计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(6): 1392−1408, 2026

SimulatorGen: 基于大语言模型多智能体的 DNN 加速器模拟器自动生成框架

夏雨欢　李　暾　周贤发　赵文博　张瑞瑜　郭　阳
（国防科技大学计算机学院　长沙　410073）
（xiayuhuan@nudt.edu.cn）

SimulatorGen: An LLM-Based Multi-Agent Framework for Automatic Generation of DNN Accelerator Simulators

Xia Yuhuan, Li Tun, Zhou Xianfa, Zhao Wenbo, Zhang Ruiyu, and Guo Yang
（*College of Computer*, *National University of Defense Technology*, *Changsha* 410073）

Abstract　With the rapid development of deep neural network (DNN) accelerators, building simulators for new architectures is costly and time-consuming. Although advances in large language models (LLMs) have opened possibilities for automated simulator generation, existing approaches suffer from limited generality, inability to construct complete systems, and high construction complexity. To address these challenges, we propose SimulatorGen, a multi-agent framework that generates DNN accelerator simulator code from natural language descriptions. First, we abstract the architecture of DNN accelerator simulators and extract twenty-three component specifications. Based on this abstraction, four collaborative agents are introduced to accomplish generation: the analyst agent retrieves domain knowledge from the simulator library via retrieval-augmented generation (RAG) and constructs structured prompts by leveraging chain-of-thought (CoT) reasoning; the coder agent generates or refines code using prompts and test feedback; the tester agent performs syntax checking, functional testing, and formal verification using the Z3 solver based on properties extracted from specifications; and the assembly agent conducts component integration, automated execution, and metric comparison to enable end-to-end construction. We evaluate SimulatorGen on twenty-three generation tasks covering diverse DNN accelerator modules and architectures. Experimental results show that SimulatorGen built on GPT-4o outperforms LLM baselines, including Claude-Sonnet-4, achieving a Pass@1 score of 82.39%. Furthermore, using the successfully generated components, we construct runnable simulators for tensor processing unit (TPU) and MAERI architectures. Compared with STONNE, the simulators built by SimulatorGen achieve relative errors ranging from 1.31% to 7.34% in energy, latency, and energy-delay product (EDP) across multiple DNN models, while maintaining functional consistency verified through testing and execution, demonstrating faithful modeling of accelerator behavior. In contrast to the single-agent SimulatorCoder, which only supports module replacement, SimulatorGen enables end-to-end generation of complete simulators, further validating the effectiveness of the proposed approach.

Key words　DNN accelerator simulator; large language model; multi-agent; code generation; retrieval-augmented generation (RAG)

收稿日期: 2026-03-12; **修回日期**: 2026-05-06
基金项目: 国家自然科学基金项目(92473115)
This work was supported by the National Natural Science Foundation of China (92473115).
通信作者: 李暾(tunli@nudt.edu.cn)

摘　要　随着深度神经网络（DNN）加速器的快速发展，为新架构构建模拟器的成本高、周期长。尽管大语言模型（LLM）的进展为模拟器生成带来了新的可能性，但现有方法仍存在通用性不足、难以生成完整系统以及构建复杂度高等问题。为此，提出SimulatorGen，一种多智能体框架，其基于自然语言描述生成DNN加速器模拟器代码。首先，对通用DNN加速器模拟器架构进行抽象并提取23条组件规范；在此基础上，引入4类协同智能体完成生成过程：分析智能体通过检索增强生成（RAG）从模拟器库中检索领域知识，并结合思维链（CoT）构建结构化提示词；编码智能体根据提示词生成代码，或基于测试反馈修复错误代码；测试智能体基于从规范中提取的属性，执行语法检查、功能测试以及使用Z3求解器的形式化验证；组装智能体负责组件组装、自动执行与指标比对，实现完整模拟器的自动构建。在覆盖多样化DNN加速器模块与架构的23个生成任务上对SimulatorGen进行评估，实验结果表明，基于GPT-4o构建的SimulatorGen的表现优于包括Claude-Sonnet-4在内的LLM基线，Pass@1达到82.39%。在成功设计的组件基础上，进一步使用SimulatorGen构建了可运行的张量处理单元（TPU）和MAERI架构的模拟器。与STONNE相比，SimulatorGen构建的模拟器在多个DNN模型上的能量、时延和能量时延乘积（EDP）指标相对误差范围为1.31%~7.34%，且功能行为通过测试与执行验证保持一致，表明其具备准确建模加速器行为的能力。同时，相比于仅支持模块替换的单智能体SimulatorCoder，SimulatorGen具备端到端生成完整模拟器的能力，进一步验证了所提方法的有效性。

关键词　DNN加速器模拟器；大语言模型；多智能体；代码生成；检索增强生成

中图法分类号　TP391；TP303

DOI: 10.7544/issn1000-1239.202660116　　**CSTR:** 32373.14.issn1000-1239.202660116

随着人工智能的快速发展，深度神经网络（deep neural network, DNN）加速器已广泛应用于云计算和边缘推理等重要而复杂的场景中[1]。为满足高吞吐、低功耗和高容错的需求，出现了各种面向DNN的加速器，如脉动阵列和空间加速器结构[2]。然而，这些加速器的设计空间迅速扩展，涉及计算、存储、调度和互连等多方面因素。设计过程日益复杂，单纯依赖人工评估已难以支持快速迭代。

随着DNN加速器设计复杂度的提升，研究人员需要能够在开发早期快速而准确地探索设计空间和评估架构改进效果的工具[3]。当前的加速器评估方法主要基于分析模型[4-5]或模拟器[6-8]。分析模型通过数学公式与简化来快速估计不同配置下的性能，但往往无法捕捉细微的执行行为，导致在许多场景中结果不够准确。随着架构变得更加复杂和灵活，分析模型的准确性与可靠性显著下降。为了提供更准确的评估，周期级模拟器（例如STONNE[6]）被视为不可或缺的工具。但是为每一种新的DNN加速器构建专用的模拟器既耗时又资源密集，如表1所示，广泛使用的开源模拟器往往包含数万行代码，这在一定程度上限制了设计空间的高效探索。

大语言模型（large language model, LLM）的出现为自动化芯片设计带来了新的机遇。已有研究开始探索利用自然语言驱动AI芯片设计，并取得了初步成果[11-12]。在诸多开发活动中，代码生成因其降低开发成本、缩短迭代周期的潜力而备受关注[13]。诸如思维链（chain-of-thought, CoT）[14]与检索增强生成（retrieval-augmented generation, RAG）[15]等LLM技术已被证明可以提升生成质量与鲁棒性。随着工具链和实践的快速演进，LLM正逐渐成为芯片设计开发流程中的关键组成部分。基于LLM的智能体逐渐在众多LLM技术中脱颖而出。智能体是一个具备环境感知、决策制定及动作执行能力的自主算法系统，是可定制的LLM实例，能够在工程工作流中复现并执行面向特定目标的离散任务[16]。有研究[17]探索了利用LLM自动生成与优化DNN加速器模拟器的方法，开发了单智能体SimulatorCoder，通过引入结合思维链推理和多轮反馈机制以提升代码生成的正确性。相关工作主要面向二维脉动阵列结构开展验证，且仅进行模块替换。然而，使用单个LLM或单智能体的研究存在显著局限性，往往难以胜任复杂任务。相

Table 1　DNN Accelerator Simulators Proposed in Our Work

表1　本文提出的DNN加速器模拟器

模拟器	计算类型	代码行数
STONNE[6]	乘加计算	19 383
SIAM[7]	存内计算	26 797
SMAUG[9]	脉动阵列	22 851
NoCDAS[10]	片上网络	29 747

比之下,多智能体协作通过明确的角色划分与职责边界,在应对代码生成中的系统性挑战方面取得了显著进展[18]。

尽管 LLM 在芯片设计任务中前景广阔,但将其应用于 DNN 加速器模拟器生成仍面临挑战:1)通用性不足。据调研,现有工作尚缺少给定任意需求或功能描述即可自动生成完整可运行模拟器的能力。2)缺乏完整模拟器生成能力。之前的基于 LLM 进行 DNN 加速器模拟器的生成工作是模块级的替换,并未实现从自然语言需求到完整模拟器的端到端自动构建。3)构建复杂度高。模拟器需要模拟硬件级计算与时序语义方面的细节。然而,不同于 DNN 加速器已有成熟设计工具链,模拟器通常是自建框架,验证过程高度依赖人工,从而限制了其开发效率。

为此,本文提出一种面向 DNN 加速器模拟器生成的多智能体框架,在组件级别分解开发任务,并协调 4 个协同工作的角色:1)分析智能体,负责领域知识检索,将需求结构化以构建领域提示词;2)编码智能体,负责在需求与测试反馈约束下生成代码;3)测试智能体,负责涵盖编译、集成测试与形式化检查的多阶段验证;4)组装智能体,负责组件组装、自动执行与指标比对,从而自动构建完整模拟器。该框架不仅支持自动化模拟器生成,还通过任务分解与可追踪的验证反馈机制促进对生成系统的程序理解。

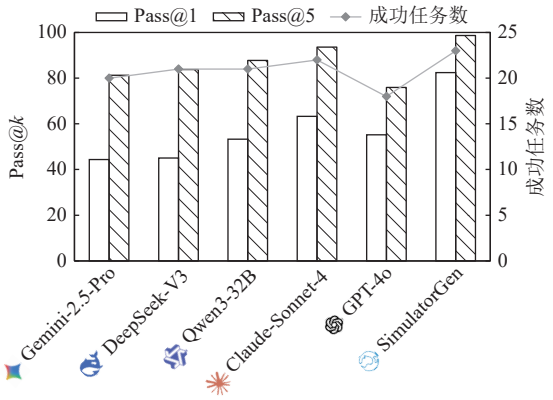
本文的主要贡献包括 3 个方面:

1)架构抽象和规范提取。对 DNN 加速器模拟器的通用体系结构进行系统性抽象,总结并提取了模拟器的自然语言功能规范,为后续基于自然语言驱动的自动生成奠定了统一描述基础。

2)整体模拟器生成。本文基于 LLM 提出了一种面向整体 DNN 加速器模拟器系统的自动生成方法,实现了从高层自然语言描述到完整可运行模拟器的端到端构建。

3)多智能体生成框架。提出一种无需微调、基于自然语言功能规范的多智能体 LLM 框架,用于 DNN 加速器模拟器生成。该框架通过定义智能体角色并进行组件级任务分解,构建模拟器组件库与任务基准;结合 CoT 与 RAG 设计上下文提示模板,引导 LLM 生成语义正确、结构完整的模块;进行迭代反馈验证,建立了一种基于反馈的测试机制,将语法与编译检查、功能测试以及基于 Z3 的形式化验证相结合,通过在测试反馈的驱动下对代码进行迭代式修正,从而显著提升了生成代码的正确性与可靠性;同时进行组件代码的系统化组装、自动执行与指标比对,实现从模块生成到完整模拟器构建与验证的端到端自动化流程。

在 23 个模拟器组件设计任务中,如图 1 所示,SimulatorGen 在生成准确率上优于 DeepSeek-V3 等多种强基线模型。此外,将生成的组件进一步组装为完整模拟器后,其周期数与能量指标均与人工设计实现的模拟器高度接近,表明该方法具有良好的实用性与可迁移性。



注:Pass@1 表示单次生成成功率,Pass@5 表示 5 个候选中至少 1 个成功的概率。

Fig. 1 Performance over twenty-three simulator component tasks

图 1 23 个模拟器组件任务的性能

1 相关工作

1.1 LLM 代码生成

代码生成(或程序合成)是指通过自动化技术自主构建软件或编写代码的过程[13]。LLM 可以根据自然语言描述生成源代码,这一过程通常被称为“自然语言到代码”任务。

为提升基于 LLM 的代码生成准确性,已有大量方法被提出,包括提示工程、RAG[15]、微调(fine-tuning)[19]、程序验证与自我修复[20],以及基于人类反馈的强化学习(reinforcement learning from human feedback, RLHF)[21]。提示词工程技术包括上下文学习(in-context learning, ICL)[22]、CoT[14] 等。RAG 是指一种在响应查询时,首先从文档库中检索相关信息,然后将检索结果与原始查询相结合以提升响应质量和准确性的方法,其通过引入外部知识库扩展了传统 LLM 的能力,有助于缓解模型生成内容中的幻觉问题。同时,微调和 RLHF 等方法通常需要准备专门的数据集用于训练模型,消耗更多计算资源,且在实践中往往不如性能强大的开源模型有效[23]。在本工作中,

则利用未经训练的LLM自动生成模拟器代码。

1.2 多智能体系统

多智能体协作是指在多智能体系统框架下，多个可交互的智能体围绕共同目标进行协调与分工。通过信息交换与联合决策，它们能够完成单个智能体难以甚至无法完成的复杂任务[24]。

单智能体在多次迭代中需要在不同子任务之间切换上下文，在硬件设计过程中采用单个智能体处理所有这些任务会导致次优结果，因为它们必须在每次交互中处理如此复杂的上下文，并通过冗长的对话历史保持一致性。相比之下，多智能体系统将任务分配给具有独立对话历史的智能体，支持专门的任务处理和更高的模块化程度。

Lin等人[25]提出了一种利用多个LLM智能体来模拟软件过程模型的代码生成框架，为LLM智能体分配对应日常开发活动的角色，包括需求工程师、架构师、开发人员等，同时利用CoT、提示组合和持续自我优化的方式协同工作，以提高代码质量。但该方法仅应用于公开基准（如HumanEval）测试，尚未涉及完整软件系统的生成。Dong等人[26]提出了一种基于LLM的协作式代码生成框架，通过角色分配与团队协作显著提升了代码生成的准确性，并为LLM在复杂任务中的应用提供了新的视角。然而，该框架在测试阶段是通过大模型直接评估生成报告，而非执行真实代码，尽管能够识别大多数逻辑错误，但无法在真实执行过程中对正确性进行充分验证。该框架将自我协作代码生成应用于复杂的游戏开发需求，并开发了天气预报网站。相比之下，本文的方法采用了更为严格、以实际执行为驱动的反馈测试机制，从而更有效地保证了代码的正确性，同时，研究对时序和功能等模拟正确性要求更高的模拟器代码生成。

1.3 模拟器生成

机器学习、大模型等人工智能技术为赋能芯片设计带来了新的机遇[27]。目前多数工作集中于手动生成模拟器或者构建模拟架构。Muñoz-Martínez等人[6]提出STONNE，一个具有可重构数据流模式的DNN加速器周期精确模拟器，支持TPU等架构。Ritik等人[8]提出SCALE-Sim，一个周期精确的模拟器框架，提供可配置的脉动阵列设计，凭借它用户可以定义一个配置文件来描述选择的架构。林涵越等人[28]提出一种通用网络处理器的结构模拟和性能仿真框架Neptune，采用同步图模拟和混合事件与时间驱动的方法以保障模拟器准确性和高效性。

在DNN加速器及其模拟器生成方面，已有一些相关研究工作。例如，Fu等人[11]提出了一种对LLM友好的硬件模板以及一种结合示例增强的提示词生成方法，用于自动化AI加速器设计。Vungarala等人[12]提出了TPU-Gen，旨在实现基于LLM的TPU精确与近似生成流程自动化，重点关注脉动阵列的微体系结构设计。文献[29]提出了体系结构描述方法，以支持基于分析模型的DNN加速器性能建模。Nayak等人[30]提出了TeAAL，一种用于简洁、精确地规范和评估稀疏张量代数加速器的语言和模拟器生成器。但其仅支持对稀疏张量代数加速器进行建模和评估，且使用分析建模。文献[17]提出了Simulator-Coder，说明LLM具备生成DNN加速器模拟器函数级代码的能力。相比之下，本文进一步关注更复杂的架构场景，采用基于LLM的多智能体协同框架，所研究的目标架构涵盖典型的二维脉动阵列加速器以及可重构数据流加速器等体系结构，且进行整体模拟器生成。

在微处理器领域，模拟器生成同样受到关注，通过硬件描述语言或者体系结构描述语言定义语法描述硬件架构，然后由编译器解析描述生成中间表示，再编译为可执行文件运行。文献[31]使用模拟器生成器，以编写的处理器规格自动生成微处理器模拟器。与上述工作相比，本工作将基于LLM的多智能体框架应用于DNN加速器模拟器的生成。

2 方法概述

本文提出一种面向DNN加速器模拟器自动生成的多智能体框架SimulatorGen。通过对模拟器架构进行抽象与规范提取，将整体模拟器进行模块分解，将模拟器生成问题转化为一组可验证、可组合的模块和组件级生成问题。同时，引入多智能体协作机制，在统一的抽象模型与规范约束下协同完成生成过程，从而提升生成结果的正确性与可靠性。

具体而言，本文的方法主要包括4个步骤。

1）对DNN加速器模拟器的通用架构进行系统性抽象，将模拟器划分为输入、计算、存储、互连和输出模块，在此基础上，对各模块组件的功能、输入输出接口和关键行为特征进行总结，进一步提取自然语言功能规范，为后续自动生成提供统一描述基础。

2）基于上述抽象架构，将生成具体和整体模拟器的问题转化为在抽象模型上生成对应模块和组件并组装的问题。给定模拟需求，如目标加速器架构、

数据流方式等,将需求映射到抽象模拟器架构中的模块上,从而明确需要生成的模块和组件集合及其功能要求。复杂的端到端模拟器生成过程被分解为模块级和组件级生成任务,最终通过组装构建完整模拟器。

3)为判断生成组件的正确性,本文基于抽象模拟器架构及其功能规范,利用 LLM 提取组件应满足的关键属性与约束条件,并据此构建相应的检查规则以提高生成结果的可靠性。

4)在上述模拟器架构抽象和模块组件划分的基础上引入 LLM,并设计多智能体协作的生成框架。通过对任务分析、代码生成、测试反馈与系统组装等职责进行角色分工,使多个智能体在统一的抽象模型与规范约束下协同工作,从而实现复杂 DNN 加速器模拟器的自动化生成。

3 模拟器架构抽象和模块组件划分

3.1 模拟器架构抽象

DNN 加速器模拟器在早期架构探索与设计优化中发挥关键作用,其通过准确模拟真实硬件的功能行为与性能特征来支持评估。与物理硬件实现不同,模拟器允许设计人员迭代和验证不同的架构配置,而无需承担实际制造的高昂成本。

基于对现有模拟器[6-10]的系统调研与分析,如图 2 所示,本文对 DNN 加速器模拟器的通用架构进行了抽象。如图 2 左图所示,模拟器的输入包括 DNN 模型,例如 ResNet50[32] 和 VGG16[33];加速器架构参数,用于实例化 DNN 模型中各算子的数值操作,以驱动加速器在功能与时序层面的硬件行为建模。如图 2 中间图所示,模拟器接收这些输入,将 DNN 模型映射到 PE 阵列上,并执行功能模拟、时序模拟以及功耗、性能与面积(power, performance, and area, PPA)估计。

功能模拟刻画加速器在执行乘加计算(multiply-accumulate, MAC)、数据路由与存储访问等操作时的行为;时序模拟提供关键架构,设计选择如何影响吞吐与延迟的周期级视角;PPA 估计则对能耗、性能与面积等关键指标进行量化评估。准确的 PPA 建模需要将微架构参数与组件级能耗模型及缓冲区延迟等相结合,例如根据存储读写事件、计算周期与片上数据移动来估算能耗。最终,模拟器输出包括 PPA、总体吞吐量等性能指标以及计算结果(图 2 右图),其中计算结果用于对功能模拟阶段的执行正确性进行验证以及指标计算。

模拟器需要对 DNN 加速器架构进行建模。图 3 中总结了一种通用的模块化建模框架。这种模块化建模方法包括可配置的分发网络、乘法器网络与归约网络,以及其关联的存储与互连子系统。每个网络由多个交换机组成,这些交换机可以通过各种拓扑结构互连形成不同类型的网络。通过组合不同网络,模拟器可灵活建模多种 DNN 加速器架构。图 3(d)与 3(e)分别展示了代表性架构 MAERI[34] 与 TPU[35]。

从 DNN 加速器架构与执行方式的角度出发,现有模拟器通常覆盖不同类型的计算组织方式,包括以乘加运算为核心的执行模型、基于脉动阵列的空间计算架构,以及面向稀疏计算优化的加速器设计。第 1 代 DNN 加速器通常针对单一主流工作负载进行高度定制,例如 TPU 采用脉动阵列微架构,并在设计阶段固化其数据流,以高效支持密集矩阵乘加运算。随着神经网络模型和工作负载特性的不断演进,下一代 DNN 加速器逐渐转向可重构架构,允许通过软件方式灵活调整数据流和计算映射策略,从而在执行周期数和能耗等方面获得更优的效率。MAERI 是一种由模块化、可配置构建块组成的 DNN 加速器架构。通过为乘法器和加法器配备小型开关,并通过新型可配置互连网络将它们连接起来,该互连网络支持任意尺寸的神经元,MAERI 能够轻松支持多种 DNN 分区和映射方式。

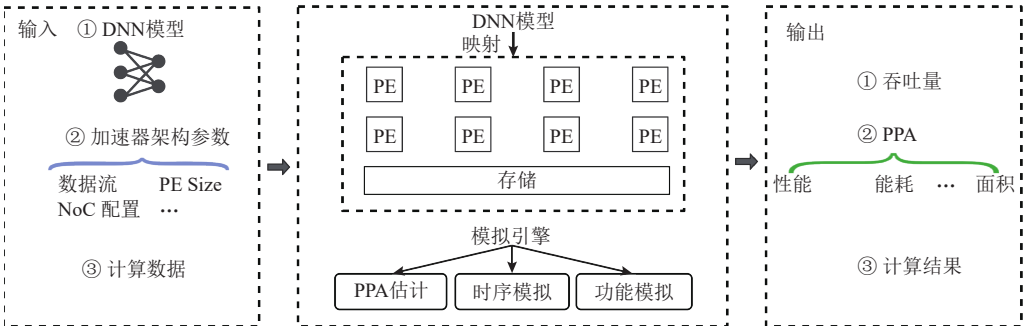


Fig. 2 Overview of DNN accelerator simulator

图 2 DNN 加速器模拟器概览

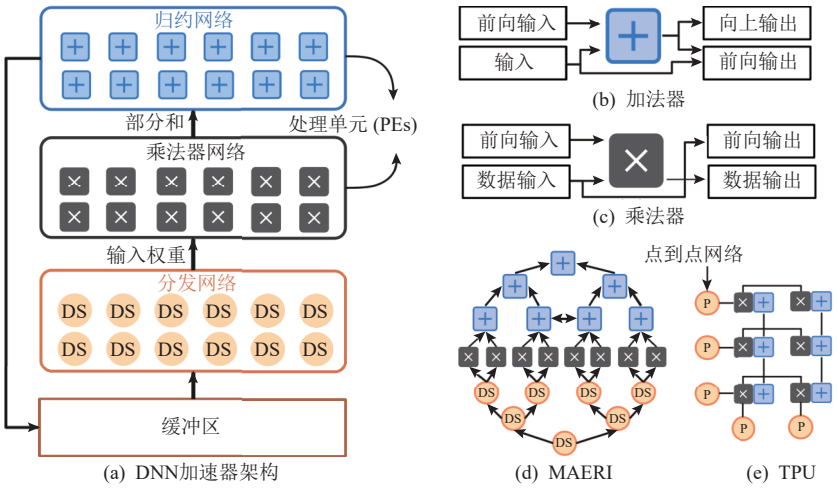


Fig. 3 Example for DNN accelerator simulation

图 3 DNN 加速器模拟实例

3.2 模拟器模块与组件划分

基于抽象的 DNN 加速器模拟器架构，本文对核心模块进行自顶向下的模块化分解，并为各类模块构建统一模板的自然语言功能规范。

具体而言，将模拟器核心功能划分为存储、计算与互连三大模块。其中，存储模块模拟本地与外部存储，同时负责读写和传输操作；计算模块模拟每个 PE 内的乘法与加法等运算；互连模块负责存储与计算模块之间的通信与同步。

在每个模块内部，进一步定义若干组件。组件是模拟器中可独立建模与生成的功能单元，例如缓冲结构、乘法单元、加法网络或路由节点等。每个模块通常包含多个组件，共同完成该模块所承担的功能职责。每个组件均通过结构化字段描述其输入输出接口、内部行为逻辑以及与性能与能量相关的语义约束，从而在抽象层面刻画其行为特征，为后续基于 LLM 的组件自动生成与组合提供了统一、可推理的功能描述基础。每条组件规范包含以下字段：

1）Component Name：组件名称。

2）Description：用自然语言概括组件的核心功能与使用场景，说明该组件在模拟器中承担的语义角色，例如作为可传输的数据单元、缓冲结构或算术执行单元等。

3）Inputs：说明组件的输入参数或初始化字段，给出名称与语义解释。

4）Outputs：说明组件对外提供的可见信息及其访问方式，明确哪些字段会被外部模块读取，以及这些信息在路由、调度或处理过程中的用途。

5）Functions：对外函数接口列表，并逐条描述每个函数的职责、触发条件与对状态或输出的影响。

6）Energy Model：当组件涉及能量评估时，在规范中给出其能量模型的参数项，说明能量与哪些事件或操作相关，例如读写、转发、乘加、累加等。

组件规范模板的设计原则是一方面保证规格描述足够明确以约束代码生成，另一方面避免引入过多实现细节以保持跨架构可迁移性。通过模板化规范，后续代码生成过程可以稳定地对齐组件接口与行为语义并支持多模块组装。如表 2 所示，本文提取了 23 条组件规范，给出了每条组件规范的名称、所属模块类别及其简要描述。

在模拟器架构抽象与规范提取过程中，在完成对模拟器结构的自顶向下分解基础上，进一步体现了该过程带来的可复用性与可参数化优势。首先，对模拟器进行架构抽象，提升了跨架构复用能力。本文将 DNN 加速器模拟器统一抽象为存储、互连与计算等功能模块及其交互接口，明确各模块的职责边界，使得同类模块能够在不同目标架构间复用，从而降低新架构适配时的重复开发成本。其次，组件规范提取可以形成可实例化的参数化模板。本文为每个模块组件提取结构化自然语言规范，其中关键字段以参数的形式保留，可根据待生成模拟器的具体需求填入不同取值，实现同一规范对不同实例的覆盖。例如，互连组件可通过拓扑类型、端口规模与路由模式等参数实例化；存储组件可通过容量和带宽等参数实例化；计算组件可通过阵列规模和数据流策略等参数实例化。

架构抽象与参数化规范共同将生成完整模拟器的复杂问题转化为在统一抽象上实例化并生成模块的问题，为后续引入多智能体进行模块生成、接口对接与系统组装奠定基础，增强了方法的扩展性。

Table 2 Overview of Module Component Specifications

表 2 模块组件规范概述

任务 ID	组件	模块	描述	任务 ID	组件	模块	描述
1	DataPackage	存储	可传输的数据单元	13	ASwitch	计算	部分和累加交换单元
2	Connection	互连	组件之间的互连结构	14	MultiplierOS	计算	乘加运算模块
3	FIFO	存储	数据包的 FIFO 缓冲区	15	TemporalRN	计算	时间维度归约网络
4	CollectionBusLine	互连	多输入数据仲裁	16	DSwitch	互连	二叉树结构的数据路由器
5	CollectionBus	互连	分层数据聚合	17	DSNetwork	互连	分层数据分发网络
6	Look-up Table	计算	基于查找表的激活操作	18	MSNetwork	计算	乘法与转发交换网络
7	Accumulator	计算	周期精确的部分和累加	19	ASNetwork	计算	树结构的部分和归约网络
8	AccumulationBuffer	存储	多累加器协同机制	20	DSNetworkTop	互连	分发控制器
9	route_1_1_plus_fw	互连	基于奇偶性的数据路由	21	FENetwork	计算	灵活的累加交换网络
10	route_2_1	互连	条件化的部分和归约	22	SDMemory	存储	片上 Scratchpad 存储器
11	FEASwitch	互连	灵活的累加与路由机制	23	OSMeshMN	互连	二维 Mesh 乘法器网络
12	MSwitch	计算	乘法与转发单元				

4 多智能体框架

设计模拟器需要解决若干关键挑战，例如实现准确的周期级模拟准确度、高效集成细粒度且可扩展的能量模型，以及在多样化 DNN 工作负载与加速器配置下确保正确性与灵活性。SimulatorGen 则通过相关 LLM 代码生成技术来针对性解决这些问题。

4.1 框架概述

SimulatorGen 是一个基于 LLM 的多智能体代码生成框架，旨在将自然语言形式表达的功能需求准确转换为 DNN 加速器模拟器代码。图 4 展示了 SimulatorGen 的核心框架，包含 4 个关键智能体：

1)分析智能体(Analyst Agent)。分析任务，基于 CoT 分解任务，同时基于检索增强构建领域提示词。

2)编码智能体(Coder Agent)。根据提示词生成代码或基于测试反馈修复并重新生成代码。

3)测试智能体(Tester Agent)。测试代码准确性并提供迭代改进反馈测试报告。

4)组装智能体(Assembly Agent)。进行组件代码组装，自动执行模拟并完成指标对比，实现完整模拟器的构建。智能体相互协作，整体流程如算法 1 所示。

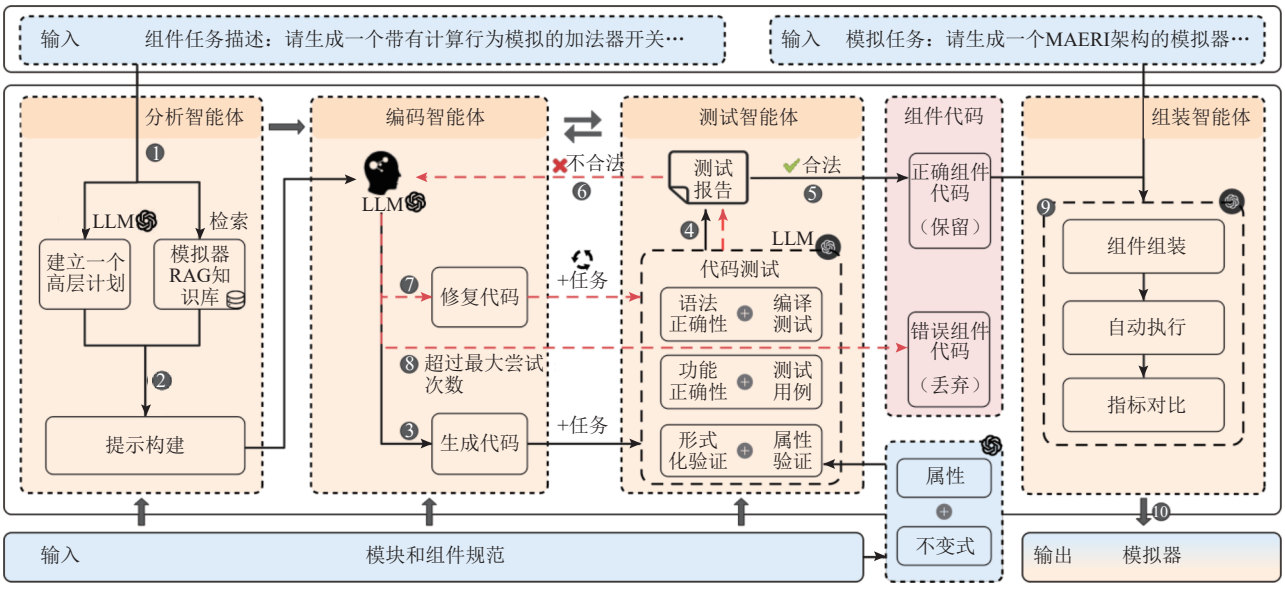


Fig. 4 Overview of SimulatorGen framework

图 4 SimulatorGen 框架概览

算法 1. 多智能体框架。

输入：组件生成任务 CT，整体模拟器生成任务 ST，模拟器 RAG 知识库 SD，任务属性和不变式 PI；

输出：完整模拟器 SC。

① 使用控制指令为 LLMs 初始化角色集合 $\{R_i\}_{i=1}^{4}$，初始化嵌入模型 EM；

② while 代码设计任务尚未完成 do

③ R_1(Analyst)：给定任务 ct_j，生成计划 $plan_j$；使用 EM 和点积从 SD 中检索相关数据 rag_j；并构造提示词 $prompt_j = ct_j + plan_j + rag_j$；

④ R_2(Coder)：生成代码 C_j；

⑤ R_3(Tester)：基于 PI_j 生成测试报告 TR_j；

⑥ while $attempts_1$ 小于最大尝试次数 do

⑦ while $attempts_2$ 小于最大尝试次数 do

⑧ if TR_j 的测试代码有效 then

⑨ 保存模拟器组件代码 C_j 到全局组件代码 C，保存报告到全局报告 TR，标记 ct_j 完成；

⑩ else

⑪ R_3: $TR_j \leftarrow NewTR_j$; /*更新测试报告*/

⑫ end if

⑬ end while

⑭ R_2: $C_j \leftarrow NewC_j$; /*更新编码智能体生成的代码*/

⑮ end while

⑯ end while

⑰ while $attempts_3$ 小于最大尝试次数 do

⑱ R_4(Assembly)：给定任务 ST，代码 C，生成整体模拟器 SC，并运行模拟器生成模拟器指标；

⑲ if 模拟器指标在预期内 then

⑳ 返回模拟器代码 SC；

㉑ else

㉒ continue;

㉓ end if

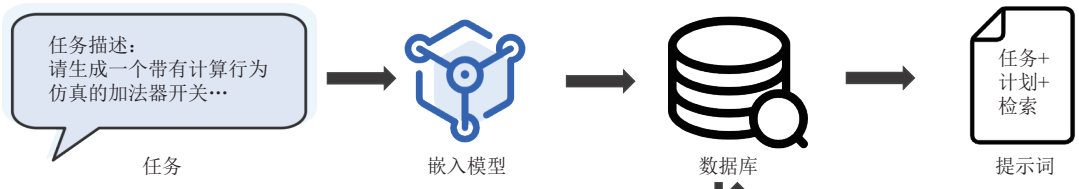
㉔ end while

算法 1 中，一个任务 CT 对应 1 个或多个代码设计任务 ct，即 $CT = \{ct_j | j = 1, \cdots\}$。

4.2 模拟器库

针对复杂的模拟器设计任务 T，LLM 往往需要获取领域知识。本文构建了一个模拟器 RAG 知识库并进行相似度检索，通过嵌入索引语料实现高效检索与排序，使编码智能体在生成代码前即可获得领域知识。模拟器库存储模拟器的基础组件信息，每个组件条目包括模块类型、功能描述、输入、输出以及代码。为保证评估公平性，模拟器库中的组件与评估任务中使用的组件不同。

模拟器库的使用流程如图 5 所示。首先，使用嵌入模型 EM 对模拟器库中的文本和代码进行编码，构建基于 FAISS 的向量库；然后，使用相同的嵌入模型 EM 将任务描述编码为查询向量，用于从库中进行相似度检索。本文选择 BGE-M3[36] 对模拟器库进行向量化处理，因为它在多粒度文本嵌入方面表现出色。



模拟库

ID	组件	模块类型	描述	输入	输出	编码
1	CollectionBus.cycle	Interconnect	Executes one simulation cycle of the interconnect bus module…	collection_ bus_lines...	CollectionBusLine sends data via output ports…	def cycle(self): …
2	ASwitch.perform_ operation_2_operands	Computing	Performs a 2-input operation on left and right packages based on…	pck_left, pck_right…	returns a new DataPackage with the sum…	def perform_operation_ 2_operands…
3	AccumulationBuffer. memory_connections	Storage	Assigns output memory connections to each accumulator instance…	memory_ connections…	assigns each accumulator its output connection …	def memory_connections …
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 5 Simulator library and prompt construction

图 5 模拟器库和提示构建

例如，对于一个需要生成具有计算行为模拟功能的加法器开关的任务，BGE-M3 会从库中检索出 2 个语义最相似的组件(例如图 5 中箭头所指的 ID=2 的输入加法器开关的组件)，并将其信息整合到生成的提示词中。此外，模拟器库可以通过添加新的通过验证的设计进行不断更新，使系统能够获取最新的领域知识。

4.3 分析智能体

分析智能体旨在通过 CoT 与 RAG 提升 LLM 对任务的理解能力，从而降低任务分析复杂度。如图 4 的步骤①~②所示，分析智能体接收组件任务描述ct_j，利用结构化 CoT 提示将任务分解为若干子任务，形成高层次计划$plan_j$，并同时通过 RAG 从模拟器库中检索相关信息rag_j(步骤①)；随后，提示词构建器整合任务、子任务、计划以及从模拟器库中检索到的信息，构建全面且富含上下文的提示词$prompt_j$(步骤②)，以指导编码器智能体生成代码。此外，提示词构建器还会标准化用户输入格式，并传达模拟器的预期功能、设计意图和架构规范。提示词如图 6 中分析智能体部分所示。

1)分析智能体

你是一名DNN 加速器模拟器专家。以下是一段任务描述：{*task_description*}。分析指令：① 让我们一步步思考。将需求拆解为若干个更容易解决的子问题，使编码员（Coder）能够更容易实现；② 制定一个高层计划，概述程序实现的主要步骤；③ 注意：该计划应保持高层次，重点在于指导编码员，而不是提供具体实现细节。请使用编号列表等方式，将你的分析写得清晰明了。

2)编码智能体

你是一名经验丰富的DNN 加速器模拟器设计专家和Python 语言专家。请以专业风格回答以下问题：基于你此前撰写的内容（以及我已发送给你的材料），请编写一个Python 文件来实现这些功能。代码只需输出1次，且只能输出1次。请注意，你所编写并发送的代码文件与测试文件位于同一目录中。
上下文代码：{*context_code*}
任务描述：{*task_description*}

3)测试智能体

我希望你在团队中充当Tester 的角色。你将接收由Coder 编写的代码，你的职责包括：① 功能完整性测试：基于需求和代码编写测试用例，用于验证设计是否完整实现了预期功能，不存在功能缺失、冗余代码或占位实现等问题；② 形式化验证：编写验证用例，将Coder 生成的代码转换为符号化路径约束，并基于已发送给你的属性和不变式，使用SMT 求解器Z3 对代码正确性进行验证。
输出要求：仅输出以下2个相互独立的代码块：
python test # your functional test code here
python z3 # your Z3 verification code here

4)组装智能体

请基于这些组件文件，搭建一个Maeri架构的模拟器。请一步步地解决这个问题。模拟器的输入是DNN模型（例如Resnet50.csv）和计算数据，将网络结构表示为卷积（CONV）操作。模拟器的输出是模拟CONV操作映射到PE上运行计算过程所需的周期数和能耗值以及计算结果。片上网络（NoC）包括增强型归约树网络、线性乘法器网络和Benes分布网络。全局缓冲区配置为108 KB，片外存储器由2个512 MB HBM2 DRAM模块组成，每个模块的带宽为256 GB/s。可以分模块逐步构建。

Fig. 6 Prompt design

图 6 提示词设计

4.4 编码智能体

编码智能体在分析智能体和测试智能体的协助与监督下负责编写正确的代码C_j。如图 4 中的步骤③和步骤⑦所示，其代码生成过程包含 2 个关键阶段，以迭代循环的方式执行：1)基于分析智能体提供的提示词，生成满足指定功能需求的代码(步骤③)；2)整合测试智能体报告中的错误反馈，对代码进行优化，修正错误并提升质量(步骤⑦)。LLM 的任务是生成 Python 代码，这得益于其在 Python 编程方面已被证实的有效性[37]。编码器能优先生成可读性强、模块化且易于维护的代码，这类代码需符合模拟器架构要求，并确保功能正确性。提示词如图 6 中编码智能体部分所示。编码智能体在每次生成代码时，其上下文包括任务规范(如 Connection 组件生成规范，包含基于 CoT 生成的任务计划，以及 RAG 检索内容)、依赖组件的源码上下文(DataPackage.py)以及最近一轮测试与形式化验证产生的错误日志，重试时只更新错误日志。

4.5 测试智能体

LLM 生成的代码通常包含各类错误。基于相关研究，通过错误反馈引导 LLM 修正这些问题至关重要。测试智能体负责测试代码的正确性并生成测试报告TR_j，具体流程如图 4 中的步骤④~⑧所示。提示词如图 6 中测试智能体部分所示。

1)代码评估。测试智能体通过多阶段过程验证生成代码的正确性(步骤④)。如图 7 所示：①语法正确性检查。通过调用 Python 内置的 *compile*() 函数对生成代码进行编译级检查，以验证其语法正确性。②功能正确性测试。基于组件输入输出规范，调用 LLM 自动生成专门用于功能验证的测试用例，以检

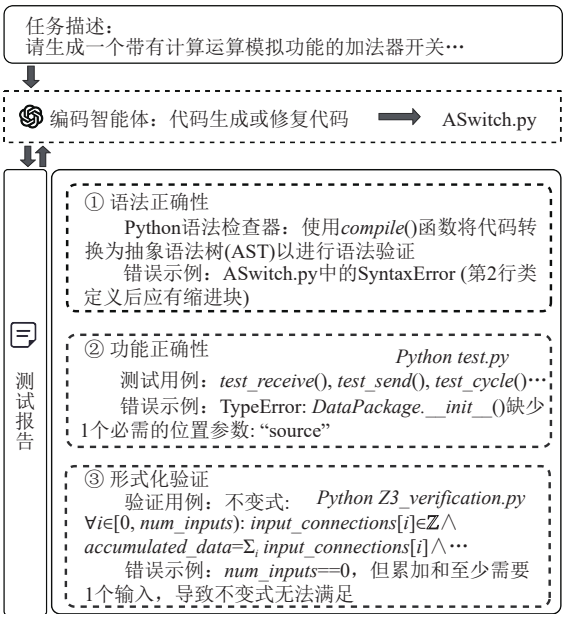


Fig. 7 Code testing stages

图 7 代码测试阶段

查是否满足任务需求及功能模拟是否完整。③形式化验证。首先调用 LLM 从任务级自然语言功能规范中自动提取属性约束与不变式条件，平均每个任务可生成约 5 条属性约束和 10 条不变式，提示词是：“你是一位形式化验证专家，从规范中提取需求和不变量”。然后，利用 LLM 将编码智能体生成的代码语义转换为符号路径约束，并将这些约束与属性、不变式一同提交至 Z3 求解器[38]进行验证，从而提高代码正确性。

测试报告 TR_j 记录语法错误信息、功能测试失败案例、违反的属性或不变式条件以及形式化验证结果等内容。值得注意的是，任务规范说明还包含对时序模拟和 PPA 估计的要求及参考数据，这对于确保生成模拟器的行为和性能准确性至关重要。测试智能体会生成测试代码，生成成功测试代码的最大尝试次数为 3，即算法 1 中的 $attempts_2 \leqslant 3$。

2)反馈自修复。如果评估结果有效(步骤⑤)，则将生成的正确代码作为输出。否则(步骤⑥)，LLM 接收测试反馈以优化并重新生成代码(步骤⑦)。如果 LLM 的代码生成尝试次数超过允许的最大值(步骤⑧)，则终止生成过程，并丢弃错误代码。在本文的实验中，该最大值设定为 3。即算法 1 中的 $attempts_1 \leqslant 3$。

4.6 组装智能体

在完成各组件代码生成与验证后，系统仍需将多个独立组件整合为可运行的完整模拟器。本文设计了组装智能体，负责组件组装、自动执行与指标比对，从而实现完整模拟器的自动构建与验证。提示词如图 6 中组装智能体部分所示。

如图 4 中的步骤⑨所示，组装智能体负责：1)组件组装。组装智能体接收模拟任务 ST、由编码智能体生成的组件代码文件 C，调用 LLM 生成完整的模拟器组装代码，包括主入口函数、模块实例化逻辑以及跨组件接口对接代码，实现从离散组件到完整系统结构的转换。2)自动执行。组装智能体将生成的组装代码与组件文件整合为完整工程结构，并自动调用运行脚本执行模拟任务。该过程支持不同网络模型与不同参数配置的批量测试，实现端到端自动模拟。3)指标比对。组装智能体自动收集模拟器输出的关键性能指标(如周期数、能量与计算结果等)，并与预期基准结果进行对比分析。当各项指标在预设误差内时，判定系统组装成功；否则将偏差信息反馈进行修正。本文将生成组装代码的最大尝试次数设为 20，即算法 1 中的 $attempts_3 \leqslant 20$。最后，由组装智能体输出任务目标模拟器 SC(步骤⑩)。

通过组件组装、自动执行、指标比对的闭环流程，组装智能体不仅验证组件接口的一致性，还确保整体架构在功能语义与性能指标上的合理性，使 SimulatorGen 具备从模块生成到系统级验证的完整自动化能力。

5 实 验

5.1 实验设计

本文通过解决以下研究问题来探索 SimulatorGen 生成 DNN 加速器模拟器代码的能力。

RQ1：SimulatorGen 框架在生成正确且可执行的 DNN 加速器模拟器代码方面的准确性和通用性如何？

RQ2：提示词设计与测试反馈策略如何影响为 DNN 加速器生成的模拟器代码的准确性？

RQ3：基于 SimulatorGen 构建的模拟器性能与人工实现相比如何？

RQ4：与单智能体 SimulatorCoder 相比，SimulatorGen 是否支持端到端系统构建能力？

为了研究 RQ1，本文设计了一个自定义基线，在 4 个广泛使用且具有不同参数量和能力的 LLM 上评估 SimulatorGen，指标包括在重复独立试验中的单元测试通过率和端到端任务成功率和运行时间。对于 RQ2，运行提示设计和测试反馈策略的控制变量实验，并通过代码生成结果分析准确率。对于 RQ3，本文将 SimulatorGen 生成的模拟器与手动实现的模拟器进

行比较，比较不同工作负载上的能量(energy)和延迟(latency)以及它们的乘积(EDP)指标。对于RQ4，基于SimulatorCoder的需求，调用SimulatorGen的多智能体框架，查看SimulatorGen是否可以生成整体的和SCALE-Sim等价的完整模拟器。

5.2 评估指标

遵循先前的代码生成研究[39]，使用Pass@k作为评估指标。具体来说，给定一个需求，允许LLM生成k个程序。如果任何生成的程序通过了所有测试用例，则认为该需求已得到解决。本文进行n次试验，并计算$\text{Pass@}k = 1 - \binom{n-c}{k} / \binom{n}{k}$，其中$c$是成功的试验次数。在本文中，$n=20$，$k \in \{1,2,\cdots,5\}$。成功任务数报告有多少个独立任务在至少1次试验中获得了成功的模拟器组件设计。

5.3 基线LLMs

本文对LLM在DNN加速器模拟器代码生成方面进行了全面评估。SimulatorGen框架的分析、编码和测试智能体基于GPT-4o[40]构建，进行组件生成，组装智能体则基于GPT-5.2[41]构建，由于其在代码理解与生成任务中的出色表现。为评估该方法的相对有效性，选取4种被广泛认可的LLM作为对比基线：Qwen3-32B[42]、Gemini-2.5-Pro[43]、Claude-Sonnet-4-5-20250929[44]和DeepSeek-V3[45]。其中，Qwen3与Claude是具备较强推理能力的指令遵循模型；Gemini是由Google公司开发的闭源LLM；而DeepSeek-V3在代码合成基准测试中已表现出良好的性能水平。本文通过API访问这些模型，包括：Qwen3-Omni-30B-A3B-Instruct、Gemini-2.5-Pro、Claude-Sonnet-4-20250514、GPT-4o、GPT-5.2和DeepSeek-V3。本文对GPT-4o测试了2种温度设置(0.2与1.0)，并基于3个代表性的模拟器生成任务(任务1、14和17)的Pass@1表现进行评估。结果表明，温度设置为0.2时可获得最高的平均Pass@1分数，在最终实验中采用该设置。所有其他LLM的参数均保持为其默认值。提示词如图6中基线LLMs部分所示。

5.4 基准任务

为支持上述评估，基于提取的模拟器的自然语言功能规范，构建了一个同样包含23个任务的综合测试基准以支持TPU和MAERI架构。如表2所示，23个任务对应23个组件的生成，任务难度依据功能复杂性及依赖关系被划分为3个等级：低难度(ID 1~8)、中难度(ID 9~15)和高难度(ID 16~23)。高难度任务的生成代码行数更多，功能更复杂，同时难度等级高的任务依赖于等级低的任务，如ID为18的任务依赖于ID为1、2、3和12的任务。每个组件均包含组件名称、功能描述、输入、输出、函数定义、能量模型(如果存在)以及测试用例。为确保评估的公平性，任务基准中所使用的组件与模拟器库使用的组件是相互独立的。

5.5 实现细节

模拟器库构建自开源的、周期精确的模拟器[6,8]。具体而言，将这些模拟器拆分为互连、存储和计算3类模块，从每个模块中提取关键代码，并将其人工提取为功能描述及其对应的Python实现，如第4.2节所示。最终得到的模拟器库包含23个条目，总计约6 097个英文单词和891行代码。实验结果显示，在算法1中将$attempts_1$和$attempts_2$的取值设置为3或5，对SimulatorGen的整体Pass@k性能影响很小，因此，最终将二者均设定为3。实验配置为Intel® CPU、32 GB内存，并运行64位Windows操作系统。实验环境为Python3.12.4，主要依赖requests2.32.3、sentence-transformers4.1.0、faiss-cpu1.11.0和z3-solver4.15.1.0。

6 评估结果

6.1 RQ1

本文将SimulatorGen框架分别部署于多种基线LLMs之上，以评估其在不同模型下的通用性。同时，本文还评估了在去除分析与测试反馈策略后，仅使用GPT-4o模型的性能表现。表3展示了SimulatorGen与所选基线模型在Pass@k和成功任务数指标上的对比结果。实验结果表明，SimulatorGen能够适配不同LLMs，并在多种模型上生成可执行的模拟器代码，体现出一定的跨模型适配能力。基于GPT-4o构建的SimulatorGen在整体代码生成准确率方面取得最佳性能，在Pass@1和Pass@5指标上相较于单独使用GPT-4o分别实现了约49.20%和29.96%的相对提升，成功解决的任务数量增加了5个。此外，不同基线模型之间在模拟器设计能力上存在差异。例如，Claude-Sonnet-4整体表现优于Gemini-2.5-Pro等模型，但与GPT-4o相比仍存在一定差距。

针对RQ1的回答：SimulatorGen不仅能够生成在功能上正确的模拟器代码，而且在整体性能上优于4种基线LLM，成功解决的任务数量最多，平均运行时间最短。尽管不同基线模型之间存在性能差异，SimulatorGen在多种模型上均能够生成可执行的模拟器代码，并在部分模型(如GPT-4o)上显著提升性

Table 3 Performance of SimulatorGen Framework Across Different Large Language Models

表 3 SimulatorGen 框架在不同大语言模型上的性能表现 %

任务 ID	Gemini-2.5-Pro		DeepSeek-V3		Qwen3-32B		Claude-Sonnet-4		GPT-4o*		SimulatorGen	
	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
1	15.00	60.09	95.00	100.00	85.00	100.00	100.00	100.00	100.00	100.00	**100.00**	**100.00**
2	95.00	100.00	50.00	98.37	95.00	100.00	65.00	99.86	100.00	100.00	**100.00**	**100.00**
3	60.00	99.64	40.00	94.89	45.00	97.02	40.00	94.89	90.00	100.00	**100.00**	**100.00**
4	30.00	87.09	70.00	99.96	60.00	99.64	35.00	91.70	95.00	100.00	**95.00**	**100.00**
5	35.00	91.70	40.00	94.89	50.00	98.37	50.00	98.37	0.00	0.00	**100.00**	**100.00**
6	60.00	99.64	25.00	80.63	70.00	99.96	55.00	99.19	100.00	100.00	**100.00**	**100.00**
7	60.00	99.64	30.00	87.09	30.00	87.09	50.00	98.37	65.00	99.86	**85.00**	**100.00**
8	85.00	100.00	90.00	100.00	80.00	100.00	85.00	100.00	80.00	100.00	**100.00**	**100.00**
9	80.00	100.00	25.00	80.63	95.00	100.00	80.00	100.00	85.00	100.00	**100.00**	**100.00**
10	50.00	98.37	80.00	100.00	70.00	99.96	70.00	99.96	75.00	99.99	**100.00**	**100.00**
11	50.00	98.37	25.00	80.63	75.00	99.99	40.00	94.89	70.00	99.96	**100.00**	**100.00**
12	60.00	99.64	0.00	0.00	60.00	99.64	90.00	100.00	70.00	99.96	**90.00**	**100.00**
13	45.00	97.02	35.00	91.70	80.00	100.00	100.00	100.00	65.00	99.86	**100.00**	**100.00**
14	40.00	94.89	40.00	94.89	55.00	99.19	85.00	100.00	70.00	99.96	**80.00**	**100.00**
15	60.00	99.64	100.00	100.00	0.00	0.00	80.00	100.00	50.00	98.37	**80.00**	**100.00**
16	50.00	98.37	60.00	99.64	50.00	98.37	90.00	100.00	0.00	0.00	**90.00**	**100.00**
17	65.00	99.86	20.00	71.83	45.00	97.02	75.00	99.99	60.00	99.64	**85.00**	**100.00**
18	0.00	0.00	70.00	99.96	60.00	99.64	80.00	100.00	35.00	91.70	**75.00**	**99.99**
19	30.00	87.09	50.00	98.37	30.00	87.09	75.00	99.99	45.00	97.02	**55.00**	**99.19**
20	30.00	87.09	30.00	87.09	40.00	94.89	45.00	97.02	0.00	0.00	**55.00**	**99.19**
21	0.00	0.00	20.00	71.83	25.00	80.63	35.00	91.70	15.00	60.09	**45.00**	**97.02**
22	20.00	71.83	40.00	94.89	25.00	80.63	0.00	0.00	0.00	0.00	**30.00**	**87.09**
23	0.00	0.00	0.00	0.00	0.00	0.00	30.00	87.09	0.00	0.00	**30.00**	**87.09**
Avg	44.35	81.30	45.00	83.80	53.26	87.79	63.26	93.61	55.22	75.93	**82.39**	**98.68**
成功任务数	20		21		21		22		18		**23**	
Time	3.72		2.91		5.1		2.55		2.90		**2.01**	

注：GPT-4o*表示单独使用的 GPT-4o 模型，其余模型均结合 SimulatorGen 框架进行评估，*Time*表示对 *n* 次实验中成功生成的 23 个组件的平均运行时间之和，单位为小时。黑体数值表示最优值。

能，体现出一定的通用性与适配能力。

6.2 RQ2

为了探究提示词设计与反馈策略对代码生成准确性的影响，本文开展了消融实验，本文评估了 4 种消融设置，其结果如表 4 所示。

1）SimulatorGen w/o Tester。为评估 Tester 模块的贡献，在 SimulatorGen 中移除了测试反馈模块。如表 4 所示，代码生成准确率出现显著下降，其中Pass@1下降了 13.26 个百分点，Pass@5下降了 12.37 个百分点，该模块的作用在高难度任务上更为显著。这表明，迭代式测试反馈机制能够有效提升 LLM 的代码生成准确性。

2）SimulatorGen w/o Analyst。移除 Analyst 模块，即禁用基于 RAG 的模拟器组件库以及任务分解功能后，性能相较于 SimulatorGen 明显下降：Pass@1下降 10.43 个百分点，Pass@5下降 2.20 个百分点，Analyst 模块对高难度任务的影响更为显著。这一结果显示结构化 CoT 提示和 RAG 策略的重要性。

3）SimulatorGen w/o RAG。在保持其他模块不变的情况下，仅移除 RAG 检索机制，会导致性能相对于 SimulatorGen 下降：Pass@1 降低 8.04 个百分点，Pass@5 降低 1.04 个百分点，同时 RAG 模块对中高难度任务的影响更为显著。这表明基于 RAG 的检索在提升代码 Pass@1 方面有明显优势。

Table 4 Ablation Study of SimulatorGen

表 4 SimulatorGen 的消融实验 %

模型	低难度		中难度		高难度		平均	
	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
GPT-4o	78.75	87.48	69.29	99.73	19.38	43.56	55.22	75.93
SimulatorGen w/o Tester	93.13	100.00	73.57	85.60	41.25	73.25	69.13	86.31
SimulatorGen w/o Analyst	96.88	100.00	76.43	97.12	43.13	92.39	71.96	96.48
SimulatorGen w/o RAG	91.25	100.00	85.71	100.00	47.50	93.23	74.35	97.64
SimulatorGen	97.50	100.00	92.86	100.00	58.13	96.20	82.39	98.68
相对提升/%	23.81	14.31	34.02	0.27	199.95	120.84	49.20	29.96

注：最后一行报告 SimulatorGen 相对于 GPT-4o 在各难度级别上的提升百分点。“w/o”表示变量不包含指定模块。

4)GPT-4o。在该实验中，移除了上述所有组件，仅评估 GPT-4o 的原始性能。如表 4 所示，其性能相较于 SimulatorGen 大幅下降：Pass@1 下降 27.17 个百分点，Pass@5 下降 22.75 个百分点，同时该性能差距在高难度任务中最为显著。这些结果进一步强调了各组件在提升代码准确性方面的重要作用。

RQ2 的回答：实验结果表明，将结构化的 CoT 提示、基于模拟器库的 RAG 策略以及迭代式测试反馈机制相结合，在提升 SimulatorGen 生成代码的准确性方面发挥了关键作用。随着任务复杂度的提升，各模块的作用愈发明显。在低难度任务中，基础模型已具备一定的生成能力，各模块带来的增益相对有限；而在中/高难度任务中，结构化提示、外部知识支持和迭代反馈机制共同作用，提升了代码生成的成功率。

6.3 RQ3

实验表明，直接以端到端方式要求 LLM 生成完全可运行的模拟器具有较高难度。在此基础上，本文利用 SimulatorGen 成功生成各个组件，构建组装智能体，引导模型对组件进行迭代式互连，在组装过程中逐步解决接口不匹配与集成问题，最终成功构建出完整可运行的 TPU 和 MAERI 架构的模拟器。其中，TPU 架构模拟器代码规模为 2 385 行，MAERI 架构模拟器代码规模为 6 207 行。

在 TPU 架构模拟器中，采用了 weight-stationary 数据流，并配置了一个 16×16 的 PE 阵列。片上网络(NoC)由点到点分发网络、线性乘法器网络和线性归约网络组成。在 MAERI 架构模拟器中，PE 配置为 256 个并行计算单元，由 256 个乘法器与 256 个加法器组成，片上网络由增强型归约树网络、线性乘法器网络以及采用 Benes 拓扑的分发网络组成。全局缓冲区配置为 108 KB，片外存储由 2 个 512 MB 的 HBM2 DRAM 模块构成，每个模块的带宽为 256 GB/s。

此外，为每个组件分配了能量与面积模型，以支持精确的 PPA 评估。鉴于模拟器的面积主要由全局缓冲区的 SRAM 结构所主导，将模拟器的面积固定为 2.4×10^6 μm^2。随后，在 4 个具有代表性的模型上，对 SimulatorGen 构建的模拟器与 STONNE 进行了延迟(latency)、能量(energy)以及能量-延迟乘积(EDP)的对比分析，这 4 个模型分别为 AlexNet[46]、MobileNetV1[47]、ResNet50[32] 和 VGG16[33]。

如表 5 所示，在基于 TPU 和 MAERI 架构分别执行 4 层神经网络模拟时，SimulatorGen 生成的模拟器与基于 RTL 的参考实现[6] 在执行周期数上的误差在 7% 以内，整体上表现出较高的一致性。其中 M、N、K 值表示矩阵乘(GEMM)的维度参数[48]。

在延迟方面，如表 6 和表 7 所示，SimulatorGen 构建的模拟器与 STONNE 的结果高度一致，TPU 架构和 MAERI 架构模拟器误差在 4% 以内，表明其具有较强的准确性与稳定性。在能量方面，如表 8 和表 9 所示，SimulatorGen 在 4 个模型上的能量结果均略低

Table 5 Latency and Its Errors Comparison Between RTL and SimulatorGen on TPU and MAERI Layers

表 5 比较 RTL 和 SimulatorGen 在 TPU 和 MAERI 层上的模拟延迟与其误差

层名称	M	N	K	延迟/cycle		误差/%
				RTL	SimulatorGen	
TPU1	16	16	32	66	68	3.03
TPU2	16	16	16	50	52	4.00
TPU3	32	32	16	200	208	4.00
TPU4	64	64	32	1 056	1 008	4.55
MAERI1	6	25	54	1 338	1 356	1.35
MAERI2	20	25	180	16 120	15 006	6.91
MAERI3	6	400	54	26 178	25 923	0.97

于 STONNE, TPU 架构模拟器误差为 1%~3%, MAERI 架构模拟器误差在 1%~6%, 说明在保持延迟精度的同时, 其能量估计与参考结果仍然较为接近。

Table 6 Latency and Its Errors Comparison Between SimulatorGen and STONNE on TPU Architecture

表 6 SimulatorGen 与 STONNE 在 TPU 架构上的延迟与其误差对比

DNN 模型	延迟/cycle		误差/%
	SimulatorGen	STONNE	
AlexNet	6.23E+06	6.43E+06	3.18
MobileNetV1	6.15E+06	6.23E+06	1.32
ResNet50	1.75E+07	1.79E+07	2.26
VGG16	6.74E+07	6.99E+07	3.50

Table 7 Latency and Its Errors Comparison Between SimulatorGen and STONNE on MAERI Architecture

表 7 SimulatorGen 与 STONNE 在 MAERI 架构上的延迟与其误差对比

DNN 模型	延迟/cycle		误差/%
	SimulatorGen	STONNE	
AlexNet	3.54E+06	3.69E+06	3.89
MobileNetV1	2.73E+06	2.70E+06	1.09
ResNet50	1.59E+07	1.64E+07	3.14
VGG16	6.14E+07	6.12E+07	0.30

Table 8 Energy and Its Errors Comparison Between SimulatorGen and STONNE on TPU Architecture

表 8 SimulatorGen 与 STONNE 在 TPU 架构上的能量与其误差对比

DNN 模型	能量/PJ		误差/%
	SimulatorGen	STONNE	
AlexNet	102.49	104.11	1.56
MobileNetV1	77.34	79.24	2.40
ResNet50	507.22	514.24	1.37
VGG16	2 022.81	2 064.09	2.00

如表 10 所示, 当将能量与延迟结合为 EDP, SimulatorGen 在不同模型上的 EDP 结果均低于 STONNE, 相对误差范围为 1.31%~7.34%。总体而言, SimulatorGen 构建的模拟器在延迟和能量 2 个维度上均与 STONNE 保持高度一致, 在 EDP 上接近, 体现良好的跨模型泛化和准确的评估能力。

在系统级组装过程中, 组装智能体最大尝试次数设为 20, 实际平均尝试次数约为 16。本文对组装失败的原因进行了分类, 集成的主要挑战来源于跨组件交互问题, 包括接口不匹配、数据格式不一致、执行流程错误以及性能指标汇总不完整等。其中, 接口不匹配主要表现为函数签名不一致、字段命名差异, 以及时序模型不统一等情况。这些问题通常需要通过多轮迭代对组件接口与执行逻辑进行对齐与修正。

Table 9 Energy and Its Errors Comparison Between SimulatorGen and STONNE on MAERI Architecture

表 9 SimulatorGen 与 STONNE 在 MAERI 架构上的能量与其误差对比

DNN 模型	能量/PJ		误差/%
	SimulatorGen	STONNE	
AlexNet	112.69	116.88	3.58
MobileNetV1	112.75	119.81	5.89
ResNet50	788.73	797.65	1.12
VGG16	3 142.70	3 194.09	1.61

Table 10 Comparison of SimulatorGen and STONNE for Simulation EDP and Error Across Different DNN models

表 10 SimulatorGen 与 STONNE 在不同 DNN 模型上的仿真 EDP 及其误差对比

DNN 模型	EDP/（PJ·cycle）				误差/%	
	SimulatorGen		STONNE			
	TPU	MAERI	TPU	MAERI	TPU	MAERI
AlexNet	6.38E+8	3.99E+8	6.70E+8	4.31E+8	4.69	7.34
MobileNetV1	4.75E+8	3.07E+8	4.94E+8	3.23E+8	3.68	4.87
ResNet50	8.86E+9	1.26E+10	9.19E+9	1.31E+10	3.60	4.23
VGG16	1.36E+11	1.93E+11	1.44E+11	1.95E+11	5.43	1.31

RQ3 的回答：由 SimulatorGen 生成的 TPU 和 MAERI 架构模拟器在精度上接近人工实现：在 AlexNet、MobileNetV1、ResNet50 和 VGG16 四个模型上, 其延迟误差控制在 4% 以内, 能量误差在 6% 以内, EDP 相对误差为 1.31%~7.34%。结果表明 SimulatorGen 生成的模拟器具备准确建模真实加速器行为的潜力。

6.4 RQ4

首先以自然语言方式对 SCALE-Sim[8] 的模块连接关系进行系统性抽象与总结, 同时将 SimulatorCoder[17] 中归纳的 138 个函数级需求以及 DNN 加速器架构规格说明作为 SimulatorGen 的任务。

实验结果表明, SimulatorGen 能够自动构建与

SCALE-Sim 在功能与时序行为上等价的完整模拟器。如表 11 所示，在 SimulatorCoder 原实验所采用的 ResNet50、Faster R-CNN、Transformer 等 8 个代表性模型上，SimulatorGen 与 SCALE-Sim 的延迟误差均为 0%；而 SimulatorCoder 与 SCALE-Sim 的延迟误差为 0~1%。该结果表明，多智能体框架在系统级一致性与周期级精度方面具备更强的稳定性。

Table 11 Latency and Error Comparison Between SimulatorGen and SimulatorCoder

表 11 SimulatorGen 与 SimulatorCoder 的延迟与误差对比

DNN 模型	延迟/cycle		误差/%
	SimulatorCoder	SimulatorGen	
NCF	552 624	552 616	<0.01
AlphaGoZero	416 502	416 494	<0.01
MobileNet	1 333 481	1 338 519	0.38
DeepSpeech2	2 198 754	2 200 287	0.07
Faster R-CNN	4 329 971	4 367 574	0.86
YOLOv2	2 556 983	2 556 974	<0.01
ResNet50	4 396 573	4 434 168	0.85
Transformer	4 871 474	4 870 117	0.03

RQ4 的回答：单智能体方法 SimulatorCoder 主要面向模块级替换，而多智能体框架 SimulatorGen 能够在统一抽象架构下完成系统级组装，具备完整模拟器的自动生成能力。从定量结果来看，SimulatorGen 的延迟误差低于 SimulatorCoder，进一步验证了多智能体协同机制在处理复杂系统构建任务时的优势。

7 讨 论

1）通用性。本文的框架在一定程度上具有通用性，其通过抽象系统架构、模块组件划分和提取规范功能描述，并借助多智能体协同完成组件生成与系统组装。基于该方法论，可进一步扩展至更广泛的应用场景，包括支持多种异构加速器架构（如 Chiplet 架构、SIGMA 等灵活可重构架构）以及更加多样化的算子类型（如稀疏矩阵乘、非规则计算等），从而在多架构、多算子条件下验证其有效性与泛化能力。本文方法可扩展至其他领域的特定开发任务，但其可迁移性的实际程度最终取决于开发者在实现层面的选择。

2）有效性威胁。实验同样面临潜在的有效性威胁。首先，LLM 的输出具有随机性，并且对检索到的知识以及提示词较为敏感。为缓解这一问题，对每个任务进行 20 次独立生成，并采用部分固定的提示词模板以降低方差。其次，LLM 的能力发展迅速，本文报告的结果仅反映了特定时间点下模型的性能。同时，对生成代码进行形式化验证仍然具有较大挑战性。验证机制主要用于提供反馈，从而提升代码正确性，并辅以对 LLM 生成代码的人工审查作为保障。本文工作鼓励未来工作更加关注生成程序的语义正确性，并探索更系统、更全面的形式化验证方法。

8 结 论

本文提出了 SimulatorGen，一种面向 DNN 加速器模拟器开发的多智能体、LLM 驱动框架。SimulatorGen 协同分析、编码、测试和组装 4 类智能体，并利用精心构建的模拟器库实现了检索增强的知识对齐、领域引导的 CoT 提示构建。结合单元测试与 Z3 属性检查的反馈驱动测试闭环以及面向系统级模拟执行的自动组件组装，在保持工程可行性的同时，持续提升生成代码的语义正确性。在包含 23 个生成任务的基准评测中，SimulatorGen 取得了 Pass@1 = 82.39% 的成绩，成功解决的任务数量最多，SimulatorGen 构建了 TPU 架构和 MAERI 架构的模拟器，在 4 个网络上的实验结果表明，其周期级行为、能量和延迟指标与人工构建的实现高度一致。相比于进行模块替换的单智能体方法 SimulatorCoder，SimulatorGen 具备完整模拟器的自动生成能力。

作者贡献声明：夏雨欢负责方案整体设计并撰写论文；李曦提出指导意见和整体思路；周贤发、赵文博负责部分实验；张瑞瑜修改论文；郭阳提出指导意见。

参 考 文 献

[1] Wang Haozhao, Qu Zhihao, Zhou Qihua, et al. A comprehensive survey on training acceleration for large machine learning models in IoT[J]. IEEE Internet of Things Journal, 2022, 9(2): 939−963

[2] Genc H N, Kim H, Ganesh P, et al. Stellar: An automated design framework for dense and sparse spatial accelerators[C]//Proc of the 57th IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2024: 409−422

[3] Sun Mijing, Xu Li, Li Zhenmin, et al. Simulators for deep neural network accelerator design and analysis: A brief review[C]//Proc of the 15th IEEE Int Conf on Anti-counterfeiting, Security, and Identification. Piscataway, NJ: IEEE, 2021: 36−42

[4] Wu Y N, Emer J S, Sze V. Accelergy: An architecture-level energy

estimation methodology for accelerator designs[C]//Proc of the IEEE/ACM Int Conf on Computer-Aided Design. Piscataway, NJ: IEEE, 2019: 1−8

[5] Parashar A, Raina P, Shao Y S, et al. Timeloop: A systematic approach to DNN accelerator evaluation[C]//Proc of the IEEE Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2019: 304−315

[6] Muñoz-Martínez F, Abellán J L, Acacio M E, et al. STONNE: Enabling cycle-level microarchitectural simulation for DNN inference accelerators[C]//Proc of the IEEE Int Symp on Workload Characterization. Piscataway, NJ: IEEE, 2021: 201−213

[7] Krishnan G, Mandal S K, Pannala M, et al. SIAM: Chiplet-based scalable in-memory acceleration with mesh for deep neural networks[J]. ACM Transactions on Embedded Computing Systems, 2021, 20(5s): 1−24

[8] Raj R, Banerjee S, Chandra N, et al. SCALE-Sim V3: A modular cycle-accurate systolic accelerator simulator for end-to-end system analysis[C]//Proc of the IEEE Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2025: 186−200

[9] Xi S, Yao Yuan, Bhardwaj K, et al. SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads[J]. ACM Transactions on Architecture and Code Optimization, 2020, 17(4): 1−26

[10] Zhu Wenyao, Chen Yizhi, Lu Zhonghai. NoCDAS: A cycle-accurate NoC-based deep neural network accelerator simulator[J]. ACM Transactions on Modeling and Computer Simulation, 2025, 35(4): 1−24

[11] Fu Yonggan, Zhang Yongan, Yu Zhongzhi, et al. GPT4AIGChip: Towards next-generation AI accelerator design automation via large language models[C]//Proc of the IEEE/ACM Int Conf on Computer-Aided Design. Piscataway, NJ: IEEE, 2023: 1−9

[12] Vungarala D, Elbtity M E, Pandit K, et al. TPU-Gen: LLM-driven custom tensor processing unit generator[C]//Proc of the IEEE Int Conf on LLM-Aided Design. Piscataway, NJ: IEEE, 2025: 1−8

[13] Jiang Juyong, Wang Fan, Shen Jiasi, et al. A survey on large language models for code generation[J]. arXiv preprint, arXiv: 2406.00515, 2024

[14] Li Jia, Li Ge, Li Yongmin, et al. Structured chain-of-thought prompting for code generation[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(2): 1−23

[15] Yang Zezhou, Chen Sirong, Gao Cuiyun, et al. An empirical study of retrieval-augmented code generation: Challenges and opportunities[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(7): 1−28

[16] Li Guopeng, Wu Ruiqi, Tan Haisheng, et al. A plan reuse mechanism for LLM-driven agent[J]. Journal of Computer Research and Development, 2024, 61(11): 2706−2720 (in Chinese)
(李国鹏, 吴瑞骐, 谈海生, 等. 面向大语言模型驱动的智能体的计划复用机制[J]. 计算机研究与发展, 2024, 61(11): 2706−2720)

[17] Xia Yuhuan, Li Tun, Zhou Hongji, et al. SimulatorCoder: DNN accelerator simulator code generation and optimization via large language models[J]. arXiv preprint, arXiv: 2602.17169, 2026

[18] Qian Chen, Xie Zihao, Wang Yifei, et al. Scaling large language model-based multi-agent collaboration[J]. arXiv preprint, arXiv: 2406.07155, 2025

[19] Wang Chong, Zhang Jian, Feng Yebo, et al. Teaching code LLMs to use autocompletion tools in repository-level code generation[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(7): 191:1−191:27

[20] Zhang Huan, Cheng Wei, Wu Yuhan, et al. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement[C]//Proc of the 39th IEEE/ACM Int Conf on Automated Software Engineering. New York: ACM, 2024: 1319−1331

[21] Chang Kaiyan, Zhu Wenlong, Wang Kun, et al. A data-centric chip design agent framework for verilog code generation[J]. ACM Transactions on Design Automation of Electronic Systems, 2025, 30(6): 1−26

[22] Li Jia, Tao Chongyang, Li Jia, et al. Large language model-aware in-context learning for code generation[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(7): 190:1−190:33

[23] Lai Yao, Lee S, Chen Guojin, et al. AnalogCoder: Analog circuit design via training-free code generation[C]//Proc of the 39th AAAI Conf on Artificial Intelligence. Palo Alto, CA: AAAI, 2025: 379−387

[24] He Junda, Treude C, Lo D. LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(5): 1−30

[25] Lin Feng, Kim D J, Chen T-H. SOEN−101: Code generation by emulating software process models using large language model agents[C]//Proc of the 47th IEEE/ACM Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2025: 1527−1539

[26] Dong Yihong, Jiang Xue, Jin Zhi, et al. Self-collaboration code generation via ChatGPT[J]. ACM Transactions on Software Engineering and Methodology, 2024, 33(7): 1−38

[27] Zhai Jianwang, Ling Zichao, Bai Chen, et al. Machine learning for microarchitecture power modeling and design space exploration: A survey[J]. Journal of Computer Research and Development, 2024, 61(6): 1351−1369
(翟建旺, 凌梓超, 白晨, 等. 机器学习辅助微架构功耗建模和设计空间探索综述[J]. 计算机研究与发展, 2024, 61(6): 1351−1369)

[28] Lin Hanyue, Wu Jingya, Lu Wenyan, et al. Neptune: A framework for generic network processor microarchitecture modeling and performance simulation[J]. Journal of Computer Research and Development, 2025, 62(5): 1091−1107
(林涵越, 吴婧雅, 卢文岩, 等. Neptune: 一种通用网络处理器微结构模拟和性能仿真框架[J]. 计算机研究与发展, 2025, 62(5): 1091−1107)

[29] Lübeck K, Jung A L F, Wedlich F, et al. Automatic generation of fast and accurate performance models for deep neural network accelerators[J]. ACM Transactions on Embedded Computing Systems, 2025, 24(2): 1−32

[30] Nayak N, Odemuyiwa T O, Ugare S, et al. TeAAL: A declarative framework for modeling sparse tensor accelerators (abstract)[C]//Proc of the 2024 ACM Workshop on Highlights of Parallel Computing. New York: ACM, 2024: 39−40

[31] Mhatre S C, Chandran P. Automatic generation of simulators for processors enhanced for security in virtualization[J]. IEEE Access,

2025, 13: 11930−11943
[32] He Kaiming, Zhang Xiangyu, Ren Shaoqing, et al. Deep residual learning for image recognition[J]. arXiv preprint, arXiv: 1512.03385, 2015
[33] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv: 1409.1556, 2015
[34] Kwon H, Samajdar A, Krishna T. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects[C]//Proc of the 23rd Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2018: 461−475
[35] Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit[C]//Proc of the 44th Annual Int Symp on Computer Architecture. New York: ACM, 2017: 1−12
[36] Chen Jianlv, Xiao Shitao, Zhang Peitian, et al. M3-Embedding: Multilinguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation[J]. arXiv preprint arXiv: 2402.03216, 2025
[37] Liu Y, Le-Cong T, Widyasari R, et al. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues[J]. ACM Transactions on Software Engineering and Methodology, 2024, 33(5): 1−26
[38] de Moura L, Bjørner N. Z3: An efficient SMT solver[C]//Proc of the 14th Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2008: 337−340
[39] Li Jia, Li Yongmin, Li Ge, et al. SKCODER: A sketch-based approach for automatic code generation[C]//Proc of the 45th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2023: 2124−2135
[40] OpenAI, Hurst A, Lerer A, et al. GPT-4o system card[J]. arXiv preprint, arXiv: 2410.21276, 2024
[41] OpenAI. Introducing GPT-5.2[EB/OL]. (2025-12-11)[2025-12-20] https://openai.com/ index/introducing-gpt-5-2
[42] Yang An, Li Anfeng, Yang Baosong, et al. Qwen3 technical report[J]. arXiv preprint, arXiv: 2505.09388, 2025
[43] Comanici G, Bieber E, Schaekermann M, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities[J]. arXiv preprint, arXiv: 2507.06261, 2025
[44] Anthropic. Claude-Sonnet-4[EB/OL]. (2025-9-29)[2025-12-29] https://www.anthropic.com/claude/sonnet
[45] Liu Aixin, Feng Bei, Xue Bing, et al. DeepSeek-V3 technical report[J]. arXiv preprint arXiv: 2412.19437, 2024
[46] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[J]. Communications of the ACM, 2017, 60(6): 84−90
[47] Howard A G, Zhu Menglong, Chen Bo, et al. MobileNets: Efficient convolutional neural networks for mobile vision applications[J]. arXiv preprint, arXiv: 1704.04861
[48] Zhou Qirui, Wen Yuanbo, Chen Ruizhi, et al. QiMeng-GEMM: Automatically generating high-performance matrix multiplication code by exploiting large language models[C]//Proc of the 39th AAAI Conf on Artificial Intelligence. Palo Alto, CA: AAAI, 2025: 22982−22990



Xia Yuhuan, born in 2000. PhD candidate. Her main research interest includes DNN accelerator simulators.
夏雨欢, 2000 年生。博士研究生。主要研究方向为 DNN 加速器模拟器。



Li Tun, born in 1974. Professor, PhD supervisor. Senior member of CCF. His main research interest includes electronic design automation.
李 暾, 1974 年生。教授, 博士生导师。CCF 高级会员。主要研究方向为电子设计自动化。



Zhou Xianfa, born in 2001. Master candidate. His main research interest includes the mapping of neural networks on in-memory accelerators.
周贤发, 2001 年生。硕士研究生。主要研究方向为神经网络在存内加速器上的映射。



Zhao Wenbo, born in 1999. PhD candidate. His main research interest includes fuzz testing in hardware development.
赵文博, 1999 年生。博士研究生。主要研究方向为硬件开发的模糊测试。



Zhang Ruiyu, born in 2000. PhD candidate. Her main research interest includes electronic design automation.
张瑞瑜, 2000 年生。博士研究生。主要研究方向为电子设计自动化。



Guo Yang, born in 1971. Professor, PhD supervisor. Senior member of CCF. His main research interests include very-large-scale integration (VLSI) and processor architecture.
郭 阳, 1971 年生。研究员, 博士生导师。CCF 高级会员。主要研究方向为超大规模集成电路、处理器体系结构。