be. If the total kernel launching overhead dominates the GPU execution time, the speedup will be limited. To this end, we propose a series of techniques to reduce the number of operators to mitigate the problem.

The first technique is to avoid invoking the heavy autograd engine. Since the number of forward operators are almost the same as that in the backward, invoking the heavy autograd engine will almost double the number of operators and bring large kernel launching overhead on CPU. To resolve this problem, we directly derive the numerical solutions of the wirelength gradient and the density gradient without invoking the autograd engine and assign a weighted accumulated gradient to each cell. This step can reduce the total kernel launching time and boost the performance significantly. It is worth noting that avoiding invoking the autograd engine will not affect our framework's extensibility since PyTorch also supports invoking the autograd engine for user-defined loss function and accumulating the separately computed numerical gradient with the backward gradient of the user defined loss function as illustrated in Figure 3.2(b).

Besides, the PyTorch in-place operators, which directly manipulate on the tensor memory without memory copying, are used as much as possible. This technique will naturally avoid redundant copying.

Finally, as frequent synchronization will interrupt the GPU pipeline and slow down the total run time, we reorder the operators that need synchronization to the end of the execution queue in each GP iteration so that the negative effects of synchronization can be alleviated.

3.3.4 Operator Skipping

It is observed that the ratio between density gradient and wirelength gradient, given as follows,

$$r = \frac{\lambda |\nabla D_{x,y}|}{|\nabla WL_{x,y}|} \tag{3.12}$$

is ultra-small in the early placement stage. Based on the observation, we propose an early-stage operator skipping technique to further boost the runtime performance. When $(r < 0.01) \wedge (iteration < 100)$, the density gradient operator will only be executed once per 20 iterations.

3.3.5 Determinism

Similar to DREAMPlace [80], we convert the floating point numbers to fixed point to avoid non-deterministic floating-point atomic-add operations. To store the converted GPU data, extra memory allocation is needed. We observe that the size of the allocated GPU memory is unchanged in each GP iteration, and dynamical allocation will cause frequent synchronization that interrupts the execution pipeline. Thus we apply a GPU memory pre-allocation technique to avoid redundant synchronization and significantly accelerate the deterministic mode.

3.4 Placement-Stage-Aware Parameters Scheduling

Preconditioning is widely used in global placement [88, 78, 59]. Given in Equation (3.13), preconditioning is applied to the optimization objective for reducing the condition number and accelerating Nesterov's optimization convergence.

$$\mathbf{H}^{-1} = (\mathbf{H_W} + \lambda \mathbf{H_D})^{-1}, \tag{3.13a}$$

$$\mathbf{H_W} = \mathrm{diag}(|S_1|, |S_2|, ..., |S_{|V|}|), \tag{3.13b}$$

$$\mathbf{H_D} = \mathrm{diag}(A_1, A_2, ..., A_{|V|}), \tag{3.13c}$$

where $S_i = \{e\}_i$ is the set of nets containing cell i, $|S_i|$ is the number of nets connecting cell i, A_i is the area of cell i, and λ is the weight of the density penalty. Recall that the placement objective is formulated as,

$$\min_p WL(p) + \lambda D(p).$$

With the preconditioner, the modified cell gradient is derived as follows,

$$\nabla p = (\mathbf{H_W} + \lambda \mathbf{H_D})^{-1}(\nabla WL_{x,y} + \lambda \nabla D_{x,y}), \tag{3.14}$$

where ∇p denotes the cells' gradient.

To measure the placement stage, we introduce the precondition weighted ratio $\omega \in (0, 1)$,

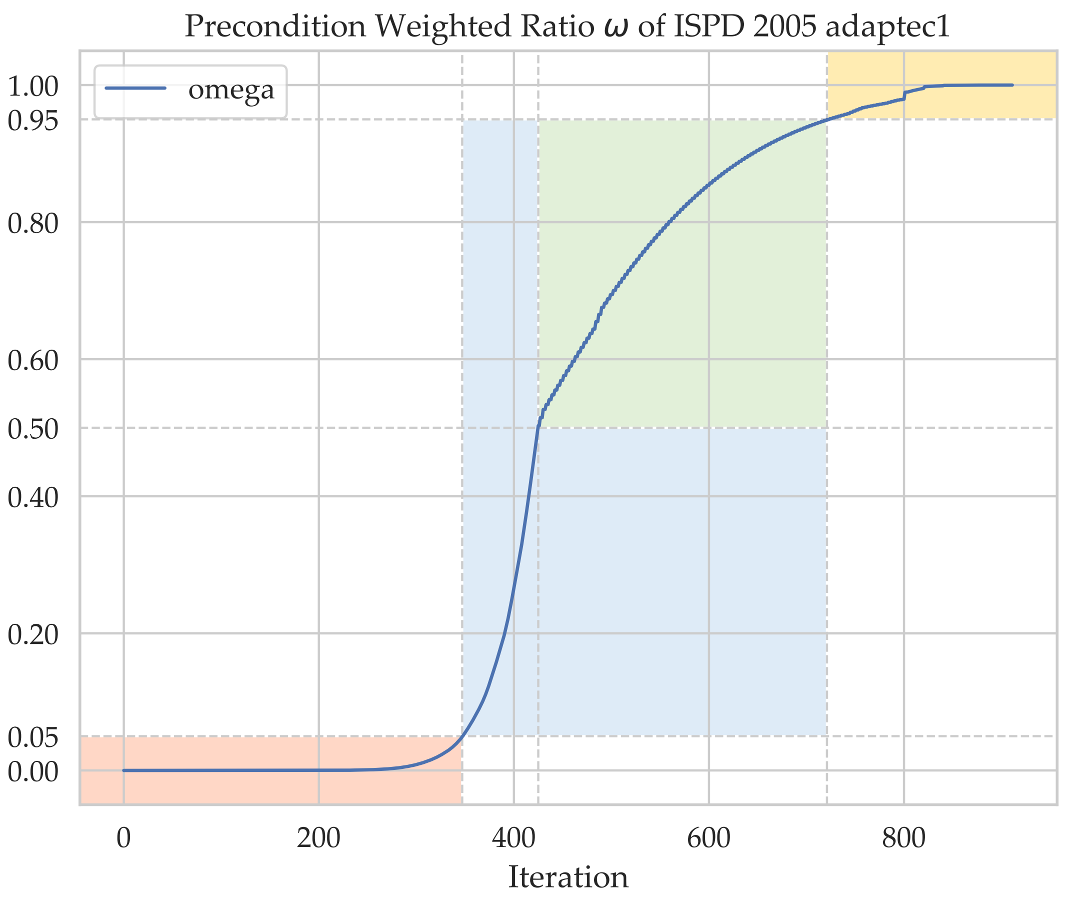


Figure 3.3: An example of the precondition weighted ratio ω of ISPD 2005 `adaptec1`.

$$\omega(\lambda) = \lambda|\mathbf{H_D}||\mathbf{H_W} + \lambda\mathbf{H_D}|^{-1} \tag{3.15a}$$

$$= \frac{\lambda \sum_{i\in V} A_i}{\sum_{i\in V} |S_i| + \lambda \sum_{i\in V} A_i}, \tag{3.15b}$$

where $|\cdot|$ is the L_1-norm operator. Note that ω is only determined by the variable λ (A_i and S_i are only related to the design technology), where λ is gradually updated during placement to enhance the importance of the density objective. Compared to $\frac{\lambda}{1+\lambda}$, the precondition weighted ratio $\omega(\lambda)$ is more relevant to the cell gradient formulated in Equation (3.14) and it further reflects the relative importance of $\lambda\mathbf{H_D}$ in $\mathbf{H}^{-1}$.

Through our investigation, $\omega(\lambda)$ successfully measures the global placement optimization stage. As shown in Figure 3.3, ω gradually increases when the placement iterates.

Algorithm 1 Placement-Stage-Aware Parameters Scheduling

```
1: $\gamma \leftarrow \gamma_0$                                        ▷ wirelength coefficient
2: $\lambda \leftarrow \lambda_0$                                        ▷ density weight
3: while $iteration$ < ITER and NOT Convergence do
4:     if $0.5 < \omega < 0.95$ and $iteration\%3 \neq 0$ then
5:         SKIP_UPDATE
6:     else
7:         $\gamma \leftarrow \gamma \times coef(overflow)$
8:         $\lambda \leftarrow \lambda \times \mu(\Delta hpwl)$              ▷ $\gamma$ and $\lambda$ are derived from [17]
9:     $\omega \leftarrow \frac{\lambda|\mathbf{H_D}|}{|\mathbf{H_W}|+\lambda|\mathbf{H_D}|}$
```

When $\omega < 0.05$ (marked in red), the optimization objective is wirelength-dominated, and cells are driven to the position with minimum wirelength. As ω rapidly grows in the intermediate stage ($0.05 < \omega < 0.95$, marked in blue), cells spread over the whole placement region, and the overlap ratio significantly decreases. At the end of the placement ($\omega > 0.95$, marked in yellow), cells are forced to a final position with minimum local penalty at the final stage. Note that the first-order derivatives of $\omega(\lambda)$ in the red and yellow areas are both small, while that in the blue area is relatively larger.

If we slow down the parameter update in the blue stage, the gradient optimizer will search the solution spaces more fine-grained, and the quality of the final solution will become better. However, a slower update of placement parameters would affect the running time so we only adopt the above slowing update technique when $0.5 < \omega < 0.95$ (marked in green in Figure 3.3) to exploit the optimization space while saving the runtime. A detailed description of this technique is given by Algorithm 1.

3.5 Extending Xplace via Neural Enhancement

In this section, we show Xplace's high extensibility by incorporating a deep neural network into its optimization process. As Xplace has a similar architecture to a normal neural network of PyTorch [104], it is natural and easy to embed a neural network as an extension. Here, we propose a neural-plugged-in framework to explore the possibility of learning-based frameworks. We also demonstarte such extension can further improve Xplace's solution quality.

Neural networks have shown great potential on mapping between dynamic systems defined by partial differential equations (PDE). Previous works of image-to-image mapping tasks are usually conducted in the spatial domain and are accurate on training datasets. However, when given a new pattern of instance, they do not perform well. Recently, convolution operation in the frequency domain is discovered to be more powerful in generalizing dynamic systems determined by a family of partial differential equations [69]. Such Fourier-Neural-Operator (FNO) is capable of learning the universal solution of a dynamic system with only limited training data.

In the global placement problem, the function of the electric field Equation (3.5) solved by Poisson's equation can be modeled as a dynamic system mapping from electron distribution to electric field, in which electron distribution is the 2-D density map of placement and the electric field is the unit moving force on x and y-axis.

As illustrated in Figure 3.4, our model is a two-path convolution network consisting of a spatial-domain path (blue) to extract the explicit information of a specific feature map and a frequency-domain path (orange) to generalize the global information of a continuous dynamic system. In order to transfer the model to multi-resolution, the input density map D is concatenated with mesh-grid index $M_x(x,y) = \frac{x}{X}$ and $M_y(x,y) = \frac{y}{Y}$, where X and Y are map sizes. The input map $I = \{D; M_x; M_y\}$ is firstly lifted to multi-channel by a fully-connected layer, denoted as $I_m = FC(I)$, and then fed into two paths.

In the Fourier path, the spatial map is transformed into the frequency space after Fast-Fourier-Transform (FFT) function $\mathcal{F}$. A low-pass-filter (LPF) L preserves a number of lower frequency components and then a linear transformation $\mathcal{W}$ is applied to the filtered map. After applying an Inverse-Fast-Fourier-Transform (IFFT) function $\mathcal{F}^{-1}$, the frequency map is transformed back to the spatial domain,

$$Freq_{layer}(I_m) = \mathcal{F}^{-1}\Big(\mathcal{W}^T \cdot L(\mathcal{F}(I_m))\Big) \tag{3.16}$$

In the spatial path, a simple pixel-wise convolution layer is operated on the feature map.

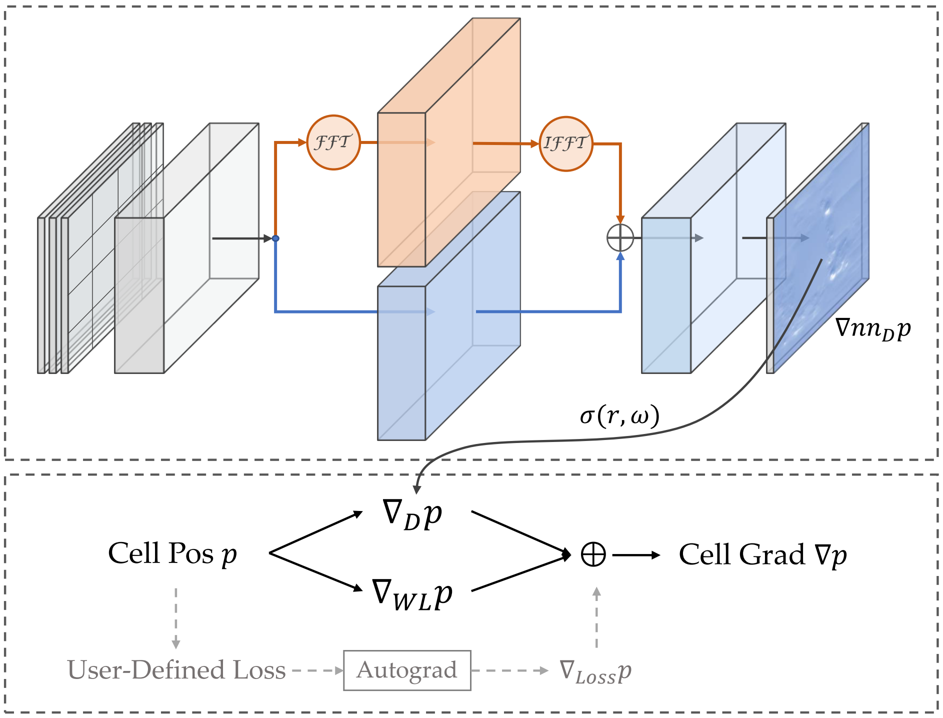


Figure 3.4: Neural-network plugged-in placement extension.

Maps from two paths are added followed with a nonlinear activation function $GELU$. The above process of a FNO is described as

$$\mathcal{O}(I_m) = GELU\Big(Conv_{2D}(I_m) + Freq_{layer}(I_m)\Big) \tag{3.17}$$

We then get the output after a down-sampling fully-connected layer $FC^{-1}\Big(\mathcal{O}(I_m)\Big)$ and a relative L_2 loss function is used for back-propagation:

$$L_2(\mathbf{x_i}, f(\mathbf{x_i};\theta)) = ||f(\mathbf{x_i},\theta) - \mathbf{y_i}||_2/||\mathbf{y_i}||_2 \tag{3.18}$$

where $\mathbf{x}_i, \mathbf{y}_i, f$ and θ are the i-th input, label, network model and network parameters. The label and prediction are normalized for evaluation.

In this model, we do not need to collect the ground-truth training data from real placement benchmarks. Rather, we can generate randomly distributed density maps and compute the numerical solution of the corresponding electric fields which will be used as labels for training.

Since we do a pixel-wise convolution on the spatial maps, the resolution of the input maps will not affect the convolution results. Moreover, in Fourier space, low-frequency components describe the global information, while high-frequency components describe the explicit local information. That means, a low-resolution image and a high-resolution image will share similar low-frequency components, with differences in high-frequency components only. As we only preserve a certain number of lower-frequency components, our model is resolution-independent. The model can be trained on low-resolution data and extended to high-resolution, which improves the adaptability of the model.

The electric fields on both the x and y directions share the same partial differential function, with only a difference in the direction. Therefore the model can be trained in only one direction and still be workable in the other direction by simply flipping the input feature map, further improving the generalization of this model.

As the model is trained on low-resolution data and only preserves lower-frequency components, the predicted density gradient will have a good global view to spread cells. Thus we insert the nn-predicted density gradient $\nabla_{nn} D_{x,y}$ into the early stage of placement to help push cells around. With the precondition weighted ratio ω defined in Section 3.4 and the gradient ratio r defined in Section 3.3.4, a smooth function $\sigma(r, \omega)$ is used to weighted-sum with the numerical solution of the density gradient, given as follows,

$$\sigma(r, \omega) = \frac{1}{1 + e^{-5(r/0.005-0.5)}} - \frac{1}{1 + e^{-5(\omega/0.05-0.5)}} \tag{3.19}$$

$$\nabla' D_{x,y} = (1 - \sigma)\nabla D_{x,y} + \sigma \nabla_{nn} D_{x,y} \tag{3.20}$$

Function σ has a bell shape, and it smoothly integrates the nn-predicted gradient with the numerical solution. When σ increases, $\nabla_{nn} D_{x,y}$ will dominate. When σ drops, $\nabla D_{x,y}$ takes effect for fine-grained placement. Note that directly introducing the external nn-predicted gradient would cause the divergence in Nesterov's optimization but using the above smoothness successfully avoids that and helps the convergence.

3.6 Experimental Results

The Xplace framework is developed with PyTorch and CUDA, and all the experiments are conducted on a Linux machine with 2.90GHz Intel Xeon CPU and a single Nvidia RTX 3090 GPU. We integrate Xplace with the legalization method discussed in DREAMPlace [80] and the detailed placer ABCDPlace [81]. Besides, we implement the deterministic mode without significantly affecting the runtime.

We test our GPU-accelerated placer on the ISPD 2005 contest benchmarks [95] and the ISPD 2015 contest benchmarks [10] with fence-region constraints removed. DREAMPlace [80] is a placement framework accelerated by GPU and shows state-of-the-art solution quality and performance compared to previous CPU-based global placers. Therefore we compare our Xplace with DREAMPlace[2] on the ISPD 2005 contest benchmarks [95] and ISPD 2015 contest benchmarks [10]. Statistics of benchmarks are given in Table 3.1.

Note that the ISPD 2015 contest benchmarks [10] are for detailed-routability-driven placement. However, the contest-provided evaluation tool is currently inaccessible. Luckily, we find that some commercial tools (e.g. Innovus [11]) can run detailed routing after fixing some errors. In this section, all experiments on ISPD 2015 contest benchmarks are conducted on this fixed version. The fixed version of ISPD 2015 contest benchmarks is released on Xplace's GitHub repository[3].

In Section 3.6.1, we experimentally verify Xplace on the ISPD 2005 as well as the ISPD 2015 contest benchmarks and compare Xplace's performance and quality with DREAMPlace. In Section 3.6.2, we conduct ablation studies to show the efficiency of our proposed operator-level optimization techniques mentioned in Section 3.3. In Section 3.6.3, we show the effectiveness of our deterministic version. In Section 3.6.4, a novel and well-designed neural

[2]`https://github.com/limbo018/DREAMPlace/tree/b31e8afa60`. We observe this version of DREAMPlace's source codes hasn't yet optimized their CUDA kernel launch bound parameters for the NVIDIA Ampere GPU architecture which RTX 3090 is powered by. For fair comparison, we optimize the launch bound parameters in DREAMPlace's source code and report the experimental results based on the modified version.

[3]`https://github.com/cuhk-eda/Xplace/tree/main/data`

Table 3.1: Benchmarks Statistics

Benchmarks	Design	#cells	#nets	Design	#cells	#nets
ISPD 2005	adaptec1	211k	221k	bigblue1	278k	284k
	adaptec2	255k	266k	bigblue2	558k	577k
	adaptec3	452k	467k	bigblue3	1097k	1123k
	adaptec4	496k	516k	bigblue4	2177k	2230k
ISPD 2015	fft_1	35k	33k	des_perf_1	113k	113k
	fft_2	35k	33k	des_perf_a	108k	115k
	fft_a	34k	32k	des_perf_b	113k	113k
	fft_b	34k	32k	edit_dist_a	127k	134k
	matrix_mult_1	160k	159k	matrix_mult_b	146k	152k
	matrix_mult_2	160k	159k	matrix_mult_c	146k	152k
	matrix_mult_a	154k	154k	pci_bridge32_a	30k	34k
	superblue12	1293k	1293k	pci_bridge32_b	29k	33k
	superblue14	634k	620k	superblue11_a	926k	936k
	superblue19	522k	512k	superblue16_a	680k	697k

network is plugged into the framework of Xplace to further improve the quality without invoking large runtime overhead.

3.6.1 Validation on Contest Benchmarks

ISPD 2005 contest benchmarks: Quantitative results on the ISPD 2005 contest benchmarks are presented in Table 3.2. We compare the HPWL, the global placement time and the total placement time (including I/O, legalization and detailed placement time) with DREAMPlace. "GP/s" and "PL/s" refer to the GP time and total placement time respectively. Experimental results show that Xplace (without NN) achieves around 1.7x GP time speedup over DREAMPlace and produces better solution quality.

ISPD 2015 contest benchmarks: Quantitative results on the ISPD 2015 contest benchmarks are presented in Table 3.3. Similar to [29], we use NTUplace4dr [48] and the embedded

Table 3.2: HPWL($\times 10^6$) and runtime (seconds) results on ISPD 2005 contest benchmarks [95]. "DM" denotes determinism.

Design	Xplace			DREAMPlace			XplaceDM			DREAMPlaceDM			Xplace-NN (DM)		
	HPWL	GP/s	PL/s	HPWL	GP/s	PL/s	HPWL	GP/s	PL/s	HPWL	GP/s	PL/s	HPWL	GP/s	PL/s
adaptec1	73.09	1.47	13.67	73.20	3.94	19.40	73.08	1.47	13.57	73.16	6.18	21.93	73.08	2.78	18.36
adaptec2	81.30	1.61	15.04	82.18	4.08	22.76	81.32	2.03	15.58	82.09	6.59	26.02	81.63	3.72	17.19
adaptec3	193.62	2.48	27.09	193.26	4.66	36.71	193.66	2.92	27.87	193.31	7.15	40.42	193.73	4.79	29.58
adaptec4	173.36	2.85	26.17	174.14	5.14	38.04	173.35	3.39	26.37	173.88	7.99	42.13	173.30	5.27	28.72
bigblue1	89.08	1.47	14.45	89.37	4.28	23.76	89.07	1.65	14.68	89.35	7.43	27.66	88.97	2.95	16.51
bigblue2	136.91	2.45	38.36	136.92	4.98	48.25	136.91	2.69	38.20	136.99	7.31	52.39	136.42	4.68	40.18
bigblue3	303.08	5.53	54.15	304.38	8.24	75.21	303.18	6.91	55.55	304.27	13.75	82.07	302.14	9.92	57.93
bigblue4	742.19	11.80	115.52	744.11	13.63	150.80	742.23	14.26	116.97	744.10	20.26	170.08	741.02	18.54	121.21
Mean	224.08	3.71	38.06	224.70	6.12	51.87	224.10	4.42	38.60	224.64	9.58	57.84	223.79	6.58	41.21
Ratio	1.0000	1.0000	1.0000	1.0028	1.6504	1.3629	1.0001	1.1908	1.0143	1.0025	2.5846	1.5198	0.9987	1.7751	1.0829

Table 3.3: HPWL($\times 10^6$), Top5 overflow, and runtime (seconds) results on the ISPD 2015 contest benchmarks [10]. OVFL-5 stands for top5 overflow. "DM" denotes determinism. The benchmarks with fence region constraints removed are labeled with †.

Design	Xplace				DREAMPlace				XplaceDM				DREAMPlaceDM				Xplace-NN (DM)			
	HPWL	OVFL-5	GP/s	PL/s	HPWL	OVFL-5	GP/s	PL/s	HPWL	OVFL-5	GP/s	PL/s	HPWL	OVFL-5	GP/s	PL/s	HPWL	OVFL-5	GP/s	PL/s
des_perf_1	1106.5	64.35	1.16	6.27	1107.3	65.33	4.07	11.76	1106.4	64.82	1.29	7.40	1107.0	65.59	6.32	14.53	1113.7	64.99	2.33	10.71
des_perf_a†	1998.8	52.50	1.25	5.89	2019.6	53.12	3.83	12.00	1997.3	52.39	1.38	6.63	2021.5	53.16	5.79	14.34	2023.4	54.52	2.60	7.62
des_perf_b†	1611.8	53.88	1.32	6.02	1609.5	54.53	3.85	11.92	1612.3	53.76	1.42	6.22	1609.7	54.40	5.29	13.87	1614.6	53.27	2.47	7.53
edit_dist_a†	4198.3	79.92	1.45	7.13	4215.7	80.58	4.19	13.97	4198.9	79.65	1.55	7.58	4215.7	80.31	5.91	15.84	4197.5	80.04	3.11	9.19
fft_1	411.5	56.34	1.18	3.23	416.1	56.78	4.82	9.45	411.0	55.66	1.25	3.60	412.1	56.23	5.42	10.08	412.0	56.63	2.48	4.99
fft_2	374.1	47.68	1.18	3.48	372.7	47.54	3.72	8.16	373.7	47.52	1.31	3.47	374.3	47.52	6.64	11.00	374.3	47.85	2.41	4.79
fft_a	625.8	34.80	1.30	3.34	629.3	35.27	3.67	7.78	626.4	34.70	1.39	3.52	628.9	34.98	5.68	10.01	626.9	34.89	2.24	4.76
fft_b	845.6	51.80	1.72	3.77	845.2	51.98	3.71	7.92	847.1	52.15	1.43	3.77	845.0	51.89	6.53	11.06	846.3	51.87	2.26	4.53
matrix_mult_1	2116.3	82.19	1.61	7.45	2129.7	82.18	3.65	14.00	2115.6	81.85	1.33	7.17	2129.1	81.99	5.07	15.28	2123.0	82.02	2.58	8.48
matrix_mult_2	2152.7	77.53	1.29	6.98	2164.3	77.47	4.32	14.74	2152.7	76.78	1.41	7.40	2163.2	77.15	7.13	17.54	2151.2	75.91	2.57	8.58
matrix_mult_a	3032.6	48.21	1.73	9.21	3036.6	47.91	4.42	16.12	3032.3	48.32	1.52	8.80	3037.0	47.96	6.67	18.14	3030.5	48.22	2.64	9.68
matrix_mult_b†	2762.6	45.05	1.65	7.72	2787.2	45.02	4.19	14.76	2762.7	45.01	1.40	7.62	2787.4	45.17	6.43	17.67	2761.3	44.21	2.47	8.90
matrix_mult_c†	2674.6	42.31	1.74	7.47	2673.1	42.31	4.26	16.21	2675.6	42.19	1.46	7.59	2674.0	42.24	5.90	18.37	2677.9	42.31	2.56	8.52
pci_bridge32_a†	360.9	30.65	1.60	3.80	361.8	30.41	3.87	8.12	361.3	30.51	1.31	3.37	362.0	30.17	5.36	9.86	355.2	30.45	2.40	4.75
pci_bridge32_b†	714.0	23.16	1.49	3.67	741.5	22.79	6.43	11.31	715.1	23.04	1.23	3.64	739.7	22.49	10.08	14.70	714.4	22.61	2.14	4.58
superblue11_a†	33521.3	54.46	2.94	39.74	33397.5	54.50	5.64	73.13	33521.8	54.58	3.95	42.24	33413.7	54.33	7.61	74.65	33526.4	54.64	4.78	42.75
superblue12	25784.5	92.44	4.72	54.77	25799.8	92.62	8.85	91.02	25792.8	92.27	5.35	58.17	25791.8	93.24	14.32	98.62	25694.8	92.68	8.10	58.76
superblue14	22776.7	63.72	2.01	28.84	23028.4	63.14	4.61	52.59	22777.5	63.77	2.12	29.10	23019.6	63.41	7.31	55.14	22784.5	63.26	3.19	30.54
superblue16_a†	25491.0	65.98	2.19	29.42	25605.4	65.98	4.25	54.93	25500.4	66.03	2.51	30.82	25599.5	65.97	6.64	58.91	25455.6	65.89	3.63	32.14
superblue19	15542.4	62.32	1.97	22.45	15630.6	61.81	4.64	42.10	15545.5	62.39	2.13	22.77	15630.7	61.89	6.83	45.73	15446.4	60.88	3.84	24.83
Mean	7405.1	56.46	1.78	13.03	7428.6	56.56	4.55	24.60	7406.3	56.37	1.84	13.54	7428.1	56.50	6.85	27.27	7396.5	56.36	3.04	14.83
Ratio	1.0000	1.0000	1.0000	1.0000	1.0032	1.0018	2.5631	1.8876	1.0002	0.9983	1.0349	1.0392	1.0031	1.0007	3.8572	2.0922	0.9988	0.9981	1.7127	1.1380

NCTUgr [84] to report the solution quality (HPWL) and the global routing (GR) routability (top5 overflow). Note that top5 overflow measures routability by taking the average overflow of the top 5% most congested GR gcells. Experimental results show that Xplace obtains around 2.6x GP time speedup than DREAMPlace and produces better HPWL with comparable top5 overflow. Considering both ISPD 2005 and ISPD 2015 benchmarks together, Xplace can achieve around 2x GP runtime speedup over DREAMPlace.

3.6.2 Ablation Studies on Operator-Level Optimization

We perform ablation studies of our proposed operator-level optimization techniques to demonstrate their effectiveness. We measure the runtime performance by the per global iteration time. Quantitative results are shown in Table 3.4. It is worth noting that operator combination, operator extraction and operator skipping techniques mainly boost the runtime performance of the larger cases while the operator reduction technique accelerates the smaller cases. These results also show that Xplace can achieve around 3x per GP iteration time speedup compared with DREAMPlace on average.

3.6.3 The Effectiveness of Determinism Implementation

We verify the solution quality and runtime performance of Xplace's deterministic version (shortened as XplaceDM) on ISPD 2005 and 2015 contest benchmarks. The results are shown in Table 3.2 and Table 3.3. Considering both ISPD 2005 and ISPD 2015 benchmarks together, XplaceDM only needs around 10% runtime overhead to support determinism with comparable solution quality. Compared to the deterministic version of DREAMPlace (denoted as DREAMPlaceDM), XplaceDM is accelerated around 3x in terms of GP time. With the well-designed parallelization of our operator-level optimization, XplaceDM also achieves around 2x GP time speedup over the non-deterministic DREAMPlace while keeping the determinism. The results successfully demonstrate the effectiveness of our determinism implementation.

Table 3.4: Ablation studies of the operator-level optimization techniques on ISPD 2005 benchmarks [95]. OR, OC, OE, OS refer to operator reduction, operator combination, operator extraction, and operator skipping respectively. Note that Xplace enables all the operator-level optimization techniques in Section 3.3.

Methods	OR	OC	OE	OS	adaptec1	adaptec2	adaptec3	adaptec4	bigblue1	bigblue2	bigblue3	bigblue4	Avg
Ratio	-	-	-	-	234%	194%	136%	124%	198%	140%	123%	121%	159%
	✓	-	-	-	110%	109%	113%	115%	105%	115%	119%	118%	113%
	✓	✓	-	-	107%	107%	107%	108%	104%	108%	113%	112%	108%
	✓	✓	✓	-	104%	102%	104%	104%	102%	104%	106%	105%	104%
Xplace	Ratio				100%	100%	100%	100%	100%	100%	100%	100%	100%
	GP / Iter Time (ms)				1.478	1.671	2.325	2.688	1.572	2.441	4.974	10.018	-
DREAMPlace	Ratio				462%	345%	288%	254%	376%	288%	199%	158%	296%
	GP / Iter Time (ms)				6.832	5.769	6.699	6.840	5.915	7.023	9.904	15.831	-

3.6.4 Neural-Enhanced Performance

The proposed model is a lightweight neural network with 471k parameters, which is only 60% of the well-known image-to-image model U-Net [108]. We perform the training based on ISPD 2005 contest benchmarks with their respective macro layouts. The density map and electric fields are used as training data and labels at every iteration with 256×256 resolution. The training scheme is given as follows,

1. Randomly select one of the macro layouts.
2. Randomly generate standard cells at a starting position and push them all over the map with only the density objective $D(p)$ for 40 iterations
3. Repeat (1) and (2) 200 times to generate 8000 samples
4. Train the model on these samples 250 iterations with 128 batch size
5. Consider (1)-(4) as one epoch and repeat 50 epochs.

Recall that the trained model has a good global view to spread cells. We explore embedding this model into our deterministic version XplaceDM. The HPWL of Xplace-NN improves around 1.3‰ in comparison with the original Xplace on the ISPD 2005 contest benchmarks as shown in Table 3.2. Xplace-NN can also obtain 1.2‰ quality improvement on ISPD 2015 benchmarks as shown in Table 3.3. Note that the proposed model is only trained on the ISPD 2005 while it can further improve the quality of the ISPD 2015 benchmarks. The results not only illustrate the effectiveness of the proposed network model but also show the high extensibility of our Xplace framework.

3.7 Concluding Remarks

In this chapter, we present Xplace, a new, fast, extensible and open-source GPU-accelerated placement framework. We also illustrate the possibility of incorporating a neural network component into a GPU-accelerated analytical placer. Experimental results on the ISPD 2005 and the ISPD 2015 benchmarks show that Xplace is efficient yet extensible.

Chapter 4

GPU-Accelerated Detailed-Routability-Driven VLSI Placement

Previous routability-driven placers either lacked the capability to handle detailed-routability optimization or relied on a time-consuming CPU-based global router for routability optimization. To address these limitations and fully leverage the power of GPUs, this chapter introduces Xplace-Route[1], a detailed-routability-driven GPU-accelerated placer built upon the deterministic version Xplace (discussed in Chapter 3), to effectively handle the detailed-routability. Our contributions are summarized as follows.

- We propose Xplace-Route, a fast detailed-routability-driven placer extended from our GPU-accelerated placement engine Xplace. A GPU-accelerated routing engine is implemented in Xplace-Route for efficient routability evaluation.

- Equipped with detailed-routability-aware techniques, Xplace-Route significantly improves detailed routability and reduces the number of violations.

[1]The source code of Xplace-Route is released on Xplace's GitHub Repository `https://github.com/cuhk-eda/Xplace`

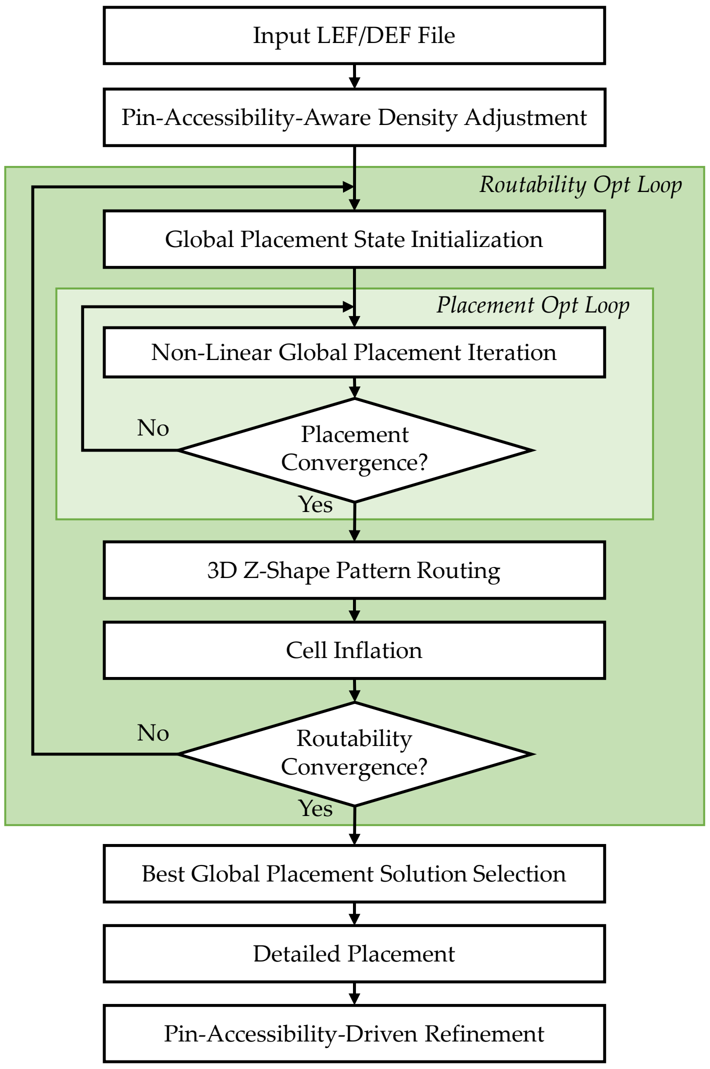

Input LEF/DEF File
Pin-Accessibility-Aware Density Adjustment
Routability Opt Loop
Global Placement State Initialization
Placement Opt Loop
Non-Linear Global Placement Iteration
No
Placement Convergence?
Yes
3D Z-Shape Pattern Routing
Cell Inflation
No
Routability Convergence?
Yes
Best Global Placement Solution Selection
Detailed Placement
Pin-Accessibility-Driven Refinement

Figure 4.1: Routability optimization flow of Xplace-Route.

- Experimental results on ISPD 2015 contest benchmarks demonstrate that Xplace-Route achieves significant quality improvement and runtime speedup.

4.1 Overview of Xplace-Route

The routability optimization flow of Xplace-Route is shown in Figure 4.1. Xplace-Route first conducts pin-accessibility-aware density adjustment after parsing the given design. Then Xplace-Route recursively improves the placement routability by a two-level nesting loop until the flow converges. The inner loop aims to optimize the cell position by the non-linear global placement objective given in Equation (3.3). With the converged placement solution given by the inner loop, Xplace-Route invokes an internal GPU-accelerated pattern router and conducts a cell inflation-based routability optimization loop. Xplace-Route will recursively re-launch the placement engine and further optimize the routability. When the nesting loop converges, Xplace-Route selects the best placement solution according to the historical routing metrics and then performs detailed placement. Finally, a pin-accessibility-driven refinement is adopted to optimize the routability further.

4.2 Detailed-Routability Optimization

4.2.1 Pin-Accessibility-Aware Density Adjustment

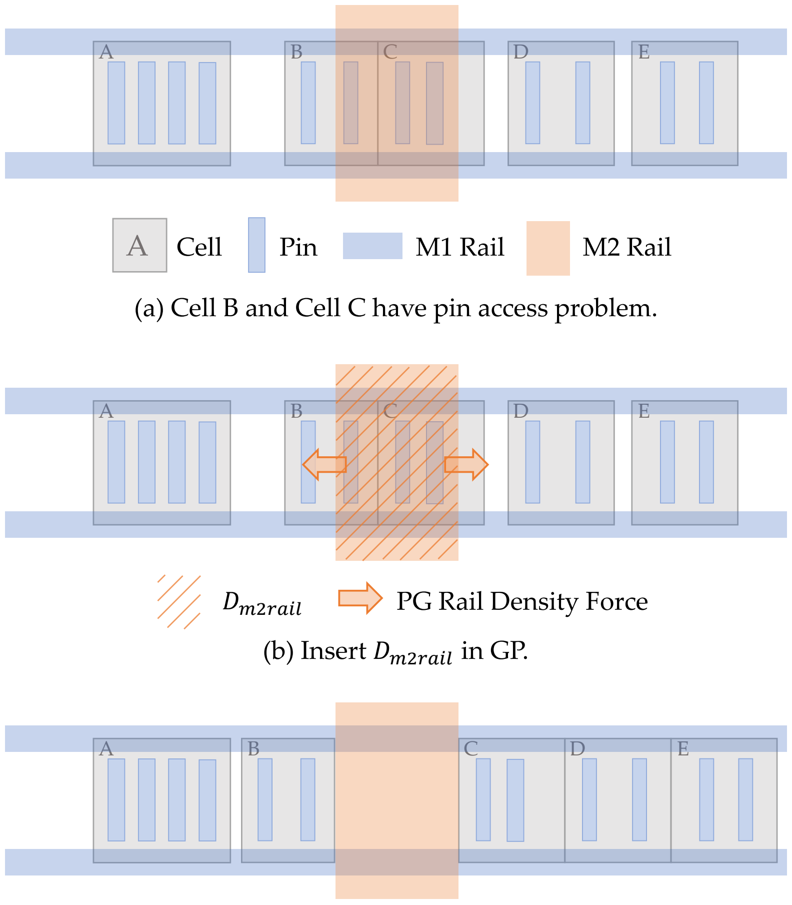
Pin-Accessibility is highly related to detailed routability [126, 55]. Low accessibility of a pin will easily lead to detailed routing violations. As shown in Figure 4.2(a), the cell B's and C's M1 signal pins are covered by an M2 power and ground (PG) rail, and it is difficult for a routing wire to access these pins without violations because of the limited routing resource on M1 layer.

To mitigate the pin-accessibility issue, we insert the PG rail density in global placement as a penalty to move the cells away from the M2 rail as illustrated in Figure 4.2(b). Besides, we observe that the area near I/O pin is usually congested, so we increase the I/O pin density to relax the congestion. To this end, the cell density D previously formulated in Equation (3.8) is modified as follows,

$$D_{route} = D + D_{m2rail} + 3 \times D_{iopin} \tag{4.1}$$

and the total density $\tilde{D}$ in Equation (3.11) used for solving Poisson's Equation is updated correspondingly as follows,

$$\tilde{D} = D_{route} + D_{fl} \tag{4.2}$$



(a) Cell B and Cell C have pin access problem.

(b) Insert D_{m2rail} in GP.

(c) Invoke PA-Refine on (a).

Figure 4.2: Pin-accessibility-aware optimization techniques.

Note that we can consider M2 rail and I/O pin as fixed macro, thus the D_{m2rail} and D_{iopin} can be pre-computed.

Compared to addressing the pin-accessibility issue during the detailed placement stage, adjusting the local placement density in global placement to ensure pin-accessibility offers valuable guidance for the gradient optimizer, enabling it to converge towards a solution by considering both M2 rail and I/O pin-accessibility.

4.2.2 Routing Congestion Map

Similar to CUGR [82] and NCTUgr [84], we divide the 3D routing region into a set of global routing cells (G-Cells) $G_R \in \mathbb{R}^{R_r \times R_c \times L}$, where R_r and R_c denote the number of rows and

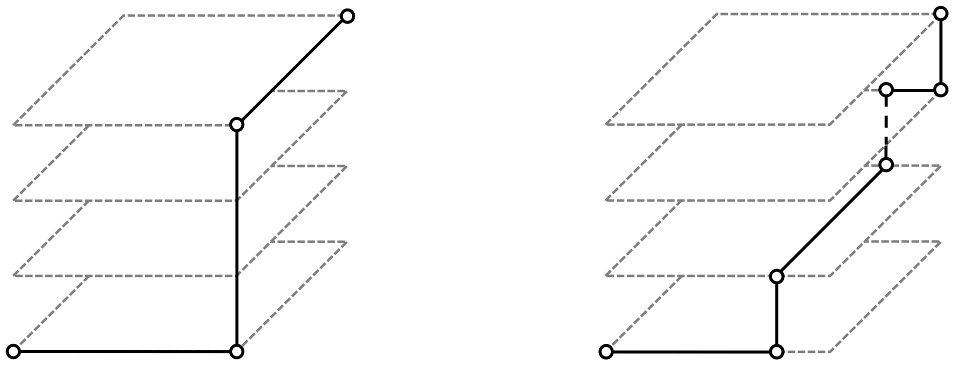

(a) L-shape **(b)** Z-shape

Figure 4.3: 3D L-shape and Z-shape pattern routing.

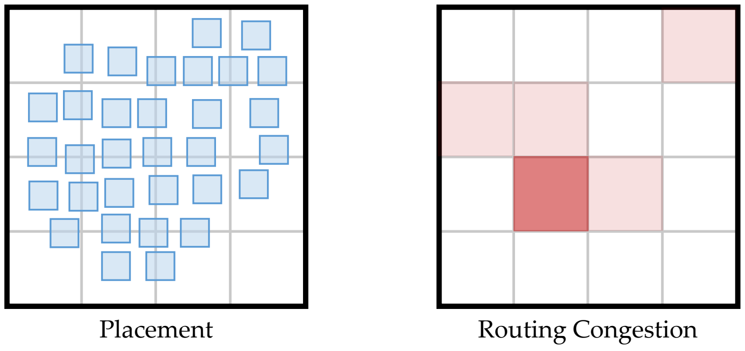
columns in the routing grid graph; L denotes the number of routing layers.

In Xplace-Route, we implement a GPU-accelerated 3D Z-shape pattern routing algorithm discussed in [77] to effectively evaluate the routability of the intermediate placement solution and provide valuable guidance for routability optimization. Reported by the internal router, we obtain the routing demand map $Dmd \in \mathbb{R}^{R_r \times R_c \times L}$ and the routing capacity map $Cap \in \mathbb{R}^{R_r \times R_c \times L}$ of a placement solution, and then we construct a routing congestion map $C \in \mathbb{R}^{R_r \times R_c}$, formulated as follows,

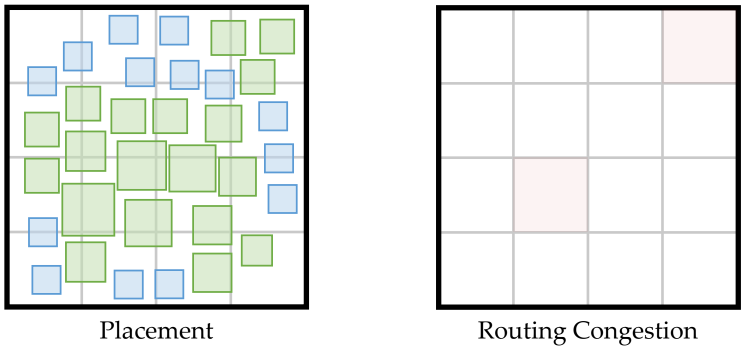
$$C_{x,y} = \max\left(\frac{\sum_{l=1}^{L} Dmd_{l,x,y}}{\sum_{l=1}^{L} Cap_{l,x,y}} - 1, 0\right),\ \forall(x,y) \in \mathbb{R}^{R_r \times R_c} \tag{4.3}$$

where Dmd is the summation of wire demand and the via demand. From Equation (4.3), an area will be classified as congested if the routing demand exceeds the available capacity in that specific area. Note that the routing solution is 3D while our placement problem is 2D. Thus, we calculate the summation across all metal layers in Equation (4.3) to reduce the dimension of the capacity and demand map from 3D to 2D.

Due to the large runtime overhead of maze routing, we choose only to launch the Z-shape pattern router (PR) to compute the congestion map and guide the cell inflation. Note that the Z-shape pattern router is also capable of generating an L-shape solution (illustrated in Figure 4.3). Empirically, such a PR-only scheme successfully enhances the placement routability with a relatively small runtime overhead.



(a) Before inflation



(b) After inflation

Figure 4.4: An example of cell inflation.

4.2.3 Cell Inflation

Recent works [40, 20, 48, 17, 110, 57, 80] have shown that inflating cells in congested areas can effectively alleviate routing congestion. As depicted in Figure 4.4, this approach involves enlarging the width and height of cells to decrease the local cell density within the congested area. Consequently, in this congested area, both the number of signal pins required tow access and the number of routing wires needed to traverse it are subsequently reduced. This reduction in demand for routing resources ultimately leads to an improvement in routability.

Similarly, in Xplace-Route, we use the routing congestion map in Equation (4.3) to compute the historical cell inflation ratio, derived as follows,

$$r_{i,t} = r_{i,t-1} \times \sqrt{\Delta r_{i,t}} \tag{4.4a}$$

$$\Delta r_{i,t} = \sum_{(x,y)\in\mathbb{R}^{R_r\times R_c}, A_i \bigcap A_{x,y} \neq \varnothing} \frac{A_i \bigcap A_{x,y}}{A_i} IR_{x,y} \tag{4.4b}$$

where $r_{i,t}$ denotes the inflation ratio of cell i's width and height in the t-th routability optimization iteration; $r_{i,0}$ is initialized as 1; A_i represents the area of cell i, and $A_{x,y}$ represents the area of the grid located at position (x, y); $IR \in \mathbb{R}^{R_r\times R_c}$ is the inflation ratio map, given as follows,

$$IR_{x,y} = \min((C_{x,y} + 1)^2, 2), \ \forall(x, y) \in \mathbb{R}^{R_r\times R_c} \tag{4.5}$$

As formulated in Equation (4.5), a cell in a more congested area will be much more inflated than one in a non-congested area. If a cell is frequently placed in a congested area, the historical inflation ratio formulated in Equation (4.4) implies that this cell will be inflated more than once to mitigate the potential routing resource scarcity.

4.2.4 Placement State Re-initialization

After the calculation of the inflation ratio, Xplace-Route will inflate cells and then re-initialize placement parameters. Different from the routability-driven DREAMPlace [80], which resets the density weight λ if the overflow ratio is smaller than a certain number, Xplace-Route waits for the global placement convergence and then re-initializes the placement state (including density weight λ, WA coefficient γ, Nesterov optimizer, etc). Since the cells are inflated, the non-linear optimized solution is largely changed. Therefore, re-initialization is necessary for the non-linear optimizer. Empirically, the re-initialization strategy assists the optimizer in searching for a better placement solution.

Figure 4.5 illustrates the HPWL and density overflow curves for ISPD 2015 `superblue12`. The dashed lines indicate the iterations where non-linear optimization converged. In this case, Xplace-Route employs non-linear placement optimization until convergence is achieved. Once a converged solution is obtained, standard cells are inflated, and the placement parameters are re-initialized. Subsequently, Xplace-Route proceeds with the next step of routability optimization. Note that one routability optimization step includes all the iterations between two consecutive dashed lines. Indeed, while the converged solution's HPWL may show a gradual increase due to cell inflation, Figure 4.7 demonstrates the corresponding gradual improvement in routability. This improvement further shows the effectiveness of our iterative routability optimization schemes. Figure 4.6 depicts all the converged solutions obtained by Xplace-Route. Note that the cell density in congested areas is gradually decreased to mitigate the routing resource scarcity problem.

4.2.5 Solution Selection Criteria

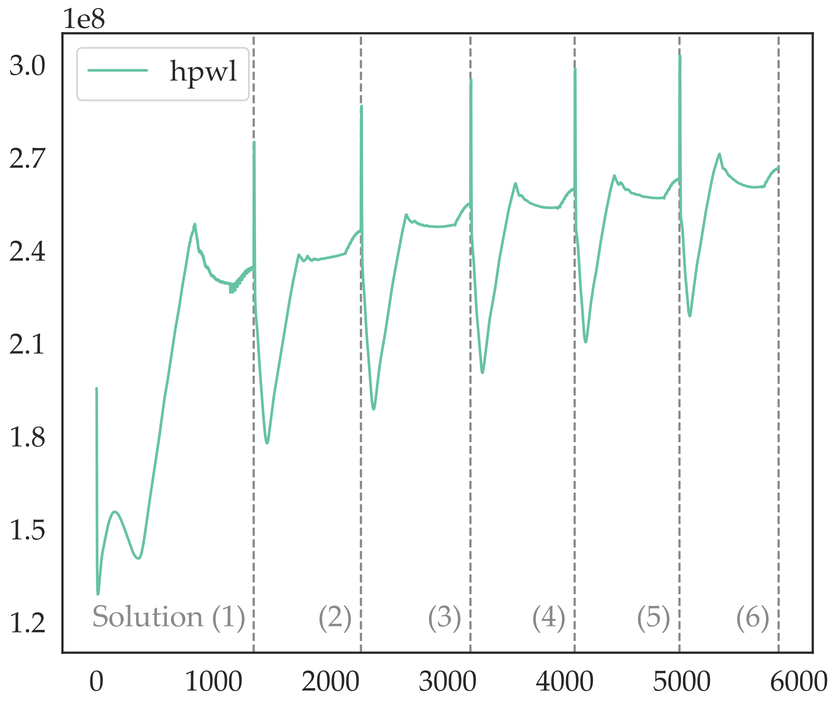
Xplace-Route recursively launches the routability optimization to reduce congestion. The routability optimization loop will be terminated if there is insufficient space to inflate the cells or the loop reaches the maximum iteration I_{route} (we set $I_{route} = 5$). After finishing the routability optimization, Xplace-Route selects the placement solution with the smallest routing cost among all the converged placement solutions (i.e. the outputs of the non-linear optimization loop). For a specific placement solution, its routing cost c_{route} is derived from the router-provided wire and via information, given as follows,

$$A_{wire,l} = WireWidth_l \times GCellSize_l \tag{4.6a}$$

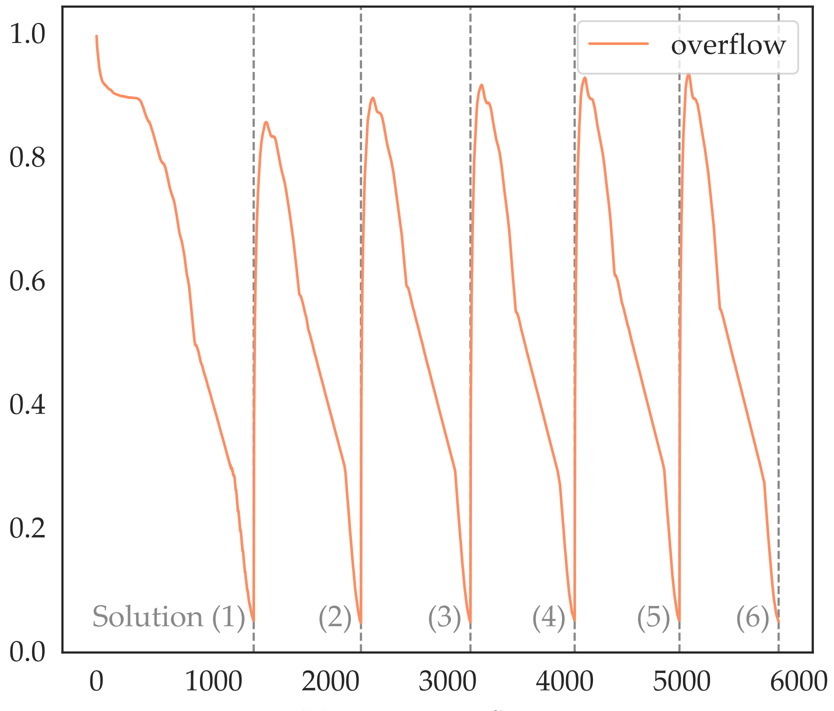
$$m_{l,x,y} = \begin{cases} 1, & \text{if } WireDmd_{l,x,y} > Cap_{l,x,y}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.6b}$$

$$c_{route} = \sum_{l,x,y} \max(WireDmd_{l,x,y} - Cap_{l,x,y}, 0) A_{wire,l} + \sum_{l,x,y} m_{l,x,y} \times ViaDmd_{l,x,y} \tag{4.6c}$$

Different from ACE [128] that computes the average of the top x% congestion, our routing cost further considers the l-th layer unit wire area $A_{wire,l}$ and the relationship between the routing capacity and the demand. In Figure 4.5, we select the fifth converged solution that has the best routing cost.

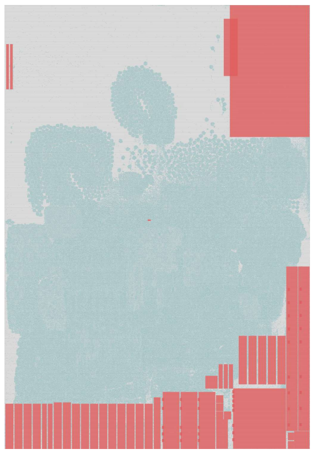


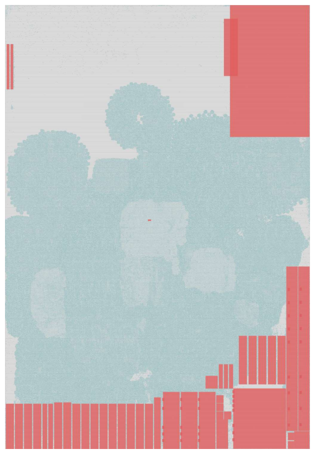
(a) HPWL

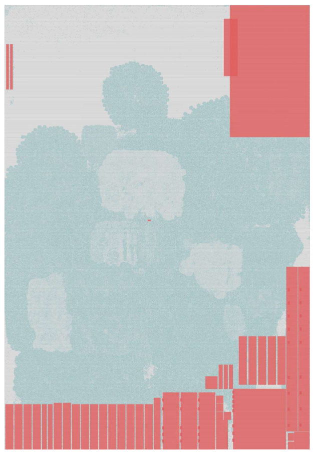


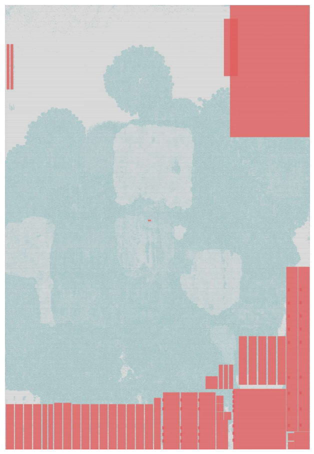
(b) Density overflow

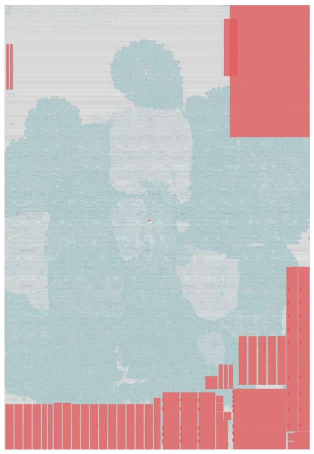
Figure 4.5: HPWL and density overflow curves for `superblue12`.


(a) Placement solution (1)


(b) Placement solution (2)


(c) Placement solution (3)


(d) Placement solution (4)


(e) Placement solution (5)

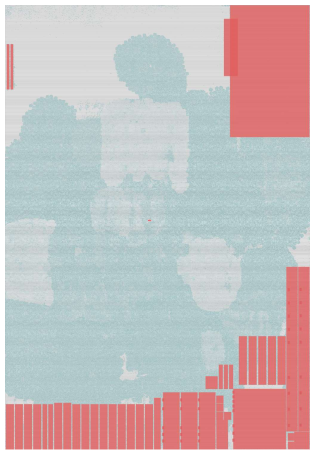
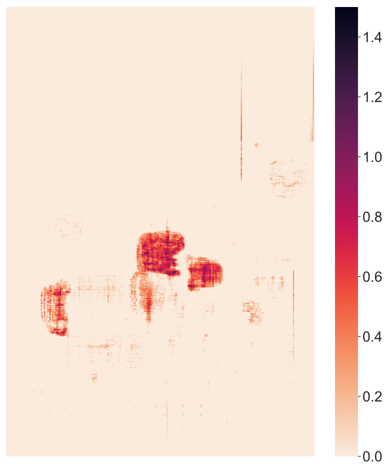
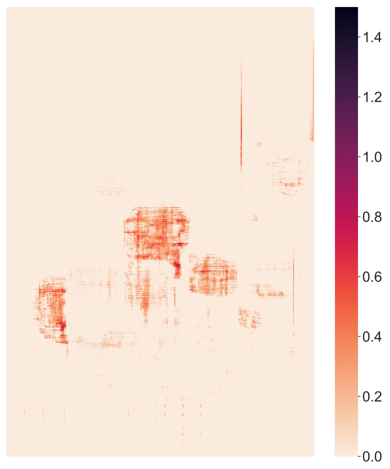

(f) Placement solution (6)

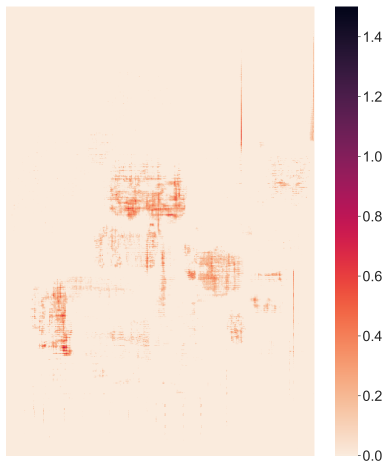
Figure 4.6: All the converged placement solutions for `superblue12`.



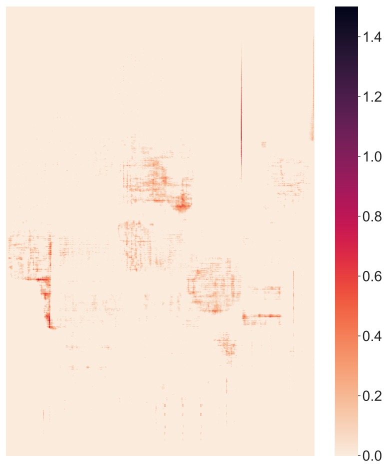
(a) Congestion map of solution (1)



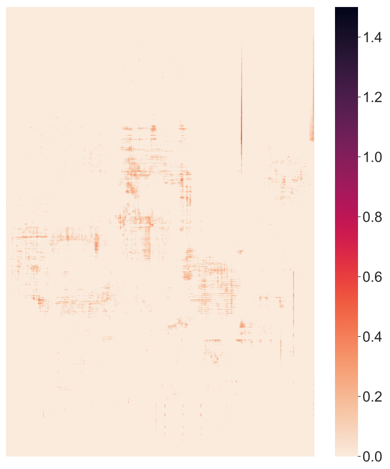
(b) Congestion map of solution (2)



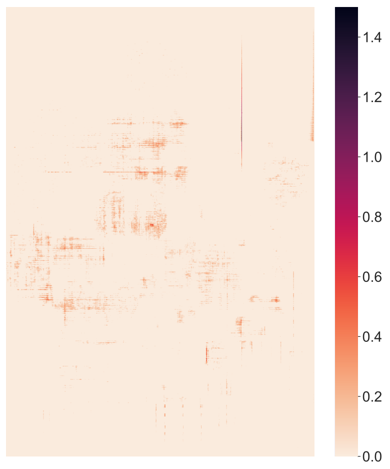
(c) Congestion map of solution (3)



(d) Congestion map of solution (4)



(e) Congestion map of solution (5)



(f) Congestion map of solution (6)

Figure 4.7: Congestion map of all the converged solutions for `superblue12`.

4.2.6 Pin-Accessibility-Driven Refinement

We integrate with ABCDPlace [81] to perform legalization and detailed placement on the selected global placement solution. However, ABCDPlace lacks detailed-routability-driven refinement. As discussed in Section 4.2.1, the pin accessibility of the placement solution can be significantly improved by moving cells outside the area covered by M2 rail. Different from [48] and [66], which adopt the pin accessibility optimization during the legalization, we propose a pin-accessibility post-refinement technique, invoked at the end of the detailed placement, to refine the PG rail-related detailed-routability.

Pin-accessibility post-refinement (shortened as PA-refine) works in a row-based manner and post-refines the legal placement solution. If a cell i overlaps with the M2 rail (i.e. the cell i is placed under the M2 rail), PA-refine will search its horizontal neighborhoods for possible movement. If it is possible to shift no more than K cells in the left or right direction to remove the overlap, PA-refine will move these cells sequentially to resolve the pin-accessibility issue. If there is insufficient space for overlap removal within K cells, we won't move them. Empirically, we use $K = 5$ to improve the pin-accessibility while avoiding large displacement. Figure 4.2(c) illustrates an example of the result of the above technique.

4.3 Experimental Results

We developed Xplace-Route with PyTorch and CUDA. Unless specified, the experiments are default conducted on a Linux machine with 2.90GHz Intel Xeon CPU and a single Nvidia RTX 3090 GPU. In this section, we will verify the detailed-routability of Xplace-Route on the fixed version[2] of ISPD 2015 detailed-routability-driven contest benchmarks [10] and demonstrate the effectiveness of our proposed routability optimization techniques.

[2] https://github.com/cuhk-eda/Xplace/tree/main/data

4.3.1 Experimental Settings

We evaluate the detailed routability of Xplace-Route, Xplace, DREAMPlace, routability-driven DREAMPlace [80]. Note that the routability-driven DREAMPlace released on GitHub[3] uses NCTUgr [84] to calculate the congestion map and compute the cell inflation ratio. However, the version of NCTUgr used by DREAMPlace does not support ISPD 2015 LEF/DEF format. Therefore we use the state-of-the-art open-sourced global placer CUGR [82] instead[4]. Besides, we also compare Xplace-Route with the state-of-the-art CPU-based detailed-routability-driven placer NTUplace4dr[48].

To verify the placement detailed-routability, we first execute different placers to generate their placement DEF files. Then we use Innovus 20.14 [11] to load the placed solutions and perform detailed routing on a Linux machine with 4x Intel Xeon E7-4830 v2 (2.20GHz). For a fair comparison, we use the same settings and parameters in Innovus to run detailed routing on these placement solutions with 10 threads enabled[5]. We measure the routability of a placement solution by the following metrics: the detailed routing wirelength (DR WL/um), the number of DR vias (#DR Vias), the number of detailed routing violations (#DRVs), and the placement time (PL/s).

4.3.2 Quantitative Results

The results are shown in Table 4.1. Xplace-Route runs in deterministic mode and successfully outperforms the other baselines in terms of detailed routing metrics. Compared to the original Xplace and DREAMPlace, Xplace-Route remarkably boosts the detailed-routability, especially #DRVs, while only taking no more than 27 seconds (on average) extra runtime overhead. Compared to the existing routability-driven placers, Xplace-Route also shows better routability with a much shorter placement elapsed time. Compared to routability-

[3]`https://github.com/limbo018/DREAMPlace/tree/b31e8afa60`

[4]`https://github.com/cuhk-eda/cu-gr`

[5]The DR evaluation script is also provided in Xplace's GitHub Repository `https://github.com/cuhk-eda/Xplace/tree/main/tool/innovus_ispd2015_fix`.

Table 4.1: DR metrics and runtime results on the ISPD 2015 contest benchmarks [10]. "DM" denotes determinism and "NonDM" denotes non-determinism. The benchmarks with fence region constraints removed are labeled with †.

Design	**Xplace-Route (DM)**					DREAMPlace (NonDM)					NTUplace4dr (DM)				
	DR WL/um	#DR Vias	#DRVs	PL/s	∗DR/s	DR WL/um	#DR Vias	#DRVs	PL/s	∗DR/s	DR WL/um	#DR Vias	#DRVs	PL/s	∗DR/s
des_perf_1	1439840	564970	10306	10	1801	1449286	570616	19945	12	2391	1509013	630344	2931	357	2721
des_perf_a†	2488402	575632	43854	23	645	2366671	565391	32381	12	652	2397042	569311	2602	302	4480
des_perf_b†	1802951	543049	1323	10	415	1810931	555033	14920	12	2018	1769864	551066	1887	332	468
edit_dist_a†	5750598	1015469	422480	26	2521	5687881	1012545	426136	14	2640	6576184	1125780	1188119	457	5736
fft_1	513869	185766	3478	13	769	522764	188497	8011	9	1375	572738	199139	1397	91	1376
fft_2	621300	191162	1186	12	953	599272	187689	8886	8	423	729743	195227	993	99	753
fft_a	1137087	192259	781	13	1127	1079780	192945	4533	8	1569	1176459	199888	1888	96	2668
fft_b	1282342	213953	16661	14	941	1252724	211800	21013	8	895	1267726	208975	39082	111	835
matrix_mult_1	2648037	828540	12373	26	6933	2712367	812150	79049	14	1344	2699517	892013	4186	339	3517
matrix_mult_2	2678569	857402	11421	26	8809	2723576	842283	69799	15	1482	2733713	906449	5062	360	8217
matrix_mult_a	3941724	848261	7054	12	6536	3881128	864485	26127	16	5073	4190876	869526	4089	381	12186
matrix_mult_b†	3649667	782850	45894	10	1301	3670952	789352	73368	15	1280	3921403	804760	61470	318	1188
matrix_mult_c†	3685351	791702	7518	10	3952	3712692	816093	26761	16	6080	4331793	855656	3784	340	5535
pci_bridge32_a†	651346	145663	4001	5	3163	650836	149042	5969	8	2649	606431	143725	1729	125	2022
pci_bridge32_b†	1007617	147953	347	5	162	1012909	150805	2158	11	320	811280	137266	426	88	119
superblue11_a†	40316503	5659845	1202	78	6906	39973449	5645598	1182	73	6953	64214794	7218914	533505	14321	16325
superblue12	42780736	10508482	29569	311	22614	42477241	11254375	3283142	91	27160	48273446	11768924	31293	7493	24229
superblue14	27959928	4341696	366	70	11901	28386548	4435721	418	53	14639	31959227	5340941	1343	3166	11635
superblue16_a†	31460619	4644486	4489	82	16539	31455744	4732382	3871	55	15284	36908305	5632423	14586	3654	30615
superblue19	20451060	3582347	8255	58	9310	20556809	3612264	7266	42	7285	22402103	4035778	8041	3245	7839
Mean	9813377	1831074	31628	41	5365	9799178	1879453	205747	25	5076	11952583	2114305	95421	1784	7123
Ratio	1.00	1.00	1.00	1.00	1.00	1.00	1.03	6.51	0.60	0.95	1.22	1.15	3.02	43.82	1.33

Design	Xplace (DM)					DREAMPlace + CUGR (NonDM)					Enhanced DREAMPlace + CUGR (NonDM)				
	DR WL/um	#DR Vias	#DRVs	PL/s	∗DR/s	DR WL/um	#DR Vias	#DRVs	PL/s	∗DR/s	DR WL/um	#DR Vias	#DRVs	PL/s	∗DR/s
des_perf_1	1447376	569167	20064	7	2726	1447442	568831	19829	204	2400	1440466	561512	19976	205	2499
des_perf_a†	2338413	558472	28028	7	589	2866082	605591	240085	115	1181	2366966	565054	33102	218	629
des_perf_b†	1816033	557017	14768	6	2310	1828936	566050	14253	173	1937	1820022	546426	13966	251	1901
edit_dist_a†	5671918	1008135	405727	8	2820	6045810	1059270	608067	164	2985	5838307	1080238	493280	202	2876
fft_1	517999	187120	7802	4	1291	516922	188519	8157	44	1317	524240	188222	8485	97	1316
fft_2	598808	188353	9200	3	394	608394	196025	5552	65	1739	629227	194278	6105	89	1552
fft_a	1093206	193126	4677	4	1476	1345219	219195	7641	49	1277	1076563	192419	4855	86	1585
fft_b	1249450	203681	32851	4	928	1405437	224950	44052	41	1224	1255215	211532	21385	90	1090
matrix_mult_1	2703662	807649	76976	7	1306	2711580	821025	79519	84	1318	2722410	812788	79271	164	1325
matrix_mult_2	2718941	840570	68624	7	1694	2710509	845701	56748	125	1681	2709817	833468	54706	160	1655
matrix_mult_a	3877049	861600	25646	9	5540	4708376	914328	41858	113	1400	3892047	865240	25770	154	5268
matrix_mult_b†	3657963	791901	66643	8	1327	4465439	864227	77742	112	1332	3674444	785578	47308	147	1178
matrix_mult_c†	3725954	814420	26195	8	6008	4831883	919329	26265	115	8360	3706182	816044	26340	144	5857
pci_bridge32_a†	642818	148475	5631	3	3130	653934	149437	5941	16	2786	646469	148451	5613	106	2877
pci_bridge32_b†	981717	149362	2076	4	276	1029276	152185	2123	23	350	1028863	151654	2148	127	338
superblue11_a†	40316503	5659845	1202	42	7399	45092596	5907673	1314	1043	7428	40335827	5637829	967	1224	6464
superblue12	42734765	10840792	3122560	58	25770	45921282	11068935	21863	2020	18623	46529015	11083954	15744	2477	21085
superblue14	27955769	4339771	347	29	13082	28925588	4234243	415	943	5143	28548859	4184970	365	1121	5816
superblue16_a†	31460619	4644486	4489	31	16771	31953926	4634390	2495	677	12834	31082724	4568077	2410	815	12601
superblue19	20468106	3584954	8289	23	9165	22266531	3709667	6116	869	5766	22642721	3733021	19036	1326	55805
Mean	9798853	1847445	196590	14	5200	10566758	1892479	63502	350	4054	10123519	1858038	44042	460	6686
Ratio	1.00	1.01	6.22	0.33	0.97	1.08	1.03	2.01	8.59	0.76	1.03	1.01	1.39	11.30	1.25

∗ The DR time (DR/s) may not precisely reflect the placement routability because Innovus will early terminate its detailed router if a design has low routability.

driven DREAMPlace, Xplace-Route achieves around 9x placement time speedup, uses 8% shorten DR WL and 3% fewer #DR Vias, and reduces 200% #DRVs. Besides, Xplace-Route is deterministic while routability-driven DREAMPlace is non-deterministic. Compared to NTUplace4dr, Xplace-Route achieves around 44x placement time speedup, uses 22% shorten DR WL and 15% fewer #DR Vias, and reduces 300% #DRVs. The better solution quality demonstrates the necessity of detailed-routing-aware techniques, and the shorter elapsed time indicates the effectiveness of our GPU-accelerated scheme.

For reference, we also report the DR time (DR/s) in Table 4.1. However, the DR runtime may not precisely reflect the routability. For example, compared to the routability-driven DREAMPlace's solution, the commercial detailed router on Xplace-Route's solution uses shorter WL, fewer vias, and fewer DRVs but uses more runtime. The reason is that the commercial detailed router will be early terminated if a design has low routability. Therefore, we choose to measure the detailed routability by DR WL, #DR Vias, and #DRVs instead of DR time.

To measure our flow efficiency, we embed our re-initializing strategy and solution selection criteria discussed in Section 4.2.4 and Section 4.2.5 into the routability-driven DREAMPlace (denoted as Enhanced DREAMPlace + CUGR in Table 4.1). The enhanced version achieves a visible routability improvement, with a reduction of DR WL by 5%, a decrease in #DR Vias by 2%, and a 44% reduction in #DRVs, and only needs 30% extra runtime overhead compared to the original routability-driven DREAMPlace. The results demonstrate the effectiveness of our re-initialization technique and solution selection criteria.

4.3.3 Ablation Studies on Detailed-Routability Optimization Techniques

We also conduct ablation studies on our proposed detailed-routability optimization techniques. Table 4.2 lists the average DR WL, #DR Vias and #DRVs of different experimental settings. The results demonstrate the efficiency of the proposed detailed-routability-driven optimization techniques in Xplace-Route. Specifically, the application of cell inflation, M2 rail density insertion, I/O pin density increment, and pin-accessibility post-refinement

Table 4.2: Ablation studies of proposed routability optimization techniques on ISPD 2015 contest benchmarks [10]. CL, D_{m2rail}, D_{iopin}, and PA-RF refer to cell inflation, M2 rail density insertion, I/O pin density increment, and pin-accessibility post-refinement respectively. Note that Xplace-Route enables all the detailed-routability optimization techniques in Section 4.2.

Methods				DR WL/um		#DR Vias		#DRVs	
CL	D_{m2rail}	D_{iopin}	PA-RF	Mean	Ratio	Mean	Ratio	Mean	Ratio
-	-	-	-	9798853	0.999	1847445	1.009	196590	6.216
✓	-	-	-	9796424	0.998	1833652	1.001	37787	1.195
✓	✓	-	-	9808884	1.000	1831921	1.000	34355	1.086
✓	✓	✓	-	9813084	1.000	1832366	1.001	34002	1.075
Xplace-Route (Ours)				9813377	1.000	1831074	1.000	31628	1.000

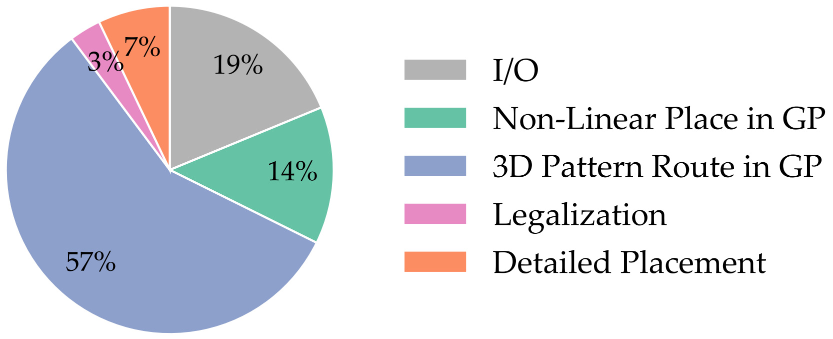


Figure 4.8: Runtime breakdown of Xplace-Route on ISPD 2015 benchmarks

achieve around 502%, 11%, 1%, and 8% #DRVs reduction respectively. Although these techniques may impact the placement solution by increasing HPWL, the final DR WL is only slightly affected, with a slight increase of around 1%. This further demonstrates the effectiveness of our proposed detailed-routability optimization techniques. Note that the cell inflation technique also contributes to the reduction of #DR Vias by 8%.

4.3.4 Runtime breakdown

Figure 4.8 shows the runtime breakdown of Xplace-Route on ISPD 2015 contest benchmarks. We measure the average runtime of different parts among all the cases in ISPD 2015 benchmarks, including I/O time, non-linear placement time in GP, 3D pattern routing time in GP, legalization time, and detailed placement time. We calculate their proportion to the runtime of routability-driven placement. The results show that our non-linear GPU-accelerated placement optimization only takes 14% of the runtime. From the percentage perspective, the pattern router in GP is the most time-consuming part and takes 57% of the runtime. However, from the absolute value perspective, the routing part only takes 23 seconds (the runtime of Xplace-Route is 41 seconds), which is still very effective.

4.4 Concluding Remarks

In this chapter, we present Xplace-Route, an extremely fast detailed-routability-driven placer built upon our GPU-accelerated placement engine Xplace. Equipped with detailed-routability-aware techniques and a GPU-accelerated routing engine, Xplace-Route effectively improves detailed routability and reduces the number of violations. Experimental results on ISPD 2015 contest benchmarks demonstrate that Xplace-Route achieves significant quality improvement and remarkable runtime speedup.

Chapter 5

Device Placement for GPUs

In Chapter 3 and Chapter 4, we explore the possibility of leveraging the power of GPUs to address the VLSI placement problem. In addition to VLSI placement, another placement problem known as device placement is also rapidly gaining attention. Different from VLSI placement which aims to optimize circuit's PPA, the objective of device placement is to accelerate DNN training on a distributed GPU cluster. As DNN models continue to grow in size and hardware topologies become more diverse, modern device placement workflows are typically decomposed into two consecutive stages: model partitioning and device mapping. The major focus of this chapter is to study and optimize this two-stage workflow, thereby effectively tackling the device placement problem on a large-scale GPU cluster. To achieve this goal, we propose Parmesan, an efficient design framework to enhance the pipeline training throughput for operator-level DNNs on systems with general topology. In this chapter, we begin by discussing the motivation behind addressing operator-level graphs and accommodating general device topologies.

5.1 Motivation

Recent pipeline scheduling literature [49, 23, 96, 67, 68, 97, 131, 103] already demonstrates its superiority in accelerating DNN training. Orthogonal to previous works that focus on improving pipeline scheduling, balanced and properly mapped partitions are also important

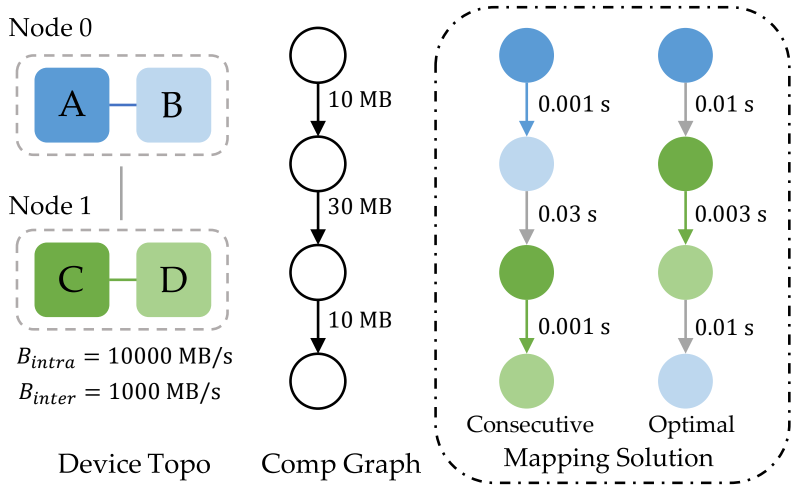


Figure 5.1: An example to show consecutive mapping cannot guarantee optimality.

to maximize training throughput. Although some existing works [121, 96, 122, 23, 136] explore mechanisms such as dynamic programming-based model partitioning and heuristic-based device mapping, they only consider layer-level graphs and flattened/hierarchical topologies without considering the potential issues behind them.

Layer-level partitioning lacks flexibility and requires pre-processing. Layer-level partitioning assumes that the number of layers is larger than the number of stages. However, this may not be true in some cases. For example, some specialized hardware has many cores with limited memory for each core, so a large number of partitions are required to fully utilize the cores. In such cases, higher flexibility can only be provided at the operator-level (op-level). Besides, since the intermediate representation (IR) graph of modern DNN compilers/frameworks is at op-level instead of layer-level, a non-trivial pre-processing is needed to generate the layer-level graph from the op-level. Such pre-processing also varies among different DNNs, which brings extra development effort. Hence, balancing the workloads among all stages and effectively handling the op-level graph should be considered when partitioning a DNN.

Heuristic-based device mapping cannot be generalized to different hardware architectures. The most commonly-used heuristic for device mapping is to put consecutive stages of a partitioned DNN training on consecutive devices (we call it consecutive mapping). Although consecutive mapping works well in some cases, it cannot guarantee good performance. We demonstrate this using a simple example of mapping four stages on a 2×2 hierarchical topology, shown in Figure 5.1. The total communication latency of consecutive mapping is around $1.4\times$ larger than the optimal solution. Admittedly, one may develop heuristics for device mapping to produce a near-optimal performance on some commonly-used hierarchical GPU architectures, but there are many dedicated hardwares other than GPUs for DNN training today which are of very different networking topologies or even integrating multiple processors and interconnections on a single chip (e.g., Google TPU, Cerebras CS-1). Existing heuristics may struggle in those scenarios, and new heuristics are needed to be invented. A good device mapping algorithm should be able to solve the general device topology mapping problem optimally. Such a general topology mapping algorithm can also assist architecture designers in estimating their hardware performance during the design loop, and enable them to design a more DNN-friendly hardware architecture.

In light of the above, we proposed Parmesan, an efficient middleware for large-scale pipeline training based on the PyTorch. Given an *arbitrary* operator-level DNN (expressed as a directed acyclic graph) and general device topology as inputs, Parmesan automatically optimizes the pipeline training throughput in an end-to-end manner.

To the best of our knowledge, Parmesan is the first work to formulate general device mapping problem for pipeline parallelism. Our device mapping formulation considers general device topology, i.e., arbitrary topology with heterogeneous interconnect bandwidths, which can be proved as an NP-complete problem. To make the above challenges solvable, Parmesan decouples the model partitioning and device mapping problems, resulting in a two-phase optimization engine as depicted in Figure 5.2. In the model partitioning stage, Parmesan performs an operator-level optimization based on dynamic programming and two well-designed techniques, considering the computation and communication overheads due to the device hardware constraints. In the device mapping stage, Parmesan conducts customized searching with an effective pruning strategy to find out the optimal placement solution efficiently. Different from previous methods, the proposed device mapping mechanism is capable of handling general topology with heterogeneous interconnect bandwidths.

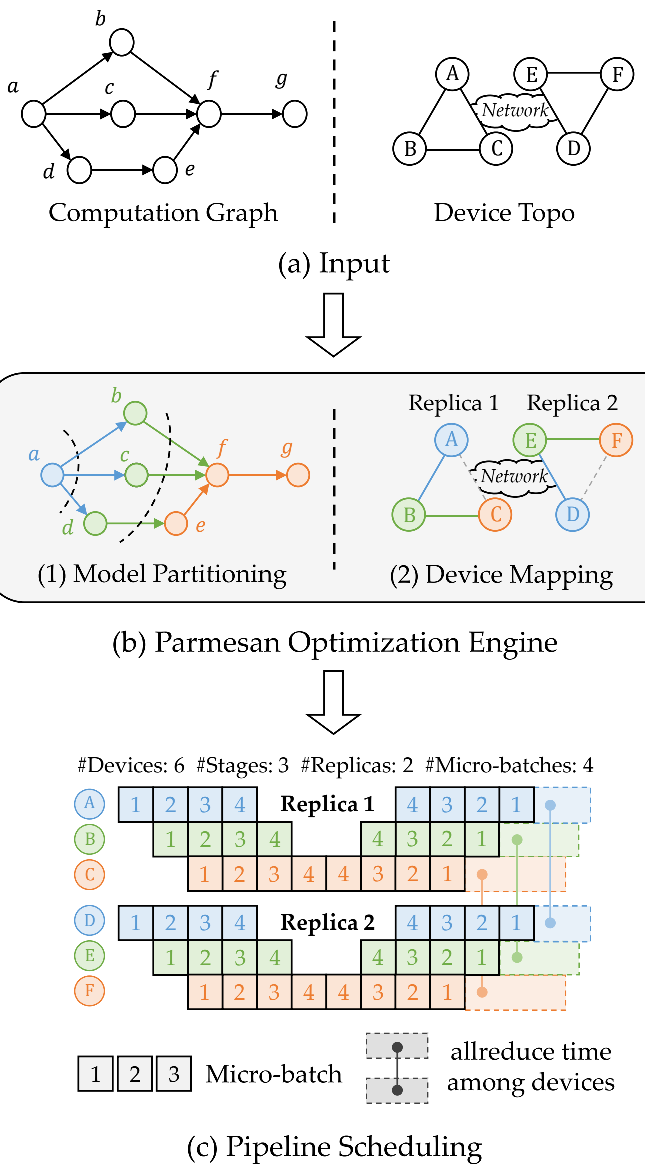


Figure 5.2: The workflow of optimizing the pipelined training throughput of using Parmesan.

Experimental results on real-world GPU clusters and simulations on non-hierarchical topologies indicate the effectiveness of Parmesan.

5.2 Overview of Parmesan

Given a neural network (NN) and a distributed system of GPU devices with heterogeneous interconnect bandwidths, our goal is to maximize the throughput of *hybrid* parallel training for the NN. We use the term *hybrid* to denote pipelined training combined with data parallelism. Since the throughput can be affected by many factors (e.g., how to partition the NN, how to map the partitions onto GPU devices) and the overall optimization problem is over-complicated, we decouple the problem into two phases, namely model partitioning (in Section 5.3) and device mapping (in Section 5.4).

The aim of the model partitioning, is to divide the NN computational graph into S connected subgraphs (stages) with R data parallel replicas per stage to fully utilize the provided GPU resources. In this phase, the heterogeneity of the interconnects is omitted for simplicity and constant bandwidth is assumed so that the problem can be transformed into one that can be optimally solved through a dynamic programming (DP) based approach. To further mitigate the high complexity of the DP while maintaining the solution quality, two techniques are applied before and after the DP. In the device mapping, the accurate interconnect bandwidths between devices will be considered. The NN partitions obtained in phase one are placed into the system, one partition per GPU device, using a well-designed nested searching approach that is optimal and is empirically shown to be efficient.

A discussion of decoupling the problem into two phases is given in Section 5.5. Section 5.6 discusses the operator-level graph extractor, profiler, and the evaluation modules.

5.3 Operator-Level Model Partitioning

Given an NN represented as a Directed Acyclic Graph (DAG), $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of operators (e.g. convolutions, additions, etc.) and $\mathcal{E}$ denotes the set of operator dependencies, the objective of our model partitioning algorithm is to find a partitioning solution to maximize the throughput of a pipeline training. Note that the objective is equivalent to finding a solution to minimize the maximum stage time [96], given by

$$\max_{\mathcal{S}} \text{ throughput} = \min_{\mathcal{S}} \max_{s \in \mathcal{S}} \{c_{\mathsf{u}}(s) + c_{\mathsf{m}}(s)\} \tag{5.1}$$

where $c_{\mathsf{u}}(s)$ and $c_{\mathsf{m}}(s)$ denote the computation time (including forward and backward execution time) and the communication time of stage s respectively, and $\mathcal{S}$ denotes a set of stages (i.e. a partitioning solution). Each stage $s \in \mathcal{S}$ contains a group of consecutive operators from $\mathcal{V}$.

The solution of model partitioning will be a set of disjoint stages $\mathcal{S}^*$ while satisfying: (a) $\mathcal{V} = \bigcup_{s \in \mathcal{S}^*} s$, (b) each stage s contains a group of consecutive operators from $\mathcal{V}$, (c) the total memory of each stage is less than the device memory. Besides, a stage-level graph $\mathcal{G}_S$ consisting of all the stages $s \in \mathcal{S}^*$ as vertices and inter-stage communications as edges is constructed and passed to the next phase (i.e. device mapping).

To solve Problem (5.1) and find a partitioning solution $\mathcal{S}^*$, several past works introduce dynamic programming-based layer-level model partitioning approaches [23, 96, 122]. However, the practicality of layer-level partitioning is limited by some critical concerns outlined in Section 5.1 (e.g., low flexibility and costly pre-processing). [121] introduces a scheme to partition the operator-level graph but yields solutions with limited qualities. In this section, we firstly introduce a DP formulation extended from [96] to tackle the operator-level partitioning problem in Section 5.3.1 and then discuss two techniques in Section 5.3.2 and Section 5.3.3 to reduce the complexity of the DP while maintaining solution quality. The details of searching the number of stages S and the number of replicas R are given in Section 5.3.4.

5.3.1 Dynamic Programming

Definition 1. A subgraph $\mathcal{H}(\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is induced if for any two vertices $u, v \in \mathcal{V}_{\mathcal{H}}$, $(u, v) \in \mathcal{E}_{\mathcal{H}}$ if and only if $(u, v) \in \mathcal{E}$.

Definition 2. An induced subgraph $\mathcal{F}(\mathcal{V}_{\mathcal{F}}, \mathcal{E}_{\mathcal{F}})$ of a directed acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a fronted subgraph if for any vertex u in $\mathcal{V}_{\mathcal{F}}$, all predecessors of u in $\mathcal{V}$ are also in $\mathcal{V}_{\mathcal{F}}$.

Definition 3. We call a graph set $F_{\mathcal{G}}$ a *fronted subgraph set* if it contains all possible fronted subgraphs of $\mathcal{G}$ (i.e. $F_{\mathcal{G}} = \{\mathcal{F} \mid \mathcal{F} \text{ is a fronted subgraph of } \mathcal{G}\}$)

Theorem 1. For a DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$, $|F_{\mathcal{G}}| \geq |\mathcal{V}|$.

Theorem 2. If a connected DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$ has a unique topological ordering, $|F_{\mathcal{G}}| = |\mathcal{V}|$.

To solve Problem (5.1) by dynamic programming, we first define a DP table as $T \in \mathbb{R}^{(|F_{\mathcal{G}}|+1) \times D \times S}$, which represents the best timing (computation and communication) achievable to partition the computational graph represented by the first parameter with d devices and s stages where $d \leq D$ and $s \leq S$. We initialize $T_{\emptyset,d,s} = 0$ for any d, s. Each item $T_{\mathcal{F},d,s} \in T$, which represents the optimal solution of partitioning a fronted graph $\mathcal{F} \in F_{\mathcal{G}}$ to s stages with assignment to d devices, is given by

$$T_{\mathcal{F},d,s} = \min_{\mathcal{F}' \in F_{\mathcal{F}} \setminus \{\mathcal{F}\}} \min_{d'=s-1}^{d-1} \max\{T_{\mathcal{F}',d',s-1}, t_{\mathcal{F}-\mathcal{F}',d-d'}\} \tag{5.2}$$

In the above formulation, the subgraph $\mathcal{F} - \mathcal{F}'$ is assigned to stage s with $d - d'$ devices (replicas). Note the range of d' represents the minimum and maximum number of devices required to partition the fronted subgraph $\mathcal{F}'$. The term $t_{\mathcal{F}-\mathcal{F}',d-d'}$ denotes the stage time of s, formulated as

$$t_{\mathcal{F}-\mathcal{F}',d-d'} = \sum_{v \in \mathcal{V}_{\mathcal{F}-\mathcal{F}'}} \{\frac{c_{\mathsf{u}}(v)}{d-d'} + \sum_{v' \in \mathrm{adj}(v) \setminus \mathcal{V}_{\mathcal{F}-\mathcal{F}'}} \frac{c_{\mathsf{m}}(v,v')}{d-d'}\} \tag{5.3}$$

where $c_{\mathsf{u}}(v)$ is the computation time for operator v and $c_{\mathsf{m}}(v, v')$ denotes the communication time between operator v and operators v', and $\mathrm{adj}(v)$ denotes v's adjacent operators (i.e., a set of all operators that directly communicate with operator v) . Note that if the memory consumed by $\mathcal{F} - \mathcal{F}'$ is larger than $(d - d') \times DM$, where DM is the device memory, we will put $t_{\mathcal{F}-\mathcal{F}',d-d'} = +\infty$. Note that real-world training with different number of devices (replicas) for each stage will invoke expensive collective communication operators. We thus empirically put $d - d'$ a constant R, and detailed discussion will be provided in Section 5.3.4.

For all $\mathcal{F} \in F_{\mathcal{G}}$, we recursively compute Equation (5.2) and then fill up the table T. The optimal value, $\min \max_{s \in \mathcal{S}} t(s)$ ($t(\cdot)$ denotes the stage time), is naturally given by $T_{\mathcal{G},D,S}$. The optimal solution $\mathcal{S}^*$ will then be derived from the computed table T.

All possible stage times described in Equation (5.3) can be pre-computed in $O(2^{|\mathcal{V}|}D)$ time. Under the assumption that $\mathcal{G}$ is a sparse DAG (i.e. the average degree of vertices $\overline{d} = O(|\mathcal{V}|)$), the dynamic programming in Equation (5.2) can run in $O(2^{|\mathcal{V}|}D^2S)$ time. If we assume that $\mathcal{G}$ is a connected DAG with a unique topological ordering (which is usually the case in practice) so that Theorem 2 applies, the complexities of these two steps are $O(|\mathcal{V}|^2D)$ and $O(|\mathcal{V}|^2D^2S)$ respectively.

5.3.2 Operator Clustering

The dynamic programming can optimally solve the partitioning problem but the running time will be extremely long if the input computation graph is large. To handle this scalability issue while maintaining quality, operator clustering will be performed before the dynamic programming process. It is observed that most of the operators in $\mathcal{V}$ are lightweight and can be clustered with other operators that are topologically closed to form hyper-operators while balancing the stage time. To this end, we perform hyper-operator merging **before** DP, called *operator clustering*, on the vanilla computation graph $\mathcal{G}$ to significantly accelerate the DP.

Now, we introduce the details of this operator clustering. We maintain a hyper-operator graph $\hat{\mathcal{G}}_k(\hat{\mathcal{V}}_k, \hat{\mathcal{E}}_k)$ during the operator clustering, where $\hat{\mathcal{V}}_k$ denotes a set of hyper-operator nodes, $\hat{\mathcal{E}}_k$ denotes their dependency edges, and k is the number of hyper-operators in $\hat{\mathcal{V}}_k$. At the beginning, this hyper-operator graph is simply initialized from the given operator graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and k is equal to $|\mathcal{V}|$.

Given an operator graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, we initialize any operator $v \in \mathcal{V}$ as an atomic-level hyper-operator $\hat{v} = \{v\}$ which only contains v itself. The initial hyper-operator set is then naturally given by $\hat{\mathcal{V}}_{|\mathcal{V}|} = \{\hat{v} \mid \hat{v} = \{v\}, \forall v \in \mathcal{V}\}$, and the edge set $\hat{E}_{|\mathcal{V}|}$ is constructed correspondingly. Starting with $\hat{\mathcal{G}}_{|\mathcal{V}|}(\hat{\mathcal{V}}_{|\mathcal{V}|}, \hat{\mathcal{E}}_{|\mathcal{V}|})$, Parmesan will recursively cluster hyper-operators. One step of clustering for a graph $\hat{\mathcal{G}}_k(\hat{\mathcal{V}}_k, \hat{\mathcal{E}}_k)$ will fuse two adjacent hyper-operators $\hat{u}, \hat{v} \in \hat{\mathcal{V}}_k$ into one with update of the corresponding dependencies in $\hat{\mathcal{E}}_k$, forming a new graph $\hat{\mathcal{G}}_{k-1}(\hat{\mathcal{V}}_{k-1}, \hat{\mathcal{E}}_{k-1})$, given as follows:

1. Select two adjacent hyper-operators $\hat{u}, \hat{v} \in \hat{\mathcal{V}}_k$ with a minimum cost in Equation (5.4) under the convexity and memory constraints.

2. Merge $\hat{v}$ into $\hat{u}$ ($\hat{u} \leftarrow \hat{u} \cup \hat{v}$) and accumulate their operator attributes like computation time and consumed memory, etc.

3. Let $\hat{\mathcal{V}}_{k-1} \leftarrow \hat{\mathcal{V}}_k \setminus \{\hat{v}\}$ and $\hat{\mathcal{E}}_{k-1} \leftarrow \hat{\mathcal{E}}_k \cup \{(\hat{u}, \hat{w}) \mid (\hat{v}, \hat{w}) \in \hat{\mathcal{E}}_k\} \cup \{(\hat{w}, \hat{u}) \mid (\hat{w}, \hat{v}) \in \hat{\mathcal{E}}_k\} \setminus \{(\hat{u}, \hat{v})\}$.

Each step of operator clustering eliminates the communication edge between $\hat{u}$ and $\hat{v}$ and leads to a relatively larger hyper-operator with accumulated computational cost. Note that we can consider a hyper-operator as one kind of operators because it also supports attributes like computation time, communication size and memory, etc.

We will next describe the criteria of selecting two adjacent hyper-operators $\hat{u}, \hat{v}$ to cluster. Before discussing the selection criteria, we first give the definition of subgraph convexity.

Definition 4. A subgraph $\mathcal{H}$ of a directed acyclic graph $\mathcal{G}$ is convex if for any two vertices $u, v \in \mathcal{H}$, there is no directed path between u, v in $\mathcal{G}$ lying outside $\mathcal{H}$.

To maintain $\hat{\mathcal{G}}_{k-1}$ as acyclic after clustering, it is not hard to see that the operators $\hat{u}, \hat{v} \in \hat{\mathcal{V}}_k$ selected to be clustered must be such that the induced subgraph $\mathcal{H}$ with $\mathcal{V}_{\mathcal{H}} = \{\hat{u}, \hat{v}\}$ of the DAG $\hat{\mathcal{G}}_k$ satisfies the **convexity constraint**. Besides, the **memory constraint** should also be satisfied, that is, the total memory consumed by the resulted hyper-operator should not exceed the device memory.

To balance the computation cost and reduce the communication cost, Parmesan first enumerates all valid operator pairs that satisfy the two aforementioned constraints and it will then cluster the target pair $(\hat{u}, \hat{v})$ with the smallest cost, given by

$$\text{cost}(\hat{u}, \hat{v}) = c_{\mathsf{u}}(\hat{u}) + c_{\mathsf{u}}(\hat{v}) - \alpha c_{\mathsf{m}}(\hat{u}, \hat{v}) \tag{5.4}$$

where α indicates relative importance between computation and communication cost. The value of α is automatically selected from 0.01, 1, and 100 with the assistance of the offline throughput simulator. Starting from the vanilla graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, Parmesan recursively performs the operator clustering until the total number of hyper-operators left is no more than a parameter K, which is set to a value that is $S \lll K \lll |\mathcal{V}|$. The resultant hyper-operator graph $\hat{\mathcal{G}}_K(\hat{\mathcal{V}}_K, \hat{\mathcal{E}}_K)$ will then be input to DP. As $K \gg S$, operator clustering will not significantly affect DP to yield computation balanced yet communication reduced stages. Besides, $K \lll |\mathcal{V}|$, the whole operator clustering step will thus run in $O(|\mathcal{V}||\mathcal{E}|)$ time and the complexity of the DP in Section 5.3.1 is significantly reduced to $O(2^K D^2 S)$. It is also found that the topological ordering of $\hat{\mathcal{G}}_K$ is unique in most of the cases and the DP will thus run in $O(K^2 D^2 S)$ time according to Theorem 2.

5.3.3 Iterative Refinement

After operator clustering and DP, an iterative refinement will be performed to further improve the partitioning result. Iterative refinement aims at reducing the communication time and balancing the computation time by fine tuning the DP result.

In iterative refinement, atomic operators on a boundary, i.e., operators that have at least one edge connecting to an operator in another stage, may be moved to a neighboring stage. Each refinement step will first find out all the valid move candidates. A move is a tuple that consists of two elements: operator to be moved and its target stage. Only atomic operators on a boundary will be considered, and invalid moves that lead to non-convexity or memory violation will be filtered out. After finding out all the valid candidates, Parmesan will calculate their movement gain according to the following three metrics:

1. decrease in the shortest path distance to the nearest reconverging operator (A reconverging vertex in a DAG is a cut vertex with at least one of its in-degree or out-degree larger than one.)

2. decrease in the total communication size

3. decrease in the maximum stage computation time.

Note that the gain values are the higher the better for all the three metrics and the $i+1$-st metric is used as a tie breaker for the i-th metric. A move with the highest gain is selected to perform one step of iterative refinement.

Parmesan iteratively refines the DP solution until no valid move can be chosen, or a limit I for the maximum number of refinement step is reached. We set I as 100. Since $I \ll |\mathcal{V}|$, the iterative refinement will thus run in $O(|\mathcal{V}|(|\mathcal{V}| + |\mathcal{E}|))$ time.

While operator clustering is a bottom-up approach to handle scalability, iterative refinement is a top-down approach to consider explicitly the influence of a move on stage partitioning to improve quality. It can remarkably mitigate the sub-optimality brought by operator clustering and can enhance the solution quality significantly.

5.3.4 Implementation Details of Operator-Level Model Partitioning

In Section 5.3.1, we discuss a dynamic programming algorithm to partition a computation graph into S stages with R replicas. And we discuss two well-designed techniques, namely operator clustering (in Section 5.3.2) and iterative refinement (in Section 5.3.3). Thus, the whole model partitioning flow in Parmesan is: (1) operator clustering, (2) dynamic programming and (3) iterative refinement. In this subsection, we will discuss several implementation details of model partitioning.

Dynamic programming for specific (S, R) configuration We first give the definition of stage replica number. Stage $s \in \mathcal{S}$ has R_s stage replicas implying there are R_s devices maintaining a replica of the entire stage s. In Section 5.3.1, we introduce a dynamic programming algorithm that supports a different replica number R_s (i.e. $d - d'$ in Equation (5.2)) for each stage s. However, employing different replica numbers for each stage introduces two additional collective communication operators, *allscatter* and *allgather*, for transmitting tensors between two adjacent stages, which will cause high communication overhead. Such communication overhead will negatively affect the real training throughput, especially for synchronous pipeline training. Therefore it is better to have the same stage replica number for all stages. In our implementation, we consider R_s as a constant R among all the stages so the DP in Equation (5.2) can be simplified as follows,

$$T_{\mathcal{F},d,s} = \min_{\mathcal{F}' \in F_{\mathcal{F}} \setminus \{\mathcal{F}\}} \min_{\substack{\forall d' \in [s-1, d-R] \\ s.t.\ d' \in \mathbb{N}}} \max\{T_{\mathcal{F}',d',s-1}, t_{\mathcal{F}-\mathcal{F}',R}\} \tag{5.5}$$

With Equation (5.5), our dynamic programming can optimally partition the DNN into S stages with R replicas. Note that Equation (5.5) solves a sub-problem of Equation (5.2). Since the extra collective communication overhead caused by having different stage replicas is expensive in real-world training, Equation (5.5) is applied in Parmesan.

Selection criteria for S **and** R As for searching the best configuration (S, R), we develop a simulator to estimate the pipeline training latency offline. We can then choose a configuration having the shortest estimated latency as the best configuration. Note that the pipeline bubble is also considered during the simulation. Since different configurations can be executed independently, such a search can be well-parallelized by multi-threading and can be finished within a reasonable amount of time.

Peak Memory Consumption Model Given the number of micro-batches MB and a computation graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the peak memory consumption of a subgraph $\mathcal{H} \subseteq \mathcal{G}$ is given as,

$$m(\mathcal{H}) = \frac{1}{MB} \sum_{v \in \mathcal{V}_{\mathcal{H}}} \{\max\{m_{\mathsf{fwd}}(v), m_{\mathsf{bwd}}(v)\} + \sum_{\substack{v' \in \mathsf{adj}(v) \\ v' \notin \mathcal{V}_{\mathcal{H}}}} M(v, v')\} \tag{5.6}$$

where $m_{\mathsf{fwd}}(v)$ and $m_{\mathsf{bwd}}(v)$ denote operator v's peak memory consumption during forward and backward propagation respectively, $M(v_i, v_j)$ denotes the communication size between operator v_i and operator v_j. This model approximates the peak memory usage of a subgraph $\mathcal{H} \subseteq \mathcal{G}$ in real-world training with *activation recomputing*.

5.4 Device Mapping for General Device Topology

In the previous phase of model partitioning, a stage-level NN graph $\mathcal{G}_S$ containing S vertices along with a constant R, indicating the number of replicas per stage are computed. For simplicity, we define a new graph $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ that is composed of R identical copies of $\mathcal{G}_S$. The objective of the device mapping problem is to obtain a bijective mapping $p : \mathcal{V}' \to \mathcal{D}$ that assigns each $s \in \mathcal{V}'$ to a unique device $d \in \mathcal{D}$ under the heterogeneous network settings, such that the min-max stage time objective

$$\min_{p} \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p) \tag{5.7}$$

is optimized. Note that $c_{\mathsf{stage}}(s, p)$ is a general notation of the stage turnaround time under a mapping p and can be instantiated differently under different scenarios.

In this section, we firstly introduce a general search algorithm that optimally finds a solution for Problem (5.7) (Section 5.4.1), and then discuss two instantiations of $c_{\mathsf{stage}}(s, p)$ as well as the corresponding search algorithm (Section 5.4.2). Equipped with a properly designed selection criterion, a combination of these two algorithms are able to efficiently produce high quality mappings for different NNs.

5.4.1 A General and Optimal Searching Algorithm

The method shown in Algorithm 2 is formulated as a nested two-level searching. The outer-level is essentially a binary search that manages an interval $[t_l, t_r]$ in which the optimal value of Problem (5.7) resides. In each iteration, the inner-level search is invoked to find whether there exists a mapping p such that the maximum stage time $t_{\max}(p) := \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p) \leq t$, where t is the mid-point of the interval. The interval shrinks according to the existence of such p at an exponential rate with respect to the number of invocations of the inner-level search. When the length of the interval becomes small enough and no more feasible mappings can be found, the algorithm returns the last found mapping.

The inner-level is a recursive, depth-first search based backtracking algorithm. A partial

Algorithm 2 Device Mapping

```
**Input:** $\mathcal{G}'$ with $c_u$ and message size as vertex and edge attributes, bandwidth $B$ between each pair of devices.
Compute initial lower bound $t_{l0}$, upper bound $t_{r0}$
$p \leftarrow \emptyset,\ t_l \leftarrow t_{l0},\ t_r \leftarrow t_{r0}$
while $t_r - t_l > \epsilon > 0$ do
    $t \leftarrow (t_l + t_r)/2$
    $p, t_p \leftarrow \text{search}(\mathcal{G}', B, t)$
    if $p = \emptyset$ then $t_l \leftarrow t$
    else $t_r \leftarrow \min\{t, t_p\}$
return $p$
```

mapping p is taken as an input state and for each time the algorithm is invoked, it tries to assign the next unmapped stage s by checking all the unallocated devices. For each candidate device d, before starting a new recursive pass with the mapping $p \cup \{(s,d)\}$, the algorithm firstly checks the stage times $c_{\text{stage}}(s', p \cup \{(s,d)\})$ of some previously assigned stages s' that can only be determined after s is assigned. The planning of what stages to be checked when assigning s is called the checking scheme of s. It only depends on the connection edges in $\mathcal{G}'$ and the instantiation of $c_{\text{stage}}(s,p)$, so it can be precomputed before the overall mapping algorithm. If any of the stage time is larger than the target t, any mapping that includes $p \cup \{(s,d)\}$ is infeasible and need not be further enumerated and checked. This is an important pruning technique to ensure the practical effectiveness of the inner-level search. The recursion terminates when a full mapping is found. A detailed description of this algorithm is given by Algorithm 3.

We provide a proof in Section 5.8.3 showing that Algorithm 2 always gives the optimal solution, provided that the initial interval contains the optimal value.

5.4.2 Two Instantiations

In the scenario of hybrid pipeline training, a natural thought on formalizing $c_{\text{stage}}(s,p)$ is to include the computational time, inter-stage communication time and inter-replica allreduce time[1], i.e., $c_{\text{stage}}(s,p) = c_u(s) + c_m(s,p) + c_{AR}(s,p)$. Despite its theoretical exactness, this

[1]In this chapter, we mainly consider the ring-allreduce.

Algorithm 3 Inner-level Search, search$(\mathcal{G}', B, t)$

```
**Input:** Graph $\mathcal{G}'$ with $c_\mathsf{u}$ and $c_\mathsf{m}$ as vertex and edge attributes, bandwidth $B$ between each pair of devices, target of the maximum stage time $t$.
**function** dfs($(\mathcal{G}', B, t, p, t_p)$)
    **if** $|p| = |V(\mathcal{G}')|$ **then return** $(p, t_p)$
    $s \leftarrow$ mapping_order$(\mathcal{G}')[|p|]$
    $t_p^{\text{old}} \leftarrow t_p$
    **for all** $d$ not assigned a stage in $p$ **do**
        satisfied $\leftarrow$ true
        **for all** $s' \in$ checking_scheme$(s)$ **do**
            $t_{s'} \leftarrow c_\mathsf{stage}(s', p \cup \{(s, d)\})$
            **if** $t_{s'} > t$ **then**
                satisfied $\leftarrow$ false; **break**
            $t_p \leftarrow \max\{t_p, t_{s'}\}$
        **if** satisfied **then**
            $p^{\text{res}},\ t_p^{\text{res}} \leftarrow$ dfs$(\mathcal{G}', B, t, p \cup \{(s, d)\}, t_p)$
            **if** $p^{\text{res}} \neq \emptyset$ **then return** $(p^{\text{res}}, t_p^{\text{res}})$
        $t_p \leftarrow t_p^{\text{old}}$
    **return** $(\emptyset, 0)$
$s_0 \leftarrow$ mapping_order$(\mathcal{G}')[0]$
**for all** $d \in \mathcal{D}$ **do**
    $p, t_p \leftarrow$ dfs$(\mathcal{G}', B, t, \{(s_0, d)\}, 0)$
    **if** $p \neq \emptyset$ **then return** $(p, t_p)$
**return** $(\emptyset, 0)$
```

formulation imposes many constraints on the checking scheme used in Algorithm 3 that the time of a particular stage cannot be determined until a considerable number of later stages have also been assigned. This in turn will lead to delayed pruning and inefficient search, and thus can be hardly applied in large-scale settings that hundreds of devices are involved. To handle this issue, we propose two kinds of instantiation for $c_\mathsf{stage}(s, p)$ such that they together provides mapping results of similar qualities as the aforementioned formulation but consumes less computational time in practice.

A key observation is that the ratio between inter-stage communication cost c_m (determined by intermediate feature maps) and inter-replica communication cost c_AR (determined by the number of NN parameters) varies with different $\mathcal{G}'$. For example, in general, CNN image models have fewer parameters than Transformer-based language models thanks to convolutions, but their intermediate tensors are larger due to the 2D nature of images. In light of this, we design the two instantiations that apply to the inter-stage and inter-replica dominant cases respectively as follows:

$$c^{\mathsf{m}}_{\mathsf{stage}}(s,p) = c_{\mathsf{u}}(s) + c_{\mathsf{m}}(s,p), \tag{5.8}$$

$$c^{\mathsf{AR}}_{\mathsf{stage}}(s,p) = c_{\mathsf{u}}(s) + c_{\mathsf{AR}}(s,p), \tag{5.9}$$

$$c_{\mathsf{m}}(s,p) = \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(p(s),p(s'))}, \tag{5.10}$$

$$c_{\mathsf{AR}}(s,p) = \max_{\substack{s_i,s_{i+1}\\ \in \mathrm{ring}(\mathrm{repl}(s))}} \left\{ 2\frac{R-1}{R} \frac{P(s)}{B(p(s_i),p(s_{i+1}))} \right\} \tag{5.11}$$

where $M(s,s')$ is the communication size between stage s and s', $B(d_1,d_2)$ denotes the bandwidth between device d_1 and d_2, $P(s)$ is the total parameter size in s and $\mathrm{ring}(\mathrm{repl}(s))$ represents the set of adjacent pairs of replicas that appear in the allreduce ring of s (e.g., if $\mathrm{repl}(s) = \{s_0,s_1,s_2\}$, then $\mathrm{ring}(\mathrm{repl}(s)) = \{(s_0,s_1),(s_1,s_2),(s_2,s_0)\}$).

With the partial assignment of devices to stage $s \in \mathcal{V}'$ and its adjacencies, $\min_d c_{\mathsf{stage}}(s,d)$ can be derived and it is always smaller than or equal to the optimal value of Problem (5.7). Thus, the *initial lower bounds* t_{l0} in Algorithm 2 of the two instantiations are computed respectively as

$$t^{\mathsf{m}}_{l0} = \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \min_{d} \min_{p^{\mathsf{m}}_{s,d}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(d,p^{\mathsf{m}}_{s,d}(s'))} \right\} \tag{5.12}$$

and

$$\begin{aligned} t^{\mathsf{AR}}_{l0} &= \max_{s \in \mathcal{V}'} \{ c_{\mathsf{u}}(s) + \min_{d} c^{\mathsf{min}}_{\mathsf{AR}}(s,d) \}, \\ c^{\mathsf{min}}_{\mathsf{AR}}(s,d) &= \min_{p^{\mathsf{AR}}_{s,d}} \max_{s' \in \mathrm{adj\_repl}(s)} \left\{ 2\frac{R-1}{R} \frac{P(s)}{B(d,p^{\mathsf{AR}}_{s,d}(s'))} \right\}, \end{aligned} \tag{5.13}$$

where $p^{\mathsf{m}}_{s,d} : \mathrm{adj}(s) \to \mathcal{D} \setminus \{d\}$ (resp. $p^{\mathsf{AR}}_{s,d} : \mathrm{adj\_repl}(s) \to \mathcal{D} \setminus \{d\}$) represents a partial assignment of the adjacent stages (resp. two adjacent replicas on the ring) of s to devices other than d. These can be computed in polynomial time using a sorting-based approach. The *initial upper bounds* are generated by taking the minimum value between a random mapping and a heuristic mapping.

Proposition 1. $t^{\mathsf{m}}_{l0} \leq \min_p \max_{s \in \mathcal{V}'} c^{\mathsf{m}}_{\mathsf{stage}}(s, p)$ and $t^{\mathsf{AR}}_{l0} \leq \min_p \max_{s \in \mathcal{V}'} c^{\mathsf{AR}}_{\mathsf{stage}}(s, p)$. Hence, the initial interval obtained as above always contain the optimal value for the respective instantiation.

Although these two instantiations of $c_{\mathsf{stage}}(s, p)$ enables effective pruning during the inner-level search and make the search trees shallower, in the worst case the inner-level search still has a complexity of $O(D!)$ (as the mapping problem is NP-complete). Therefore, we perform an enhancement and a heuristic on Algorithm 2 to further accelerate the overall device mapping. The enhancement is to launch a batch of inner-level searches with different targets (t) using multi-threading. The targets are selected such that the interval is evenly divided. The lower bound would be updated as the largest value of all the not-found targets, and the upper bound would be the smallest value of all the found targets.

As a heuristic, we enable timeout to prevent search tasks from running over long, and the results for the early-stopped threads are regarded as not found. Admittedly, this heuristic would affect the optimality of device mapping, but as the experiments in Section 5.7.4 show, in general, the resulting quality is rarely affected.

For a certain $\mathcal{G}'$, one of the two instantiations is automatically selected based on the ratio between the allreduce communication size and the total inter-stage communication size. If this ratio is larger than one, $c^{\mathsf{AR}}_{\mathsf{stage}}(s, p)$ will be applied, otherwise $c^{\mathsf{m}}_{\mathsf{stage}}(s, p)$ will be applied.

5.5 Discussions

Theorem 3. The device mapping problem is NP-complete.

As Theorem 3 shown, the device mapping problem (phase 2) for general topology is an NP-complete problem. If we consider the various bandwidth inside the general topology during model partitioning (phase 1), that is, do phase 1 and phase 2 simultaneously, the overall complexity will be too high to be solved effectively. To make the problem solvable, we decouple the whole problem into two phases (model partitioning and device mapping). However, such decoupling unavoidably brings an issue to phase 1: we cannot foresee which two devices a partition and its adjacent partition will be mapped to, so the bandwidth between these two devices is hard to determine beforehand, especially for a general topology. However, the communication size between two adjacent partitions can be captured.

As a result, in phase 1, we assume the flattened device topology and aim at balancing the computation time while reducing the communication size. In phase 2, we take the device topology into account explicitly, and optimally map the partitions generated by phase 1 onto a user given general topology, and consider the various communication bandwidths between different devices. Empirically, such a two-phase paradigm works well within an acceptable runtime.

5.6 Implementation

We implement Parmesan with PyTorch [104], Numba [64], and NetworkX [36]. The core of Parmesan contains around 12k lines of Python. In this section, we will introduce other components in Parmesan. To simplify the notation, we will use the word *optimizer* to denote the whole optimization process (including model partitioning and device mapping).

Figure 5.3 describes the whole workflow of Parmesan. Given a DNN model written in PyTorch, our graph extractor will first conduct just-in-time (JIT) tracing and automatically extract the operator-level computation graph. We will then launch the profiler to profile the operator attributes (including forward/backward time/memory) and compute the total parameter size of this operator (allreduce time is highly correlated to the parameter size).

As for the device topology, we first adopt our communication profiler to measure the point-to-point (*p2p*) communication overhead between every pair of devices. Then we construct a p2p bandwidth look-up-table (LUT) based on the postal model [8, 28] to represent the device topology.

After building the operator-level computation graph and the device topology LUT, our optimizer will take them as input and output the partitioning result and mapping plan.

Parmesan's pipeline scheduler and simulator evaluate the solution quality and provide instructive information for further development. Inspired by FlexFlow [50], we develop

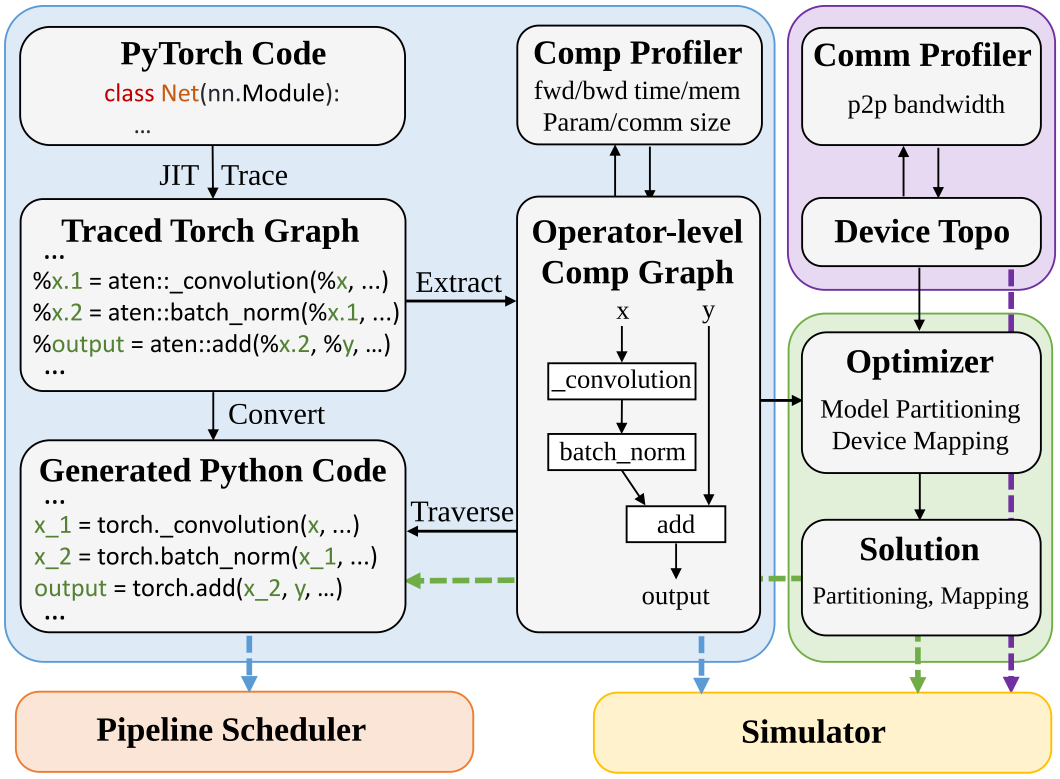


Figure 5.3: The whole workflow of Parmesan.

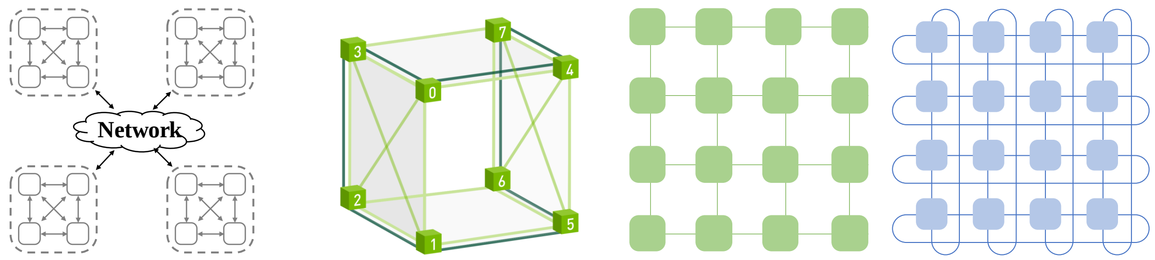
a task-graph-based simulator to tackle the general device topology simulation problem. Our simulator considers pipelined forward/backward propagations, pipeline bubbles, and gradient allreduce. For the real-world evaluation, we design a GPipe [49] fashion pipeline scheduler in PyTorch with CUDA 11.3 and adopt NCCL 2.10.3 distributed backend for both the p2p communication between pipeline stages and allreduce between the stage replicas. Note that the Python code snippets executed by our pipeline scheduler are automatically generated from the operator-level graph and the optimized solution.

Besides, Parmesan supports writing/reading computation graphs, device topologies, and optimized solutions. Thus, one can further explore some more algorithms/flows and evaluate their performance based on Parmesan. Meanwhile, Parmesan's optimizer and simulator are independent of the deep learning framework. Provided the computation graph extracted by other DL frameworks (like Tensorflow), Parmesan can automatically conduct model partitioning and device mapping for the given network and simulate the solution performance.

5.7 Experimental Results

We evaluate our proposed method through measuring (1) the latency of running a real scheduled pipeline training, and (2) the simulated pipeline running time for non-regular topology. We use a synchronous pipeline[2] where each training step consists of four micro-batches followed by an allreduce and parameter update, with activation recomputing enabled. We mainly conduct experiments on ResNet [37], BERT [21] and Swin Transformer [85]. Two real machine architectures and a series of synthetic architectures are considered in the experiments.

[2]We mainly consider synchronous pipelined training whose training accuracy is not affected theoretically.



(a) Two-level hierarchy (b) intra-DGX-1 connection (c) 2d mesh (d) 2d torus

Figure 5.4: Visualization of different device topologies.

5.7.1 Experimental Settings

In this subsection, we will describe the detailed settings of different device topologies and different NNs.

Device Topologies We consider various device topologies including two-level hierarchical, Alibaba cloud 4x DGX-1, Mesh/torus, and Randomly generated architectures.

Two-level hierarchical architecture is a four-machine system that has two Xeon 4114 CPUs and four Titan V GPUs per node. GPUs are connected by PCI-e intra-node. The inter-node connection is Ethernet. According the profiled results, intra-node bandwidth is 11GB/s, inter-node bandwidth is 1.1GB/s. An illustration of two-level hierarchical architecture is shown in Figure 5.4(a).

Alibaba cloud 4x DGX-1 architecture consists of 4 ecs.gn6e-c12g1.24xlarge virtual machine instances, where each machine contains 8 V100 GPUs. There are three types of bandwidth in intra-node, that is, PCI-e, single NVLinks and double NVLinks, their profiled bandwidth are 10.1GB/s, 21.4GB/s and 43.2GB/s separately. The inter-node connection relies on Ethernet and its profiled bandwidth is 1.4GB/s. Figure 5.4(b) illustrates its intra-node connections[3]. Each corner denotes one V100 GPU, a single edge denotes a single NVLink connection, and a double-edge denotes a double NVLink connection. Device pairs without direct edge in Figure 5.4(b) are connected by PCI-e.

[3]Figure 5.4(b) is from DGX-1 Whitepaper in https://www.nvidia.com/en-us/data-center/dgx-1/.

Table 5.1: Synthesised bandwidths mesh/torus architecture

#Hops	1	2	3	4	5	6	7	8	9	10	11	12
Bandwidth (GB/s)	78.1	39.0	24.4	14.6	9.77	7.81	5.86	4.4	2.93	1.46	0.88	0.78
#Hops	13	14	15	16	17	18	19	20	21-31	31-51	≥ 52	
Bandwidth (GB/s)	0.68	0.59	0.49	0.39	0.29	0.19	0.098	0.098	0.088	0.078	0.068	

Mesh/torus architectures. Since it's difficult for us to access a real mesh/torus kind system, we set the bandwidth between any two devices according to the number of hops between them. Note that Table 5.1 is generated by us as an example and it may not be the same as the real mesh/torus architectures. Examples of 4×4 2d mesh/tours architectures are shown in Figure 5.4(c) and Figure 5.4(d).

Randomly generated architectures. We randomly generated three kinds of fully heterogeneous device topologies, named random_blk_1, random_blk_2 and uniform_dist. (1) random_blk_1 contains several nodes with randomly generated intra-node bandwidth and fixed inter-node bandwidth. Suppose the number of devices is D, we firstly generate a list of random integer $[b_1, b_2, ..., b_L]$ whose summation is D, where L denotes the number of nodes and b_i denotes the number of devices in node n_i. We then randomly generated L intra-node bandwidths for all nodes following a uniform distribution $\mathcal{U}_{[1e-4,1e-2]}$ MB/us (1 GB/s = $\frac{1024}{10^6}$ MB/us) and set the inter-node bandwidth as $1e-4$ MB/us. Note that the only homogeneity in this architecture is the intra-node connection for each node n_i. (2) random_blk_2 also needs a list of random integer $[b_1, b_2, ..., b_L]$ similar to random_blk_1,. Besides, the intra-node bandwidths in random_blk_2 are randomly generated following $\mathcal{U}_{[1e-5,1e-2]}$ MB/us. Note that the intra-node bandwidths for each node n_i are heterogeneous. Then we compute the average value of all the intra-node bandwidths as $\overline{BW}_{intra}$, and let the base inter-node bandwidth as $BW_{inter} = \overline{BW}_{intra}/10$. The inter-node bandwidth between node n_i and n_j is set as $BW_{inter}/|i-j|$. (3) uniform_dist: all the device-to-device bandwidths are randomly generated following $\mathcal{U}_{[1e-5,1e-2]}$ MB/us.

Visualizations for different device topologies' bandwidth lookup-tables (a table to

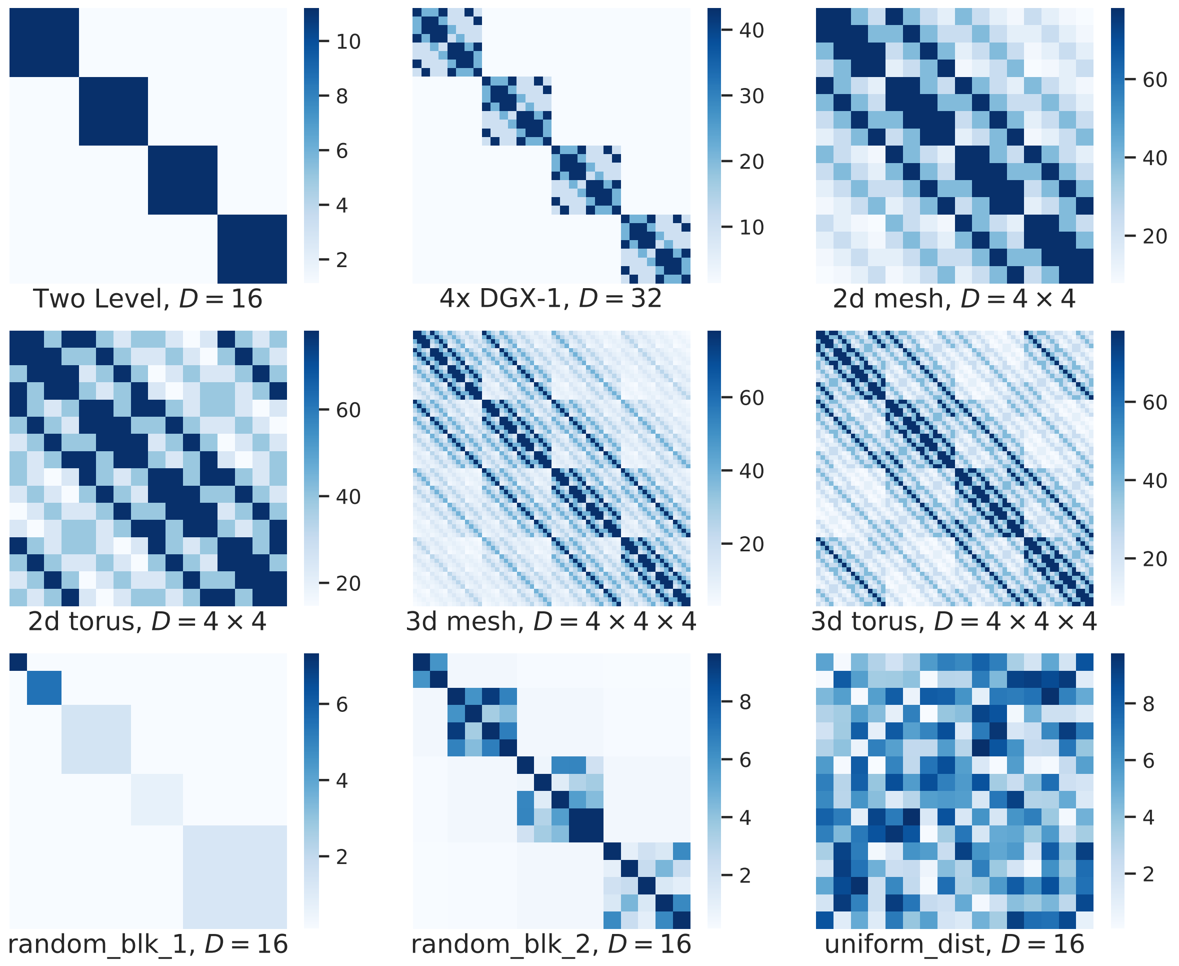


Figure 5.5: Visualizations for different architectures' bandwidth look-up-tables (a table to represent the device-to-device bandwidths).

represent the device-to-device bandwidths) are shown in Figure 5.5.

Model Settings We conduct experiments on various DNN models[4] including ResNet [37], SwinTransformer [85], BERT [21] and SemanticFPN [61]. The model settings for experiments on two-level hierarchical architecture and simulation on non-regular architecture are given in Table 5.2, and the model settings for experiments on Alibaba cloud 4x DGX-1 architecture

[4]Codes of ResNet, SwinTransformer, BERT are from TorchVision library, https://github.com/microsoft/Swin-Transformer, and https://github.com/codertimo/BERT-pytorch respectively. And we re-implement SemanticFPN following MMSegmentation (https://github.com/open-mmlab/mmsegmentation) and TorchVision library.

Table 5.2: DNN Models used in two-level hierarchical and non-regular architectures

	ResNet-152	Swin-L	SemanticFPN	BERT-L
Layers	[3, 8, 36, 3]	[2, 2, 18, 2]	Backbone: ResNet-50	24
Embed Dim	/	192	/	1024
Heads	/	[6, 12, 24, 48]	/	16
K	32	32	32	48
#Params	60M	197M	30M	365M
#Classes	1000	1000	150	/
Image Size	224x224x3	224x224x3	512x512x3	/
Vocab Size	/	/	/	30533
Seq Length	/	/	/	512
Micro-Batch Size	64	32	16	4

Table 5.3: DNN Models used in cloud 4x DGX-1 architecture

	Swin-L	Swin-U	BERT-24	BERT-36	BERT-48
Layers	[2, 2, 18, 2]	[2, 2, 24, 2]	24	36	48
Embed Dim	192	256	1024	1024	1024
Heads	[6, 12, 24, 48]	[8, 16, 32, 64]	16	16	16
K	32	48	48	72	96
#Params	197M	424M	365M	516M	667M
#Classes	1000	1000	/	/	/
Image Size	384x384x3	384x384x3	/	/	/
Vocab Size	/	/	30533	30533	30533
Seq Length	/	/	512	512	512
Micro-Batch Size	16	16	8	8	8

Table 5.4: A comparison with relevant baselines on the two-level hierarchical architecture. Speedup by our mapping is marked in brown.

Task	Res152 Classifi.			BERT-L Pre-train			Swin-L Pre-train		
(S,R)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)
Pipedream	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x
+ our map	7.6x	2.4x	1.0x	1.0x	1.0x	1.0x	2.5x	1.4x	1.2x
RaNNC	0.9x	1.3x	0.8x	0.8x	0.9x	0.9x	-	-	-
+ our map	7.2x	2.5x	0.9x	0.8x	0.9x	0.9x	-	-	-
Parmesan	**8.1x**	**2.6x**	**1.2x**	**1.1x**	**1.1x**	**1.0x**	**2.5x**	**1.6x**	**1.3x**

are given in Table 5.3. Note that we generate Swin-U, BERT-36 and BERT-48 by modifying the open source code, and BERT-24 is identical to BERT-L. We use SGD as these DNNs' optimizers.

5.7.2 Validation on Different Device Topologies

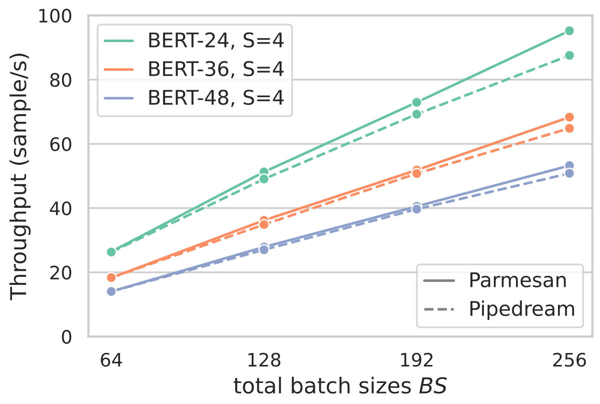
Two-level hierarchical architecture. We compare the throughput of Parmesan with RaNNC (operator-level) and Pipedream (layer-level) since they both provide I/O interface and support configuring the number of stages. As device mapping is not provided in these baselines, we put consecutive stages on consecutive devices (CS map). Note that CS map is the most commonly used heuristic for device mapping, which aims to alleviate the allreduce time. The whole evaluation flow can be summarized as follows: (1) acquire the DNN partitioning results from the partitioner of these baselines; (2) adopt the CS map/our map to these partitioning; (3) invoke our synchronous pipeline scheduler to measure the training latency and throughput. The performance of these two baselines, RaNNC and Pipedream, are then compared with the full flow of Parmesan. We fix the number of micro-batches MB as 4. Note that Pipedream fails to partition the given operator-level graph (BERT and Swin) within 2 hours, so we develop several pre-processing techniques for Pipedream to generate the layer-level graph input from the operator-level graph. Meanwhile, RaNNC reports an unknown error when partitioning Swin (marked as "-" in Table 5.4).

We measure the relative speedup of the synchronous pipeline training on a two-level hierarchical architecture (described in Section 5.7.1), and the results are shown in Table 5.4. By default, we apply CS map for Pipedream and RaNNC. To demonstrate the improvement of our mapping algorithm, we also apply our method to map their model partitions (+ our map). Since BERT is built with repeating blocks and is allreduce-heavy, our mapping algorithm produces similar solutions as CS map. However, Parmesan still obtains 1.1$\times$ speedup compared to Pipedream due to a better-balanced model partitioning for BERT. For the vision model ResNet-152 and Swin-L, our mapping algorithm speeds up the throughput of PipeDream and RaNNC significantly compared to CS map. Unlike BERT, ResNet and Swin are both p2p-heavy (Swin-L is also allreduce-heavy), CS map, as a kind of allreduce-first mapping, is no longer suitable for these cases. In contrast, our mapping algorithm can optimally map the partitions without human-in-the-loop. Note that our mapping results on Pipedream's Swin-L $(8, 2)$ and $(16, 1)$ and RaNNC's ResNet-152 $(16, 1)$ are neither consecutive mapping nor p2p-first, which further demonstrates the effectiveness of our mapping algorithm comparing to human-designed heuristic. Besides, our results achieve around 10% speedup for ResNet and Swin compared to other partitioning algorithms with our mapping, which indicates the better quality of our operator-level model partitioning algorithm. Finally, the results of our full-flow algorithm compared to other baselines demonstrate the superiority of Parmesan.

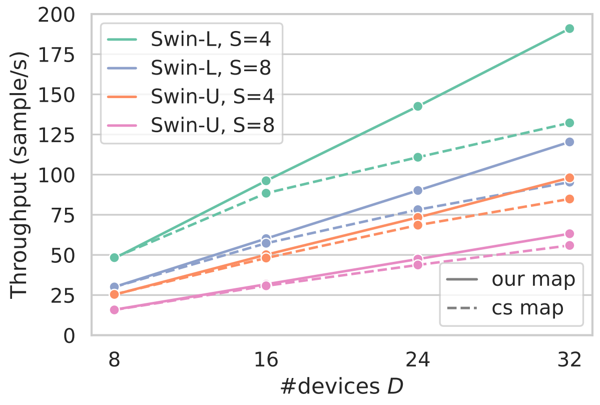
4x DGX-1 architecture. We conduct larger-scale experiments on the Alibaba cloud 4x DGX-1 architecture, in which the throughput under different settings are compared. DGX-1 is not a completely hierarchical architecture. Although inter-DGX-1 connection still relies on Ethernet, there are three types of intra-DGX-1 connections, namely double NVLinks, single NVLink, and PCI-e (described in Section 5.7.1). To examine the scalability of Parmesan, we enlarge the BERT-24 (i.e. BERT-L) to BERT-48 with 48 transformer layers with 667M parameters. We also enlarge Swin-L 2.15x to construct Swin-U which contains 424M parameters. The detailed settings of our enlarged models are discussed in Section 5.7.1.

Figure 5.6 shows a comparison of the throughput under different settings on the 4x DGX-1 architecture. In Figure 5.6(a), we compare the solution quality of different BERTs with Pipedream. We fix the number of micro-batches MB as 4 and the size of each micro-batch mBS as 8, and change the number of replicas R and the number of BERT layers to measure the throughput under different total batch sizes BS ($BS = mBS \times R \times MB$). This experiment demonstrates the effectiveness of our full-flow algorithm when scaling up the BERT size and the total batch size compared to Pipedream + CS Map. In Figure 5.6(b), we use our partitioning algorithm and fix $mBS = 16, MB = 4$, then we modify the number of devices D ($D = S \times R$) to evaluate the throughput of different Swin models under different mapping algorithms. Our mapping algorithm consistently achieves better solution qualities when scaling to large systems and large models compared to CS Map since our mapping algorithm takes the heterogeneity of the intra-DGX-1 connection into account. In Figure 5.6(c), we keep using our partitioning algorithm and measure the Swin-U throughput under various (S, R, MB) configurations and different mapping algorithms while maintaining $mBS = 16, D = 32$. The results show that our mapping algorithm consistently speeds up the throughput of CS Map under different (S, R, MB) configurations. All these experiments demonstrate the high extensibility and high efficiency of Parmesan.

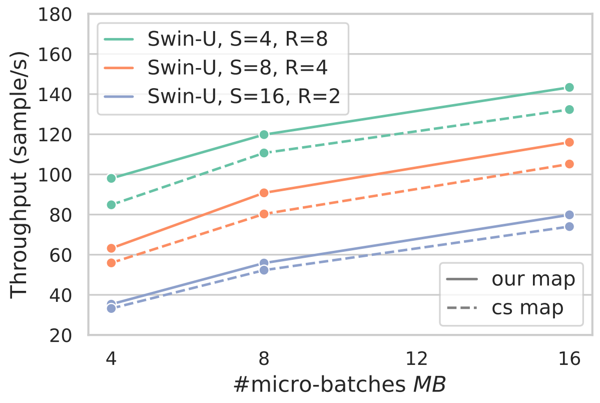
Besides, we compare the throughput with Pipedream on 4x DGX-1 architecture shown in Table 5.5. Compared to the CS map, our mapping algorithm consistently obtains remarkable speedup on the partitioning results generated by both two algorithms (ours and Pipedream's). Since Pipedream's partition needs to communicate between stages more intensively while 4x DGX-1 architecture has extremely large intra-node bandwidth (double NVLink), the inter-stage communication overhead of Pipedream's partitions is significantly reduced by applying our mapping. The remaining factor that mainly affects the throughput is whether the computation is balanced or not. Since Pipedream's partition and our partition are both computationally balanced, the throughput of Pipedream's partitioning + our mapping looks close to ours. Nevertheless, our partitioning algorithm still considers communication overhead and achieves significant speedup compared to Pipedream when



(a) $mBS = 8, MB = 4$



(b) $mBS = 16, MB = 4$



(c) $mBS = 16, D = 32$

Figure 5.6: Throughput of different BERTs and Swins on 4x DGX-1 Architecture.

Table 5.5: A comparison with Pipedream on 4x DGX-1. "PD" is short for Pipedream and "Par" is short for partitioning.

Task	Swin-L Pre-train			Swin-U Pre-train		
(S,R)	(4,8)	(8,4)	(16,2)	(4,8)	(8,4)	(16,2)
PD Par + CS map	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x
PD Par + Our map	**2.7x**	**2.5x**	1.3x	**1.9x**	**1.3x**	1.2x
Our Par + CS Map	1.9x	1.9x	1.2x	1.6x	1.2x	1.4x
Parmesan	**2.7x**	**2.5x**	**1.5x**	**1.9x**	**1.3x**	**1.5x**

Table 5.6: Simulation results for different topologies. Speedup by our mapping over the CS mapping is shown. "-" denotes the number of devices D ($D = S \times R$) cannot form a specific torus/mesh architecture.

(S,R)	(4,16)	(8,8)	(16,4)	(4,64)	(8,32)	(16,16)	(4,128)	(8,64)	(16,32)
2d mesh	1.1x	1.0x	2.7x	5.6x	2.5x	1.4x	-	-	-
2d torus	1.1x	1.0x	2.6x	1.6x	1.5x	1.0x	-	-	-
3d mesh	1.0x	1.1x	1.1x	-	-	-	1.3x	1.8x	1.2x
3d torus	1.0x	1.1x	1.0x	-	-	-	1.1x	1.0x	2.6x
random_blk_1	1.5x	1.8x	1.5x	1.1x	1.4x	1.9x	1.1x	1.7x	1.9x
random_blk_2	2.1x	1.6x	3.0x	1.4x	1.3x	3.7x	1.5x	1.6x	4.5x
uniform_dist	33.5x	11.4x	6.7x	8.1x	5.1x	7.2x	23.3x	8.9x	5.5x

fixing the mapping as CS map. The results further demonstrate the effectiveness of our mapping algorithm and our operator-level partitioning scheme.

Non-regular architecture. We mentioned above that DGX-1 has some intra-node heterogeneity, but it still resembles the traditional hierarchical architecture to some extent. To evaluate the performance of our mapping algorithm for general device topology, architectures of specialized hardware can be considered. Unfortunately, it is difficult to verify our algorithm on those hardwares due to unavailability or incompatibility with the commonly used programming interface. Hence, we developed a simulator to simulate the performance and conduct a comparison with the human-designed heuristic on popular grid-based

Table 5.7: The relative speedup of our refinement technique. "RF" is short for refinement.

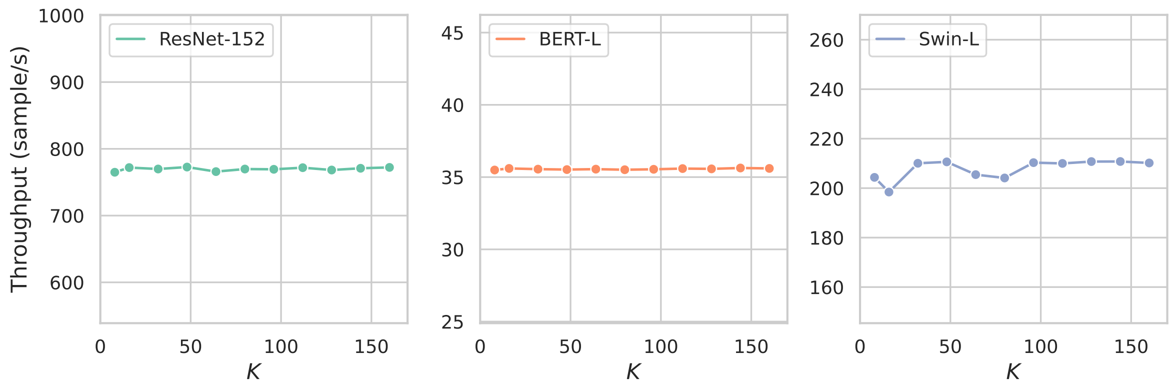
	Res152 Classfi.			BERT-L Pre-train			Swin-L Pre-train		
(S, R)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)
w/o RF	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
w/ RF	**1.04x**	**1.09x**	**1.14x**	**1.00x**	**1.17x**	**1.25x**	**1.38x**	**1.28x**	**1.12x**

architectures (mesh and torus) and fully heterogeneous topologies. The experiments are conducted on SemanticFPN [61] whose partitioning results include more skip-connections, bringing more challenges to solving the problem. We randomly generate three kinds of fully heterogeneous topologies, named random_blk_1, random_blk_2 and uniform_dist (described in Section 5.7.1). As shown in Table 5.6, our mapping algorithm consistently yields significant speedup over CS mapping on various topologies. The results show that our proposed general topology mapping algorithm can solve the mapping problem once and for all without human-in-the-loop.

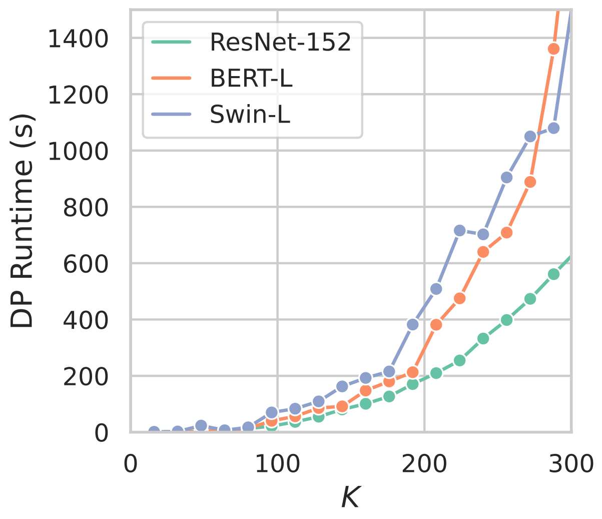
5.7.3 Ablation Studies on Model Partitioning

Effectiveness of Clustering. We measure the pipeline training throughput under different settings of K on the two-level hierarchical architecture. We fix the device mapping for each group with the same value of $S = 4, R = 4$. As shown in Figure 5.7(a), changing K affects the throughput mildly especially for ResNet and BERT. Meanwhile, we also measure the impact on the DP running time of changing K to demonstrate that operator clustering can significantly improve the efficiency of DP. As shown in Figure 5.7(b), the DP runtime (s) increases drastically with increasing the K. These two experiments illustrate that operator clustering is able to reduce the running time of DP with very minimal effect on the partitioning quality.

Effectiveness of Refinement. We measure the relative speedup of pipeline training throughput on the two-level hierarchical architecture to demonstrate the effectiveness of refinement. For each DNN model, we use the same K to generate the model partitions



(a) Throughput (sample/s)



(b) DP runtime (s)

Figure 5.7: Effectiveness of operator clustering. (a) The impact of different K on the pipeline throughput for different models. (b) The DP runtime (s) of different models with setting different K.

and fix the device mapping for each group (S, R). As shown in Table 5.7, refinement can significantly increase the throughput for most cases. The results demonstrate the effectiveness of the refinement step in fixing the partitioning result obtained by operator clustering and DP, and mitigating the sub-optimality gap brought by operator clustering.

5.7.4 Impact of Timeout Heuristic of Device Mapping

As mentioned in Section 5.4, a timeout heuristic is applied to speedup the searching during device mapping. In this paragraph, we make inspections on its runtime and resulted optimization quality. We conduct experiments on mapping the partitioned SemanticFPN on non-regular architectures under different (S, R) pairs. Table 5.8 shows the runtime of two instantiations of our device mapping. Our mapping algorithm successfully generates results for up to 512 devices within 40 minutes. Table 5.9 illustrates the result quality generated by mapping with timeout by comparing them with the optimal version (i.e., without the timeout heuristic). For the cases of the grid-based topologies with the number of devices $D = S \times R \leq 16$, the searching equipped with timeout is able to produce identical min-max stage time as by the optimal version. However, due to the increasing complexity, the optimal version fails to obtain the solution within 2 hours while our device mapping with timeout heuristic successes. "-" in Table 5.8 and Table 5.9 denotes the number of devices D ($D = S \times R$) cannot form a specific torus/mesh architecture.

5.7.5 Comparison to Heuristic Mappings

Our previous discussion mainly compares with CS map, which is a kind of allreduce-first mapping. However, some networks are not allreduce-heavy, so applying the CS map to them may not be suitable. For example, a model with large inter-stage (p2p) communication (e.g., ResNet and Swin) may further speed up by other mappings. Intuitively, the most straightforward way is to develop a heuristic-based p2p sequential mapping (p2p map) for such cases. However, it is still non-trivial for humans to automatically select which heuristics to map. Although the ensemble of CS map (allreduce-first) and p2p map (p2p-first) works

Table 5.8: Runtime (in seconds) of device mapping with different (S, R) and device topologies. p2p and allr refer to our two instantiations of device mapping respectively.

Topology	Alg.	(2, 8)	(4, 4)	(8, 2)	(4, 16)	(8, 8)	(16, 4)	(4, 64)	(8, 32)	(16, 16)	(4, 128)	(8, 64)	(16, 32)
2d mesh	p2p	19.8	0.6	4.0	605.5	908.1	908.0	609.7	762.0	762.0	-	-	-
	allr	0.4	0.9	0.4	1.3	1.5	1.6	611.4	6.4	48.7	-	-	-
2d torus	p2p	0.2	0.6	3.8	454.3	605.8	605.8	609.8	761.5	761.9	-	-	-
	allr	10.1	0.4	0.5	1.6	1.4	1.8	480.3	6.9	6.9	-	-	-
3d mesh	p2p	-	-	-	605.4	756.7	756.7	-	-	-	616.8	769.9	770.8
	allr	-	-	-	42.5	1.6	1.6	-	-	-	661.1	88.0	57.6
3d torus	p2p	-	-	-	325.2	454.9	756.9	-	-	-	617.0	618.9	619.2
	allr	-	-	-	1.7	1.5	1.7	-	-	-	82.1	15.5	36.5
random_blk_1	p2p	50.5	12.1	234.4	1379.4	1357.2	490.7	345.9	607.8	608.1	354.1	1221.5	1223.5
	allr	869.4	12.0	9.7	905.4	1021.5	760.1	911.0	910.5	910.7	925.0	922.6	923.3
random_blk_2	p2p	1057.8	1209.3	1119.9	455.9	757.5	1208.4	1212.8	1063.8	1214.5	1221.8	1223.6	1224.6
	allr	906.7	304.3	3.0	908.0	1361.2	646.7	1402.3	1086.3	931.3	1842.8	1091.0	1096.0
uniform_dist	p2p	681.7	320.7	478.2	987.0	781.1	1096.9	949.0	1251.4	1402.7	1282.5	1410.3	1541.0
	allr	185.5	453.9	0.6	644.7	454.1	303.1	368.1	217.6	350.7	334.9	205.8	208.6

Table 5.9: Comparison on results quality of device mapping with timeout heuristic to that of the optimal version (i.e., without the timeout heuristic). Figures in the table are the obtained objective values of Problem (5.7) for each combination of algorithm and (S, R). Differences are underlined. "*" denotes failure in obtaining results within 2 hours. "tmo" is short for timeout.

Topology	Alg.	(2, 8)	(4, 4)	(8, 2)	(4, 16)	(8, 8)	(16, 4)	(4, 64)	(8, 32)	(16, 16)	(4, 128)	(8, 64)	(16, 32)
2d mesh	p2p w/ tmo	177625	98940	60715	105340	66200	48829	105340	85851	114891	-	-	-
	p2p opt	177625	98940	60715	*	*	*	*	*	*	-	-	-
	allr w/ tmo	169473	81709	47752	82287	47789	28622	86123	47766	28927	-	-	-
	allr opt	169473	81709	47752	82287	47789	28622	*	*	28927	-	-	-
2d torus	p2p w/ tmo	177625	98940	60715	104140	62067	46091	105340	85851	79498	-	-	-
	p2p opt	177625	98940	60715	*	60715	39691	*	*	*	-	-	-
	allr w/ tmo	169473	81618	47752	82287	47763	28622	82455	47794	28927	-	-	-
	allr opt	169473	81618	47752	82287	47763	28622	82455	47794	28927	-	-	-
3d mesh	p2p w/ tmo	-	-	-	104140	64715	47691	-	-	-	105340	85851	65356
	p2p opt	-	-	-	*	60715	39691	-	-	-	*	*	*
	allr w/ tmo	-	-	-	82287	47763	28622	-	-	-	82483	47830	28978
	allr opt	-	-	-	82287	47763	28622	-	-	-	82483	47830	28978
3d torus	p2p w/ tmo	-	-	-	104140	63424	47691	-	-	-	105340	81195	46091
	p2p opt	-	-	-	*	60715	*	-	-	-	*	*	*
	allr w/ tmo	-	-	-	82287	47789	28622	-	-	-	82483	47795	28978
	allr opt	-	-	-	82287	47789	28622	-	-	-	82483	47795	28978
random_blk_1	p2p w/ tmo	7848025	5561027	5983883	7067273	5329127	5285295	10617386	10335205	11163700	10661165	11511750	14437823
	p2p opt	7848025	5561027	5983883	*	*	*	*	*	*	*	*	*
	allr w/ tmo	2717063	359296	96651	2757132	123523	44337	2891041	1933769	2238277	2913360	1964333	2312136
	allr opt	2717063	359296	96651	*	*	*	*	*	*	*	*	*
random_blk_2	p2p w/ tmo	1926463	2274920	1542303	2340990	1728740	1770727	3728927	3112257	7265727	3728927	3582268	7308867
	p2p opt	*	*	*	*	*	*	*	*	*	*	*	*
	allr w/ tmo	737258	127727	49239	604081	63963	42814	1196374	661812	462752	1797052	1090929	561108
	allr opt	*	127727	49239	*	*	*	*	*	*	*	*	*
uniform_dist	p2p w/ tmo	256915	267108	187708	254602	196161	201261	255566	314564	313764	253809	447725	484707
	p2p opt	*	266651	187708	*	*	*	*	*	*	*	*	*
	allr w/ tmo	184743	102645	49154	106767	56403	40923	107532	57963	44909	107609	58195	45580
	allr opt	184743	*	49154	*	*	*	*	*	*	*	*	*

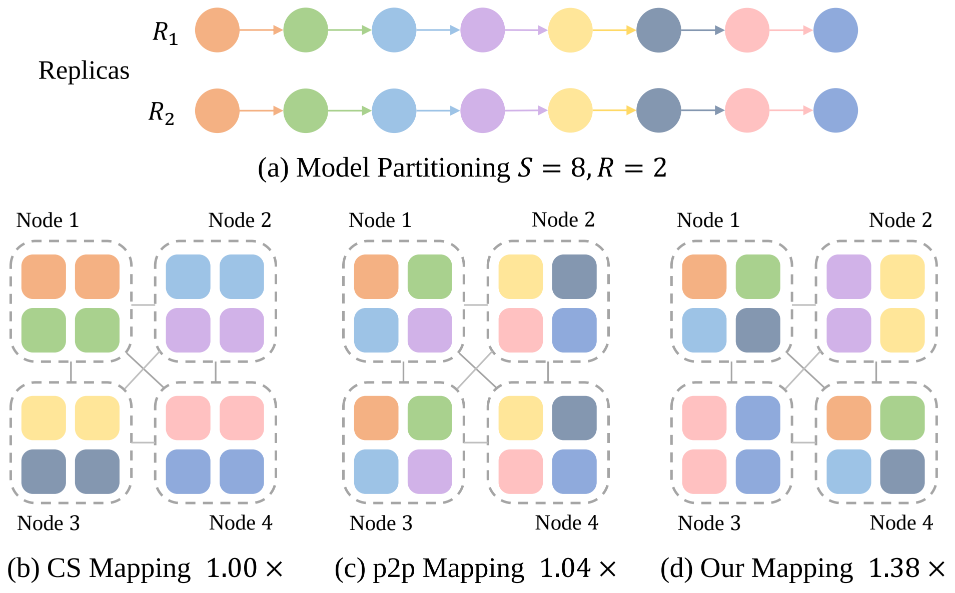


Figure 5.8: A visualization of different mapping algorithms on a two-level hierarchical architecture for Pipedream's Swin-L $(8, 2)$ partitioned results.

Table 5.10: The cases that our mapping algorithm maps the model partitioning in a non-regular manner (i.e., neither p2p-sequential nor allreduce-first).

Model	(S, R)	Partition	CS Map	p2p Map	Our Map
Res152	(16,1)	RaNNC	1.00x	1.00x	**1.12x**
Swin-L	(8,2)	Pipedream	1.00x	1.04x	**1.38x**
Swin-L	(16,1)	Pipedream	1.00x	1.00x	**1.17x**

well in most cases on the two-level hierarchy experiments in Section 5.7.2, for some cases, they are not the best. In contrast, our mapping algorithm generally generates the best mapping solution among all the aforementioned mappings. As cases shown in Table 5.10, our mapping algorithm maps the model partitioning in a non-regular manner and produces a better solution than the previous two heuristics. Note that p2p map is identical to CS map when $R = 1$. A visualization of different mapping algorithms for Pipedream's Swin-L $S = 8, R = 2$ partitioned results are illustrated in Figure 5.8. It is also non-trivial for humans to invent a heuristic to produce a solution with similar performance as Figure 5.8(d).

Admittedly, if we compare our mapping with the integration of the CS map and p2p

Table 5.11: Simulation results for different topologies. Speedup by our mapping over the heuristic mapping is shown. "-" denotes the number of devices D ($D = S \times R$) cannot form a specific torus/mesh architecture.

(S, R)	(4, 16)	(8, 8)	(16, 4)	(4, 64)	(8, 32)	(16, 16)	(4, 128)	(8, 64)	(16, 32)
2d mesh	1.0x	1.0x	1.2x	7.0x	2.5x	1.4x	-	-	-
2d torus	1.0x	1.0x	1.0x	1.0x	1.1x	1.0x	-	-	-
3d mesh	1.0x	1.0x	1.2x	-	-	-	1.6x	1.8x	1.0x
3d torus	1.0x	1.0x	1.0x	-	-	-	1.0x	1.0x	1.0x
random_blk_1	1.1x	1.0x	1.0x	1.3x	1.0x	1.0x	1.2x	1.0x	1.0x
random_blk_2	2.1x	1.8x	1.0x	1.1x	1.1x	2.1x	1.2x	1.1x	1.7x
uniform_dist	16.0x	11.7x	6.7x	12.1x	5.1x	9.8x	17.5x	7.7x	5.5x

map, except in the cases mentioned above, the other two-level hierarchy experiments might be seemingly underwhelming. However, when tackling the non-regular architecture (grid-based and random-based), our mapping algorithm shows its effectiveness compared to these two heuristics. We compare the ensembling solution (the solution with the best simulated throughput) of the two instantiations of our mapping algorithm with the ensembling results of the two heuristics (CS map and p2p map) under SemanticFPN on the non-regular architecture. As shown in Table 5.11, our mapping successfully speeds up the simulated throughput of most cases compared to the ensemble heuristic. The results not only indicate the limitation of heuristic mapping but demonstrate the strong demand of our effective general topology mapping algorithm.

5.7.6 Flow Runtime

Parmesan successfully generates partitioning and mapping solutions within 60 minutes for all the abovementioned experiments (Section 5.7). The average runtime of these experiments is around 8 minutes. Compared to the training time of recently advanced DNNs which requires hundreds of hours, our optimization process only consumes a small portion of that time but brings significant speedup.

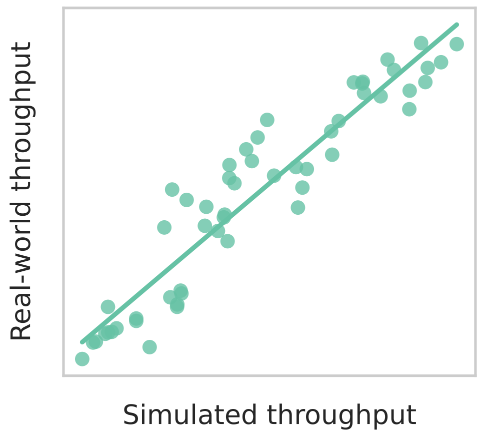


Figure 5.9: A comparison between the simulated and real-world throughput.

5.7.7 Simulator's Accuracy

We evaluate the simulator's accuracy by comparing the simulated results with real-world execution. We measure the real-world training throughputs on different models, different (S, R) configurations, and different mappings on the two-level hierarchy and compare them with the simulated results. The results are shown in Figure 5.9. Spearman's rank correlation coefficient between simulated and real-world throughput is around 0.95. There indeed exists small variations for some cases, but it should be understandable since there could be many factors that affect the real measured throughput due to, e.g., non-determinism in networking. Our simulator aims to give a reference value that is useful for developing an optimizer and exploring new partitioning and mapping algorithms.

With our well-designed optimizer, simulator, pipeline scheduler, and unified IO format, we believe our proposed general device topology mapping algorithm will have a high potential to provide valuable information for architecture designers and assist them in designing a more DNN-friendly hardware.

5.8 Proofs

5.8.1 Proof of Theorem 1

Proof. For any vertex u in $\mathcal{G}$, there is always a fronted graph $\mathcal{F}_u$ whose $\mathcal{V}_{\mathcal{F}_u} = \{u\} \cup \{v \mid v \text{ is a predecessor of } u\}$. We claim that $\forall\, u, v \in \mathcal{V}$, $\mathcal{F}_u \neq \mathcal{F}_v$. Then we can easily derive $|F_\mathcal{G}| \geq |\mathcal{V}|$ because $\mathcal{G}$ has $|\mathcal{V}|$ unique vertices. Now we prove the claim that $\forall\, u, v \in \mathcal{V}$, $\mathcal{F}_u \neq \mathcal{F}_v$. Suppose $\exists\, u, v$, s.t. $\mathcal{F}_u = \mathcal{F}_v$. Since $\mathcal{F}_u = \mathcal{F}_v$, we can get $u \in \mathcal{V}_{\mathcal{F}_v}$ and $v \in \mathcal{V}_{\mathcal{F}_u}$, then we have $\{u, v\} \subseteq \mathcal{V}_{\mathcal{F}_u}$ and $\{u, v\} \subseteq \mathcal{V}_{\mathcal{F}_v}$. This implies v is a predecessor of u and u is a predecessor of v, that is, there is a cycle in $\mathcal{G}$. This contradicts that $\mathcal{G}$ is a DAG. Thus $\forall\, u, v \in \mathcal{V}$, $\mathcal{F}_u \neq \mathcal{F}_v$. □

5.8.2 Proof of Theorem 2

Proof. Suppose v_i is the i-th vertex in the unique topological ordering of $\mathcal{G}$, we have $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$. For any $\mathcal{F} \in F_\mathcal{G}$, if $v_i \in \mathcal{F}$ and $i \geq j$ for any $v_j \in \mathcal{V}_\mathcal{F}$, we have $\mathcal{V}_\mathcal{F} = \{v_1, ..., v_i\}$, which implies the vertex with the largest topological ordering will determine $\mathcal{F}$ uniquely. Because $\mathcal{G}$ has $|\mathcal{V}|$ vertices and their topological ordering are distinct and unique, we have $|F_\mathcal{G}| = |\mathcal{V}|$. □

5.8.3 Proof for Optimality of Algorithm 2

Proposition 2. Algorithm 2 always returns the optimal solution p^* of Problem (5.7), if the initial interval $[t_l^{(0)}, t_r^{(0)}]$ contains the optimal value $t_{\max}(p^*) = \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p^*)$ of Problem (5.7).

Proof. We firstly prove that $t_{\max}(p^*) \in [t_l^{(i)}, t_r^{(i)}], \forall i$ by induction, where i is the iteration number of the while-loop in Algorithm 2. The base case $i = 0$ is already ensured as the precondition. For the inductive step, suppose $t_{\max}(p^*) \in [t_l^{(i)}, t_r^{(i)}]$ after iteration i. Consider two possible cases in iteration $i + 1$: the inner-level search either founds or is unable to found a solution for target $t \in [t_l^{(i)}, t_r^{(i)}]$. For the former case, $t_{\max}(p^*) \leq t_p \leq \min\{t, t_p\} =$

$t_r^{(i+1)}$, and since $t_{\max}(p^*) \geq t_l^{(i)} = t_l^{(i+1)}$, we have $t_{\max}(p^*) \in [t_l^{(i+1)}, t_r^{(i+1)}]$. For the latter case, there is no solution p such that $t_p \leq t$, i.e., $t_{\max}(p^*) \geq t = t_l^{(i+1)}$. Combined with $t_{\max}(p^*) \leq t_r^{(i)} = t_r^{(i+1)}$, we obtain $t_{\max}(p^*) \in [t_l^{(i+1)}, t_r^{(i+1)}]$.

It is easy to see that the interval $[t_l^{(i)}, t_r^{(i)}]$ is always shrinking with the increment of i, i.e., $t_l^{(i+1)} \geq t_l^{(i)}, t_r^{(i)} \leq t_r^{(i+1)}, \forall i$. When the length of the interval decreases to a value small enough (ϵ in Algorithm 2) such that no more solution can be found that has an objective value within this interval (since there is only a finite number of possible solutions), the last found solution is the optimal one. $\square$

5.8.4 Proof of Proposition 1

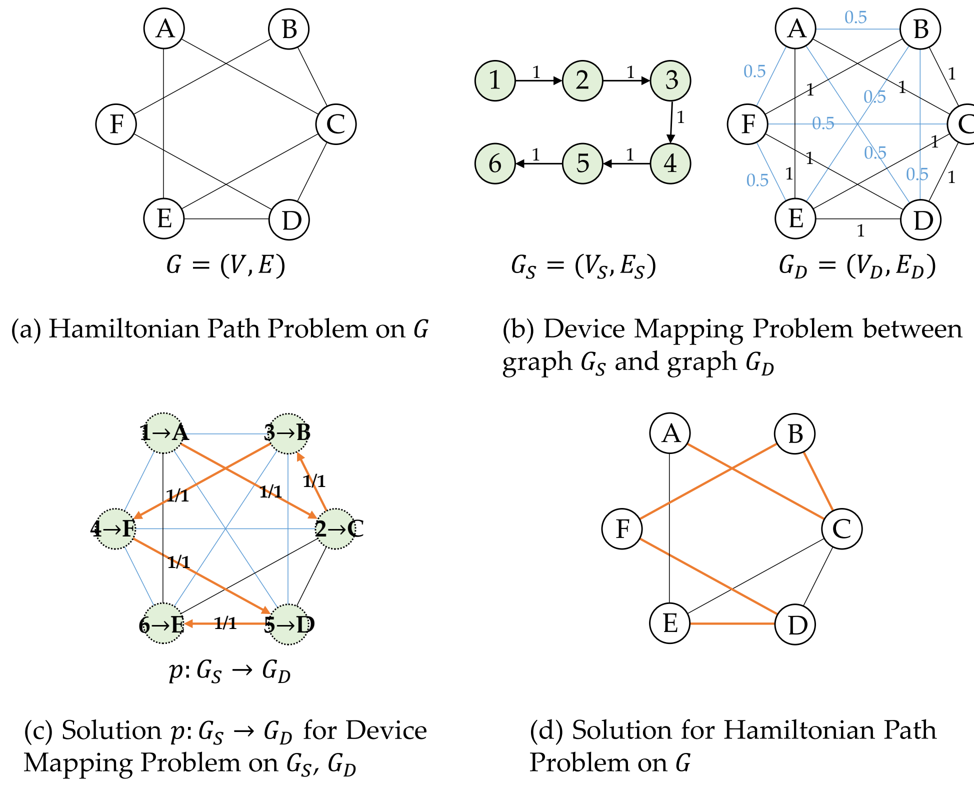
Proof. We firstly prove $t_{l0}^{\mathsf{m}} \leq \min_p \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}^{\mathsf{m}}(s, p)$. For any stage $s \in \mathcal{V}'$ and any mapping p, we have

$$\min_d \min_{p_{s,d}^{\mathsf{m}}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(d, p_{s,d}^{\mathsf{m}}(s'))} \leq \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(p(s), p(s'))}.$$

This is because the LHS is essentially the smallest possible inter-stage communication time of s (note that the LHS is independent of p), and the RHS is the inter-stage communication time of s under p. Since this inequality holds for any $s \in \mathcal{V}'$, we get

$$\begin{aligned} t_{l0}^{\mathsf{m}} &= \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \min_d \min_{p_{s,d}^{\mathsf{m}}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(d, p_{s,d}^{\mathsf{m}}(s'))} \right\} \\ &\leq \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(p(s), p(s'))} \right\} \\ &= \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}^{\mathsf{m}}(s, p). \end{aligned}$$

Also, the above inequality holds for any p, and thus we obtain $t_{l0}^{\mathsf{m}} \leq \min_p \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}^{\mathsf{m}}(s, p)$ as desired.



(a) Hamiltonian Path Problem on G

(b) Device Mapping Problem between graph G_S and graph G_D

(c) Solution $p: G_S \to G_D$ for Device Mapping Problem on G_S, G_D

(d) Solution for Hamiltonian Path Problem on G

Figure 5.10: The Hamiltonian path problem on G can be reduced to the device mapping problem between the stage graph G_S and the device topology graph G_D. The mapping solution p of DMP with a maximum stage time (= communication data size / bandwidth) less than or equal to 1 will give a path connecting all the corresponding vertices in G.

For the second inequality, we claim that

$$\min_{d} \min_{p_{s,d}^{\text{AR}}} \max_{s' \in \text{adj\_repl}(s)} \left\{ 2\frac{R-1}{R} \frac{P(s)}{B(d, p_{s,d}^{\text{AR}}(s'))} \right\}$$
$$\leq \max_{\substack{s_i, s_{i+1} \\ \in \text{ring}(\text{repl}(s))}} \left\{ 2\frac{R-1}{R} \frac{P(s)}{B(p(s_i), p(s_{i+1}))} \right\}$$

holds for any stage $s \in \mathcal{V}'$ and any mapping p, since the LHS is no greater than the smallest possible inter-replica communication time of s (note that on the LHS, the maximization is over a subset of the ring only), and the RHS is the inter-replica communication time of s under p. The remaining part of the proof essentially follows the argument in the proof of the first inequality. □

5.8.5 Proof of Theorem 3

Proof. We are going to prove the NP-completeness of the decision problem version of the Device Mapping Problem (DMP) by reducing from the Hamiltonian Path Problem on undirected graph $G(V, E)$. The decision problem version of DMP is to decide whether there is a mapping p that the maximum stage time is less than or equal to a threshold t,

$$\max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p) \leq t. \tag{5.14}$$

First of all, the decision version of DMP is NP since the correctness of a mapping can be checked in polynomial time. Next, we can reduce the Hamiltonian Path Problem to the decision version DMP by (1) putting $n = |V|$ and there is a corresponding device d_i for each node $v_i \in V$ for $i = 1 \ldots n$, (2) setting the computation times of all the stages to zero; (3) for any edge $(v_i, v_j) \in E$, put the bandwidth between device d_i and d_j as 1, while the bandwidths of all the remaining pairs of devices are 0.5; (4) putting the threshold t as 1 and having the set of stage S just n stages $\{s_1, \ldots, s_n\}$ such that there is data communication of size 1 from s_i to s_{i+1} for all $i = 1 \ldots n - 1$ only. It is not hard to see a Hamiltonian path in G will correspond to a mapping p of the n stages to the set of n devices with a maximum stage time less than or equal to 1, and the vice versa that a mapping p will give a path connecting the n corresponding vertices in G. $\square$

Figure 5.10 gives an example of the above reduction.

5.9 Concluding Remarks

In this chapter, we propose an efficient design framework, named Parmesan, to map the training of a large DNN onto a system with general device topology to maximize the throughput. We demonstrate the effectiveness of our proposed framework on different GPU topologies and show the superiority of our mapping algorithm when tackling non-hierarchical architectures compared to existing heuristics.

Conclusion

In this thesis, we explore the interaction between placement and GPU, focusing specifically on addressing two fundamental challenges: GPU-accelerated VLSI placement in EDA and device placement for GPUs in MLSys.

For GPU-accelerated VLSI placement, we introduce Xplace, an extremely fast and extensible GPU-accelerated placement framework. We employ operator-level optimization techniques to enable effective parallelization and we propose a placement-stage-aware parameter scheduling technique to enhance the placement quality. Compared to the state-of-the-art placer, Xplace shows superior performance and quality. Furthermore, we investigate the potential of integrating neural guidance into a GPU-accelerated analytical placer to further boost the solution quality.

In addition to optimizing wirelength and runtime performance in VLSI placement, we extend Xplace to address detailed-routability and present Xplace-Route. Xplace-Route combines a GPU-accelerated placement engine and a GPU-accelerated routing engine to facilitate routability-driven placement. Through the incorporation of several proposed detailed-routability optimization techniques, Xplace-Route achieves superior detailed-routability and significant runtime speedup.

Regarding device placement for GPUs, we present Parmesan, an efficient design framework to maximize the DNN training throughput for operator-level DNNs on systems with general topology. Parmesan works in an end-to-end manner and solves the whole optimization problem in two phases. The first phase aims at producing well-balanced partitions, and the second phase works towards placing the DNN on devices connected by an arbitrary topology network, considering the heterogeneity of the interconnection bandwidth. Experimental results show that Parmesan speeds up the pipeline training throughput on systems with different GPU topologies and is able to handle the mapping problem for heterogeneously interconnected architectures.

In conclusion, this thesis studies the potential of leveraging GPUs in two fundamental yet challenging placement problems in the fields of EDA and MLSys. With the continuous advancements in GPU computational power and the increasing complexity of these placement problems, we believe the works presented in this thesis will effectively enhance the placement performance and successfully demonstrate their practicality in real-world productions.

References

[1] Martıén Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI’16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.

[2] Saurabh N. Adya, Igor L. Markov, and Paul G. Villarrubia. “On Whitespace and Stability in Mixed-Size Placement and Physical Synthesis”. In: *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*. USA: IEEE Computer Society, 2003, p. 311. ISBN: 1581137621.

[3] Anthony Agnesina, Kyungwook Chang, and Sung Kyu Lim. “VLSI Placement Parameter Optimization using Deep Reinforcement Learning”. In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9.

[4] Anthony Agnesina, Puranjay Rajvanshi, Tian Yang, Geraldo Pradipta, Austin Jiao, Ben Keller, Brucek Khailany, and Haoxing Ren. “AutoDMP: Automated DREAMPlace-Based Macro Placement”. In: *Proceedings of the 2023 International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 149–157. ISBN: 9781450399784. DOI: 10.1145/3569052.3578923. URL: https://doi.org/10.1145/3569052.3578923.

[5] A.R. Agnihotri, S. Ono, Chen Li, M.C. Yildiz, A. Khatkhate, Cheng-Kok Koh, and P.H. Madden. “Mixed block placement via fractional cut recursive bisection”. In: *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems 24.5 (2005), pp. 748–761. DOI: 10.1109/TCAD.2005.846363.

[6] N. Ahmed, T. Natarajan, and K.R. Rao. "Discrete Cosine Transform". In: *IEEE Transactions on Computers* C-23.1 (1974), pp. 90–93. DOI: 10.1109/T-C.1974.223784.

[7] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. "Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 472–487. ISBN: 9781450391627.

[8] Amotz Bar-Noy and Shlomo Kipnis. "Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems". In: *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '92. San Diego, California, USA: Association for Computing Machinery, 1992, pp. 13–22. ISBN: 089791483X.

[9] Ulrich Brenner and André Rohe. "An Effective Congestion Driven Placement Framework". In: *Proceedings of the 2002 International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2002, pp. 6–11. ISBN: 1581134606. DOI: 10.1145/505388.505391. URL: https://doi.org/10.1145/505388.505391.

[10] Ismail S. Bustany, David Chinnery, Joseph R. Shinnerl, and Vladimir Yutsis. "ISPD 2015 Benchmarks with Fence Regions and Routing Blockages for Detailed-Routing-Driven Placement". In: *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 157–164. ISBN: 9781450333993. DOI: 10.1145/2717764.2723572.

[11] *Cadence Innovus Implementation System*. https://www.cadence.com.

[12] Tony Chan, Jason Cong, and Kenton Sze. "Multilevel Generalized Force-Directed Method for Circuit Placement". In: *Proceedings of the 2005 International Symposium on*

Physical Design. San Francisco, California, USA: Association for Computing Machinery, 2005, pp. 185–192. ISBN: 1595930213. DOI: 10.1145/1055137.1055177.

[13] Tony F. Chan, Jason Cong, Joseph R Shinnerl, Kenton Sze, and Min Xie. “MPL6: Enhanced Multilevel Mixed-Size Placement”. In: *Proceedings of the 2006 International Symposium on Physical Design*. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 212–214. ISBN: 1595932992. DOI: 10.1145/1123008.1123055.

[14] Jingsong Chen, Jian Kuang, Guowei Zhao, Dennis J.-H. Huang, and Evangeline F. Y. Young. “PROS 2.0: A Plug-In for Routability Optimization and Routed Wirelength Estimation Using Deep Learning”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.1 (2023), pp. 164–177. DOI: 10.1109/TCAD.2022.3168259.

[15] Pengwen Chen, Chung-Kuan Cheng, Albert Chern, Chester Holtz, Aoxi Li, and Yucheng Wang. “Placement Initialization via Sequential Subspace Optimization with Sphere Constraints”. In: *Proceedings of the 2023 International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 133–140. ISBN: 9781450399784. DOI: 10.1145/3569052.3571877. URL: https://doi.org/10.1145/3569052.3571877.

[16] Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang. “NTUplace3: An Analytical Placer for Large-Scale Mixed-Size Designs With Preplaced Blocks and Density Constraints”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), pp. 1228–1240. DOI: 10.1109/TCAD.2008.923063.

[17] Chung-Kuan Cheng, Andrew B. Kahng, Ilgweon Kang, and Lutong Wang. “RePlAce: Advancing Solution Quality and Routability Validation in Global Placement”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.9 (2019), pp. 1717–1730. DOI: 10.1109/TCAD.2018.2859220.

[18] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. "Project adam: Building an efficient and scalable deep learning training system". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 571–582.

[19] Jason Cong and Yi Zou. "Parallel Multi-Level Analytical Global Placement on Graphics Processing Units". In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. San Jose, California: Association for Computing Machinery, 2009, pp. 681–688. ISBN: 9781605588001. DOI: 10.1145/1687399.1687525.

[20] Nima Karimpour Darav, Andrew Kennings, Aysa Fakheri Tabrizi, David Westwick, and Laleh Behjat. "Eh?Placer: A High-Performance Modern Technology-Driven Placer". In: *ACM Trans. Des. Autom. Electron. Syst.* 21.3 (2016). ISSN: 1084-4309.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[22] H. Eisenmann and F.M. Johannes. "Generic global placement and floorplanning". In: *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*. 1998, pp. 269–274. DOI: 10.1145/277044.277119.

[23] Shiqing Fan et al. "DAPPLE: a pipelined data parallel approach for training large models". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2021).

[24] Yuanxiang Gao, Li Chen, and Baochun Li. "Spotlight: Optimizing device placement for training deep neural networks". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 1676–1684.

[25] Amur Ghose, Vincent Zhang, Yingxue Zhang, Dong Li, Wulong Liu, and Mark Coates. "Generalizable Cross-Graph Embedding for GNN-Based Congestion Prediction". In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Munich,

Germany: IEEE Press, 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643446. URL: https://doi.org/10.1109/ICCAD51958.2021.9643446.

[26] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. "Accurate, large minibatch sgd: Training imagenet in 1 hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[27] OpenAI. *GPT-4*. https://openai.com/research/gpt-4. 2023.

[28] William Gropp, Luke N Olson, and Philipp Samfass. "Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. 2016, pp. 41–50.

[29] Jiaqi Gu, Zixuan Jiang, Yibo Lin, and David Z. Pan. "DREAMPlace 3.0: Multi-Electrostatics Based Robust VLSI Placement with Region Constraints". In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9.

[30] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. "GPU-accelerated Critical Path Generation with Path Constraints". In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643504.

[31] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. "GPU-accelerated Path-based Timing Analysis". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 721–726. DOI: 10.1109/DAC18074.2021.9586316.

[32] Zizheng Guo, Feng Gu, and Yibo Lin. "GPU-Accelerated Rectilinear Steiner Tree Generation". In: *2022 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM. 2022.

[33] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. "GPU-Accelerated Static Timing Analysis". In: *2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM. 2020. DOI: 10.1145/3400302.3415631.

[34] Zizheng Guo and Yibo Lin. “Differentiable-Timing-Driven Global Placement”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1315–1320. ISBN: 9781450391429. DOI: 10.1145/3489517.3530486. URL: https://doi.org/10.1145/3489517.3530486.

[35] Zizheng Guo, Jing Mai, and Yibo Lin. “Ultrafast CPU/GPU Kernels for Density Accumulation in Placement”. In: *Proceedings of the 58th Annual Design Automation Conference 2021*. ACM. 2021.

[36] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[38] Xu He, Tao Huang, Wing-Kai Chow, Jian Kuang, Ka-Chun Lam, Wenzan Cai, and Evangeline F.Y. Young. “Ripple 2.0: High quality routability-driven placement via global router integration”. In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–6.

[39] Xu He, Tao Huang, Linfu Xiao, Haitong Tian, Guxin Cui, and Evangeline F.Y. Young. “Ripple: An effective routability-driven placer by iterative cell movement”. In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, pp. 74–79. DOI: 10.1109/ICCAD.2011.6105308.

[40] Xu He, Tao Huang, Linfu Xiao, Haitong Tian, and Evangeline F. Y. Young. “Ripple: A Robust and Effective Routability-Driven Placer”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.10 (2013), pp. 1546–1556. DOI: 10.1109/TCAD.2013.2265371.

[41] Zhuolun He, Yuzhe Ma, and Bei Yu. “X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms”. In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. New York, NY, USA: Association for Computing Machinery, 2022. ISBN: 9781450392174. DOI: 10.1145/3508352.3549383. URL: https://doi.org/10.1145/3508352.3549383.

[42] Magnus R Hestenes, Eduard Stiefel, et al. “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436.

[43] Wenting Hou, Hong Yu, Xianlong Hong, Yici Cai, Weimin Wu, Jun Gu, and W.H. Kao. “A new congestion-driven placement algorithm based on cell inflation”. In: *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001*. 2001, pp. 605–608. DOI: 10.1109/ASPDAC.2001.913375.

[44] Meng-Kai Hsu, Valeriy Balabanov, and Yao-Wen Chang. “TSV-Aware Analytical Placement for 3-D IC Designs Based on a Novel Weighted-Average Wirelength Model”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.4 (2013), pp. 497–509. DOI: 10.1109/TCAD.2012.2226584.

[45] Meng-Kai Hsu, Yao-Wen Chang, and Valeriy Balabanov. “TSV-Aware Analytical Placement for 3D IC Designs”. In: *Proceedings of the 48th Design Automation Conference*. San Diego, California: Association for Computing Machinery, 2011, pp. 664–669. ISBN: 9781450306362. DOI: 10.1145/2024724.2024875.

[46] Meng-Kai Hsu, Yi-Fang Chen, Chau-Chin Huang, Sheng Chou, Tzu-Hen Lin, Tung-Chieh Chen, and Yao-Wen Chang. “NTUplace4h: A Novel Routability-Driven Placement Algorithm for Hierarchical Mixed-Size Circuit Designs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.12 (2014), pp. 1914–1927. DOI: 10.1109/TCAD.2014.2360453.

[47] Bo Hu and Malgorzata Marek-Sadowska. “FAR: Fixed-Points Addition & Relaxation Based Placement”. In: *Proceedings of the 2002 International Symposium on Physical*

Design. New York, NY, USA: Association for Computing Machinery, 2002, pp. 161–166. ISBN: 1581134606. DOI: 10.1145/505388.505426. URL: https://doi.org/10.1145/505388.505426.

[48] Chau-Chin Huang, Hsin-Ying Lee, Bo-Qiao Lin, Sheng-Wei Yang, Chin-Hao Chang, Szu-To Chen, Yao-Wen Chang, Tung-Chieh Chen, and Ismail Bustany. "NTUplace4dr: A Detailed-Routing-Driven Placer for Mixed-Size Circuit Designs With Technology and Region Constraints". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.3 (2018), pp. 669–681. DOI: 10.1109/TCAD.2017.2712665.

[49] Yanping Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019.

[50] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks." In: *Proceedings of Machine Learning and Systems* (2019), pp. 1–13.

[51] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[52] A.B. Kahng, S. Reda, and Qinke Wang. "Architecture and details of a high quality, large-scale analytical placer". In: *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.* 2005, pp. 891–898. DOI: 10.1109/ICCAD.2005.1560188.

[53] A.B. Kahng and Qinke Wang. "Implementation and extensibility of an analytic placer". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.5 (2005), pp. 734–747. DOI: 10.1109/TCAD.2005.846366.

[54] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*. Vol. 312. Springer, 2011.

[55] Andrew B. Kahng, Lutong Wang, and Bangqi Xu. "The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218532.

[56] A. Kennings and K.P. Vorwerk. "Force-Directed Methods for Generic Placement". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.10 (2006), pp. 2076–2087. DOI: 10.1109/TCAD.2005.862748.

[57] Myung-Chul Kim, Jin Hu, Dong-Jin Lee, and Igor L. Markov. "A SimPLR method for routability-driven placement". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, pp. 67–73. DOI: 10.1109/ICCAD.2011.6105307.

[58] Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov. "SimPL: An effective placement algorithm". In: *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2010, pp. 649–656. DOI: 10.1109/ICCAD.2010.5654229.

[59] Myung-Chul Kim and Igor L. Markov. "ComPLx: A Competitive Primal-Dual Lagrange Optimization for Global Placement". In: *Proceedings of the 49th Annual Design Automation Conference*. San Francisco, California: Association for Computing Machinery, 2012, pp. 747–752. ISBN: 9781450311991. DOI: 10.1145/2228360.2228496.

[60] Myung-Chul Kim, Natarajan Viswanathan, Charles J. Alpert, Igor L. Markov, and Shyam Ramji. "MAPLE: Multilevel Adaptive Placement for Mixed-Size Designs". In: *Proceedings of the 2012 ACM International Symposium on International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 193–200. ISBN: 9781450311670. DOI: 10.1145/2160916.2160958. URL: https://doi.org/10.1145/2160916.2160958.

[61] Alexander Kirillov, Ross Girshick, Kaiming He, and Piotr Dollár. "Panoptic feature pyramid networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 6399–6408.

[62] Jürgen M Kleinhans, Georg Sigl, Frank M Johannes, and Kurt J Antreich. "GORDIAN: VLSI placement by quadratic programming and slicing optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.3 (1991), pp. 356–365.

[63] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *arXiv preprint arXiv:1404.5997* (2014).

[64] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-Based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[65] Chen Li, Min Xie, Cheng-Kok Koh, J. Cong, and P.H. Madden. "Routability-driven placement and white space allocation". In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* 2004, pp. 394–401. DOI: 10.1109/ICCAD.2004.1382607.

[66] Haocheng Li, Wing-Kai Chow, Gengjie Chen, Bei Yu, and Evangeline F.Y. Young. "Pin-Accessible Legalization for Mixed-Cell-Height Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.1 (2022), pp. 143–154. DOI: 10.1109/TCAD.2021.3053223.

[67] Shigang Li and Torsten Hoefler. "Chimera: efficiently training large-scale neural networks with bidirectional pipelines". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.

[68] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale

Language Models”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 6543–6552.

[69] Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. “Fourier Neural Operator for Parametric Partial Differential Equations”. In: *International Conference on Learning Representations*. 2021.

[70] Peiyu Liao, Dawei Guo, Zizheng Guo, Siting Liu, Yibo Lin, and Bei Yu. “DREAMPlace 4.0: Timing-driven Placement with Momentum-based Net Weighting and Lagrangian-based Refinement”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023), pp. 1–1. DOI: 10.1109/TCAD.2023.3240132.

[71] Chun-Xun Lin and Martin D. F. Wong. “Accelerate analytical placement with GPU: A generic approach”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 1345–1350. DOI: 10.23919/DATE.2018.8342222.

[72] Jai-Ming Lin, Hao-Yuan Hsieh, Hsuan Kung, and Hao-Jia Lin. “Routability-driven Analytical Placement with Precise Penalty Models for Large-Scale 3D ICs”. In: *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2022, pp. 1–8.

[73] Jai-Ming Lin, Chung-Wei Huang, Liang-Chi Zane, Min-Chia Tsai, Che-Li Lin, and Chen-Fa Tsai. “Routability-driven Global Placer Target on Removing Global and Local Congestion for VLSI Designs”. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–8. DOI: 10.1109/ICCAD51958.2021.9643544.

[74] Mark Po-Hung Lin, Chih-Cheng Hsu, and Yu-Chuan Chen. “Clock-Tree Aware Multibit Flip-Flop Generation During Placement for Power Optimization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.2 (2015), pp. 280–292. DOI: 10.1109/TCAD.2014.2376988.

[75] Shiju Lin, Jinwei Liu, Tianji Liu, Martin D. F. Wong, and Evangeline F. Y. Young. “NovelRewrite: Node-Level Parallel AIG Rewriting”. In: *Proceedings of the 59th ACM/IEEE*

Design Automation Conference. New York, NY, USA: Association for Computing Machinery, 2022, pp. 427–432. ISBN: 9781450391429. DOI: 10.1145/3489517.3530462. URL: https://doi.org/10.1145/3489517.3530462.

[76] Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. "GAMER: GPU-Accelerated Maze Routing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.2 (2023), pp. 583–593. DOI: 10.1109/TCAD.2022.3184281.

[77] Shiju Lin and Martin D.F. Wong. "Superfast Full-Scale GPU-Accelerated Global Routing". In: *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2022, pp. 1–8.

[78] Tao Lin, Chris Chu, Joseph R. Shinnerl, Ismail Bustany, and Ivailo Nedelchev. "POLAR: Placement based on novel rough legalization and refinement". In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 357–362. DOI: 10.1109/ICCAD.2013.6691143.

[79] Tao Lin, Chris Chu, and Gang Wu. "POLAR 3.0: An ultrafast global placement engine". In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, pp. 520–527. DOI: 10.1109/ICCAD.2015.7372614.

[80] Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Brucek Khailany, and David Z. Pan. "DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.4 (2021), pp. 748–761. DOI: 10.1109/TCAD.2020.3003843.

[81] Yibo Lin, Wuxi Li, Jiaqi Gu, Haoxing Ren, Brucek Khailany, and David Z. Pan. "ABCDPlace: Accelerated Batch-Based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 5083–5096. DOI: 10.1109/TCAD.2020.2971531.

[82] Jinwei Liu, Chak-Wa Pui, Fangzhou Wang, and Evangeline F. Y. Young. "CUGR: Detailed-Routability-Driven 3D Global Routing with Probabilistic Resource Model". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218646.

[83] Siting Liu, Qi Sun, Peiyu Liao, Yibo Lin, and Bei Yu. "Global Placement with Deep Learning-Enabled Explicit Routability Optimization". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1821–1824. DOI: 10.23919/DATE51398.2021.9473959.

[84] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. "NCTU-GR 2.0: Multithreaded Collision-Aware Global Routing With Bounded-Length Maze Routing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.5 (2013), pp. 709–722. DOI: 10.1109/TCAD.2012.2235124.

[85] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. "Swin transformer: Hierarchical vision transformer using shifted windows". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 10012–10022.

[86] Jinan Lou, S. Thakur, S. Krishnamoorthy, and H.S. Sheng. "Estimating routing congestion using probabilistic analysis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.1 (2002), pp. 32–41. DOI: 10.1109/43.974135.

[87] Yi-Chen Lu, Sai Pentapati, and Sung Kyu Lim. "The Law of Attraction: Affinity-Aware Placement Optimization Using Graph Neural Networks". In: *Proceedings of the 2021 International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 7–14. ISBN: 9781450383004. DOI: 10.1145/3439706.3447045. URL: https://doi.org/10.1145/3439706.3447045.

[88] Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis Jen-Hsin Huang, Chin-Chi Teng, and Chung-Kuan Cheng. "EPlace: Electrostatics-Based Placement

Using Fast Fourier Transform and Nesterov's Method". In: *ACM Trans. Des. Autom. Electron. Syst.* 20.2 (2015). ISSN: 1084-4309. DOI: 10.1145/2699873.

[89] Jingwei Lu, Hao Zhuang, Ilgweon Kang, Pengwen Chen, and Chung-Kuan Cheng. "EPlace-3D: Electrostatics Based Placement for 3D-ICs". In: *Proceedings of the 2016 on International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 11–18. ISBN: 9781450340397. DOI: 10.1145/2872334.2872361. URL: https://doi.org/10.1145/2872334.2872361.

[90] Jingwei Lu et al. "ePlace-MS: Electrostatics-Based Placement for Mixed-Size Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.5 (2015), pp. 685–698.

[91] Tao Luo and David Z. Pan. "DPlace2.0: A stable and efficient analytical placement based on diffusion". In: *2008 Asia and South Pacific Design Automation Conference*. 2008, pp. 346–351. DOI: 10.1109/ASPDAC.2008.4483972.

[92] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. "A graph placement methodology for fast chip design". In: *Nature* 594.7862 (2021), pp. 207–212.

[93] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. "Device placement optimization with reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2430–2439.

[94] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. "Module placement on BSG-structure and IC layout applications". In: *Proceedings of International Conference on Computer Aided Design*. 1996, pp. 484–491. DOI: 10.1109/ICCAD.1996.569870.

[95] Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Bruce Winter, and Mehmet Yildiz. "The ISPD2005 Placement Contest and Benchmark Suite". In: *Proceedings of the 2005 International Symposium on Physical Design*. San Francisco, California, USA:

Association for Computing Machinery, 2005, pp. 216–220. ISBN: 1595930213. DOI: 10.1145/1055137.1055182.

[96] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei A. Zaharia. "PipeDream: generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[97] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. "Memory-efficient pipeline-parallel dnn training". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 7937–7947.

[98] Deepak Narayanan et al. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476209. URL: https://doi.org/10.1145/3458817.3476209.

[99] Yurii Evgen'evich Nesterov. "A method of solving a convex programming problem with convergence rate O\bigl(k^2\bigr)". In: *Doklady Akademii Nauk*. Vol. 269. 3. Russian Academy of Sciences. 1983, pp. 543–547.

[100] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" In: *Queue* 6.2 (2008), pp. 40–53.

[101] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. "Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs". In: *International Conference on Learning Representations*. 2019.

[102] Min Pan and Chris Chu. "IPR: An Integrated Placement and Routing Algorithm". In: *2007 44th ACM/IEEE Design Automation Conference*. 2007, pp. 59–62.

[103] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. "Hetpipe: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism". In: *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 2020, pp. 307–321.

[104] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[105] Pitch Patarasuk and Xin Yuan. "Bandwidth optimal all-reduce algorithms for clusters of workstations". In: *Journal of Parallel and Distributed Computing* 69.2 (2009), pp. 117–124.

[106] Rajat Raina, Anand Madhavan, and Andrew Y Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 873–880.

[107] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '20. Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3505–3506. ISBN: 9781450379984.

[108] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*. Vol. 9351. Springer, 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[109] J.A. Roy, S.N. Adya, D.A. Papa, and I.L. Markov. "Min-cut floorplacement". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.7 (2006), pp. 1313–1326. DOI: 10.1109/TCAD.2005.855969.

[110] Jarrod A. Roy, Natarajan Viswanathan, Gi-Joon Nam, Charles J. Alpert, and Igor L. Markov. "CRISP: Congestion Reduction by Iterated Spreading during Placement". In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. San Jose, California: Association for Computing Machinery, 2009, pp. 357–362. ISBN: 9781605588001. DOI: 10.1145/1687399.1687467.

[111] Donald M. Schuler and Ernst G. Ulrich. "Clustering and Linear Placement". In: *Proceedings of the 9th Design Automation Workshop*. New York, NY, USA: Association for Computing Machinery, 1972, pp. 50–56. ISBN: 9781450374583. DOI: 10.1145/800153.804929. URL: https://doi.org/10.1145/800153.804929.

[112] C. Sechen and A. Sangiovanni-Vincentelli. "The TimberWolf placement and routing package". In: *IEEE Journal of Solid-State Circuits* 20.2 (1985), pp. 510–522. DOI: 10.1109/JSSC.1985.1052337.

[113] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. "Megatron-LM: Training multi-billion parameter language models using model parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[114] Gunilla Sköllermo. "A Fourier method for the numerical solution of Poisson's equation". In: *Mathematics of Computation* 29.131 (1975), pp. 697–711.

[115] Peter Spindler and Frank M. Johannes. "Fast and Accurate Routing Demand Estimation for Efficient Routability-driven Placement". In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364463.

[116] Peter Spindler, Ulf Schlichtmann, and Frank M. Johannes. "Kraftwerk2—A Fast Force-Directed Quadratic Placement Approach Using an Accurate Net Model". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.8 (2008), pp. 1398–1411. DOI: 10.1109/TCAD.2008.925783.

[117] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[118] Fan-Keng Sun and Yao-Wen Chang. “Big: A Bivariate Gradient-Based Wirelength Model for Analytical Circuit Placement”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[119] Wern-Jieh Sun and C. Sechen. “Efficient and effective placement for very large circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.3 (1995), pp. 349–359. DOI: 10.1109/43.365125.

[120] Taraneh Taghavi, Xiaojian Yang, and Bo-Kyung choi. “Dragon2005: Large-Scale Mixed-Size Placement Tool”. In: *Proceedings of the 2005 International Symposium on Physical Design*. San Francisco, California, USA: Association for Computing Machinery, 2005, pp. 245–247. ISBN: 1595930213. DOI: 10.1145/1055137.1055191. URL: https://doi.org/10.1145/1055137.1055191.

[121] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. “Automatic Graph Partitioning for Very Large-scale Deep Learning”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), pp. 1004–1013.

[122] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. “Piper: Multidimensional Planner for DNN Parallelization”. In: *Advances in Neural Information Processing Systems* 34 (2021).

[123] *Transistor count (Wikipedia)*. https://en.wikipedia.org/wiki/Transistor_count.

[124] N. Viswanathan and C.C.-N. Chu. “FastPlace: efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.5 (2005), pp. 722–733. DOI: 10.1109/TCAD.2005.846365.

[125] Natarajan Viswanathan, Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Haoxing Ren, and Chris Chu. “RQL: Global Placement via Relaxed Quadratic Spreading and

Linearization”. In: *2007 44th ACM/IEEE Design Automation Conference*. 2007, pp. 453–458.

[126] Fangzhou Wang, Jinwei Liu, and Evangeline F.Y. Young. “FastPass: Fast Pin Access Analysis with Incremental SAT Solving”. In: *Proceedings of the 2023 International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 9–16. ISBN: 9781450399784. DOI: 10.1145/3569052.3571879. URL: https://doi.org/10.1145/3569052.3571879.

[127] Fangzhou Wang, Lixin Liu, Jingsong Chen, Jinwei Liu, Xinshi Zang, and Martin D.F. Wong. “Starfish: An Efficient P&amp;R Co-Optimization Engine with A*-Based Partial Rerouting”. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Munich, Germany: IEEE Press, 2021, pp. 1–9.

[128] Yaoguang Wei, Cliff Sze, Natarajan Viswanathan, Zhuo Li, Charles J. Alpert, Lakshmi Reddy, Andrew D. Huber, Gustavo E. Tellez, Douglas Keller, and Sachin S. Sapatnekar. “GLARE: Global and Local Wiring Aware Routability Evaluation”. In: *Proceedings of the 49th Annual Design Automation Conference*. San Francisco, California: Association for Computing Machinery, 2012, pp. 768–773. ISBN: 9781450311991. DOI: 10.1145/2228360.2228499.

[129] Jurjen Westra, Chris Bartels, and Patrick Groeneveld. “Probabilistic Congestion Prediction”. In: *Proceedings of the 2004 International Symposium on Physical Design*. New York, NY, USA: Association for Computing Machinery, 2004, pp. 204–209. ISBN: 1581138172. DOI: 10.1145/981066.981110. URL: https://doi.org/10.1145/981066.981110.

[130] Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. “RouteNet: Routability prediction for Mixed-Size Designs Using Convolutional Neural Network”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, pp. 1–8. DOI: 10.1145/3240765.3240843.

[131] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. "PipeMare: Asynchronous Pipeline Parallel DNN Training". In: *Proceedings of Machine Learning and Systems*. Vol. 3. 2021, pp. 269–296.

[132] Haoyu Yang, Kit Fung, Yuxuan Zhao, Yibo Lin, and Bei Yu. "Mixed-Cell-Height Legalization on CPU-GPU Heterogeneous Systems". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 784–789. DOI: 10.23919/DATE54114.2022.9774671.

[133] Xiaojian Yang, Bo-Kyung Choi, and M. Sarrafzadeh. "Routability-driven white space allocation for fixed-die standard-cell placement". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.4 (2003), pp. 410–419. DOI: 10.1109/TCAD.2003.809660.

[134] Cunxi Yu and Zhiru Zhang. "Painting on Placement: Forecasting Routing Congestion Using Conditional Generative Adversarial Nets". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3317876. URL: https://doi.org/10.1145/3316781.3317876.

[135] Yanheng Zhang and Chris Chu. "CROP: Fast and effective congestion refinement of placement". In: *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. 2009, pp. 344–350.

[136] Lianmin Zheng et al. "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning". In: *Proc. OSDI*. 2022.

[137] Wenxing Zhu, Zhipeng Huang, Jianli Chen, and Yao-Wen Chang. "Analytical Solution of Poisson's Equation and Its Application to VLSI Global Placement". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, pp. 1–8. DOI: 10.1145/3240765.3240779.

[138] Ziran Zhu, Jianli Chen, Zheng Peng, Wenxing Zhu, and Yao-Wen Chang. "Generalized Augmented Lagrangian and Its Applications to VLSI Global Placement". In:

2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465922.

[139] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. "Parallelized Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. Vol. 23. 2010.

List of Publications

[1] Lixin Liu, Bangqi Fu, Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. “Xplace: An Extremely Fast and Extensible Placement Framework”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (under review)* (2023).

[2] Lixin Liu, Tianji Liu, Bentian Jiang, and Evangeline F. Y. Young. “Parmesan: Efficient Partitioning and Mapping Flow for DNN Training on General Device Topology”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (under review)* (2023).

[3] Bentian Jiang, Lixin Liu, Yuzhe Ma, Bei Yu, and Evangeline F. Y. Young. “Neural-ILT 2.0: Migrating ILT to Domain-Specific and Multitask-Enabled Neural Network”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.8 (2022), pp. 2671–2684.

[4] Bentian Jiang, Jingsong Chen, Jinwei Liu, Lixin Liu, Fangzhou Wang, Xiaopeng Zhang, and Evangeline F. Y. Young. “CU.POKer: Placing DNNs on WSE With Optimal Kernel Sizing and Efficient Protocol Optimization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.6 (2022), pp. 1888–1901.

[5] Lixin Liu, Bangqi Fu, Martin D. F. Wong, and Evangeline F. Y. Young. “Xplace: An Extremely Fast and Extensible Global Placement Framework”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. San Francisco, California: Association for Computing Machinery, 2022, pp. 1309–1314.

[6] Fangzhou Wang, Lixin Liu, Jingsong Chen, Jinwei Liu, Xinshi Zang, and Martin D.F. Wong. “Starfish: An Efficient P&amp;R Co-Optimization Engine with A*-Based Partial Rerouting”. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Munich, Germany: IEEE Press, 2021, pp. 1–9.

[7] Bentian Jiang, Xiaopeng Zhang, Lixin Liu, and Evangeline F.Y. Young. “Building up End-to-End Mask Optimization Framework with Self-Training”. In: *Proceedings of the 2021 International Symposium on Physical Design*. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 63–70.

[8] Bentian Jiang, Lixin Liu, Yuzhe Ma, Hang Zhang, Bei Yu, and Evangeline F. Y. Young. “Neural-ILT: Migrating ILT to Neural Networks for Mask Printability and Complexity Co-Optimization”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. Virtual Event, USA: Association for Computing Machinery, 2020.

[9] Bentian Jiang, Jingsong Chen, Jinwei Liu, Lixin Liu, Fangzhou Wang, Xiaopeng Zhang, and Evangeline F. Y. Young. “CU.POKer: Placing DNNs on Wafer-Scale AI Accelerator with Optimal Kernel Sizing”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. Virtual Event, USA: Association for Computing Machinery, 2020.

[10] Weiyang Liu, Rongmei Lin, Zhen Liu, Lixin Liu, Zhiding Yu, Bo Dai, and Le Song. “Learning towards Minimum Hyperspherical Energy”. In: *NeurIPS*. 2018.