Subitizing-Inspired Large Language Models for Floorplanning

Chen-Chen Yeh
Dept. of Computer Science & Engineering
Yuan Ze University
Taoyuan, 320315
s1111511@mail.yzu.edu.tw

Shao-Chien Lu
Dept. of Computer Science & Engineering
Yuan Ze University
Taoyuan, 320315
s1111534@mail.yzu.edu.tw

Hui-Lin Cho
Dept. of Computer Science & Engineering
Yuan Ze University
Taoyuan, 320315
s1112003@mail.yzu.edu.tw

Yu-Cheng Lin
Dept. of Computer Science & Engineering
Yuan Ze University
Taoyuan, 320315
linyu@saturn.yzu.edu.tw

Rung-Bin Lin
Dept. of Computer Science & Engineering
Yuan Ze University
Taoyuan, 320315
csrlin@saturn.yzu.edu.tw

Abstract— We present a novel approach to solving the floorplanning problem by leveraging fine-tuned Large Language Models (LLMs). Inspired by subitizing, the human ability to instantly count small numbers of items at a glance, we hypothesize that LLMs can similarly address floorplanning challenges accurately. Our experimental results demonstrate that LLMs achieve high success and optimal rates while attaining relatively low average dead space. These findings underscore the potential of LLMs as promising solutions for complex optimization tasks in VLSI design.

I. Introduction

As semiconductor manufacturing processes advance, chip designs increasingly incorporate various Intellectual Properties (IPs) to expedite development cycles. While integrating IPs accelerates the design process, it concurrently amplifies the complexity of chip layout allocation. Floorplanning, a pivotal stage in the Electronic Design Automation (EDA) workflow, entails the strategic placement of functional modules on a chip during the initial design phase. Effective floorplanning must navigate multiple constraints, including minimizing dead space and wire length, managing thermal dissipation, and optimizing the utilization of chip real estate. Additionally, the ability to rotate modules or adjust their width-to-height ratios can enhance their fitting within the designated chip area, further complicating the layout process.

Over the years, several representations for floorplanning results have been introduced, such as sequence pairs [1], normalized Polish expressions [2], slicing trees [3], and B*-trees [4]. Due to the NP-hard nature of the floorplanning problem, traditional approaches have predominantly employed optimization techniques like Simulated Annealing (SA) and Integer Linear Programming (ILP).

SA-based floorplanning has been extensively explored in early research. For instance, [2] utilized normalized Polish expressions in conjunction with SA to design efficient floorplans. Building upon this, [5] extended SA methodologies to accommodate three-dimensional floorplanning. Further advancements were made by [6], who introduced a fast three-stage SA algorithm based on B*-tree representation, achieving high success rates with reduced dead space. Similarly, [7] presented a hybrid SA approach that incorporated a novel greedy strategy alongside B*-tree representations, enhancing non-slicing floorplanning accuracy.

ILP approaches have been investigated for floorplanning. [8] employed ILP to simultaneously minimize interconnection delays and chip area, addressing critical performance metrics in floorplanning. Additionally, [9] demonstrated the application of ILP in reticle design and wafer dicing processes for multiple project wafers, highlighting its versatility in related semiconductor manufacturing tasks.

Furthermore, machine learning (ML) has opened new avenues for addressing floorplanning challenges. Researchers have increasingly explored ML-based solutions to complement or replace traditional optimization techniques. For example, [10] introduced GoodFloorplan, which leverages reinforcement learning to minimize both area and wire length, yielding impressive results on MCNC and GSRC benchmarks. Building on this, [11] adopted deep reinforcement learning strategies based on sequence pairs to achieve superior floorplanning solutions. Furthermore, [12] integrated hybrid reinforcement learning with genetic algorithms (GA), demonstrating significant reductions in wire length and area in the resultant floorplans.

Large Language Models (LLMs) have revolutionized various domains by automating complex tasks and enhancing decision-making processes. Despite their transformative impact, the application of LLMs to floorplanning remains relatively unexplored, primarily due to challenges related to data representation and the scarcity of large-scale, high-quality datasets. Inspired by the concept of subitizing, as first introduced by [13], which describes the human ability to instantly and accurately recognize the number

of items in a small set, we hypothesize that LLMs can similarly provide rapid and accurate solutions to floorplanning problems after exposure to a sufficient number of examples.

Supporting this hypothesis, [14] provided experimental evidence indicating that human reaction times for recognizing fewer than four items range between 40 to 100 milliseconds per item. However, for larger quantities, reaction times escalate to 250 to 350 milliseconds, suggesting a shift in cognitive processing strategies. They proposed three theories of enumeration: density-based, pattern-based, and working memory explanations. Drawing an analogy, we posit that LLMs can efficiently recognize and generate optimal solutions for small-scale floorplanning tasks by leveraging pattern recognition and memory-based strategies akin to human cognitive processes.

The paper is organized as follows: Section II provides a detailed description of the floorplanning problem, including the calculation of dead space and the iterative process for optimizing module placements. Section III outlines our proposed methodology, which comprises two primary stages: fine-tuning and inference. Section IV presents the experimental results of our approach, comparing two methodologies—local fine-tuning with Unsloth and leveraging the OpenAI API to fine-tune GPT4o-mini. Finally, Section V concludes the paper and outlines future research directions.

II. Problem Description

In the initial stages of the EDA workflow, floorplanning involves determining the approximate placement of each module on a chip. Given a set of n modules $P = \{p_1, p_2, \ldots, p_n\}$, each with specified dimensions—width w_i and height h_i—the objective is to arrange all modules within a two-dimensional space to minimize dead space. In this context, each module possesses a fixed shape, prohibiting rotations or dimensional adjustments to simplify the floorplanning process for LLMs. For example, Fig. 1 illustrates three modules needing to be floorplanned and one possible floorplan. As the number of modules increases, the problem becomes significantly more complex.

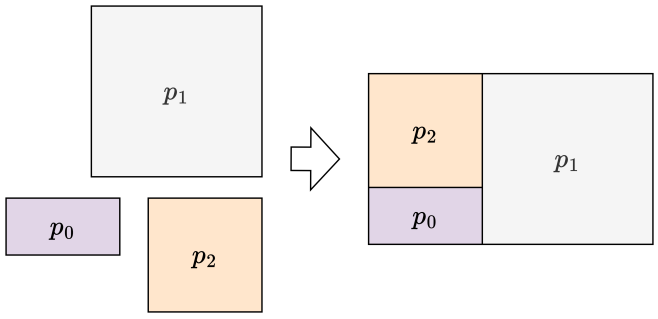


Fig. 1.: Floorplanning problem with three modules

To record floorplanning results, a straightforward approach is to directly store the coordinates of each module. However, this method becomes inefficient for large-scale floorplanning tasks, requiring substantial memory and computational resources. For representing a floorplan, there are two primary categories: non-slicing and slicing floorplans. A slicing floorplan implies that modules are obtained through horizontal or vertical cuts, whereas a non-slicing floorplan captures the relative vertical and horizontal relationships between modules.

This paper focuses on slicing floorplan representation, although potentially losing some possible floorplans, offers easier generation and storage. Fig. 2 illustrates a slicing floorplan and its corresponding slicing tree. We first slice vertically to separate p_1 from the composite module containing p_0 and p_2, denoting the root node as "V" (vertical slicing). The left (or bottom) module is assigned as the left child node to maintain consistency. Subsequently, a horizontal slice separates p_0 and p_2, represented by the "H" in the tree. The final tree structure is shown in the figure.

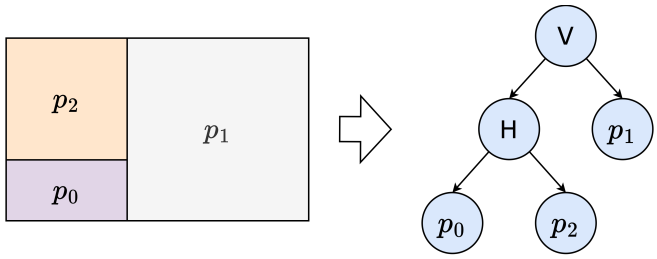


Fig. 2.: Turning a slicing floorplan into a slicing tree

Dead space is defined as the unused area that arises from gaps between adjacent modules. The calculation of dead space depends on the relative positioning of the modules. Specifically, when two modules are placed side by side along the x-axis, the dead space is computed as the product of the width of the module with the smaller height and the absolute difference in their heights. Similarly, when two modules are stacked along the y-axis, the dead space is computed as the product of the height of the module with the smaller width and the absolute difference in their widths.

For two adjacent modules p_i and p_j with dimensions (w_i, h_i) and (w_j, h_j), respectively, the dead space is defined as follows:

$$\textbf{dead\_space}(p_i, p_{i+1}) = \begin{cases} \textbf{ds}_{\textbf{hor}}(p_i, p_{i+1}), & \textbf{if } p_i \textbf{ and } p_{i+1} \\ & \textbf{are horizontally adjacent,} \\ \textbf{ds}_{\textbf{ver}}(p_i, p_{i+1}), & \textbf{if } p_i \textbf{ and } p_{i+1} \\ & \textbf{are vertically adjacent.} \end{cases} \tag{1}$$

where the $\textbf{ds}_{\textbf{hor}}(p_i, p_{i+1})$ and $\textbf{ds}_{\textbf{ver}}(p_i, p_{i+1})$ functions are defined as follows:

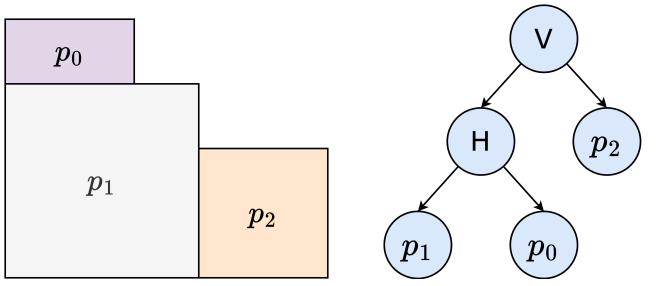
$$\textbf{ds}_{\textbf{hor}}(p_i, p_j) = \begin{cases} w_i \times |h_i - h_j|, & \textbf{if } h_i \leq h_j, \\ w_j \times |h_i - h_j|, & \textbf{otherwise}. \end{cases} \tag{2}$$

$$\textbf{ds}_{\textbf{ver}}(p_i, p_j) = \begin{cases} h_i \times |w_i - w_j|, & \textbf{if } w_i \leq w_j, \\ h_j \times |w_i - w_j|, & \textbf{otherwise}. \end{cases} \tag{3}$$

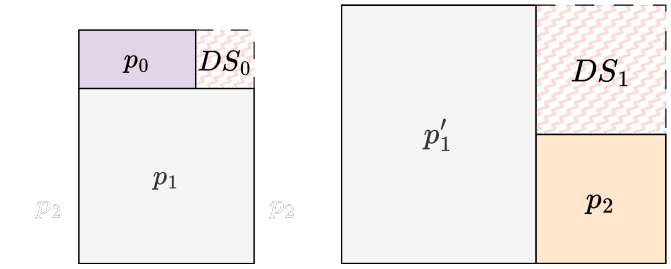
For scenarios involving more than two modules, the dead space calculation is generalized through an iterative process that selects two modules, calculates their dead space, and merges them into a single composite module. This process continues until all modules are combined into a single composite module. The dimensions of the composite module are updated as follows:

$$\textbf{if horizontally adjacent:} \quad w_{\textbf{new}} = w_i + w_{i+1}, \quad h_{\textbf{new}} = \max(h_i, h_{i+1}), \tag{4}$$

$$\textbf{if vertically adjacent:} \quad w_{\textbf{new}} = \max(w_i, w_{i+1}), \quad h_{\textbf{new}} = h_i + h_{i+1}. \tag{5}$$



(a) Non-optimal floorplan (b) Slicing tree for the floorplan



(c) Dead space for horizontal adjacency (d) Dead space for vertical adjacency

Fig. 3.: Dead space calculation for floorplanning (DS: Dead Space)

Fig. 3 illustrates examples of dead space calculations for horizontally and vertically adjacent modules. Fig. 3a shows a non-optimal floorplan, and Fig. 3b depicts the corresponding slicing tree. We can traverse the tree in post-order, calculate the dead space between two adjacent modules, merge them into a composite module, and update the dimensions. Fig. 3c and 3d show the dead space calculation for horizontal and vertical adjacency, respectively. In this case, the total dead space is computed as $DS_0 + DS_1$. The optimal floorplan is defined as the one with the minimum dead space.

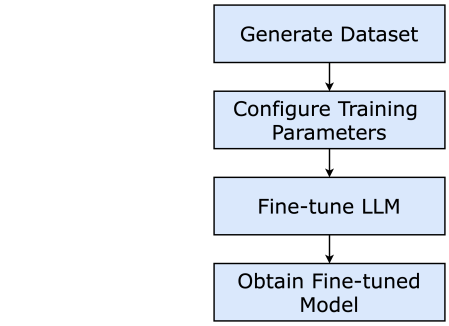
III. Methodology

Fig. 4 illustrates an overview of our proposed workflow, which comprises two primary stages: fine-tuning and inference.

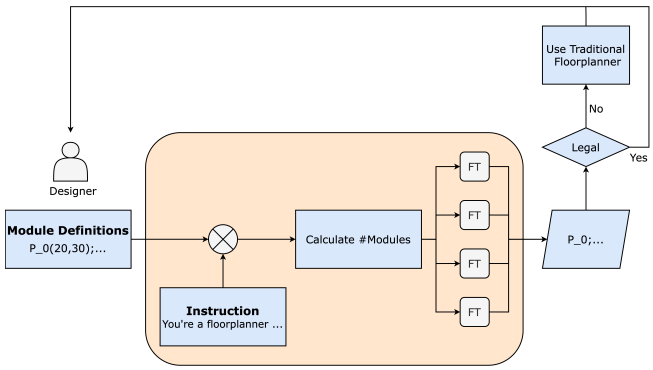
A. Fine-tuning Stage

The fine-tuning stage, depicted in Fig. 4a, involves generating a comprehensive dataset and subsequently fine-tuning a Large Language Model (LLM) to predict optimal floorplanning solutions.

Dataset Generation: The scarcity of extensive floorplanning datasets has historically limited the effectiveness of machine learning approaches in this domain. While one might consider using an existing floorplanner to generate large datasets, this approach is time-consuming and relies on obtaining circuit data (e.g., from the MCNC benchmarks), which is not always feasible. Moreover, even real data generated by floorplanners may not be optimal. To overcome these challenges and inspired by [15], we introduce a novel data generation technique based on recursive slicing and tree encoding. The process, illustrated in Fig. 5, comprises the following steps:



(a) Fine-tuning stage



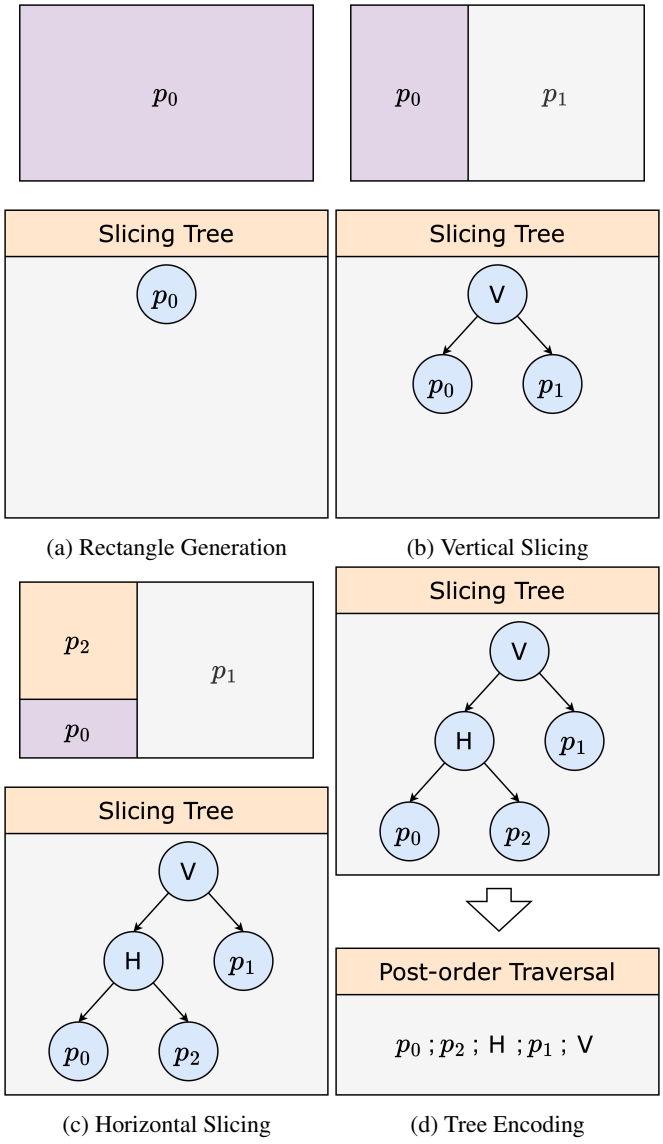
(b) Inference stage (FT: Fine-tuned model)

Fig. 4.: Overview of our workflow

1. **Initial Rectangle Generation: Generate a rectangle representing the chip boundary with random width and height.**
2. **Recursive Slicing:**
   - **Randomly select a rectangle (module) and slice it either horizontally or vertically.**
   - **Each slicing operation yields two adjacent modules; the slicing direction determines whether the width or height remains constant.**
   - **Designate the left (or bottom) module as the left child node in the slicing tree.**
   - **Continue the recursive slicing until a complete slicing tree is formed.**
3. **Tree Encoding: Encode the resultant slicing tree using post-order traversal. Each module is labeled as p_i, and slicing operations are denoted by 'H' (horizontal) or 'V' (vertical), separated by semicolons.**

We choose a slicing tree representation because it enables easier dataset generation compared to alternative methods. In particular, using post-order traversal (instead of pre-order) avoids the generation of long sequences of consecutive "H" or "V" operations, which can lead to errors in LLM outputs. For example, a pre-order traversal might yield "V;H;p_0;p_2;p_1," which becomes increasingly problematic as dataset size grows. This encoding effectively captures the hierarchical structure of the floorplan, enabling the creation of large-scale datasets necessary for robust LLM fine-tuning.

The fine-tuning process involves training the LLM on datasets tailored to specific module counts (16 and 24 modules) to validate our hypothesis that LLMs can perform floorplanning in a



(a) Rectangle Generation (b) Vertical Slicing

(c) Horizontal Slicing (d) Tree Encoding

Fig. 5.: Illustration of the recursive slicing and tree encoding process.

manner akin to human subitizing. The input to the LLM consists of module names along with their respective widths and heights, combined with background instructions. The desired output is a slicing tree in post-order traversal that represents the optimal floorplan.

B. Inference Stage

In the inference stage, as illustrated in Fig. 4b, the fine-tuned LLM receives a set of modules as input and generates a corresponding floorplanning solution. However, due to inherent limitations in language models, the LLM occasionally produces not optimal results or even illegal slicing trees. Consequently, the LLM's output cannot be used directly; instead, it should be integrated with a traditional floorplanner to provide guidance, rather than replacing conventional floorplanning methods.

C. Experimental Setup

We conducted experiments using two approaches. First, we utilize Unsloth [16], an efficient framework for locally fine-tuning open-source LLMs. Second, we leverage the OpenAI API to fine-tune a proprietary LLM, specifically GPT4o-mini [17].

C.1. Dataset Specifications

- **Module Counts: Experiments were performed on datasets with 16 and 24 modules. These counts were inspired by the ami33 and ami49 circuits from the MCNC benchmark, originally containing 33 and 49 modules, respectively. To align with our model requirements, we combined two to four modules with identical widths or heights into single modules.**
- **Dataset Sizes: The training datasets comprised 80,000 and 120,000 optimal floorplans for the 16-module and 24-module cases, respectively. Larger module counts necessitated larger datasets to enhance fine-tuning performance.**
- **Test Samples: Each model was evaluated on 50 unseen test samples generated using the same recursive slicing and encoding methodology as the training data.**

C.2. Metrics for Evaluation

- **Success Rate (S): Percentage of test samples yielding a legal slicing tree.**
- **Optimal Result Rate (O): Percentage of floorplans with zero dead space, indicating an optimal solution.**
- **Dead Space Ratio (D): Ratio of dead space to the summation of all module areas in the floorplan.**

The metrics are mathematically defined as:

$$S = \frac{N_{\text{legal}}}{N_{\text{total}}} \times 100\% \tag{6}$$

$$O = \frac{N_{\text{optimal}}}{N_{\text{total}}} \times 100\% \tag{7}$$

$$D = \frac{\sum_{i=1}^{n-1} \textbf{dead\_space}(p_i, p_{i+1})}{\sum_{i=1}^{n} (w_i \times h_i)} \tag{8}$$

where N_{legal} is the number of test samples with legal slicing trees, N_{total} is the total number of test samples, N_{optimal} is the number of optimal floorplans, and dead_space(p_i, p_{i+1}) is the dead space between two adjacent modules and can be calculated using Equations 1.

C.3. Model Fine-tuning Details

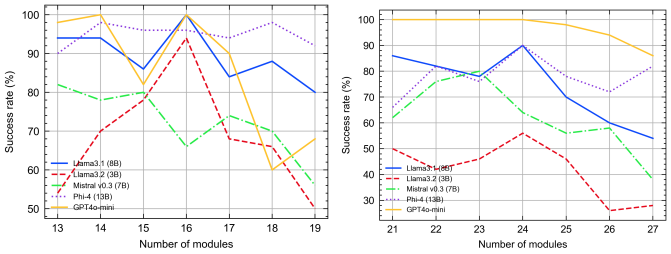
- **Local LLM Fine-tuning: Models fine-tuned locally include LLaMA 3.1 (8B) [18], LLaMA 3.2 (3B) [19], Mistral v0.3 (7B) [20], and Phi-4 (13B) [21]. The Unsloth framework was chosen for its efficiency, achieving 2-5x faster fine-tuning with 70% less memory usage compared to traditional frameworks. Fine-tuning was performed on a single NVIDIA GeForce RTX 4070 Ti Super with 16 GB of VRAM over 200 epochs, completing in approximately 30 minutes.**
- **OpenAI API Fine-tuning: Due to hardware constraints that made it challenging to fine-tune larger LLMs locally, we opted to fine-tune OpenAI's GPT4o-mini. Initial fine-tuning**

on the 16-module dataset yielded promising results. Although we expanded the dataset size to test the capability of LLMs locally, budget limitations restricted us to re-fine-tuning only the 16-module model, resulting in 6,253 samples for GPT4o-mini.

IV. Experimental Results

We present our experimental results for two approaches: (1) using Unsloth [16], an efficient framework for fine-tuning open-source LLMs locally, and (2) leveraging the OpenAI API to fine-tune a proprietary LLM for the floorplanning task. The experiments are conducted on datasets containing 16 and 24 modules. The choice of these module counts was inspired by the ami33 and ami49 circuits from the MCNC benchmark, which originally contain 33 and 49 modules, respectively. To align with the requirements of our fine-tuned models, we combine two to four modules (with the same width or height) into a single module. However, since benchmark data does not always form rectangles, our experiments are based on custom benchmarks that will be made publicly available in the future. For each test sample, the LLM generates five outputs, and the best result is selected for evaluation. The equations used to compute these metrics are provided in Equations 6 to 8.

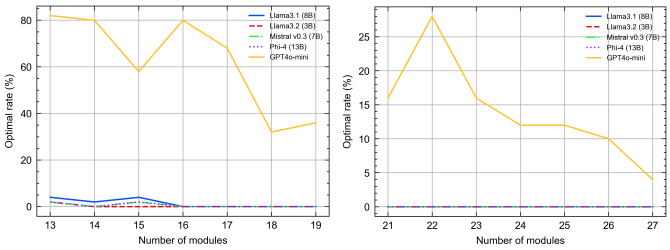
Fig. 6 illustrates the success rate for the different fine-tuned LLMs, which is defined as the percentage of test samples that produce a legal slicing tree. In the 16-module scenario (Fig. 6a), GPT4o-mini, Phi-4 (13B), and Llama3.1 (8B) achieve high success rates—with nearly 100% at 16 modules—while Mistral v0.3 (7B) shows a noticeable decline at 16 modules, failing to meet our hypothesis that the model should complete the specified module count. Similarly, in the 24-module scenario (Fig. 6b), GPT4o-mini demonstrates near-perfect performance at lower module counts with slight declines as the module count increases. In contrast, Phi-4 (13B) and Llama3.1 (8B) perform well for the 16-module case, whereas Mistral v0.3 (7B) and Llama3.2 (3B) achieve only around a 60% success rate. Overall, these results indicate that GPT4o-mini, Phi-4 (13B), and Llama3.1 (8B) are the most robust models for generating legal slicing trees in our floorplanning tasks.



(a) 16-module scenario (b) 24-module scenario

Fig. 6.: Success rate of fine-tuned LLMs for different module counts.

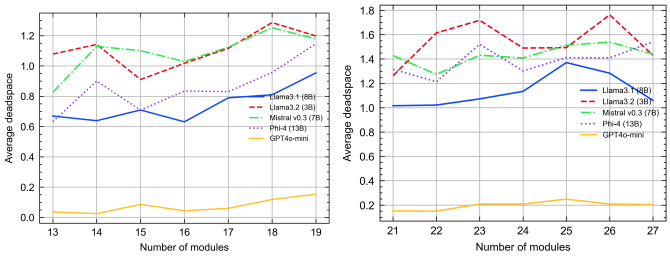
Fig. 7 shows the optimal rate for two fine-tuned LLMs, defined as the percentage of floorplanning results that achieve an optimal floorplan. In the 16-module scenario (Fig. 7a), GPT4o-mini exhibits substantially higher optimal rates (ranging from 57% to 82%) compared to the other models. In contrast, Llama3.1 (8B) records optimal rates up to only 4% for some module counts, while Llama3.2 (3B), Mistral v0.3 (7B), and Phi-4 (13B) achieve negligible rates (mostly between 0% and 2%). In the 24-module scenario (Fig. 7b), GPT4o-mini attains optimal rates up to 28% at 22 modules, though it decreases to 12% at 24 modules. The other models consistently produce 0% optimal results. These findings indicate that GPT4o-mini is significantly more effective at generating optimal floorplanning solutions, especially as the number of modules increases, though its performance decreases with higher module counts, reflecting inherent limitations.



(a) 16-module scenario (b) 24-module scenario

Fig. 7.: Optimal rate of fine-tuned LLMs for different module counts.

Fig. 8 presents the average dead space values for the legal slicing trees generated by the fine-tuned LLMs. (Only legal slicing trees are considered in the computation.) Lower values indicate a more efficient floorplan with less unused space. In the 16-module scenario (Fig. 8a), GPT4o-mini achieves very low average dead space values (ranging from 0.03 to 0.15), whereas the other models exhibit significantly higher values: Llama3.1 (8B) ranges from 0.63 to 0.95, Llama3.2 (3B) from 0.91 to 1.20, Mistral v0.3 (7B) from 0.83 to 1.18, and Phi-4 (13B) from 0.63 to 1.15. In the 24-module scenario (Fig. 8b), GPT4o-mini maintains low average dead space values (between 0.15 and 0.25), while Llama3.1 (8B) ranges from 1.02 to 1.37, Llama3.2 (3B) from 1.26 to 1.76, Mistral v0.3 (7B) from 1.27 to 1.54, and Phi-4 (13B) from 1.21 to 1.54. These results demonstrate that GPT4o-mini not only generates legal slicing trees at a higher success rate but also produces significantly lower dead space, indicating a more efficient allocation of chip area.



(a) 16-module scenario (b) 24-module scenario

Fig. 8.: Average dead space of fine-tuned LLMs for different module counts.

V. Conclusion

We introduced an innovative approach to solving the floorplanning problem in VLSI design by leveraging fine-tuned Large Lan-

guage Models (LLMs). We developed an efficient representation and a novel method for generating high-quality datasets necessary for effective LLM fine-tuning. Our experimental evaluation revealed that fine-tuned LLMs, particularly GPT4o-mini, achieve high success and optimal rates while producing significantly lower average dead space, resulting in more efficient floorplans compared to traditional and other ML-based methods. These findings support our hypothesis that LLMs can perform floorplanning tasks in a manner akin to human subitizing, demonstrating their potential as powerful tools for addressing complex optimization problems in VLSI design.

Despite these promising results, the performance of GPT4o-mini diminishes at 24-module scenario, highlighting the need for further research to enhance scalability and incorporate additional design constraints. Future work should explore the integration of advanced LLM architectures and the inclusion of comprehensive optimization objectives—such as wire length minimization, thermal management, and power consumption—to develop more robust and practical floorplanning solutions. Our findings pave the way for more intelligent and efficient design automation processes in VLSI design.

Acknowledgments

We would like to express our gratitude to OpenAI for providing access to ChatGPT enhancing the clarity, coherence, and overall quality of this paper. The successful completion of this research was supported by the academic resources and research infrastructure provided by the Taiwan Semiconductor Research Institute, National Institutes of Applied Research. We hereby express our sincere gratitude. This work was supported by National Science Council of Taiwan under Grant Numbers 113-2640-E-155-001.

References

[1] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 472–479, 1995.

[2] DF Wong and CL Liu. A new algorithm for floorplan design. In *23rd ACM/IEEE Design Automation Conference*, pages 101–107. IEEE, 1986.

[3] Ralph HJM Otten. Automatic floorplan design. In *19th Design Automation Conference*, pages 261–267. IEEE, 1982.

[4] Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu, and Shu-Wei Wu. B*-trees: A new representation for non-slicing floorplans. In *Proceedings of the 37th Annual Design Automation Conference*, pages 458–463, 2000.

[5] Lei Cheng, Liang Deng, and Martin DF Wong. Floorplanning for 3-d vlsi design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 405–411, 2005.

[6] Tung-Chieh Chen and Yao-Wen Chang. Modern floorplanning based on b/sup*/-tree and fast simulated annealing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(4):637–650, 2006.

[7] Jianli Chen, Wenxing Zhu, and M Montaz Ali. A hybrid simulated annealing algorithm for nonslicing vlsi floorplanning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(4):544–553, 2010.

[8] Suphachai Sutanthavibul, Eugene Shragowitz, and J Ben Rosen. An analytical approach to floorplan design and optimization. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 187–192, 1991.

[9] Meng-Chiou Wu and Rung-Bin Lin. Reticle floorplanning and wafer dicing for multiple project wafers. In *Sixth international symposium on quality electronic design (isqed'05)*, pages 610–615. IEEE, 2005.

[10] Qi Xu, Hao Geng, Song Chen, Bo Yuan, Cheng Zhuo, Yi Kang, and Xiaoqing Wen. Goodfloorplan: Graph convolutional network and reinforcement learning-based floorplanning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(10):3492–3502, 2021.

[11] Shenglu Yu, Shimin Du, and Chang Yang. A deep reinforcement learning floorplanning algorithm based on sequence pairs. *Applied Sciences*, 14(7):2905, 2024.

[12] Ke Liu, Jian Gu, Hao Gu, and Ziran Zhu. A hybrid reinforcement learning and genetic algorithm for vlsi floorplanning. In *Proceedings of the 2023 15th International Conference on Machine Learning and Computing*, pages 412–418, 2023.

[13] Edna L Kaufman, Miles W Lord, Thomas Whelan Reese, and John Volkmann. The discrimination of visual number. *The American journal of psychology*, 62(4):498–525, 1949.

[14] Lana M Trick and Zenon W Pylyshyn. Why are small and large numbers enumerated differently? a limited-capacity preattentive stage in vision. *Psychological review*, 101(1):80, 1994.

[15] Chin-Chih Chang, Jason Cong, and Min Xie. Optimality and scalability study of existing placement algorithms. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 621–627, 2003.

[16] Michael Han Daniel Han and Unsloth team. Unsloth, 2023.

[17] OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence`, 2024.

[18] Meta. Introducing llama 3.1: Our most capable models to date. `https://ai.meta.com/blog/meta-llama-3-1`, 2024.

[19] Meta. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. `https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices`, 2024.

[20] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[21] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.