UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

SOFIA POPSIN GOMES

High-Level Synthesis with Large Language Models: AI-Automated Directive Injection

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Gabriel Luca Nazar
Coadvisor: Prof. Dr. André Grahl Pereira

Porto Alegre
July 2026

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Marcia Cristina Bernardes Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitora de Graduação: Profª. Nádya Pesce da Silveira
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do Curso de Engenharia de Computação: Prof. Paulo Francisco Butzen
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“Sometimes, the best way to convince someone they are wrong is to let them go their own way.”

— Unknown

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Prof. Dr. Gabriel Luca Nazar, and my co-advisor, Prof. Dr. André Grahl Pereira, for their guidance, patience, and availability throughout this work. The knowledge and direction I received from both of you were essential for this research to come together.

To my parents, Paulo and Joseida, I am deeply grateful for your constant support. You encouraged me and stood by my side through every difficulty, and this achievement belongs to you as much as it does to me.

To my family, thank you for understanding my absence, always justified by my dedication to my studies. Your support, even from a distance, was greatly appreciated.

To all the friends I have made throughout my life, thank you for being part of my story and for shaping, each in your own way, the person I am today.

To the women I met during my undergraduate years, thank you for contributing to this journey. The friendship, exchange, and support among us made this experience worthwhile.

To all of you, my sincere gratitude.

ABSTRACT

This work investigates the use of Large Language Models (LLMs) for the automated selection and injection of optimization directives in High-Level Synthesis (HLS) projects. The manual choice of these directives directly impacts performance, area utilization, and power consumption in FPGA-based systems, but typically requires prior experience and specialized domain knowledge. An automated pipeline was developed to leverage commercial LLMs from the Google Gemini and Gemma families to analyze C/C++ source code and inject suitable pragmas to optimize loops, functions, and memory accesses, using a zero-shot prompt engineering strategy. The pipeline was evaluated against the MLDHLS benchmark suite using AMD Vitis HLS, and results were compared against an unoptimized baseline, the MLDHLS best-known solutions, and the HGBO state-of-the-art framework. The findings demonstrate that frontier LLMs can produce competitive directive configurations in a single inference pass, in some cases surpassing both the dataset reference and the iterative ML-based baseline, while also revealing systematic failure modes specific to zero-shot HLS optimization.

Keywords: High-Level Synthesis (HLS). Large Language Models (LLMs). Design Space Exploration (DSE). Automated Hardware Design.

Síntese de Alto Nível com Modelos de Linguagem de Grande Porte: Injeção de Diretivas Automatizada por IA

RESUMO

Este trabalho investiga o uso de Modelos de Linguagem de Grande Porte (LLMs) para a seleção e injeção automatizada de diretivas de otimização em projetos de Síntese de Alto Nível (HLS). A escolha manual dessas diretivas impacta diretamente o desempenho, a utilização de área e o consumo de potência em sistemas baseados em FPGA, mas tipicamente requer experiência prévia e conhecimento especializado de domínio. Um pipeline automatizado foi desenvolvido para utilizar LLMs comerciais das famílias Google Gemini e Gemma na análise de código-fonte C/C++ e na injeção de pragmas adequados para otimizar laços, funções e acessos à memória, por meio de uma estratégia de engenharia de prompt zero-shot. O pipeline foi avaliado sobre o conjunto de benchmarks MLDHLS utilizando o AMD Vitis HLS, e os resultados foram comparados contra uma baseline não otimizada, as melhores soluções conhecidas do dataset MLDHLS e o framework de estado da arte HGBO. Os resultados demonstram que LLMs de fronteira são capazes de produzir configurações de diretivas competitivas em uma única inferência, superando em alguns casos tanto a referência do dataset quanto a baseline iterativa baseada em aprendizado de máquina, ao mesmo tempo em que revelam modos de falha sistemáticos específicos à otimização HLS por abordagem zero-shot.

Palavras-chave: Síntese de Alto Nível (HLS). Modelos de Linguagem de Grande Porte (LLMs). Exploração de Espaço de Projeto (DSE). Projeto de Hardware Automatizado..

LIST OF FIGURES

Figure 2.1 Traditional RTL flow compared to the HLS-based flow. 15
Figure 2.2 Optimization directives guiding the HLS translation process. 16
Figure 2.3 Illustration of the PIPELINE, UNROLL, and ARRAY_PARTITION directives. 16
Figure 2.4 Design space configurations across the Power, Performance, and Area (PPA) trade-off. 17
Figure 2.5 Simplified overview of LLM training from massive datasets. 18
Figure 2.6 Prompt engineering strategy combining task definition, persona, and RAG. 18

Figure 3.1 Automated directive injection pipeline overview. 23
Figure 3.2 Exploratory test: Bubble Sort directive injection via ChatGPT. 24
Figure 3.3 Prompt structure for the `bbgemm` benchmark (abbreviated) 31
Figure 3.4 `run_hls.tcl` script for disabling automatic optimizations. 37

Figure 4.1 Number of source files per benchmark versus synthesis failure count. 53
Figure 4.2 Total lines of code per benchmark versus synthesis failure count. 53

LIST OF TABLES

Table 3.1 MLDHLS benchmark suite: algorithms and descriptions.....25
Table 3.2 Input size per benchmark (number of source files and lines of code).....26
Table 3.3 Unoptimized baseline synthesis latency results. 26
Table 3.4 MLDHLS dataset best-known latency results.....27
Table 3.5 HGBO best latency results under PPA configuration (100 iterations). 28
Table 3.6 Overview of evaluated LLM architectures and capabilities. 32

Table 4.1 Gemini-2.5-Flash synthesis results per run. 39
Table 4.2 Gemini-2.5-Flash average clock cycles and timing status per benchmark. 39
Table 4.3 Resource utilization: extreme-acceleration cases (Gemini-2.5-Flash).....41
Table 4.4 Gemini-3-Flash synthesis results per run. 42
Table 4.5 Gemini-3-Flash average clock cycles and timing status per benchmark. 42
Table 4.6 Resource utilization: extreme-acceleration cases (Gemini-3-Flash).....43
Table 4.7 Gemma-3-27b-it synthesis results per run. 44
Table 4.8 Gemma-3-27b-it average clock cycles and timing status per benchmark.....45
Table 4.9 Gemma-4-31b-it synthesis results per run. 46
Table 4.10 Gemma-4-31b-it average clock cycles and timing status per benchmark. 47
Table 4.11 Resource utilization: extreme-acceleration cases (Gemma-4-31b-it). 49
Table 4.12 Global latency performance comparison across LLM architectures (cycles).50
Table 4.13 Synthesis outcome summary per model (out of 10 benchmarks). 52

LIST OF ABBREVIATIONS AND ACRONYMS

ADRS	Average Distance from Reference Set
API	Application Programming Interface
AXI	Advanced eXtensible Interface
BOME	Bayesian Optimization-based Multi-objective Exploration
BRAM	Block Random-Access Memory
DSE	Design Space Exploration
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HGBO	Hierarchical Graph Bayesian Optimization
HGP	Hierarchical Graph Neural Network Predictor
HLS	High-Level Synthesis
ICL	In-Context Learning
II	Initiation Interval
LLM	Large Language Model
LUT	Look-Up Table
MLDHLS	Multiple Loop Directive HLS Dataset
NSGA-II	Non-dominated Sorting Genetic Algorithm II
PLDA	Power, Latency, Delay, and Area
PPA	Power, Performance, and Area
QoR	Quality of Results
RAG	Retrieval-Augmented Generation

RTL	Register-Transfer Level
SA	Simulated Annealing
TCL	Tool Command Language
TDM	Tree-structured Design Space Modeler

CONTENTS

1 INTRODUCTION **12**
1.1 Use of Artificial Intelligence Tools **14**
2 LITERATURE REVIEW **15**
2.1 High-Level Synthesis (HLS) **15**
2.2 Optimization Directives (Pragmas) **16**
2.3 Design Space Exploration (DSE) **17**
2.4 Large Language Models (LLMs) and Prompt Engineering **18**
2.5 Related Work **19**
3 METHODOLOGY **22**
3.1 Preliminary Exploratory Tests **22**
3.2 Benchmark Suite (MLDHLS dataset) **23**
3.3 Reference Points for Comparison **24**
3.3.1 Unoptimized Baseline 26
3.3.2 MLDHLS Best Known Solutions 27
3.3.3 HGBO State-of-the-Art Baseline 28
3.4 Prompt Engineering **29**
3.5 Automated Directive Injection **30**
3.6 Codes Extraction **33**
3.7 Synthesis **34**
3.7.1 Synthesis Configuration and Tcl Scripts 35
4 EXPERIMENTAL RESULTS AND DISCUSSION **38**
4.1 Gemini-2.5-Flash **38**
4.2 Gemini-3-Flash **41**
4.3 Gemma-3-27b-it **44**
4.4 Gemma-4-31b-it **46**
4.5 Comparative Analysis **49**
4.5.1 Performance Against Reference Points 50
4.5.2 Cross-Model Convergence 51
4.5.3 Synthesis Coverage and Validity Summary 51
4.5.4 Input Complexity and Model Performance 52
4.6 Synthesis Failure Analysis **53**
4.6.1 Pragma Scope Violations 54
4.6.2 AXI Interface Bundle Conflicts 54
4.6.3 Incomplete Code Delivery 54
4.6.4 Variable Hallucination in Directive Arguments 55
4.6.5 Conflicting Pragma Combinations 55
4.6.6 Source Code Corruption 55
5 CONCLUSION **57**
5.1 Future Work **58**
REFERENCES **60**

1 INTRODUCTION

As Moore's Law and Dennard Scaling reached their end, new ways to improve performance started being explored. One such approach is the use of customized dedicated hardware (HENNESSY; PATTERSON, 2019), which is increasingly present and necessary in both embedded applications and heterogeneous microprocessor architectures. This is due to its ability to improve computational performance and reduce energy consumption in the applications where it is applied.

However, despite the significant contribution of customized hardware to computational efficiency, developing these devices is time-consuming and requires extensive knowledge and experience from the designer. These circuits are typically described at the Register-Transfer Level (RTL) using Hardware Description Languages (HDLs) such as VHDL and Verilog. This approach has always posed challenges, but the situation worsened as designs became increasingly complex (CHI et al., 2022), making the design process both costly and time-consuming.

In this context, driven by the need for a more efficient alternative, High-Level Synthesis (HLS) matured and established itself. It made domain-specific hardware more accessible to developers, who no longer required deep knowledge of HDLs, since HLS accepts languages such as C and C++ (NUMAN et al., 2020). These languages operate at higher levels of abstraction and possess a broader user base.

However, since these languages were originally designed for software execution, they primarily describe *what* is being computed. While HLS tools possess default behaviors to map this code into hardware, the resulting synthesis often fails to meet specific performance or area requirements. Code optimizations can be guided through specific hardware directives (Advanced Micro Devices, Inc., 2023), allowing the designer to refine the final implementation and achieve desired architectural characteristics.

These specifications are implemented as code directives, provided via external scripts or directly within the source code as pragmas (Advanced Micro Devices, Inc., 2023). These pragmas guide the HLS tool to override its default heuristics with specific optimization instructions. They are typically inserted as code annotations, resembling special comments. By leveraging these directives, the designer can specify desired architectural characteristics, constraints, and optimization strategies, such as parallelism and pipelining, without modifying the original algorithmic logic.

Nevertheless, this shift in focus has introduced a new challenge to hardware de-

velopment. Currently, the difficulty lies in determining where to apply these directives, which ones to select, and how to configure them (SCHAFER; WANG, 2020). This challenge is driven by the vast number of possible configurations, which creates an extremely large design space that is difficult to navigate.

In this scenario, Generative Artificial Intelligence, specifically Large Language Models (LLMs), emerge as a promising solution. With the increasing use of these tools for code generation, they have become a viable alternative for the directive selection task. Unlike conventional Design Space Exploration (DSE) methods, which typically rely on heuristic algorithms (SCHAFER; WANG, 2020; SOHRABIZADEH et al., 2022), LLMs can interpret C/C++ code semantics and suggest optimizations based on learned patterns (YAO et al., 2025), acting as an expert assistant.

The objective of this work is to investigate the viability of using commercial LLMs, specifically those provided by the Google Gemini API (TEAM, 2025a; TEAM, 2024), for the automated injection of optimization directives. This work proposes the development of an automated pipeline that utilizes Prompt Engineering (Zero-Shot) to analyze kernels and insert pragmas, evaluating whether this low-cost approach can compete with traditional methods and established baselines.

The contributions of this work are:

- A functional and automated pipeline for the integration between the Google Gemini API and the AMD Vitis HLS tool, enabling zero-shot directive injection at scale;
- A systematic evaluation of four commercial LLMs against established HLS optimization baselines;
- A taxonomy of recurring failure modes in zero-shot LLM-based directive injection, characterizing systematic, model-specific limitations.

The remainder of this work is organized as follows: Section 2 provides the theoretical background and reviews related works that represent the current state of the art. Section 3 details the methodology used for the pipeline development and the evaluation criteria. Section 4 presents the results, and Section 5 concludes the work with final considerations.

1.1 Use of Artificial Intelligence Tools

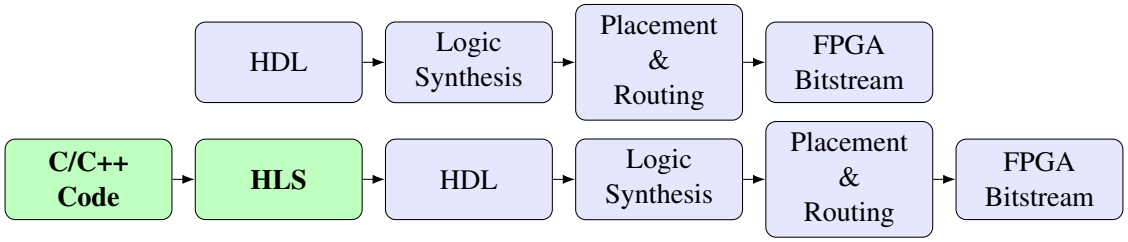
Generative Artificial Intelligence (AI) tools were used as a writing aid during the preparation of this text, specifically to support the organization of the semantic structure of certain passages. All AI-assisted content was critically reviewed, edited, and validated by the author, who assumes full responsibility for the content, analyses, and conclusions presented in this work. It is worth noting that this disclosure refers exclusively to writing assistance. The use of Large Language Models as the object of investigation in the automated directive injection pipeline, described throughout this work, is a separate and central contribution of the work itself.

2 LITERATURE REVIEW

This section presents the theoretical foundations essential to this work. The subjects covered include key aspects of hardware development, such as High-Level Synthesis (HLS), optimization directives, and DSE (*Design Space Exploration*). Furthermore, this section covers fundamental concepts of Artificial Intelligence, specifically Large Language Models (LLMs) and Prompt Engineering. Finally, a review of related work is provided, representing the state of the art in leveraging LLMs for HLS directive selection.

2.1 High-Level Synthesis (HLS)

Figure 2.1: Traditional RTL flow compared to the HLS-based flow.



Source: The author

High-Level Synthesis (HLS) functions similarly to a compiler, translating code written typically in C, C++, or SystemC into Hardware Description Languages (HDLs). It employs C-based languages to describe the behavioral aspects of a circuit, whereas Register-Transfer Level (RTL) focuses on the structural implementation, such as registers, operations, and interconnections. This approach simplifies the design process by abstracting low-level complexities (LAHTI; HäMäLäINEN, 2025), thereby enabling significant productivity gains in hardware development.

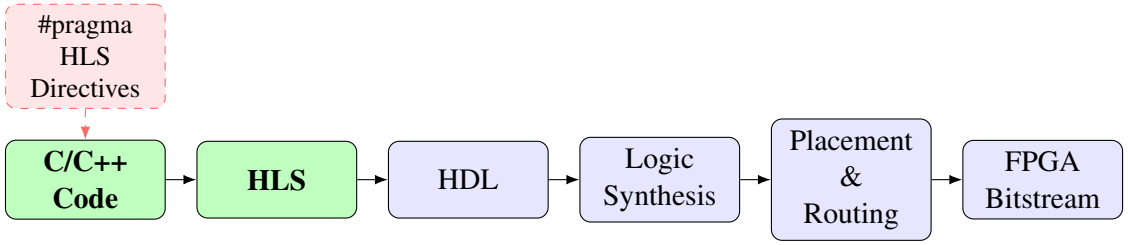
The internal HLS synthesis process traditionally follows a sequence of foundational steps established in early synthesis literature and still central to modern tools. These steps (allocation, scheduling, and binding) act as an automated "black box" within the tool, transforming the behavioral description into a timed structural implementation. As described in classic synthesis theory and consistently applied in contemporary frameworks (GAJSKI et al., 1992; SCHAFER; WANG, 2020), these stages consist of:

- **Allocation:** Determines the necessary hardware resources, such as functional units, registers, and memory blocks;

- **Scheduling:** Assigns operations to specific clock cycles and defines the design timing, resulting in a timed model;
- **Binding:** Maps logical operations and variables to specific physical hardware resources; for example, an addition operation is mapped to a specific adder.

2.2 Optimization Directives (Pragmas)

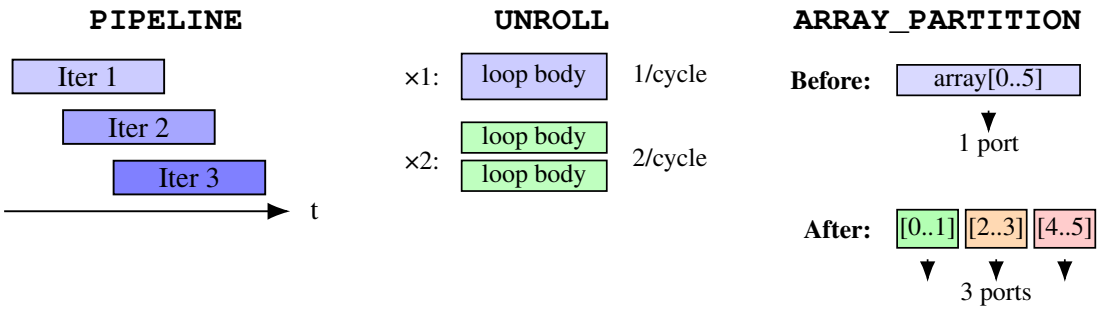
Figure 2.2: Optimization directives guiding the HLS translation process.



Source: The author

Code directives serve as a guide for the synthesis tool, providing instructions on how the high-level code should be optimized during its transformation into hardware (Advanced Micro Devices, Inc., 2023). One of the most frequently used directives is `PIPELINE`, which instructs the tool to reduce the Initiation Interval (II) of a function or loop, enabling new iterations to begin before the previous ones have finished. Another fundamental directive is `UNROLL`, which guides the tool to replicate the loop hardware. This often allows for parallel execution, improving latency at the cost of increased resource consumption and area overhead. Finally, the `ARRAY_PARTITION` directive partitions an array into smaller segments, allowing the use of multiple registers or memory banks instead of a single large block. This enables simultaneous data access and avoids memory bottlenecks.

Figure 2.3: Illustration of the PIPELINE, UNROLL, and ARRAY_PARTITION directives.



Source: The author

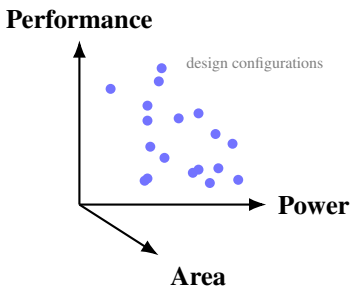
2.3 Design Space Exploration (DSE)

Design Space Exploration (DSE) is an engineering process that aims to identify the optimal design solution by navigating the set of possible configurations within the "design space". Rooted in classic optimization theory, this process frequently represents a fundamental multi-objective challenge involving Power, Performance, and Area (PPA). As established in foundational EDA literature (GAJSKI et al., 1992; MICHELI, 1994) and still central to contemporary HLS research (MOLINA et al., 2022), enhancing performance typically requires trade-offs that impact power consumption or silicon area, and vice-versa.

In this context, the total number of clock cycles acts as the primary indicator of performance in this work. Since all benchmarks were synthesized under a fixed 10 ns clock period constraint, minimizing cycle count is directly equivalent to minimizing execution latency, making it a representative measure of kernel throughput. This metric was selected as the primary optimization objective communicated to the LLM during the prompt engineering stage.

The challenge lies in the fact that the definition of an "optimal solution" depends strictly on the specific requirements of the application. Moreover, given the vast number of possible combinations and the significant computational cost of synthesis (SCHAFER; WANG, 2020), an exhaustive exploration of the design space is computationally infeasible.

Figure 2.4: Design space configurations across the Power, Performance, and Area (PPA) trade-off.

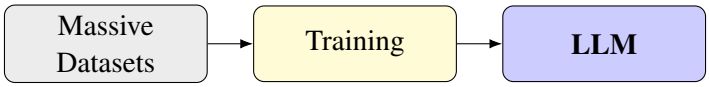


Source: The author

2.4 Large Language Models (LLMs) and Prompt Engineering

Large Language Models (LLMs) are advanced Artificial Intelligence systems trained on massive datasets to understand and generate both human language and programming code (ROMERA-PAREDES et al., 2024). These models represent a significant breakthrough in the field of Deep Learning and are based on the neural network architecture known as the Transformer.

Figure 2.5: Simplified overview of LLM training from massive datasets.

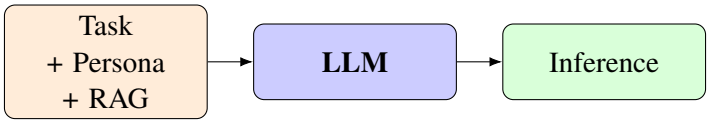


Source: The author

Through In-Context Learning (ICL) (BROWN et al., 2020), an LLM can learn to perform new tasks using only the information and instructions provided within the prompt, without requiring updates to its internal parameters, such as fine-tuning.

Furthermore, Prompt Engineering can be utilized to instruct an LLM to adopt a specific persona (e.g., a hardware specialist). By doing so, the model leverages its latent domain-specific knowledge, thereby enhancing the precision and technical quality of its outputs.

Figure 2.6: Prompt engineering strategy combining task definition, persona, and RAG.



Source: The author

In this work, the Google Gemini API was selected as the primary tool for this purpose, based on both technical and operational requirements. Primarily, the model's native API support was a prerequisite for the development of the proposed automated pipeline, enabling the programmatic interaction necessary for large-scale code analysis and synthesis.

Additionally, Gemini provides a robust free-access tier that ensures the economic feasibility of extensive experimentation. This allows for multiple inference iterations across the benchmark suite without the financial constraints often associated with high-performance commercial models (TEAM, 2025a; TEAM, 2024), effectively lowering the barrier to entry for automated hardware design exploration.

2.5 Related Work

Traditionally, Design Space Exploration (DSE) for HLS directive selection has relied on heuristics and iterative search strategies. Consequently, this process is often extremely time-consuming and computationally expensive, as each HLS synthesis cycle requires significant time and energy. Moreover, the vast number of possible directive combinations creates an exponential search space that grows alongside code complexity.

Tools such as AutoDSE (SOHRABIZADEH et al., 2022), frequently employed as baselines, rely on full code synthesis to evaluate each candidate configuration, making them susceptible to these time constraints. As a result, such tools often fail to explore the design space comprehensively (MASHNOOR et al., 2025), frequently converging on local optima rather than identifying a global optimum.

To avoid running the complete synthesis flow for every configuration iteration, researchers have introduced machine learning (ML) models for early-stage quality of results (QoR) prediction. In (KUANG et al., 2023), this predictive approach was combined with a multi-objective search strategy to propose an automatic framework called HGBO-DSE. This infrastructure integrates a Hierarchical Graph Neural Network Predictor (HGP) to infer expected power, performance, and area (PPA) outcomes, a Tree-structured Design Space Modeler (TDM) to prune invalid directive configurations, and a Bayesian Optimization-based Multi-objective Exploration engine (BOME) to navigate the design space.

Driven by this architecture, the framework achieved significant performance gains. The HGP module outperformed state-of-the-art graph architectures such as IronMan-Pro (WU; XIE; HAO, 2023), PowerGear (LIN et al., 2022) and PNA-R(WU et al., 2022) in predicting resource and power metrics (except on the critical path metric) accelerating the overall DSE process by up to 24x, with an average speedup of 14x compared to full FPGA implementation flows. Furthermore, in terms of the Average Distance from Reference Set (ADRS), the BOME engine outperformed traditional simulated annealing (SA) (MAHAPATRA; SCHAFER, 2014) and NSGA-II (SCHAFER, 2017) algorithms by 94.65% and 61.81% respectively, while improving the product of power, latency, delay and area (PLDA) of the Pareto solution by 72.00% and 30.47% over SA and NSGA-II respectively.

In (YAO et al., 2025), an LLM was employed for both project information extraction and as an autonomous agent responsible for directive selection. Both components

were refined through fine-tuning to enhance their performance. By utilizing the Llama-13B model alongside the Vivado HLS tool, this approach achieved a 15% improvement in the ADRS metric compared to the second-best state-of-the-art method analyzed, a multi-fidelity optimization algorithm proposed by (SUN et al., 2022).

However, both these strategies (KUANG et al., 2023; YAO et al., 2025) demand the creation of specialized training datasets and significant computational resources for model training, making them less accessible compared to off-the-shelf API solutions.

To mitigate these costs, the studies that introduced the RALAD (XU; HU; HUANG, 2024) and TimelyHLS (MASHNOOR et al., 2025) frameworks employed Retrieval-Augmented Generation (RAG) (LEWIS et al., 2021). RAG is a technique that optimizes LLM output by referencing a specific, reliable knowledge base outside of its initial training data before generating a response. This approach allows the model to access up-to-date or domain-specific technical information without the need for costly retraining.

In the case of the RALAD framework, the contextual knowledge used to enrich the prompts was sourced from the book *'Parallel Programming for FPGAs'* (KASTNER; MATAI; NEUENDORFFER, 2018). The TimelyHLS framework, in addition to RAG, implemented a cyclic feedback loop by incorporating synthesis logs and reports provided by the HLS tool to refine the model's suggestions. Using the CodeLlama-13B model with Vivado HLS, RALAD reported speedups of up to 9.7x over the unoptimized source code, whereas TimelyHLS, using GPT-4 and Claude-3.5-Sonnet within the Vitis HLS environment, achieved speedups of up to 4x over the unoptimized HLS baseline. While both frameworks showed significant performance improvements, these engineering approaches require integrating external data sources and iterative execution loops alongside the core model invocation.

Another approach involves the use of multi-agent systems, as implemented in the LLM-DSE framework (WANG et al., 2025). In this study, two agents, referred to as "Specialists", each with a distinct focus, are responsible for proposing diverse sets of directives. These proposals are submitted to an "Arbitrator" agent, which filters the most promising candidates for synthesis. The resulting synthesis reports and logs are then delivered to the "Critic" agent, which processes this feedback and determines which information should be propagated. Finally, the "Router" agent receives this data and decides which specialist is best suited to further improve the directives for a given project.

Although this framework outperformed AutoDSE, achieving an average speedup of 2.55x in terms of execution clock cycles using the Merlin Compiler, it faces the chal-

lenge of “Context Explosion”. This phenomenon occurs when the Router agent becomes overloaded with tasks, potentially compromising the precision of its decisions. Consequently, the high token consumption required to maintain this multi-agent coordination over multiple exploration iterations significantly exceeds that of single-model direct generation methods. This architectural overhead results in a solution characterized by increased operational costs and high structural complexity.

Considering the approaches of these previous studies, it is evident that existing frameworks often complement the language model with external mechanisms, such as fine-tuning on specialized datasets, deploying multi-agent coordination, or integrating RAG infrastructures. This research, however, explores a different trade-off: evaluating the effectiveness of state-of-the-art commercial Large Language Models applied directly to HLS directive selection without external structural dependencies.

The goal is to analyze the vast latent knowledge and native reasoning capabilities already present in these massive models. By doing so, we contrast the specialized engineering frameworks with a generalist approach that benefits from the continuous evolution and increasing accessibility of powerful off-the-shelf LLMs, evaluating if their out-of-the-box performance can match or exceed specialized solutions.

3 METHODOLOGY

The diagram in Figure 3.1 illustrates the automated directive injection pipeline proposed in this work, with numbered markers indicating each stage described below. The pipeline consists of the following stages:

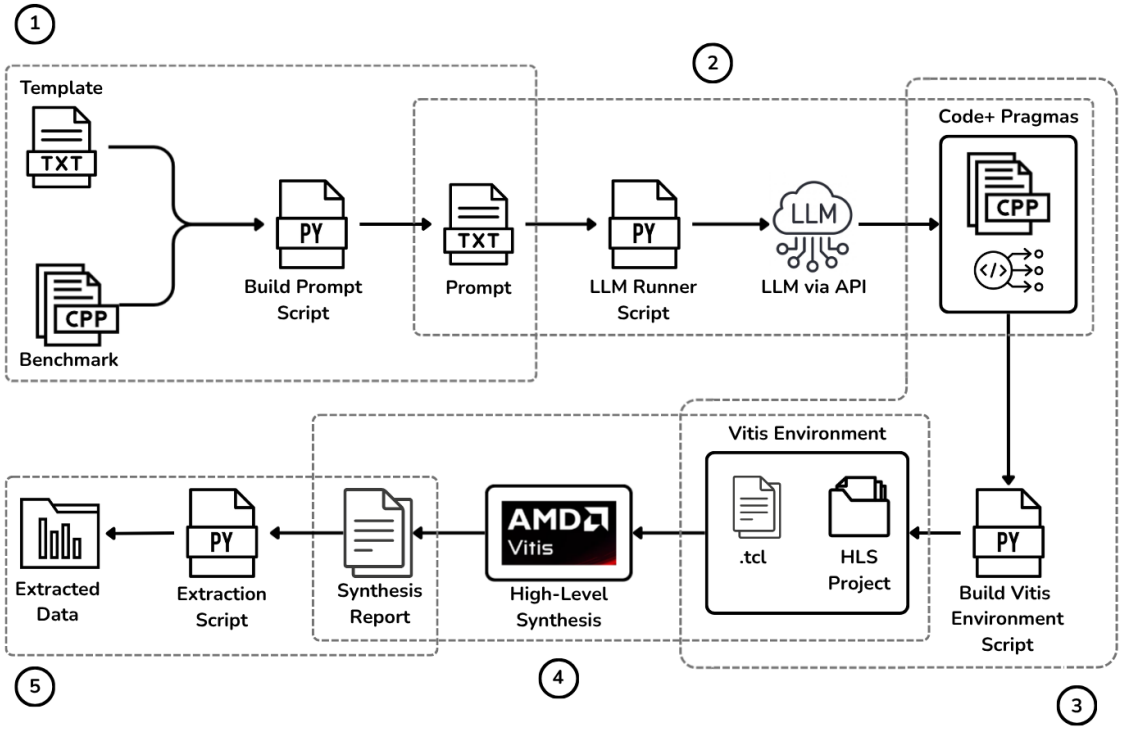
1. **Prompt Construction:** A Python script combines the instruction template with the benchmark source files to produce a structured prompt in `.txt` format. The prompt design is detailed in Section 3.4.
2. **LLM Inference:** The prompt is submitted to the selected language model via the Google Gemini API, which returns the source code containing the suggested optimization directives. The models and multi-run execution strategy are described in Section 3.5.
3. **Vitis Environment Setup:** A Python script organizes the model output into an HLS project structure alongside the corresponding `.tcl` configuration script. The directory structure is detailed in Section 3.6.
4. **High-Level Synthesis:** AMD Vitis HLS executes the `.tcl` script to synthesize the optimized design and produce a synthesis report. The synthesis flow and its configuration are described in Section 3.7.
5. **Data Extraction:** A Python script parses the synthesis reports and extracts the latency and timing results for analysis, which are presented in Chapter 4.

3.1 Preliminary Exploratory Tests

During the initial phases of this research, exploratory tests were conducted using the ChatGPT web interface to assess the baseline capability of Large Language Models in understanding High-Level Synthesis semantics. Elementary algorithms, such as Matrix Multiplication, Bubble Sort, and Vector Max Value search, were provided to the model with a simple instruction to optimize the code for the Vitis HLS 2023.2 tool.

The model correctly identified the main optimization opportunities for the kernel, demonstrating that generalist LLMs possess sufficient HLS domain knowledge to serve as a starting point for automated directive injection. These observations directly motivated the development of the fully automated API pipeline detailed in the subsequent sections.

Figure 3.1: Automated directive injection pipeline overview.



Source: The author

3.2 Benchmark Suite (MLDHLS dataset)

Building upon the promising results observed in the exploratory phase, a rigorous evaluation of the automated pipeline required a standardized and more complex set of algorithms. Therefore, the MLDHLS (Multiple Loop Directive HLS) dataset (ANDRADE, 2025) was utilized in this study, as it provides comprehensive Design Space Exploration (DSE) results for a diverse set of kernels, enabling a direct performance comparison between the baseline implementations, the LLM-generated configurations, and the dataset's own established reference results.

The ten benchmarks selected for evaluation are exactly those provided by the MLDHLS dataset. This deliberate choice ensures that the best-known solutions documented in the dataset serve as a valid and direct reference bound for the LLM-generated configurations. Table 3.1 summarizes each algorithm and its computational domain.

Together, these benchmarks span a representative range of computational profiles: from encryption and signal processing kernels to linear algebra and memory-intensive stencil operations. This diversity is essential to assess whether the LLM-driven directive injection pipeline generalizes across structurally distinct algorithmic patterns, rather than excelling only in a narrow class of workloads.

Figure 3.2: Exploratory test: Bubble Sort directive injection via ChatGPT.

Model Input

Choose the best directives to achieve the best possible performance (the lowest number of cycles possible) in the Vitis 2023.2 tool for the following code:

```
#define N 30
// Function to perform Bubble Sort
void bubbleSort(int arr[N]) {
    for (int i = 0; i < N - 1; i++) {
        for (int j = 0; j < N - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Model Output

To achieve the best possible performance (i.e., the lowest number of cycles) in Vitis 2023.2, you need to apply loop unrolling, loop pipelining, and memory optimization directives. Below are the key pragmas you should use:

```
void bubbleSort(int arr[N]) {
#pragma HLS array_partition variable=arr complete
    for (int i = 0; i < N - 1; i++) {
#pragma HLS pipeline II=1
#pragma HLS unroll factor=2
        for (int j = 0; j < N - i - 1; j++) {
...
```

Source: The author

3.3 Reference Points for Comparison

Before evaluating the directive selection capabilities of the Large Language Models, it is essential to establish stable points of comparison to quantify performance. This section presents the reference latency values, measured in clock cycles, structured around three distinct reference points:

- **Unoptimized Baseline:** The original benchmark kernels synthesized without any optimization directives, establishing the raw sequential circuit latency as a lower bound for comparison.
- **MLDHLS Best Known Solutions:** The optimal latency results provided by the

Table 3.1: MLDHLS benchmark suite: algorithms and descriptions.

Benchmark	**Description**
ADPCM	Adaptive Differential Pulse-Code Modulation; an audio compression algorithm that encodes the difference between consecutive samples.
AES	Advanced Encryption Standard; a symmetric block cipher widely used for secure data encryption.
BACKPROP	Backpropagation; the core training algorithm for neural networks, computing gradients through successive layers.
GEMM	General Matrix Multiplication; a foundational linear algebra operation central to many scientific and machine learning workloads.
GSM	Global System for Mobile Communications encoder; implements the speech coding algorithm used in digital mobile telephony.
GRAMSCHMIDT	Gram-Schmidt orthogonalization; a numerical method for producing an orthonormal basis from a set of vectors.
KNN	K-Nearest Neighbors; a classification algorithm that determines class membership based on distance to neighboring data points.
SHA	Secure Hash Algorithm; a cryptographic hash function used to produce fixed-size digests from arbitrary input.
STENCIL3D	3D stencil computation; a memory-intensive iterative kernel that updates each point in a 3D grid based on its neighbors.
TRANS_FFT	Transposed Fast Fourier Transform; a communication-bound variant of the FFT used in distributed spectral analysis.

Source: The author

MLDHLS dataset, representing the minimum values discovered through a Random Search exploring an average of 348.8 configurations per benchmark.

- **HGBO State-of-the-Art:** The latency results achieved by the Hierarchical Graph Bayesian Optimization framework, representing the current machine learning-based state of the art for this benchmark suite.

Together, these three reference points form a robust comparative basis, allowing the contribution of the automated directive injection pipeline to be isolated and quantified against both the unoptimized case and established optimization frameworks.

Table 3.2: Input size per benchmark (number of source files and lines of code).

Benchmark	# Files	# Lines
adpcm	2	640
aes	8	1,032
backprop	2	409
gemm	2	62
gramschmidt	2	227
gsm	7	550
knn	2	87
sha	2	261
stencil3d	2	76
transposed_fft	2	437

Source: The author

3.3.1 Unoptimized Baseline

To evaluate the effectiveness of the directives proposed by the Large Language Models, a reference baseline was established by synthesizing the original benchmarks without any optimization pragmas. This evaluation was performed under the controlled environment detailed in Section 3.7.1, ensuring that the Vitis HLS automatic pipelining and array partitioning heuristics were completely disabled.

Table 3.3 presents the synthesis results for the unoptimized execution branch across all ten evaluated algorithms, with performance quantified by total kernel latency in clock cycles.

Table 3.3: Unoptimized baseline synthesis latency results.

Benchmark	Clock Cycles
adpcm	1,717
aes	13,960
backprop	258,334,952
gemm	741,381
gsm	51,266
gramschmidt	20,461,812,225
knn	13,599
sha	25,421
stencil3d	9,475
transposed_fft	4,905

Source: The author

As observed in Table 3.3, kernels with complex multi-nested loops or memory-

intensive computational structures exhibit severe latency bottlenecks when left unoptimized. The `gramschmidt` and `backprop` kernels are the most notable examples, reaching cycle counts in the tens of billions and hundreds of millions respectively, a direct consequence of sequential execution chains and unmitigated loop overheads.

Under the restricted synthesis constraints established to disable the tool's native heuristics, the compiler is forced to map each design sequentially. These results therefore reflect the raw algorithmic complexity of each kernel and reinforce the necessity of explicit directive injection to guide hardware parallelization.

3.3.2 MLDHLS Best Known Solutions

To establish a reference for optimization quality, the best-known results documented in the MLDHLS dataset were extracted. These values represent the absolute minimum latency configurations discovered through a Random Search exploring an average of 348.8 configurations per benchmark, and serve as a reference point for what random search can achieve within this benchmark suite.

Table 3.4 compiles these optimal configurations for each of the ten benchmark algorithms, detailing the specific solution identifier alongside its synthesis latency in clock cycles.

Table 3.4: MLDHLS dataset best-known latency results.

Benchmark	**Best Solution**	**Clock Cycles**	**Speedup vs. Baseline**
adpcm	solution144	1,240	1.38×
aes	solution240	847	16.48×
backprop	solution283	5,191,225	49.77×
gemm	solution132	16,721	44.33×
gsm	solution169	809	63.37×
gramschmidt	solution252	544,821,946	37.55×
knn	solution307	2,149	6.33×
sha	solution78	25,879	0.98×
stencil3d	solution440	295	32.12×
transposed_fft	solution249	2,149	2.28×

Source: (ANDRADE, 2025)

The latency reductions relative to the unoptimized baseline are substantial across

nearly all benchmarks. For instance, `stencil3d` drops from 9,475 to just 295 cycles, and `gsm` from 51,266 to 809, reflecting the significant impact that well-chosen directives can have on memory-bound and loop-intensive kernels. Notably, `sha` presents an exception: its best-known solution (25,879 cycles) is marginally higher than the unoptimized result (25,421 cycles), suggesting that the directive combinations explored by the Random Search did not yield a net benefit for this particular kernel.

3.3.3 HGBO State-of-the-Art Baseline

To contextualize the LLM directive injection results against a dedicated machine learning framework, the best latency results produced by the Hierarchical Graph Bayesian Optimization (HGBO) infrastructure were extracted. As described in Section 2.5, HGBO combines graph-based performance prediction with Bayesian optimization to navigate the design space efficiently, representing the current state of the art for directive injection. The framework was evaluated under its standard configuration, targeting the multi-objective PPA Pareto frontier across 100 iterations.

Table 3.5 compiles the quantitative latency outcomes achieved by this framework across all ten benchmark algorithms.

Table 3.5: HGBO best latency results under PPA configuration (100 iterations).

Benchmark	**Project**	**Clock Cycles**	**Speedup vs. Baseline**	**Speedup vs. MLDHLS**
adpcm	prj_98	1,133	1.52×	0.91×
aes	prj_45	513	27.21×	1.65×
backprop	prj_40	7,148,529	36.14×	0.73×
gemm	prj_79	20,609	35.97×	0.81×
gsm	prj_61	1,152	44.50×	0.70×
gramschmidt	prj_64	543,706,113	37.63×	1.00×
knn	prj_8	2,285	5.95×	0.94×
sha	prj_9	28,958	0.88×	0.89×
stencil3d	prj_39	1,734	5.46×	0.17×
transposed_fft	prj_9	1,349	3.64×	0.63×

Source: The author

The results demonstrate that HGBO, even when optimizing for the broader PPA objective rather than latency exclusively, consistently yields substantial reductions over the unoptimized baseline across all ten benchmarks. For compute-intensive kernels such

as `aes` and `gsm`, the framework achieves reductions of approximately 96% relative to the unoptimized baseline, reaching 513 and 1,152 cycles respectively. Similarly, `knn` is reduced from 13,599 to 2,285 cycles, closely approaching the MLDHLS best-known solution of 2,149 cycles.

However, two benchmarks warrant closer attention. For `stencil3d`, HGBO reaches 1,734 cycles, which, while a significant improvement over the unoptimized baseline of 9,475 cycles, remains considerably above the MLDHLS best-known result of 295 cycles. This gap suggests that the PPA multi-objective criterion led the framework to trade latency for gains in power or area in this particular kernel. For `sha`, HGBO produces 28,958 cycles, which is marginally higher than both the unoptimized baseline (25,421 cycles) and the MLDHLS best-known result (25,879 cycles), reinforcing the observation from the previous subsection that this kernel is particularly resistant to directive-based latency optimization.

These reference values establish a strong competitive baseline, as HGBO benefits from 100 synthesis-informed iterations to converge on its solutions. The LLM-driven pipeline, by contrast, produces its directive configurations in a single inference step without any iterative feedback, making the comparison an informative measure of how much optimization capability can be extracted from a model's latent knowledge alone.

3.4 Prompt Engineering

The prompt structure was composed of two main parts: the template (instructions) and the source code. These components were concatenated into a text file (`.txt`) and submitted to the model via API, resulting in an individual prompt for each benchmark. The code section contained the C/C++ kernel files to be analyzed, all concatenated using XML-style tags in the format `<file filename>` and `</file filename>` to mark the beginning and end of each file. This structure allowed the model to identify the boundaries between header and source files within a single prompt. Meanwhile, the template section was organized as follows:

1. **Role Attribution (Persona):** The LLM was instructed to act as an expert HLS engineer. This strategy was adopted because assigning a specific role significantly enhances the quality and precision of the responses in coding tasks (SILVER et al., 2025).

2. **Task Definition:** The prompt explicitly established that the objective was to optimize the C/C++ code for the Vitis HLS platform, with the primary goal of minimizing the total number of clock cycles.
3. **Output Constraint:** The model was requested to return strictly the source code containing the recommended directives, without additional explanatory text. The instruction explicitly required the model to maintain the original tag-based format (`<file filename>...</file filename>`) in its response. This was essential to enable the post-processing script to automatically parse the output and restore the files to their original formats (`.c, .cpp, .h`) for synthesis.
4. **Directives Insertion Instructions (Scope):** Explicit rules regarding pragma positioning were included. During previous tests, it was observed that the model occasionally inserted directives incorrectly outside the function scope. Therefore, the prompt reinforced the Vitis HLS syntax rule requiring all pragmas to be contained within the function body.

An example of the final prompt layout generated by the automation script using a real benchmark instance is illustrated in Figure 3.3. This structure demonstrates how the optimization rules and the original source code are combined into a single prompt before being sent to the LLM. Note that the `LOOP_TRIPCOUNT` pragma already present in the source code is part of the original benchmark and serves only as metadata to inform the HLS tool of the expected iteration count; it does not constitute an optimization directive and was not inserted by the pipeline.

3.5 Automated Directive Injection

For the Automated Directive Injection stage, models from the Google Gemini API family, developed by Google DeepMind, were utilized, specifically the Gemini-2.5-Flash, Gemini-3-Flash, Gemma-3-27b-it, and Gemma-4-31b-it versions. The selection of these models was primarily based on their integration capabilities and execution viability.

Figure 3.3: Prompt structure for the `bbgemm` benchmark (abbreviated).

Model Input

You are a specialist in optimizing hardware designs generated with high-level synthesis. Choose the best directives to achieve the lowest number of cycles possible in the Vitis 2023.2 tool for the following code. Provide only the code with additional directives, without any explanation or additional text. Mark the beginning and end of each file and specify its name using XML-style tags, exactly like this: '<file filename>' and '</file filename>'.
'#pragma HLS' is only allowed in function scope
top function: bbgemm

```
<file gemm.h>
/* Implementation based on algorithm described in: Lam et al.
ASPLOS 1991 */

#define TYPE double
#define row_size 64
#define block_size 8
#define N row_size*col_size
...
void bbgemm(TYPE m1[N], TYPE m2[N], TYPE prod[N]);
</file gemm.h>

<file gemm.c>
/* Implementation based on algorithm described in: Lam et al.
ASPLOS 1991 */

#include "gemm.h"

void bbgemm(TYPE m1[N], TYPE m2[N], TYPE prod[N]){
    int i, k, j, jj, kk;
    ...
    loopjj:for (jj = 0; jj < row_size; jj += block_size){
        #pragma HLS LOOP_TRIPCOUNT min=8 max=8 avg=8
        ...
        loopj:for (j = 0; j < block_size; ++j){
            mul = temp_x * m2[k_row + j + jj];
            prod[i_row + j + jj] += mul;
        }
    }
}
</file gemm.c>
```

Source: The author

Table 3.6: Overview of evaluated LLM architectures and capabilities.

Model	**Parameters**	**Architecture**	**Key Capabilities**
Gemini-2.5-Flash	N/A (proprietary)	Sparse MoE	Hybrid reasoning with configurable thinking budget; strong coding and multimodal performance
Gemini-3-Flash	N/A (proprietary)	Based on Gemini 3 Pro	Native multimodal reasoning (text, image, audio, video); 1M token context window
Gemma-3-27b-it	27B (dense)	Decoder-only Transformer	Instruction-tuned open model; multimodal via SigLIP encoder; 128K context window
Gemma-4-31b-it	31B (dense)	Decoder-only Transformer	Instruction-tuned open model; enhanced coding and agentic capabilities; 256K context window

Source: (Google DeepMind, 2025a; Google DeepMind, 2025b; Google DeepMind, 2025c; TEAM, 2025b)

Initially, exploratory tests were conducted through a chat interface (ChatGPT) during the preliminary design stage, previously illustrated in Figure 3.2. However, the operational workflow was migrated to the Google ecosystem due to the efficiency of the Gemini API for automated scripting. The Python client library (`google-generativeai`) offered low integration complexity, enabling the construction of the Automated Directive Injection pipeline proposed in this work.

Additionally, the choice also considered economic viability and research reproducibility. Access to the API through free developer tiers (via Google AI Studio) allows the experiments to be conducted and replicated without financial barriers. This strategy avoids the prohibitive costs associated with alternative high-performance commercial APIs (TEAM, 2025a; TEAM, 2024), ensuring a highly accessible framework for academic research.

To address the inherent non-determinism and stochastic variability of Large Language Models (SONG et al., 2025), the automated pipeline executed a minimum of six independent inference runs per benchmark for each evaluated model. This multiple-iteration strategy was necessary to mitigate sampling randomness, ensure research reproducibility, and collect reliable data points for evaluation. Consequently, it allows for a robust statistical analysis of the generated hardware configurations, separating accidental synthesis anomalies from stable optimization patterns learned by the models.

3.6 Codes Extraction

After receiving the responses from the models, the code extraction was performed. The extracted source files were organized into a directory structure, as shown in Listing 3.1, where the HLS tool was subsequently executed.

It is worth noting that during the evaluation of Gemini-3-Flash, it was observed that the model did not consistently follow the exact tag format specified in the prompt (`<file filename>...</file filename>`), occasionally using alternative conventions such as `<filename>` or omitting delimiters entirely. Rather than discarding these responses, the extraction script was extended to recognize and handle common tag variations, allowing the pipeline to recover the source files whenever the content was structurally intact. This decision was deliberate: the primary objective of this work is to evaluate the model's capability to insert correct and effective HLS directives, not to assess its ability to follow a specific output formatting convention.

The directory hierarchy was structured to maintain a clear distinction between the baseline and the LLM-generated designs. The root directory, `vitis`, was organized as follows:

- **Baseline Branch:** Contained within the `MLDHLS/baseline` path, storing the original projects for each benchmark. Each sub-folder included the source files (`src/`) and a dedicated `run_hls.tcl` script for synthesis.
- **Model Evaluation Branch:** Organized by model name (e.g., `gemini-2.5-flash`) and further divided into execution runs (`run_n`). This allowed for the independent storage and synthesis of code modified with directives suggested across different iterations.
- **Automation Scripts:** Execution was managed by Shell scripts (`.sh`) located in `vitis_run` folders, which orchestrated the batch synthesis of all benchmarks within their respective branches.

To verify that the models restricted their modifications to pragma insertion without altering the original algorithmic logic, a systematic diff-based validation step was integrated into the pipeline. For each run and each benchmark, the generated source files were compared line-by-line against the original benchmark files using a Python script. Each file was classified as `SUCCESS` if only `#pragma HLS` directives were added, `WARNING` if comments were also introduced, or `CRITICAL_ERROR` if any original lines were re-

Listing 3.1 – Directory hierarchy for automated execution.

```
vitis/
|-- MLDHLS/
    |-- baseline/
    |   |-- <benchmark_name>/
    |   |   |-- hls_project/
    |   |   |-- src/
    |   |   |-- run_hls.tcl
    |   |-- vitis_run/
    |       |-- init_MLDHLS_base.sh
    |-- <model_name>/
        |-- run_n/
        |   |-- <benchmark_name>/
        |       |-- hls_project/
        |       |-- src/
        |       |-- run_hls.tcl
        |-- vitis_run/
            |-- init_<model_name>_<run>_ai.sh
```

moved or modified, or if non-pragma additions were detected. This validation confirmed that most files were modified exclusively through pragma insertion, while cases classified as `CRITICAL_ERROR` correspond directly to the source code corruption and signature modification failures described in Section 4.6.

3.7 Synthesis

For the synthesis stage, the AMD Vitis HLS (version 2023.2) was chosen. This tool performs high-level synthesis by transforming C/C++ specifications into RTL (Verilog/VHDL) through scheduling and binding processes (Advanced Micro Devices, Inc., 2023). The selection was driven by the fact that Vitis HLS is the state-of-the-art environment for Xilinx/AMD devices, offering robust support for standard optimization pragmas. Furthermore, its command-line interface (`vitis_hls`) enables full automation via scripts, which was an essential attribute for the pipeline developed in this research.

It is important to note that all synthesis evaluations in this work were performed exclusively using the C Synthesis stage (`csynth_design`) of Vitis HLS, which transforms the C/C++ source into a Register-Transfer Level (RTL) description and produces the latency estimates. The subsequent implementation stages (e.g., placement and routing) were excluded from the scope of this evaluation because the primary metric of interest, total clock cycles, is reported directly by the C Synthesis stage without requiring a full

implementation run. Consequently, all latency values and timing slack figures reported here are derived from C Synthesis reports and reflect pre-implementation estimates rather than post-routing measurements.

After the directory structure was organized (as detailed in Section 3.6 and visualized in Listing 3.1), the tool was executed in batch mode using Tcl scripts to synthesize each benchmark project. The detailed configuration of these execution scripts is discussed in Section 3.7.1. At the end of this process, the results were extracted from the synthesis reports, focusing primarily on latency. In this context, latency is defined as the total execution time, calculated as the product of the number of clock cycles and the target clock period (10 ns). This metric was selected as the primary target objective for the LLM during the prompt engineering stage.

In this context, the total number of clock cycles was selected as the primary metric extracted from synthesis reports and communicated to the LLM as the optimization objective during the prompt engineering stage. Since all benchmarks were synthesized under a fixed clock period of 10 ns, the absolute execution time is directly proportional to cycle count and can be obtained by multiplying the two values; however, as the clock period is constant across all experiments, cycle count alone is sufficient for comparison purposes.

The synthesis results, which are comprehensively detailed and discussed in Chapter 4, enabled an evaluation of the LLMs' capacity to identify patterns within the C/C++ code and suggest optimization directives. While the suggested pragmas demonstrated potential for enhancing hardware performance compared to the baseline, the effectiveness of these optimizations varied across different benchmarks. This analysis provides insights into the model's ability to reduce the manual complexity of HLS optimization while also highlighting the current limitations and challenges in achieving consistent performance gains through LLM-driven design space exploration.

3.7.1 Synthesis Configuration and Tcl Scripts

To ensure a fair and accurate evaluation of the directives proposed by the LLMs, it was crucial to isolate their impact from the default optimizations automatically applied by the Vitis HLS tool. By default, Vitis HLS employs internal heuristics that may automatically pipeline certain loops or optimize memory accesses, which could mask the actual effect of the AI-generated pragmas.

To prevent this interference, specific configuration commands were included within the `run_hls.tcl` script used to guide the synthesis process, as illustrated in Figure 3.4. Specifically, flags such as `config_compile -pipeline_loops 999` (which sets the automatic pipelining threshold to a sufficiently high loop bound to avoid unintentional optimizations) and `config_array_partition -throughput_driven off` were added to explicitly disable the tool's default behavioral optimizations. Consequently, this setup guarantees that any performance improvement or architectural change observed in the synthesized hardware is exclusively the result of the explicit directives injected by the language models. This approach establishes a controlled and deterministic baseline for the experimental comparison.

Figure 3.4: `run_hls.tcl` script for disabling automatic optimizations.

run_hls.tcl

```
# Set important values
set project_name hls_project
set solution_name solution_main
set top_function bbgemm
set src_file_1 src/gemm.c
set src_file_2 src/gemm.h
set part_name xcu50-fsvh2104-2-e

# Create HLS project
open_project $project_name
set_top $top_function

# Add source files
add_files $src_file_1
add_files $src_file_2

# Create solution
open_solution "$solution_name" -flow_target vitis

# Set the target FPGA
set_part $part_name

# Create clock (10ns -> 100MHz)
create_clock -period 10 -name default

# Add compilation configuration directives
config_compile -pipeline_loops 999
config_array_partition -throughput_driven off

# Run synthesis
csynth_design

exit
```

Source: The author

4 EXPERIMENTAL RESULTS AND DISCUSSION

This chapter presents the evaluation of the proposed automated directive injection pipeline across four models from the Google Gemini and Gemma families. The primary metric for comparison is the total number of clock cycles reported after C Synthesis in Vitis HLS. To account for the stochastic variability of the language models, the results presented for the LLM-driven approaches represent the mean across the multiple inference runs executed for each benchmark.

The following notation conventions are used consistently throughout all tables in this chapter:

- A dash (–) indicates that the synthesis process failed entirely, either due to pragma scope violations, syntax errors, incomplete code delivery, or other compiler-level failures.
- (†) marks results where the design synthesized successfully in terms of cycle count, but the slack reported by Vitis HLS was negative, indicating that the critical path exceeded the 10 ns clock period constraint.
- (§) marks valid syntheses where the resulting clock cycle count exceeds the unoptimized baseline, meaning the directives actively degraded performance.

It is worth noting that the negative slack values reported in this chapter reflect the critical path estimated at the C Synthesis stage, and were treated as invalid for the purposes of this analysis since timing closure could not be confirmed without proceeding to placement and routing. This is a conservative criterion: the actual implementation flow could, in some cases, resolve marginal violations through physical-level optimizations such as logic restructuring and floorplanning. Exploring whether these specific cases would close timing after full implementation is left as a direction for future work.

4.1 Gemini-2.5-Flash

Table 4.1 presents the raw synthesis results across all six inference runs for the Gemini-2.5-Flash model. Each entry reports the obtained latency in clock cycles alongside the timing slack value in nanoseconds.

Table 4.2 consolidates these results, separating valid configurations from those that present synthesis failure and negative slack.

Table 4.1: Gemini-2.5-Flash synthesis results per run.

Benchmark	Run 1		Run 2		Run 3		Run 4		Run 5		Run 6	
	Cycles	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*
adpcm	393	−16.24	393	−16.24	393	−16.24	–	–	–	–	393	−16.24
aes	–	–	–	–	–	–	–	–	–	–	–	–
backprop	2,112,224	−0.86	2,112,224	−0.86	2,112,224	−0.86	–	–	2,112,224	−0.86	–	–
gemm	–	–	–	–	–	–	–	–	–	–	–	–
gramschmidt	–	–	–	–	–	–	–	–	–	–	–	–
gsm	–	–	87	+0.08	–	–	–	–	87	+0.08	87	+0.08
knn	572	0.00	572	0.00	–	–	572	0.00	572	0.00	572	0.00
sha	20,859	−78.14	–	–	20,859	−78.14	20,859	−78.14	–	–	20,859	−78.14
stencil3d	3,220	+0.73	891	+0.13	891	+0.13	3,220	+0.73	891	+0.13	891	+0.13
trans_fft	2,879	−0.12	2,879	−0.12	2,879	−0.12	2,879	−0.12	2,879	−0.12	2,879	−0.12

Source: The author

Table 4.2: Gemini-2.5-Flash average clock cycles and timing status per benchmark.

Benchmark	**Baseline**	**Avg. Clock Cycles**	**Avg. Slack (ns)**	**Status**
adpcm	1,717	393	−16.24	Negative slack
aes	13,960	–	–	Synthesis failure
backprop	258,334,952	2,112,224	−0.86	Negative slack
gemm	741,381	–	–	Synthesis failure
gramschmidt	20,461,812,225	–	–	Synthesis failure
gsm	51,266	87	+0.08	**Valid**
knn	13,599	572	0.00	**Valid**
sha	25,421	20,859	−78.14	Negative slack
stencil3d	9,475	1,667	+0.33	**Valid**
transposed_fft	4,905	2,879	−0.12	Negative slack

Source: The author

The Gemini-2.5-Flash model produced synthesizable output for seven out of ten benchmarks across at least one run, with `aes`, `gemm`, and `gramschmidt` failing across all six runs. The nature of these failures, however, differs between benchmarks. For `aes`, the model corrupted the source code structure in every run, introducing unbalanced braces and undeclared identifiers that rendered the file unparseable by the HLS compiler. For `gemm` and `gramschmidt`, the generated code was syntactically intact, but the model inserted conflicting AXI interface bundle directives (`s_axilite` ports assigned to both `control` and `control_r` bundles), which Vitis HLS rejects as an invalid interface configuration.

Among the seven benchmarks that produced at least one synthesizable result, the failure pattern was not uniform across runs. For `gsm`, three out of six runs failed due to a pragma conflict between `INLINE` and `PIPELINE` applied to the same function, a semantically contradictory combination that the model generated inconsistently. For `knn`, a single run failed due to the same AXI bundle conflict observed in `gemm`, while the remaining five runs synthesized correctly. For `backprop` and `sha`, two runs each

failed due to pragmas inserted outside function scope, violating the fundamental Vitis HLS scoping rule that was explicitly stated in the prompt.

Among the seven benchmarks with at least one successful synthesis, only `gsm`, `knn`, and `stencil3d` produced timing-compliant results. The remaining four (`adpcm`, `backprop`, `sha`, and `transposed_fft`) synthesized but violated the 10 ns clock constraint across all valid runs, and are therefore reported as negative slack.

For `gsm`, the achieved latency of 87 cycles represents a reduction of over 99.8% relative to the baseline of 51,266 cycles. Analysis of the generated code reveals that the model applied `INLINE` directives to all auxiliary functions, effectively flattening the entire call hierarchy into a single block, combined with `PIPELINE II=1` at the top-level function and `ARRAY_PARTITION complete` on the internal working arrays. This combination allowed the HLS scheduler to treat the entire kernel as a single pipelined datapath, eliminating function call overhead and enabling multiple data samples to be processed concurrently across different pipeline stages, while the complete array partitioning allowed operations within each stage to execute in parallel rather than competing for shared memory ports.

For `knn`, all five valid runs produced a consistent latency of 572 cycles with zero slack, a reduction of approximately 95.8% relative to the baseline of 13,599 cycles, surpassing both the MLDHLS best-known solution of 2,149 cycles and the HGBO reference of 2,285 cycles. Analysis of the generated code reveals that the model applied `ARRAY_PARTITION complete` to the three position arrays (`position_x`, `position_y`, `position_z`), while partitioning the larger neighbor list (`NL`) into blocks of size `maxNeighbors`, alongside `UNROLL` on the inner neighbor loop and seven dedicated `m_axi` interface ports, one per array. This combination allowed the scheduler to fetch all neighbor positions for a given atom in parallel within a single pipelined iteration of the outer loop, at the cost of replicating the position-lookup logic once for each neighbor and maintaining seven dedicated AXI memory interfaces, one per array.

Given the magnitude of these latency reductions, it became relevant to investigate their underlying hardware cost. To contextualize the cost of the two extreme-acceleration cases, Table 4.3 presents the resource utilization reported by C Synthesis for the `gsm` and `knn` baselines alongside their respective Gemini-2.5-Flash configurations.

Table 4.3: Resource utilization: extreme-acceleration cases (Gemini-2.5-Flash).

Benchmark	Configuration	Cycles	BRAM	DSP	FF	LUT
GSM	Baseline	51,266	30 (1%)	45 (∼0%)	24,994 (1%)	74,628 (8%)
	Gemini-2.5-Flash	87	0 (–)	1,006 (16%)	27,065 (1%)	89,187 (10%)
KNN	Baseline	13,599	30 (1%)	41 (∼0%)	49,385 (2%)	124,920 (14%)
	Gemini-2.5-Flash	572	3,642 (135%)	1,840 (30%)	1,076,737 (61%)	1,106,547 (126%)

Source: The author

For `gsm`, the most significant change is in DSP consumption, which increases from 45 (∼0%) to 1,006 (16%), reflecting the parallel arithmetic units instantiated to sustain the `PIPELINE II=1` across the flattened kernel. LUT usage grows moderately from 74,628 (8%) to 89,187 (10%), while FF utilization remains essentially unchanged at approximately 1%. Notably, BRAM usage drops from 30 (1%) to zero, as the complete array partitioning strategy replaced shared memory blocks with distributed register files, enabling simultaneous access to all array elements at the cost of higher combinational logic. All four resource metrics remain within the physical capacity of the target device, confirming that this directive configuration represents a feasible implementation.

By contrast, the `knn` configuration presents a markedly different resource profile. BRAM utilization increases from 30 (1%) to 3,642 (135%) and LUT from 124,920 (14%) to 1,106,547 (126%), both exceeding the physical capacity of the target device. FF usage also grows substantially, from 49,385 (2%) to 1,076,737 (61%), while DSP increases to 1,840 (30%), remaining within budget. This growth is consistent with the directive strategy described above: the complete partitioning of the three position arrays drives the FF increase, while the combination of block-partitioned neighbor access, full loop unrolling, and seven dedicated AXI master interfaces contributes to the disproportionate LUT growth. This outcome is consistent with the optimization objective communicated to the model: the prompt requested the lowest possible cycle count without specifying any resource constraints or naming the target device, and the model responded by generating a maximally parallel design unconstrained by area budgets. It is worth noting that designs with comparable or larger resource footprints can be accommodated by larger devices.

4.2 Gemini-3-Flash

Table 4.4 presents the synthesis results across all six inference runs for the Gemini-3-Flash model. The same notation conventions defined at the beginning of this chapter apply throughout.

Table 4.4: Gemini-3-Flash synthesis results per run.

Benchmark	Run 1		Run 2		Run 3		Run 4		Run 5		Run 6	
	Cycles	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*
adpcm	–	–	–	–	–	–	–	–	–	–	–	–
aes	–	–	–	–	–	–	–	–	–	–	–	–
backprop	–	–	–	–	–	–	–	–	–	–	–	–
gemm	–	–	–	–	–	–	–	–	–	–	–	–
gramschmidt	–	–	–	–	–	–	–	–	405,215,745	−6.88	405,215,745	−6.88
gsm	–	–	–	–	–	–	–	–	–	–	–	–
knn	1,086	0.00	1,086	0.00	1,086	0.00	1,086	0.00	1,086	0.00	1,086	0.00
sha	–	–	–	–	–	–	–	–	–	–	–	–
stencil3d	283	0.00	283	0.00	283	0.00	283	0.00	1	0.64	318	0.00
trans_fft	–	–	–	–	–	–	–	–	–	–	–	–

Source: The author

Table 4.5 summarizes these outcomes, distinguishing timing-compliant results from those that violated the clock constraint.

Table 4.5: Gemini-3-Flash average clock cycles and timing status per benchmark.

Benchmark	Baseline	Avg. Clock Cycles	Avg. Slack (ns)	Status
adpcm	1,717	–	–	Synthesis failure
aes	13,960	–	–	Synthesis failure
backprop	258,334,952	–	–	Synthesis failure
gemm	741,381	–	–	Synthesis failure
gramschmidt	20,461,812,225	405,215,745	−6.88	Negative slack
gsm	51,266	–	–	Synthesis failure
knn	13,599	1,086	0.00	**Valid**
sha	25,421	–	–	Synthesis failure
stencil3d	9,475	241.83	+0.11	**Valid**
transposed_fft	4,905	–	–	Synthesis failure

Source: The author

The Gemini-3-Flash model produced synthesizable output for only three out of ten benchmarks, with the remaining seven failing across all six runs due to incomplete code delivery, as described in Section 4.6. Among the three benchmarks that generated at least one result, the outcomes varied considerably in both reliability and validity.

For `gramschmidt`, runs 5 and 6 synthesized but violated the timing constraint with a slack of −6.88 ns. The reported latency of 405,215,745 cycles still represents a substantial reduction from the baseline of 20,461,812,225 cycles, and is competitive with the MLDHLS best-known solution of 544,821,946 cycles and the HGBO reference of 543,706,113 cycles. However, since the design does not meet the clock period requirement, this result is reported as a negative slack and excluded from valid comparisons.

For `knn`, all six runs synthesized successfully and deterministically, producing 1,086 cycles with zero slack. This represents a reduction of approximately 92% relative to the baseline of 13,599 cycles, surpassing both the MLDHLS best-known solution of 2,149 cycles and the HGBO reference of 2,285 cycles. Analysis of the generated code reveals that the model applied `ARRAY_PARTITION complete` to the three position arrays (`position_x`, `position_y`, `position_z`), while partitioning the larger neighbor

list (NL) cyclically, matching the UNROLL factor applied to the inner neighbor loop. This more selective partitioning strategy, combined with PIPELINE II=1 on the outer loop, allowed the scheduler to compute all neighbor interactions for a given atom in parallel within a single pipelined iteration, while avoiding the disproportionate resource cost of fully partitioning the largest array in the kernel. The complete consistency across runs makes this the most reliable result produced by this model.

For stencil3d, all six runs produced timing-compliant results. Runs 1 through 4 consistently yielded 283 cycles with zero slack. Analysis of the generated code reveals that the model applied ARRAY_PARTITION complete to all three kernel arrays and UNROLL on every nested loop level, including the doubly-nested pair at the core of the stencil computation. Run 5 reported a latency of 1 cycle, an outlier compared to the other five runs. Analysis of the generated code reveals that run 5 applied the same ARRAY_PARTITION complete and UNROLL strategy as runs 1–4, with the addition of INTERFACE directives. Analysis of the generated code reveals that run 6 differs from runs 1–4 by the addition of PIPELINE II=1 directives on the outer loops, producing a timing-compliant result of 318 cycles. The mean over all six runs is 241.83 cycles, below both the MLDHLS optimum of 295 cycles and the HGBO reference of 1,734 cycles, representing a strong result achieved in a single inference pass without any iterative feedback.

To contextualize the resource cost behind these results, Table 4.6 presents the utilization estimates reported by C Synthesis for the knn and stencil3d baselines alongside their respective Gemini-3-Flash configurations.

Table 4.6: Resource utilization: extreme-acceleration cases (Gemini-3-Flash).

Benchmark	Configuration	Cycles	BRAM	DSP	FF	LUT
KNN	Baseline	13,599	30 (1%)	41 (∼0%)	49,385 (2%)	124,920 (14%)
	Gemini-3-Flash	1,086	1,470 (54%)	616 (10%)	738,492 (42%)	510,740 (58%)
STENCIL3D	Baseline	9,475	16 (∼0%)	3 (∼0%)	9,875 (∼0%)	19,023 (2%)
	Gemini-3 (runs 1–4)	283	4,096 (152%)	7,056 (118%)	2,088,176 (119%)	2,382,158 (273%)
	Gemini-3 (run 5)	1	0 (–)	7,056 (118%)	286,902 (16%)	570,054 (65%)
	Gemini-3 (run 6)	318	4,096 (152%)	504 (8%)	1,914,720 (109%)	2,191,408 (251%)

Source: The author

For knn, all four resource metrics remain within the physical capacity of the target device, though with substantial growth relative to the baseline. BRAM and FF increase the most, reaching 54% and 42% respectively, a reflection of the mixed partitioning strategy described above: the cyclic partitioning of NL retains a moderate number of BRAM banks rather than eliminating memory blocks entirely, while the fully partitioned position arrays

and the unrolled neighbor loop drive the FF and LUT growth. This confirms that, for this benchmark, the latency gain is achieved through a genuine and physically realizable increase in hardware parallelism, achieved by balancing aggressive parallelism against the resource cost of partitioning the kernel's largest array.

For `stencil3d`, by contrast, all three resource profiles exceed the physical capacity of the target device in at least one metric, with LUT reaching 273% and 251% in runs 1–4 and run 6, respectively. The particularly high DSP consumption in runs 1–4 (118%) reflects the aggressive arithmetic parallelism enabled by the unrolled loop structure described above, applied in addition to the complete array partitioning, whereas run 6 achieved a comparable latency with substantially fewer DSPs (8%) by reusing arithmetic units across iterations through its added `PIPELINE II=1` directives, at the cost of a modest increase from 283 to 318 cycles. Run 5 shares the same 118% DSP utilization as runs 1–4, confirming that its underlying arithmetic structure is unchanged, but its BRAM utilization drops to zero and its FF and LUT usage fall to 16% and 65% respectively. As with the `knn` case for Gemini-2.5-Flash, the resulting cycle counts at or below the MLDHLS optimum of 295 cycles are best understood as the outcome of an unconstrained optimization: the model identified valid scheduling solutions with minimal latency, but at a resource cost the target device cannot physically accommodate.

4.3 Gemma-3-27b-it

Table 4.7 presents the synthesis results across all six inference runs for the Gemma-3-27b-it model.

Table 4.7: Gemma-3-27b-it synthesis results per run.

Benchmark	Run 1		Run 2		Run 3		Run 4		Run 5		Run 6	
	Cycles	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*
adpcm	11,588§	0.00	11,588§	0.00	11,588§	0.00	11,588§	0.00	11,588§	0.00	11,588§	0.00
aes	–	–	–	–	–	–	–	–	–	–	–	–
backprop	–	–	–	–	–	–	–	–	–	–	–	–
gemm	–	–	–	–	–	–	–	–	–	–	–	–
gramschmidt	19,790,461,441	−7.09	19,790,461,441	−7.09	19,790,461,441	−7.09	19,790,461,441	−7.09	19,790,461,441	−7.09	19,790,461,441	−7.09
gsm	–	–	–	–	–	–	–	–	–	–	–	–
knn	13,599	−0.35	13,599	−0.35	13,599	−0.35	13,599	−0.35	13,599	−0.35	13,599	−0.35
sha	31,723§	0.00	31,723§	0.00	59,037§	0.00	59,037§	0.00	59,037§	0.00	31,723§	0.00
stencil3d	–	–	–	–	–	–	–	–	–	–	–	–
trans_fft	–	–	–	–	–	–	–	–	–	–	–	–

§ Valid synthesis but latency exceeds the unoptimized baseline.

Source: The author

Table 4.8 presents a consolidated view of these results, grouped by synthesis validity and timing compliance.

Table 4.8: Gemma-3-27b-it average clock cycles and timing status per benchmark.

Benchmark	Baseline	Avg. Clock Cycles	Avg. Slack (ns)	Status
adpcm	1,717	11,588	0.00	Valid, worse than baseline[§]
aes	13,960	–	–	Synthesis failure
backprop	258,334,952	–	–	Synthesis failure
gemm	741,381	–	–	Synthesis failure
gramschmidt	20,461,812,225	19,790,461,441	−7.09	Negative slack
gsm	51,266	–	–	Synthesis failure
knn	13,599	13,599	−0.35	Negative slack
sha	25,421	45,380	0.00	Valid, worse than baseline[§]
stencil3d	9,475	–	–	Synthesis failure
transposed_fft	4,905	–	–	Synthesis failure

[§] Valid synthesis but result is worse than the unoptimized baseline.

Source: The author

The Gemma-3-27b-it model produced the weakest overall results among all evaluated models, with six out of ten benchmarks failing to synthesize entirely and no benchmark yielding a valid result that improved upon the unoptimized baseline.

The synthesis failures follow three distinct patterns. For `gemm` and `gsm`, the model delivered the main source files but omitted the corresponding header files (`gemm.h` and `gsm_lpc.h`), preventing compilation. For `stencil3d` and `transposed_fft`, pragmas were consistently inserted outside function scope across all six runs, violating the explicit constraint stated in the prompt. For `backprop`, the model introduced a more severe error: it modified the function signatures in the source file, causing type conflicts with the declarations in the header. This indicates that the model altered the original code structure beyond pragma insertion, which is outside the intended scope of the task.

Among the four benchmarks that produced synthesizable output, none achieved a latency improvement over the baseline. For `adpcm`, all six runs produced 11,588 cycles, a 6.7× degradation relative to the baseline of 1,717 cycles. Analysis of the generated code reveals that the model inserted `ARRAY_PARTITION complete` directives on all global arrays, including large read-only arrays such as `qq6_code6_table[64]` and `decis_levl[30]`. Fully partitioning large arrays that are accessed sequentially generates an excessive number of parallel memory ports without a corresponding increase in parallel accesses, imposing scheduling overhead that outweighs any potential benefit and ultimately degrades performance relative to the unoptimized sequential baseline. For `sha`, the results were not only worse than the baseline of 25,421 cycles but also inconsistent across runs, with two distinct outcomes of 31,723 and 59,037 cycles, indicating that

the model generated structurally different directive sets between runs without converging on a stable strategy.

For `gramschmidt` and `knn`, the designs synthesized but violated the timing constraint. The `knn` result is particularly telling: the reported latency of 13,599 cycles is identical to the unoptimized baseline, meaning the inserted directives had no measurable effect on the cycle count, yet still introduced a marginal negative slack of -0.35 ns. This combination, no latency gain with a timing penalty, represents the worst possible outcome of directive injection and suggests that the model inserted directives that were syntactically valid but semantically ineffective for this kernel.

Overall, the Gemma-3-27b-it results indicate that this model lacks the HLS domain knowledge required to reliably generate correct and effective optimization directives, even for benchmarks where the task is straightforward. The consistent degradation and scope violations across all runs suggest a fundamental limitation of this model rather than random generation artifacts.

4.4 Gemma-4-31b-it

Table 4.9 presents the synthesis results across all six inference runs for the Gemma-4-31b-it model.

Table 4.9: Gemma-4-31b-it synthesis results per run.

Benchmark	**Run 1**		**Run 2**		**Run 3**		**Run 4**		**Run 5**		**Run 6**	
	Cycles	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*	*Cycles*	*Slack (ns)*
adpcm	394	−16.24	394	−16.24	394	−16.24	394	−16.24	394	−16.24	394	−16.24
aes	–	–	–	–	–	–	–	–	–	–	–	–
backprop	2,112,470	−0.86	2,112,470	−0.86	2,112,470	−0.86	2,112,470	−0.86	2,112,470	−0.86	2,112,470	−0.86
gemm	–	–	–	–	–	–	–	–	–	–	–	–
gramschmidt	530,519,041	−0.02	530,519,041	−0.02	530,519,041	−0.02	530,519,041	−0.02	530,519,041	−0.02	530,519,041	−0.02
gsm	–	–	–	–	–	–	–	–	–	–	–	–
knn	1,014	0.00	1,014	0.00	1,014	0.00	1,014	0.00	1,014	0.00	1,014	0.00
sha	25,294	0.00	25,294	0.00	25,294	0.00	25,294	0.00	25,294	0.00	25,294	0.00
stencil3d	318	0.00	318	0.00	318	0.00	318	0.00	318	0.00	318	0.00
trans_fft	–	–	–	–	–	–	–	–	–	–	–	–

Source: The author

Table 4.10 aggregates the per-run data, classifying each benchmark according to its synthesis outcome and timing status.

Table 4.10: Gemma-4-31b-it average clock cycles and timing status per benchmark.

Benchmark	Baseline	Avg. Clock Cycles	Avg. Slack (ns)	Status
adpcm	1,717	394	−16.24	Negative slack
aes	13,960	–	–	Synthesis failure
backprop	258,334,952	2,112,470	−0.86	Negative slack
gemm	741,381	–	–	Synthesis failure
gramschmidt	20,461,812,225	530,519,041	−0.02	Negative slack
gsm	51,266	–	–	Synthesis failure
knn	13,599	1,014	0.00	**Valid**
sha	25,421	25,294	0.00	**Valid**
stencil3d	9,475	318	0.00	**Valid**
transposed_fft	4,905	–	–	Synthesis failure

Source: The author

The Gemma-4-31b-it model represents a substantial improvement over its predecessor Gemma-3-27b-it, producing synthesizable output for six out of ten benchmarks and, crucially, achieving valid timing-compliant results for three of them. The model also demonstrated a high degree of determinism: for every benchmark that synthesized, the result was identical in every run. Analysis of the generated code reveals that the model consistently applied the same directive strategy for each kernel.

The four synthesis failures exhibit two distinct error patterns. For `aes` and `gsm`, the model introduced undeclared identifiers directly within the directive arguments. For example, referencing a non-existent variable `word` in an `ARRAY_PARTITION` directive for `aes`, and variables `ACF`, `P`, and `K` in `gsm`. This indicates that the model hallucinated variable names that do not exist in the original source, inserting them as partition targets. For `transposed_fft`, run 1 exhibited a similar hallucination of array names, while runs 2 through 6 produced a pragma conflict between `UNROLL` and `PIPELINE` on the same loop, a semantically contradictory combination.

For `gemm`, no synthesis report was produced across any run. Post-hoc inspection of the generated code and the partial synthesis log reveals a two-stage failure. The model applied `ARRAY_PARTITION complete` to all three arrays (`m1`, `m2`, and `prod`), each containing 4,096 `double` elements, and additionally injected interface directives of the same type responsible for the bundle conflicts described in Section 4.6. The interaction between complete partitioning and these interface directives caused Vitis HLS to attempt the generation of one dedicated hardware port per partition element, resulting in 4,096 individual ports for `prod` alone. The tool never advanced past the RTL interface gener-

ation stage, and the host machine became unresponsive before any synthesis result could be produced.

Among the three cases with negative slack, the `gramschmidt` case is noteworthy: the slack of -0.02 ns is the most marginal violation in the entire experiment, and the achieved latency of 530,519,041 cycles represents a reduction of over 97% from the baseline of 20,461,812,225 cycles, closely approaching both the MLDHLS best-known solution of 544,821,946 cycles and the HGBO reference of 543,706,113 cycles. A minor clock relaxation or a small adjustment to the most critical path directive would likely render this result valid. The `adpcm` and `backprop` violations are noteworthy in that Gemini-2.5-Flash produced nearly identical results for both benchmarks; this cross-model convergence is analyzed in detail in Section 4.5.2.

The three valid results are the strongest aspect of this model's performance. For `sha`, all six runs produced 25,294 cycles with perfect consistency, making it the only benchmark across all evaluated models where `sha` was both synthesized correctly and improved upon the baseline of 25,421 cycles, a modest but reliable gain.

For `stencil3d`, the consistent 318 cycle result in every run is timing-compliant and represents a 96.6% reduction from the baseline, though it remains above the MLDHLS optimum of 295 cycles and the Gemini-3 result of 283 cycles. Analysis of the generated code reveals that the model applied `ARRAY_PARTITION complete` to all kernel arrays, combined with `UNROLL` on every nested loop level, including the doubly-nested pair at the core of the stencil computation. This combination allowed the HLS scheduler to unroll the entire 3D neighbor computation for each grid point into parallel combinational logic, rather than iterating sequentially across the stencil's spatial dimensions, at the cost of replicating the corresponding arithmetic and addressing logic for every unrolled iteration.

For `knn`, all six runs produced 1,014 cycles with zero slack, representing a 92.5% reduction from the baseline of 13,599 cycles and outperforming both the MLDHLS best-known solution of 2,149 cycles and the HGBO reference of 2,285 cycles. Analysis of the generated code reveals that the model applied `ARRAY_PARTITION complete` to all four kernel arrays, including the neighbor list (`NL`), combined with `PIPELINE` on the outer atom loop and `UNROLL` on the inner neighbor loop, replicating the distance and force computation logic for each neighbor. This combination allowed the HLS scheduler to compute the interaction with all neighbors of a given atom in parallel within a single pipelined iteration, rather than sequentially accumulating their contributions, at the cost of

replicating the corresponding distance and force computation logic once for each neighbor. Combined with the deterministic consistency across runs, this is one of the strongest individual results produced by any model in this experiment.

To contextualize the resource cost behind the `knn` and `stencil3d` results, Table 4.11 presents the utilization estimates reported by C Synthesis for both benchmarks' baselines alongside their respective Gemma-4-31b-it configurations.

Table 4.11: Resource utilization: extreme-acceleration cases (Gemma-4-31b-it).

Benchmark	Configuration	Cycles	BRAM	DSP	FF	LUT
KNN	Baseline	13,599	30 (1%)	41 (∼0%)	49,385 (2%)	124,920 (14%)
	Gemma-4-31b-it	1,014	8,734 (324%)	616 (10%)	3,741,083 (214%)	3,860,878 (442%)
STENCIL3D	Baseline	9,475	16 (∼0%)	3 (∼0%)	9,875 (∼0%)	19,023 (2%)
	Gemma-4-31b-it	318	4,096 (152%)	504 (8%)	1,914,720 (109%)	2,191,408 (251%)

Source: The author

For `knn`, the resource cost behind this result is substantial: BRAM reaches 324%, FF 214%, and LUT 442% of the device capacity, while DSP grows more moderately to 10%, confirming that the parallel replication strategy described above produced exactly the kind of resource explosion expected from full partitioning and unrolling applied to arrays and loops of this size. A similarly aggressive strategy was applied to `stencil3d`, where the replicated logic described above pushed LUT and FF utilization to 251% and 109% of the device capacity, respectively, while still achieving a hardware-realizable scheduling solution with 318 cycles of latency. As discussed for the Gemini-2.5-Flash `knn` case, this behavior reflects the model fulfilling the stated objective of cycle minimization in the absence of any resource constraints in the prompt.

4.5 Comparative Analysis

Table 4.12 consolidates the average latency results across all four evaluated models alongside the three reference points established in Chapter 3.

Table 4.12: Global latency performance comparison across LLM architectures (cycles).

Benchmark	Baseline	MLDHLS	HGBO	Gemini-2.5	Gemini-3	Gemma-3	Gemma-4
adpcm	1,717	1,240	1,133	393†	–	11,588§	394†
aes	13,960	847	513	–	–	–	–
backprop	258,334,952	5,191,225	7,148,529	2,112,224†	–	–	2,112,470†
gemm	741,381	16,721	20,609	–	–	–	–
gramschmidt	20,461,812,225	544,821,946	543,706,113	–	405,215,745†	19,790,461,441†	530,519,041†
gsm	51,266	809	1,152	87	–	–	–
knn	13,599	2,149	2,285	572	1,086	13,599†	1,014
sha	25,421	25,879	28,958	20,859†	–	45,380§	25,294
stencil3d	9,475	295	1,734	1,667	241.83	–	318
transposed_fft	4,905	2,149	1,349	2,879†	–	–	–

† Negative slack § Valid but worse than baseline.

Source: The author

4.5.1 Performance Against Reference Points

Among the four evaluated models, Gemini-2.5-Flash achieved the broadest coverage, producing synthesizable output for seven out of ten benchmarks. However, when restricting the analysis to timing-compliant results only, this coverage narrows to three benchmarks: `gsm`, `knn`, and `stencil3d`. Within these valid results, the model's performance is remarkable. For `gsm`, the achieved latency of 87 cycles surpasses both the MLDHLS best-known solution of 809 cycles and the HGBO reference of 1,152 cycles by a wide margin. For `knn`, the result of 572 cycles similarly outperforms both references. These two benchmarks represent the strongest individual results produced by any model in this experiment when considering only timing-valid outputs.

Gemini-3-Flash produced the fewest synthesizable results overall, but its valid outputs for `knn` and `stencil3d` are among the best in the experiment. The mean of 241.83 cycles for `stencil3d` is the only result across all models to fall below the MLDHLS optimum of 295 cycles, representing a particularly strong outcome given that it was achieved in a single inference pass without any iterative feedback. The model's critical limitation, however, is its failure to deliver complete code for most benchmarks, which rendered seven out of ten kernels non synthesizable.

Gemma-3-27b-it is the weakest model overall. It is the only model that produced results consistently worse than the unoptimized baseline for synthesizable benchmarks, with `adpcm` degrading by 6.7× and `sha` producing latencies between 1.25× and 2.3× above the baseline. No valid result from this model improved upon any reference point.

Gemma-4-31b-it demonstrates a clear generational improvement over Gemma-3,

achieving three fully valid and consistent results. For `knn` and `stencil3d`, the model outperforms both MLDHLS and HGBO references. For `sha`, it is the only model in the experiment to produce a valid improvement over the baseline for this particularly resistant benchmark. Additionally, the `gramschmidt` negative slack of -0.02 ns is the most marginal in the entire experiment, suggesting that a near-optimal directive set was identified but narrowly failed the timing closure requirement.

4.5.2 Cross-Model Convergence

A notable pattern emerges when comparing the numeric results across models: Gemini-2.5-Flash and Gemma-4-31b-it independently converged on nearly identical directive strategies for two benchmarks. For `adpcm`, both models achieved similar cycle counts (393 and 394 respectively) with an identical slack of -16.24 ns. Analysis of the generated code reveals that both applied the same core strategy: aggressive `ARRAY_PARTITION` on the same set of global variables, combined with `PIPELINE` and `UNROLL` on the same loops. Gemini-2.5-Flash additionally inserted AXI interface directives, but the near-identical results suggest these had negligible impact on latency. The shared timing bottleneck is therefore attributable to the common array partitioning strategy. For `backprop`, the results are 2,112,224 and 2,112,470 cycles with the same -0.86 ns slack, despite the strategies being structurally different: Gemini-2.5-Flash applied `UNROLL complete` and `LOOP_FLATTEN` alongside AXI interface directives, while Gemma-4-31b-it relied on `PIPELINE II=1` on inner loops with local array partitioning. The convergence on the same timing boundary from two distinct approaches suggests a fundamental physical constraint in this kernel that both strategies independently exposed. In both cases, the marginal slack of -0.86 ns indicates that a minor clock relaxation or targeted pragma adjustment would likely be sufficient to achieve a valid result.

4.5.3 Synthesis Coverage and Validity Summary

Table 4.13 summarizes the synthesis outcomes for each model across the ten benchmarks, categorizing each result into one of four states: valid improvement, negative slack, worse than baseline, or synthesis failure.

The table reveals two distinct failure profiles among the evaluated models. Gemini-

Table 4.13: Synthesis outcome summary per model (out of 10 benchmarks).

Model	Valid Improvement	Negative Slack	Worse than Baseline	Synthesis Failure
Gemini-2.5-Flash	3	4	0	3
Gemini-3-Flash	2	1	0	7
Gemma-3-27b-it	0	2	2	6
Gemma-4-31b-it	3	3	0	4

Source: The author

3-Flash concentrates its failures in the synthesis category due to incomplete code delivery, but its synthesizable results are consistently valid and competitive. Gemma-3-27b-it, by contrast, achieves a higher synthesis rate but produces no useful results, its outputs either violate timing or degrade performance. Gemini-2.5-Flash and Gemma-4-31b-it share a similar valid/violation split, with Gemma-4 showing stronger determinism across runs.

4.5.4 Input Complexity and Model Performance

Figures 4.1 and 4.2 were created based on the input size metrics reported in Table 3.2, together with the aggregate synthesis failure count per benchmark (out of the four evaluated models).

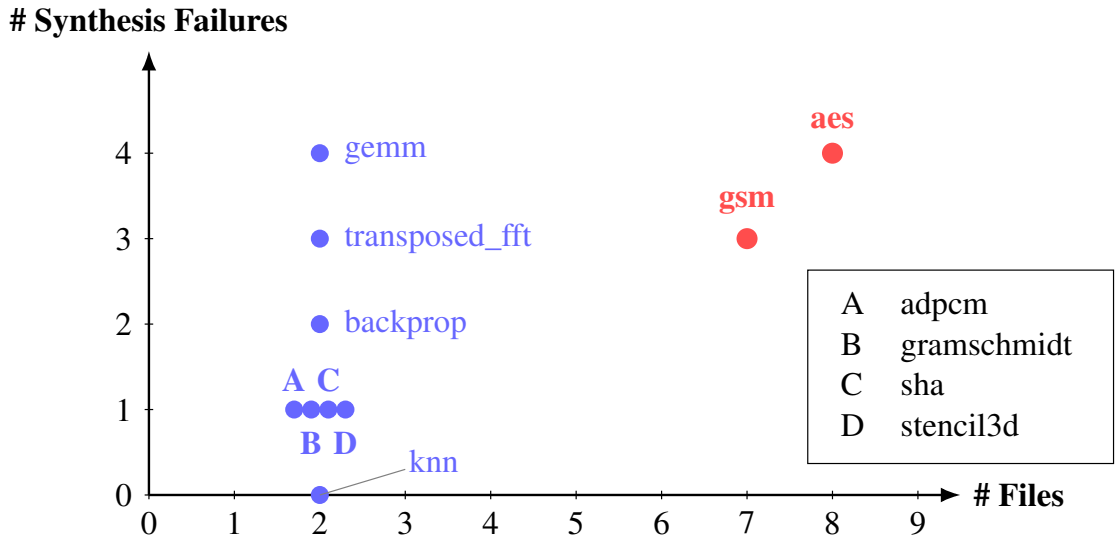
Analysis of Figure 4.1 shows that the two benchmarks with the highest file counts, `aes` (8 files) and `gsm` (7 files), were also among those with the highest failure counts (4 and 3 failures out of 4 models, respectively). This pattern extends to the rest of the benchmark set: the majority of benchmarks with only two files had zero or one failure (5 out of 8 benchmarks).

Analysis of Figure 4.2 shows a tendency between number of lines and number of failures across most benchmarks, progressing from `stencil3d` and `knn` (fewest lines, fewest failures) through `gramschmidt`, `sha`, `backprop`, `transposed_fft`, and `gsm`, up to `aes` (most lines, most failures). Two benchmarks clearly deviate from this trend. `gemm` (62 lines, the smallest benchmark in the dataset) reaches the maximum failure count of 4, driven by the AXI interface bundle conflicts and interface-related synthesis crash documented in Section 4.6, rather than by any structural complexity related to code length. Conversely, `adpcm` (640 lines) shows only a single failure despite its comparatively high line count, as three of the four models produced valid or negative-slack results for this benchmark rather than failing outright.

Taken together, total lines of code shows a more discernible trend with failure count than file count does in this dataset, although the two outliers identified for line count, `gemm` and `adpcm`, indicate that this relationship is not absolute. Given the limited

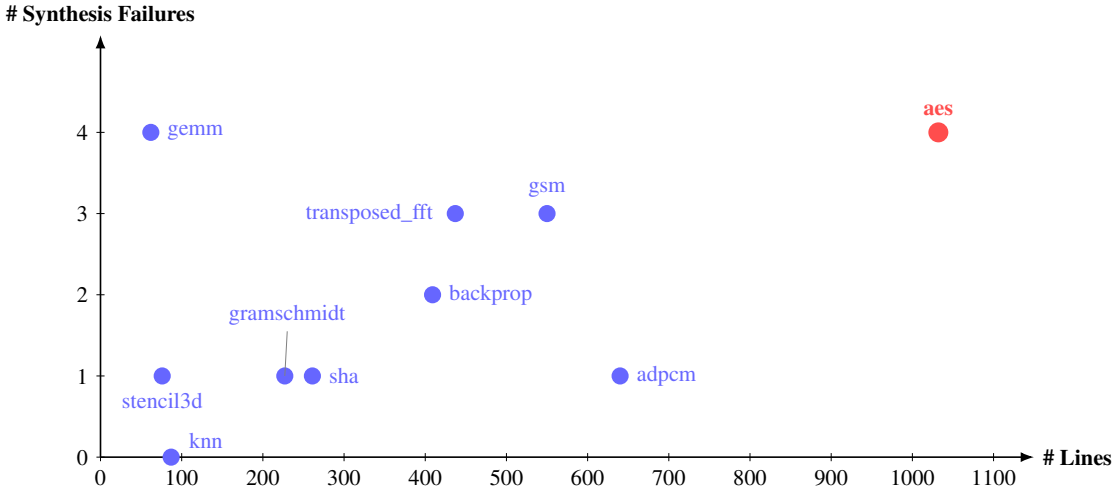
sample size (ten benchmarks), these observations are presented as descriptive patterns drawn directly from the data, not as proven relationships.

Figure 4.1: Number of source files per benchmark versus synthesis failure count.



Source: The author

Figure 4.2: Total lines of code per benchmark versus synthesis failure count.



Source: The author

4.6 Synthesis Failure Analysis

Across all four models and ten benchmarks, synthesis failures constituted a significant portion of the total runs. This section consolidates the observed failure patterns into a taxonomy of root causes, which provides insight into the specific limitations of zero-shot LLM directive injection and informs potential mitigation strategies.

4.6.1 Pragma Scope Violations

The most frequently recurring failure across all models was the insertion of `#pragma HLS` directives outside function scope. Vitis HLS requires all pragmas to be placed within the body of the function they target; placing them at file scope or between function definitions results in a hard compiler error. This constraint was explicitly stated in the prompt for all experiments, yet violations were observed in Gemini-2.5-Flash (`backprop`, `sha`), Gemma-3-27b-it (`stencil3d`, `transposed_fft`, `aes`), indicating that explicit instruction alone is insufficient to reliably enforce this constraint in zero-shot settings.

4.6.2 AXI Interface Bundle Conflicts

Gemini-2.5-Flash consistently generated conflicting `s_axilite` interface directives for `gemm` and `gramschmidt` across all six runs. The error indicates that the model assigned different AXI bundle names (`control` and `control_r`) to ports that Vitis HLS requires to share a single bundle in kernel mode. This is a Vitis-specific interface constraint that differs from standard HLS pragma behavior, and the model's consistent failure on these benchmarks suggests a gap in its knowledge of the Vitis kernel compilation flow as opposed to the more general HLS synthesis flow.

4.6.3 Incomplete Code Delivery

Gemini-3-Flash failed to deliver the complete set of source files for eight out of ten benchmarks across most runs. After the extraction script was updated to handle tag format variations, the remaining failures were traced to truncated responses: the model delivered one or more files from a multi-file benchmark but omitted others, leaving the HLS project without a compilable design unit. For `aes`, `gsm`, and `gramschmidt`, the missing files were identifiable headers; for other benchmarks, the absence of any source file produced the generic `HLS 200-70` elaboration error. This behavior is consistent with output length limitations or loss of instruction-following coherence during long generations.

4.6.4 Variable Hallucination in Directive Arguments

Gemma-4-31b-it introduced a qualitatively distinct failure mode: hallucinated variable names inserted as arguments to `ARRAY_PARTITION` directives. For `aes`, the model referenced a non-existent identifier `word`; for `gsm`, it referenced `ACF`, `P`, and `K`. These identifiers do not exist in the original source code of the respective benchmarks. This failure mode is particularly insidious because the generated code is syntactically well-formed, the pragmas follow correct Vitis HLS syntax, but reference variables that the compiler cannot resolve. It suggests that the model identified the correct directive type (`ARRAY_PARTITION`) and even the correct insertion location, but failed to ground the variable name argument in the actual source context.

4.6.5 Conflicting Pragma Combinations

Several models generated semantically contradictory directive combinations that the HLS compiler rejects at synthesis time. The most common was the simultaneous application of `INLINE` and `PIPELINE` to the same function, observed in Gemini-2.5-Flash for `gsm`. A related conflict between `UNROLL` and `PIPELINE` on the same loop was observed in Gemma-4-31b-it for `transposed_fft`. In both cases, the directives are individually valid but mutually exclusive: `INLINE` removes the function boundary that `PIPELINE` requires, and complete `UNROLL` eliminates the loop structure that `PIPELINE` targets. These errors indicate that the models lack a representation of pragma interaction constraints, treating each directive as independent rather than as part of an interdependent optimization strategy.

4.6.6 Source Code Corruption

Gemini-2.5-Flash corrupted the source code of `aes` across all runs by introducing unbalanced braces and undeclared identifiers into the body of `aes_key.c`, producing cascading parse errors that prevented compilation entirely. Gemma-3-27b-it exhibited a related but distinct corruption for `backprop`: rather than introducing syntactic errors, the model modified the function signatures in the source file, causing type conflicts with the declarations in the header. Both cases represent a failure mode where the model altered

the original algorithmic logic rather than restricting its output to pragma insertion, which was the explicit task specification.

5 CONCLUSION

This work investigated the viability of using commercial Large Language Models for automated HLS directive injection through a zero-shot prompt engineering approach, without fine-tuning, retrieval-augmented generation, or multi-agent coordination. The proposed pipeline integrated the Google Gemini API with the AMD Vitis HLS tool and was evaluated across four models and ten benchmarks from the MLDHLS dataset.

The results demonstrate that frontier LLMs possess latent HLS domain knowledge sufficient to produce competitive optimization directives in a single inference pass. Among the evaluated models, Gemini-2.5-Flash and Gemma-4-31b-it achieved the strongest overall performance, each producing three timing-compliant results that improved upon the unoptimized baseline. For `gsm` and `knn`, Gemini-2.5-Flash surpassed both the MLDHLS best-known solutions and the HGBO state-of-the-art reference across 100 synthesis-informed iterations. Gemini-3-Flash, despite its high synthesis failure rate due to incomplete code delivery, produced the single best result for `stencil3d` across all models, falling below the MLDHLS optimum of 295 cycles with a mean of 241.83 cycles. Gemma-4-31b-it demonstrated the highest determinism of all evaluated models, producing identical results across all six runs for every synthesizable benchmark, and was the only model to improve upon the baseline for the particularly resistant `sha` kernel.

It is important to qualify the magnitude of some of these results. Several of the strongest-performing configurations, including the `knn` cases for Gemini-2.5-Flash and Gemma-4-31b-it, and the majority of `stencil3d` configurations across Gemini-3-Flash and Gemma-4-31b-it, exceed the physical resource capacity of the target device, most notably in BRAM, FF, and LUT utilization. These configurations remain valid outcomes of the C Synthesis stage and a faithful reflection of the optimization objective communicated to the models, which specified only cycle count minimization without any resource constraint or target-device information. However, this also means that a portion of the most impressive latency reductions reported in this work would require a larger FPGA device, or a revised prompt incorporating explicit resource budgets, before being physically realizable on the evaluated hardware.

At the same time, the evaluation revealed systematic failure modes that constrain the practical applicability of the zero-shot approach. Pragma scope violations, AXI interface bundle conflicts, incomplete code delivery, variable hallucination in directive arguments, and conflicting pragma combinations were consistently observed across models,

even when the relevant constraints were explicitly stated in the prompt. These failures were not random: each model exhibited a characteristic error profile that reflects structural gaps in its HLS domain knowledge. Furthermore, designs with negative slack affected a significant portion of the synthesized results, indicating that models tend to prioritize cycle count reduction without an inherent awareness of the physical timing boundaries imposed by the target FPGA fabric.

Taken together, these findings suggest that zero-shot LLM directive injection occupies a viable but bounded niche in the HLS optimization landscape. It is most effective for benchmarks with clear loop structures and memory access patterns that align with common HLS optimization idioms, and least effective for kernels with complex multi-file structures, specialized interface requirements, or tight timing constraints. The approach is best understood not as a replacement for iterative DSE frameworks, but as a low-cost first pass that can rapidly identify strong directive candidates for a subset of benchmarks, potentially reducing the number of synthesis iterations required by more expensive methods.

5.1 Future Work

Several directions emerge directly from the limitations and observations of this work.

The prompt engineering strategy represents the most accessible improvement path. This work used a fixed zero-shot template across all models and benchmarks; few-shot examples with known-valid directive configurations, chain-of-thought reasoning instructions, or model-specific prompt adaptations could meaningfully reduce the pragma scope violation rate and improve timing compliance. In particular, providing one or two examples of correctly scoped pragmas for a simple kernel before presenting the target benchmark may be sufficient to eliminate the most frequent failure mode observed, without requiring any changes to the pipeline infrastructure.

Extending the optimization objective beyond latency is another natural direction. This work focused exclusively on cycle count minimization, but practical hardware design requires navigating trade-offs between power, performance, and area. Future work could reformulate the prompt to target multi-objective optimization, instructing the model to balance latency against resource utilization or power consumption, and evaluate results using PPA metrics rather than latency alone. This would also enable a fairer comparison

against HGBO, which was originally designed for PPA optimization.

Finally, extending the evaluation to a broader set of models, including non-Google providers and open-source alternatives, would allow for a more comprehensive characterization of the relationship between model size, architecture, training data, and HLS domain knowledge. The four models evaluated in this work represent only a subset of the available landscape, and a systematic study across a wider and more diverse set of models could clarify which characteristics most influence directive generation quality, informing model selection criteria for automated hardware design pipelines.

REFERENCES

Advanced Micro Devices, Inc. *HLS Pragmas • Vitis Unified IDE and Common Command-Line Reference Manual (UG1553)*. 2023. Available from Internet: <https://docs.amd.com/r/en-US/ug1553-vitis-ide/HLS-Pragmas>.

ANDRADE, H. C. *MLDHLS: Multiple Loop Directive HLS Dataset*. 2025. Accessed: 2025-11-24. Available from Internet: <https://github.com/HeitorCAndrade/MLDHLS>.

BROWN, T. et al. Language models are few-shot learners. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020. v. 33, p. 1877–1901. Available from Internet: <https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html>.

CHI, Y. et al. *Democratizing Domain-Specific Computing*. 2022. Available from Internet: <https://arxiv.org/abs/2209.02951>.

GAJSKI, D. D. et al. *High — Level Synthesis*. Springer US, 1992. ISBN 978-1-4613-6617-1 978-1-4615-3636-9. Available from Internet: <http://link.springer.com/10.1007/978-1-4615-3636-9>.

Google DeepMind. *Gemini 2.5 Flash Model Card*. [S.l.], 2025. Available from Internet: <https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-2-5-Flash-Model-Card.pdf>.

Google DeepMind. *Gemini 3 Flash Model Card*. [S.l.], 2025. Available from Internet: <https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf>.

Google DeepMind. *Gemma 4 Model Overview*. 2025. <https://ai.google.dev/gemma/docs/core/model_card_4?utm_source=deepmind.google&utm_medium=referral&utm_campaign=gdm&utm_content&hl=pt-br>. Accessed: 2025.

HENNESSY, J. L.; PATTERSON, D. A. A new golden age for computer architecture. v. 62, n. 2, p. 48–60, 2019. ISSN 0001-0782. Available from Internet: <https://dl.acm.org/doi/10.1145/3282307>.

KASTNER, R.; MATAI, J.; NEUENDORFFER, S. *Parallel Programming for FPGAs*. 2018. Available from Internet: <https://arxiv.org/abs/1805.03648>.

KUANG, H. et al. Hgbo-dse: Hierarchical gnn and bayesian optimization based hls design space exploration. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. [S.l.: s.n.], 2023. p. 106–114.

LAHTI, S.; HäMäLäINEN, T. D. High-level synthesis for FPGAs—a hardware engineer's perspective. v. 13, p. 28574–28593, 2025. ISSN 2169-3536. Available from Internet: <https://ieeexplore.ieee.org/abstract/document/10878985>.

LEWIS, P. et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. Available from Internet: <https://arxiv.org/abs/2005.11401>.

LIN, Z. et al. Powergear: Early-stage power estimation in fpga hls via heterogeneous edge-centric gnns. In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*. [S.l.: s.n.], 2022. p. 1341–1346.

MAHAPATRA, A.; SCHAFER, B. C. Machine-learning based simulated annealer method for high level synthesis design space exploration. In: *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*. [S.l.: s.n.], 2014. p. 1–6.

MASHNOOR, N. et al. TimelyHLS: LLM-based timing-aware and architecture-specific FPGA HLS optimization. In: *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. [s.n.], 2025. p. 1–6. ISSN: 2996-5330. Available from Internet: <https://ieeexplore.ieee.org/abstract/document/11125726>.

MICHELI, G. D. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994. (Electrical and Computer Engineering Series). ISBN 978-0-07-016333-1. Available from Internet: <https://books.google.fr/books?id=FESDQgAACAAJ>.

MOLINA, R. S. et al. High-level synthesis hardware design for FPGA-based accelerators: Models, methodologies, and frameworks. v. 10, p. 90429–90455, 2022. ISSN 2169-3536. Available from Internet: <https://ieeexplore.ieee.org/document/9864576/>.

NUMAN, M. W. et al. Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains. v. 8, p. 174692–174722, 2020. ISSN 2169-3536. Available from Internet: <https://ieeexplore.ieee.org/document/9195872>.

ROMERA-PAREDES, B. et al. Mathematical discoveries from program search with large language models. v. 625, n. 7995, p. 468–475, 2024. ISSN 1476-4687. Publisher: Nature Publishing Group. Available from Internet: <https://www.nature.com/articles/s41586-023-06924-6>.

SCHAFER, B. C. Parallel high-level synthesis design space exploration for behavioral ips of exact latencies. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, ACM New York, NY, USA, v. 22, n. 4, p. 1–20, 2017.

SCHAFER, B. C.; WANG, Z. High-level synthesis design space exploration: Past, present, and future. v. 39, n. 10, p. 2628–2639, 2020. ISSN 1937-4151. Available from Internet: <https://ieeexplore.ieee.org/abstract/document/8847448>.

SILVER, T. et al. Classical planning with llm-generated heuristics: Challenging the state of the art with python code. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [s.n.], 2025. Available from Internet: <https://neurips.cc/virtual/2025/loc/san-diego/poster/117784>.

SOHRABIZADEH, A. et al. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Trans. Des. Autom. Electron. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 27, n. 4, feb. 2022. ISSN 1084-4309. Available from Internet: <https://doi.org/10.1145/3494534>.

SONG, Y. et al. The good, the bad, and the greedy: Evaluation of LLMs should not ignore non-determinism. In: CHIRUZZO, L.; RITTER, A.; WANG, L. (Ed.). *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of*

the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers). Albuquerque, New Mexico: Association for Computational Linguistics, 2025. p. 4195–4206. ISBN 979-8-89176-189-6. Available from Internet: <https://aclanthology.org/2025.naacl-long.211/>.

SUN, Q. et al. Correlated multi-objective multi-fidelity optimization for hls directives design. *ACM Trans. Des. Autom. Electron. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 27, n. 4, mar. 2022. ISSN 1084-4309. Available from Internet: <https://doi.org/10.1145/3503540>.

TEAM, G. *Gemma 2: Improving Open Language Models at a Practical Size*. 2024. Available from Internet: <https://arxiv.org/abs/2408.00118>.

TEAM, G. *Gemini: A Family of Highly Capable Multimodal Models*. 2025. Available from Internet: <https://arxiv.org/abs/2312.11805>.

TEAM, G. *Gemma 3 Technical Report*. 2025. Available from Internet: <https://arxiv.org/abs/2503.19786>.

WANG, H. et al. *LLM-DSE: Searching Accelerator Parameters with LLM Agents*. arXiv, 2025. Available from Internet: <http://arxiv.org/abs/2505.12188>.

WU, N.; XIE, Y.; HAO, C. Ironman-pro: Multiobjective design space exploration in hls via reinforcement learning and graph neural network-based modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 42, n. 3, p. 900–913, 2023.

WU, N. et al. High-level synthesis performance prediction using gnns: benchmarking, modeling, and advancing. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2022. (DAC '22), p. 49–54. ISBN 9781450391429. Available from Internet: <https://doi.org/10.1145/3489517.3530408>.

XU, H.; HU, H.; HUANG, S. *Optimizing High-Level Synthesis Designs with Retrieval-Augmented Large Language Models*. arXiv, 2024. Available from Internet: <http://arxiv.org/abs/2410.07356>.

YAO, X. et al. High-level synthesis directives design optimization via large language model. v. 30, n. 5, p. 78:1–78:24, 2025. ISSN 1084-4309. Available from Internet: <https://dl.acm.org/doi/10.1145/3747291>.