ChipNeMo: Domain-Adapted LLMs for Chip Design

Mingjie Liu[§], Teo Ene[§], Robert Kirby[§], Chris Cheng[§], Nathaniel Pinckney[§], Rongjian Liang[§]
Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, Bonita Bhaskaran
Bryan Catanzaro, Arjun Chaudhuri, Sharon Clay, Bill Dally, Laura Dang, Parikshit Deshpande
Siddhanth Dhodhi, Sameer Halepete, Eric Hill, Jiashang Hu, Sumit Jain, Brucek Khailany
Kishor Kunal, Xiaowei Li, Hao Liu, Stuart Oberman, Sujeet Omar, Sreedhar Pratty, Ambar Sarkar
Zhengjiang Shao, Hanfei Sun, Pratik P Suthar, Varun Tej, Kaizhe Xu, Haoxing Ren
NVIDIA

***Abstract*—ChipNeMo aims to explore the applications of large language models (LLMs) for industrial chip design. Instead of directly deploying off-the-shelf commercial or open-source LLMs, we instead adopt the following domain adaptation techniques: custom tokenizers, domain-adaptive continued pretraining, supervised fine-tuning (SFT) with domain-specific instructions, and domain-adapted retrieval models. We evaluate these methods on three selected LLM applications for chip design: an engineering assistant chatbot, EDA script generation, and bug summarization and analysis. Our results show that these domain adaptation techniques enable significant LLM performance improvements over general-purpose base models across the three evaluated applications, enabling up to 5x model size reduction with similar or better performance on a range of design tasks. Our findings also indicate that there's still room for improvement between our current results and ideal outcomes. We believe that further investigation of domain-adapted LLM approaches will help close this gap in the future.**

I. Introduction

Over the last few decades, Electronic Design Automation (EDA) algorithms and tools have provided huge gains in chip design productivity. Coupled with the exponential increases in transistor densities provided by Moore's law, EDA has enabled the development of feature-rich complex SoC designs with billions of transistors. More recently, researchers have been exploring ways to apply AI to EDA algorithms and the chip design process to further improve chip design productivity [1] [2] [3]. However, many time-consuming chip design tasks that involve interfacing with natural languages or programming languages still have not been automated. The latest advancements in commercial (ChatGPT, Bard, etc.) and open-source (Vicuna [4], LLaMA2 [5], etc.) large language models (LLM) provide an unprecedented opportunity to help automate these language-related chip design tasks. Indeed, early academic research [6] [7] [8] has explored applications of LLMs for generating RTL that can perform simple tasks in small design modules as well as generating scripts for EDA tools.

We believe that LLMs have the potential to help chip design productivity by using generative AI to automate many language-related chip design tasks such as code generation, responses to engineering questions via a natural language interface, analysis and report generation, and bug triage. In this study, we focus on these three specific LLM applications: an **engineering assistant chatbot** for GPU ASIC and Architecture design engineers, which understands internal HW designs and is capable of explaining complex design topics; **EDA scripts generation** for two domain specific tools based on Python and Tcl for VLSI timing analysis tasks specified in English; **bug summarization and analysis** as part of an internal bug and issue tracking system.

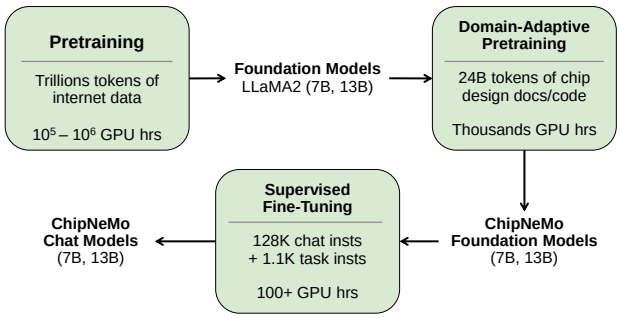


Fig. 1: ChipNeMo Training Flow

Although general-purpose LLMs trained on vast amounts of internet data exhibit remarkable capabilities in generative AI tasks across diverse domains (as demonstrated by Bubeck et al. in [9]), recent work such as BloombergGPT [10] and BioMedLLM [11] demonstrate that domain-specific LLM models can outperform a general purpose model on domain-specific tasks. In the hardware design domain, [6] [12] showed that open-source LLMs (CodeGen [13]) fine-tuned on additional Verilog data can outperform state-of-art OpenAI models. Customizing LLMs in this manner also avoids security risks associated with sending proprietary chip design data to third party LLMs via APIs. However, it would be prohibitively expensive to train domain-specific models for every domain from scratch, since this often requires millions of GPU training hours. To cost-effectively train domain-specific models, we instead propose to combine the following techniques: Domain-Adaptive Pre-Training (DAPT) [14] of foundation models with domain-adapted tokenizers, model alignment using general and domain-specific instructions, and retrieval-augmented generation (RAG) [15] with a trained domain-adapted retrieval model.

As shown in Figure 1, our approach is to start with a base

[§]Equal contribution

foundational model and apply DAPT followed by Supervised Fine-Tuning (SFT). DAPT, also known as continued pretraining with in-domain data, has been shown to be effective in areas such as biomedical and computer science publications, news, and reviews. In our case, we construct our domain-specific pre-training dataset from a collection of proprietary hardware-related code (e.g. software, RTL, verification testbenches, etc.) and natural language datasets (e.g. hardware specifications, documentation, etc.). We clean up and preprocess the raw dataset, then continued-pretrain a foundation model with the domain-specific data. We call the resulting model a ChipNeMo Foundation Model. DAPT is done on a fraction of the tokens used in pre-training, and is much cheaper, only requiring a few thousand GPU hours. As described in Section V, we find this approach to be more effective than Parameter Efficient Training (PEFT) techniques such as LoRA [16] for our use cases.

LLM tokenizers convert text into sequences of tokens for LLM training. A domain-specific tokenizer improves the tokenization efficiency by tailoring rules and patterns for domain-specific terms such as keywords commonly found in RTL. For DAPT, we cannot retrain a new domain-specific tokenizer from scratch, since it would make the foundation model invalid. Instead of restricting ChipNeMo to the pre-trained general-purpose tokenizer used by the foundation model, we instead adapt the pre-trained tokenizer to our chip design dataset, only adding new tokens for domain-specific terms.

ChipNeMo foundation models are completion models which require supervised-fine-tuning (SFT) to adapt to tasks such as chat. We use largely publicly available general-purpose chat instruction datasets for multi-turn chat together with a small amount of domain-specific instruction datasets to perform SFT on the ChipNeMo foundation model, which produces the ChipNeMo Chat model. We observe that SFT with a general purpose chat instruction dataset is adequate to align the ChipNeMo foundation models with queries in the chip design domain. We also added a small amount of task-specific SFT instruction data, which further improves the alignment. We trained multiple ChipNeMo Foundation and Chat models based on variants of LLaMA2 models used as the base foundation model.

To improve performance on the engineering assistant chatbot application, we also leverage Retrieval Augmented Generation (RAG). RAG is an *open-book* approach for giving LLMs precise context for user queries. It retrieves relevant in-domain knowledge from its data store to augment the response generation given a user query. This method shows significant improvement in grounding the model to the context of a particular question. Crucially we observed significant improvements in retrieval hit rate when finetuning a pretrained retrieval model with domain data. This led to even further improvements in model quality.

We highlight the following contributions and findings related to adapting LLMs to the chip design domain:

- We demonstrate domain-adapted LLM effectiveness on three use-cases: an engineering assistant chatbot, EDA tool script generation, and bug summarization and analysis. We achieve a score of 7.4 out of 10 point scale for engineering assistant chatbot responses based on expert evaluations, achieve more than 50% correctness in EDA script generation, and expert evaluation rating of 4 to 5 out of 7 point scale for summarizations and assignment identification tasks.
- Domain-adapted ChipNeMo models dramatically outperforms all vanilla LLMs evaluated on both multiple-choice domain-specific AutoEval benchmarks and human evaluations for applications.
- For tasks where it is possible for the model to generate text from the prompt context (e.g. chat with RAG hits, summarization, code generation with provided documentation), domain-adaptation closes the gap between a state-of-the-art LLaMA2 70B model and a much smaller 13B model (a small incremental training cost enables up to 5x parameter reduction for reduced inference cost).
- Customized tokenizers reduce DAPT token count by up to 3.3% without hurting effectiveness on applications.
- SFT on an additional $1.1K$ domain-specific instructions significantly improves applications proficiency by up to 0.33 out of 10-point scale, 18% correctness and 0.79 out of 7-point scale in engineering assistant chatbot, EDA scripts generation, and bug summarization and analysis, respectively.
- Fine-tuning our ChipNeMo retrieval model with domain-specific data improves the retriever hit rate by 30% over a pre-trained state-of-the-art retriever, in turn improving overall quality of RAG responses.

The paper is organized as follows. Section II describes our dataset and auto evaluation benchmarks for domain knowledge verification. Section III outlines domain adaptation and training methods used including the adapted tokenizer, DAPT, SFT, and RAG. Section IV provides details of each application and the experimental setup. Section V describes the experimental results including human evaluations for each application. Section VI discusses ChipNeMo limitations and future work. Section VII describes relevant LLM methods and other work targeting LLMs for chip design. Finally, complete results along with additional model training details and examples of text generated by the application use-cases are illustrated in the Appendix.

II. Dataset

A. DAPT Dataset

During Domain-Adaptive Pre-Training (DAPT), we assemble a dataset from a combination of NVIDIA-proprietary chip design specific data sources and publicly available datasets.

Chip Design Datasets: Our internal dataset consists of a diverse range of text sources pertinent to chip design, spanning design, verification, infrastructure, and internal documentation. Table I provides a breakdown of the data collected after filtering, and the corresponding number of tokens using the LLaMA2 tokenizer. We construct the dataset by gathering all relevant internal data, then filtering by file type, based

on filename extensions and distinguishing between machine-generated and human-written content. Although we evaluated on three specific use cases, we did not specifically limit the dataset to sources known to be relevant to these use cases since we believed that incorporating additional domain knowledge would improve performance. After collection, cleaning, and filtering, the internal data training corpus has 23.1 billion tokens. Further details of the data collection process are covered in Appendix A.

Public Datasets: We augment the chip design specific data with a sample of publicly available data from various sources, a common practice in the development of foundational large language models. Our approach was to reuse public training data from other language models, with the stipulation that it must be publicly accessible and compatible with open sourcing. These datasets exhibit a high degree of correlation with the pretraining data used in LLaMA2 [5], with the intention of preserving general knowledge and natural language capabilities during DAPT. The public datasets used by ChipNeMo can be categorized into two groups, natural language and code. For the natural language component, we draw from Wikipedia data [17], as it is widely regarded for its high data quality. For code, we leverage GitHub data [18], focusing on programming languages also present in our internal data chip design dataset such as C++, Python, and Verilog. To ensure that the overall dataset is representative of pre-training distributions, we perform a sub-sampling operation that results in approximately 9.2% of the total training tokens being sampled from these public datasets, with a balanced representation of natural language and code.

Data Blend: A significant proportion of the domain data we gathered is comprised of unannotated code from diverse origins. In an effort to enhance the model's comprehension of domain-specific knowledge, we conducted downsampling of code data while concurrently upsampling natural language data, specifically design documentation, over a span of 2 to 4 training epochs. We also increased the representation of data that we deemed more pertinent to downstream applications, such as human-written EDA tool scripts. Furthermore, we incorporated publicly available domain data for 1 epoch. Details of the token distribution for training are shown in Table I.

B. SFT Instruction Data

During Supervised Fine-Tuning (SFT), we employ a general chat SFT instruction dataset that is accessible for commercial use. The dataset is comprised largely of publicly available instruction following datasets including OASST [19], FLAN [20], P3 [21] and a small amount of a broad domain proprietary dataset comprising various topics such as brainstorming, open-ended question answering, rewriting, summarization etc. It's important to note that the SFT instruction data we discuss here is focused on general natural language tasks and does not contain any information or tasks related to the downstream use cases in chip design. In total, this dataset comprises 128,000 training samples.

Additionally, we meticulously assembled a domain-specific instruction dataset for aligning the model to downstream use cases. These examples have been meticulously crafted by subject matter experts and are formatted as single-turn questions and answers. Table II depicts the quantity of our domain-specific instruction dataset. It's worth noting that the total number of training samples in the domain-specific instruction dataset is quite small when compared to the extensive amount of generative chat instruction data.

C. AutoEval

In order to quickly and quantitatively assess the accuracy of various models, we established evaluation criteria structured as multiple-choice question-and-answer formats for each use case, designed to closely align with established benchmarks, such as MMLU [22]. In the process of formulating these multiple-choice questions, collaboration with domain experts was pivotal. The goal was to ensure that each question included at least one complex answer choice, thereby posing a challenge to individuals with limited domain expertise. Careful attention was also given to prevent any inadvertent contamination of the questions with data from our domain-specific SFT. In addition to the per-use-case benchmarks, an additional benchmark was created for general circuit design knowledge, covering both analog and digital design topics. The number of multiple-choice questions for evaluation benchmark are shown in Table III.

When we report results on the above benchmarks, we take average results obtained from five distinct runs to mitigate the effects of variance and noise in the testing process. Each iteration employs a set of 5-shot examples, with variations introduced across each individual runs.

In addition to these domain-specific evaluation benchmarks, we also include commonly-used publicly available LLM academic benchmarks. Furthermore, we measure the model's code generation capabilities, by evaluating HumanEval [23] for Python and VerilogEval [12] for Verilog.

III. ChipNeMo Domain Adaptation Methods

ChipNeMo implements multiple domain adaptation techniques to adapt LLMs to the chip design domain. These techniques include custom tokenizers for chip design data, domain adaptive pretraining with large corpus of domain data, supervised-fine-tuning with domain specific tasks, and retrieval-augmented generation with a fine-tuned retrieval model. We will illustrate the details of each technique in this section.

A. Tokenizer

When adapting a pre-trained tokenizer, the main goals are to improve tokenization efficiency on domain-specific data, maintain efficiency and language model performance on general datasets, and minimize the effort for retraining/fine-tuning. To achieve this, we've developed a four-step approach:

- Step 1: Training a tokenizer from scratch using domain-specific data.
- Step 2: From the vocabulary of the new tokenizer, identifying tokens that are absent in the general-purpose tokenizer and are rarely found in general-purpose datasets.

Data Source Type	Data Percentage (%)	Data Tokens (B)	Training Percentage (%)	Training Tokens (B)
Bug Summary	9.5%	2.4	10.0%	2.4
Design Source	47.0%	11.9	24.5%	5.9
Documentation	17.8%	4.5	34.0%	8.2
Verification	9.1%	2.3	10.4%	2.5
Other	7.9%	2.0	12.0%	2.9
Wikipedia	5.9%	1.5	6.2%	1.5
Github	2.8%	0.7	3.0%	0.7
Total	100.0%	25.3	100.0%	24.1

TABLE I: Breakdown of Data by Source. Token count measured with original LLaMA2 tokenizer.

Domain Source	Number of Samples
Design Knowledge	280
EDA Script Generation	480
Bug summarization and analysis	392
Total	1152

TABLE II: Breakdown of Domain SFT Data.

Domain Source	Number of Questions
Design Knowledge (Design)	94
EDA Script Generation (Scripting)	74
Bug Summarization and Analysis (Bugs)	70
Open Domain Circuit Design (Circuits)	227

TABLE III: Domain-specific Evaluation Benchmark.

- Step 3: Expanding the general-purpose tokenizer with the newly identified tokens at Step 2.
- Step 4: Initializing the embeddings of the new tokens by utilizing the general-purpose tokenizer.

Specifically for Step 4, when a new token is encountered, it is tokenized using the pretrained general-purpose tokenizer. The embedding of the new token is determined by averaging the embeddings of the tokens generated by the general-purpose tokenizer [24], and the output layer weights initialized to zero.

Step 2 helps maintain the performance of the pre-trained LLM on general datasets by selectively introducing new tokens that are infrequently encountered in general-purpose datasets. And Step 4 reduces the effort required for retraining/finetuning the LLM via initialization of the embeddings of new tokens guided by the general-purpose tokenizer.

B. Domain Adaptive Pretraining

In our study, we apply DAPT on pretrained foundation base models LLaMA2 7B/13B. Each DAPT model is initialized using the weights of their corresponding pretrained foundational base models. We name our DAPT models **ChipNeMo**. We employ tokenizer augmentation as depicted in Section III-A and initialize embedding weight accordingly [24]. We conduct further pretraining on domain-specific data by employing the standard autoregressive language modeling objective. All model training procedures are conducted using the NVIDIA NeMo framework [25], incorporating techniques such as tensor parallelism [26] and flash attention [27] for enhanced efficiency.

Our models undergo a consistent training regimen with similar configurations. A small learning rate of $5 \cdot 10^{-6}$ is employed, and training is facilitated using the Adam optimizer, without the use of learning rate schedulers. The global batch size is set at 256, and a context window of 4096 tokens is applied, resulting in an effective batch size of 1M tokens. Detailed training hyperparameters are provided in Appendix B. The toal number of training steps is set to 23,200, equating to roughly 1 epoch of the data blend.

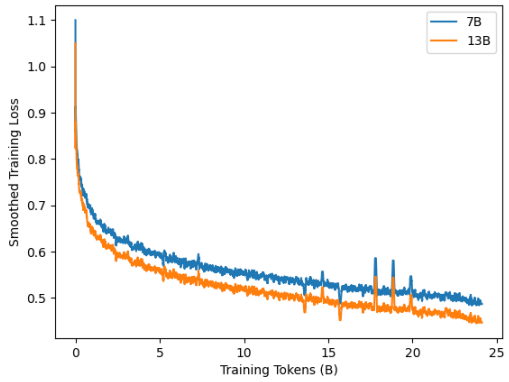


Fig. 2: Smoothed Training Loss for ChipNeMo with Tokenizer Augmentation.

Figure 2 illustrates the training loss of ChipNeMo under the specified hyperparameters. We do observe spikes in the training loss. In contrast to the hypothesis in [28], we postulate that in our scenario, these spikes can be attributed to "bad data" since these irregularities seem to consistently occur in similar training steps for the same model, even across different model sizes. We chose not to address this issue, as these anomalies did not appear to significantly impede subsequent training steps (with no noticeable degradation in validation loss), possibly due to our application of a low learning rate.

C. Supervised Fine-Tuning

After DAPT, we perform model alignment with supervised fine-tuning (SFT). We adopt the identical hyperparameter training configuration as DAPT for all models, with the exception of using a reduced global batch size of 128. All SFT data is structured according to the chat template below:

```
<extra_id_0>System\n{system}
<extra_id_1>User\n{user_utterance}
<extra_id_1>Assistant\n{chipnemo_response}
...
```

We employ an autoregressive optimization objective, implementing a strategy where losses associated with tokens originating

from the system and user prompts are masked [5]. This approach ensures that during backpropagation, our focus is exclusively directed towards the optimization of answer tokens.

We combine our domain SFT dataset, comprising approximately 1.1k samples, with the more extensive general chat SFT dataset of 128k samples. We then engaged in fine-tuning for a single epoch after applying a random shuffle to the data. We conducted experiments involving augmentation of the domain-specific SFT dataset for more than one epoch. However, it became evident that the model rapidly exhibited signs of overfitting when presented with in-domain questions, often repeating irrelevant answers from the domain SFT dataset.

Additionally, we conducted an additional SFT using solely the general chat dataset, excluding any domain-specific SFT data. For clarity, we designate all our ChipNeMo models as follows:

1) **ChipNeMo-Chat:** Models fine-tuned with both domain and general chat data;
2) **ChipNeMo-Chat (noDSFT):** Models fine-tuned with general chat data exclusively.

We also experimented with DAPT directly on a chat aligned model, such as the LLaMA2-Chat model. We found that DAPT significantly degraded the model's alignment, making the resulting model useless for downstream tasks.

D. Retrieval-Augmented Generation

It is well known that LLMs can generate inaccurate text, so-called *hallucination* [29]. Although the phenomenon is not completely understood, we still must mitigate *hallucinations* since they are particularly problematic in an engineering assistant chatbot context, where accuracy is critical. Our proposal is to leverage the retrieval augmented generation (RAG) method. RAG tries to retrieve relevant passages from a database to be included in the prompt together with the question, which grounds the LLM to produce more accurate answers. We find that using a domain adapted language model for RAG significantly improves answer quality on our domain specific questions. Also, we find that fine-tuning an off-the-shelf unsupervised pre-trained dense retrieval model with a modest amount of domain specific training data significantly improves retrieval accuracy. Our domain-adapted RAG implementation diagram is illustrated on Figure 3.

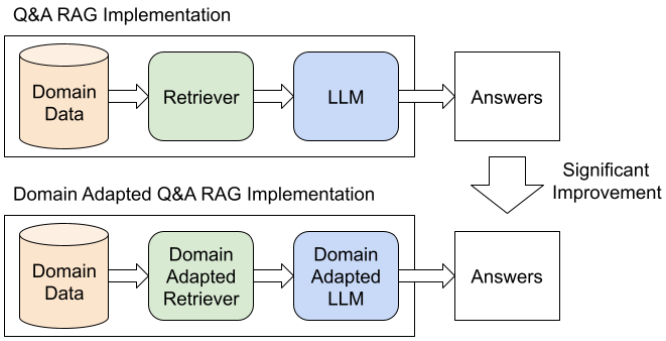


Fig. 3: RAG Implementation Variations

We created our domain adapted retrieval model by fine-tuning the *e5_small_unsupervised* model [30] with 3000 domain specific auto-generated samples using the Tevatron framework [31]. The sample generation and training process are covered in Appendix C.

Even with the significant gains that come with fine-tuning a retrieval model, the fact remains that retrieval still struggles with queries that do not map directly to passages in the document corpus or require more context not present in the passage. Unfortunately, these queries are also more representative of queries that will be asked by engineers in real situations. Combining retrieval with a domain adapted language model is one way to address this issue.

IV. LLM Applications

We conducted a survey of potential LLM applications within our design teams and categorized them into four buckets: **code generation**, **question & answer**, **analysis and reporting**, and **triage**. Code generation refers to LLM generating design code, testbenches, assertions, internal tools scripts, etc.; Q & A refers to an LLM answering questions about designs, tools, infrastructures, etc.; Analysis and reporting refers to an LLM analyzing data and providing reports; triage refers to an LLM helping debug design or tool problems given logs and reports. We selected one key application from each category to study in this work, except for the **triage** category which we leave for further research. The motivation and technical details of each application are given below.

A. Engineering Assistant Chatbot

This application aims to help design engineers with answers to their architecture, design, verification, and build questions, which could significantly improve their overall productivity without impacting the productivity of others. It is observed that design engineers often enjoy brainstorming, designing hardware, and writing code, but can be slowed down waiting for answers on design knowledge they lack. Design productivity can also be enhanced by avoiding having engineers write code based on mistaken assumptions or debugging code that they are unfamiliar with. Internal studies have shown that up to 60% of a typical chip designer's time is spent in debug or checklist related tasks across a range of topics including design specifications, testbench construction, architecture definition, and tools or infrastructure. Experts on these issues are often spread around the globe in a multinational company, such that it is not always convenient to find immediate help. Therefore, an engineering assistant chatbot based on knowledge extracted from internal design documents, code, any recorded data about designs and technical communications such as emails and corporate instant communications, etc. could help significantly improve design productivity. We implemented this application with the domain-adapted RAG method mentioned in Section III-D.

B. EDA Script Generation

Another common task in an industrial chip design flow is writing EDA scripts to accomplish a variety of tasks such

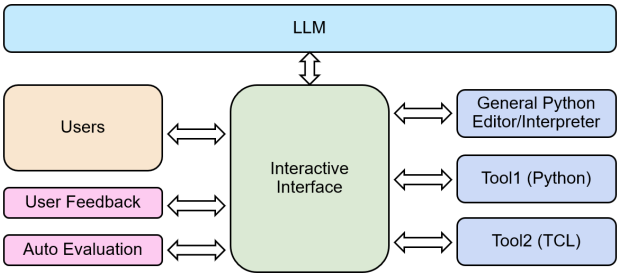


Fig. 4: LLM script generator integration with EDA tools

as design implementation, introspection and transformation. These scripts often leverage both tool-specific and custom internal script libraries. Learning these libraries, navigating tool documentation, and writing and debugging these scripts, can take up a significant amount of engineering time.

LLMs have proven adept at small scale code generation on a wide array of tasks [32] and therefore customizing these models to accelerate engineer productivity in this domain specific task is a natural fit. In this work we focus on generating two different types of scripts from natural language task descriptions. The first are scripts which leverage Tool1, an internal python library for design editing and analysis. The second are Tcl scripts that use the command interface provided by Tool2, which is a leading industrial static timing analysis tool.

In order to build our domain-specific fine-tuning dataset for this task, production scripts for both tools were collected from design experts. We observed that our DAPT models can generate reasonable inline comments for the code. This enabled us to use these models to improve the quality of collected scripts by generating additional inline comments. Human experts later verified and corrected these comments and created an associated prompt. These prompts and code pairs make up the data used for DSFT in the format discussed in Section III-C.

To provide and collect feedback in the most meaningful way, we spent significant effort building the flow shown in Fig. 4 where engineers can both query the model and run generated code through the same interface. This allows us to be confident in the *correctness* of generated code as well as provide accurate feedback by allowing engineers to see how many corrections they might need to get a functioning script. We support Tool1 and Tool2 integration by establishing interactive connections to tool servers.

Additionally, we provide a user feedback form, allowing us to compare different models and glean valuable insights from user feedback. This valuable information can aid us in further refining our models.

C. Bug Summarization and Analysis

Tracking the reporting, triage, debug and resolution of various features and bugs across stages of the production flow is a time-consuming process. Engineering managers spend a lot of time reviewing internal issue tracking databases to build understanding of the state of the project and help speed their execution. Therefore, a tool that is able to look at all supporting information and quickly summarize both technical and managerial data as well as suggest next steps would boost team productivity. We focus on using LLMs to generate three different outputs - one focused on technical details, one on managerial details and one recommending task assignment.

To study these tasks we used NVIDIA's internal bug database, NVBugs. This database is used for bug reporting, tracking and resolution as well as general task and feature tracking across the company. We expect ChipNeMo models to perform well on this task as a large amount of bug data was included in the DAPT dataset. Additionally, we built a domain-specific SFT dataset for this task that includes examples of the bug summarizing and task assignment tasks.

Often, bug descriptions contain large snippets of log files or code dumps along with long comment histories. In such cases, the bug text is too large for our LLM context windows. To work around this, we implemented two solutions. First, we found and replaced long path names with shorter aliases to allow the model to associate paths that occur in multiple places in the bug without needing to process the entire string. Second, we split the summarization task into an incremental task where the model is tasked with accumulating data across multiple summary and bug data chunks. We use a hierarchical approach where the bug is first separated into chunks that fit into the context window. Those chunks are then summarized and the summaries are accumulated then separated into chunks. This process is repeated until the entire set of summaries fits into a single context window and a single summary is generated. We use this same approach independent of the LLM used for summarization.

V. Evaluations

We evaluate our training methodology and application performance in this section. We study both 7B and 13B models in the training methodology evaluation, and only 13B models in the application performance evaluation. For comparison, we also evaluate two baseline chat models: LLaMA2-13B-Chat* and LLaMA2-70B-Chat. LLaMA2-13B-Chat* is the foundation LLaMA2 13B base model fine-tuned with our general purpose chat instruction dataset, which is different from the original LLaMA2-13B-Chat model trained with reinforcement learning from human feedback (RLHF). We chose to do so for fair comparison of domain adapted models and base models under the same model alignment approach. LLaMA2-70B-Chat is the publicly released LLaMA2-Chat model trained with RLHF, which is considered as the state-of-the-art(SOTA) open-source chat model.

A. Tokenizer

We adapt the LLaMA2 tokenizer (containing 32K tokens) to chip design datasets using the previously outlined four-step process. Approximately 9K new tokens are added to the LLaMA2 tokenizer. The adapted tokenizers can improve tokenization efficiency by 1.6% to 3.3% across various chip design datasets as shown in Figure 5. We observe no obvious

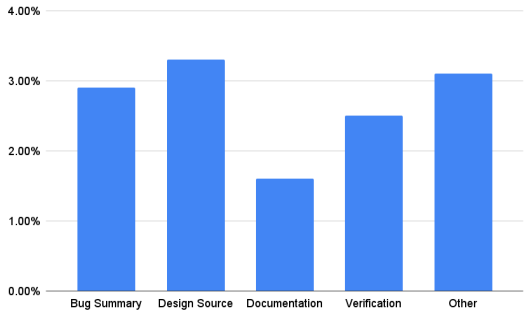


Fig. 5: ChipNeMo Tokenizer Augmentation Improvements.

changes to tokenizer efficiency on public data. Importantly, we have not observed significant decline in the LLM's accuracy on public benchmarks when using the custom augmented tokenizers even prior to DAPT.

B. Domain Adaptive Pretraining

Figure 6 presents the outcomes for ChipNeMo models on the AutoEval benchmark for chip design domain and open domain academic benchmarks. Our research findings can be summarized as follows:

1) DAPT models exhibit a slight degradation in accuracy on open-domain academic benchmarks.
2) DAPT exerts a substantial positive impact on tasks within the domain itself. This effect is manifested in significant improvements in internal design knowledge as well as general circuit design knowledge.
3) The use of larger and more performant foundational models yields better zero-shot results on domain-specific tasks. Furthermore, the employment of superior base models results in enhanced domain models post-DAPT, leading to heightened performance on in-domain tasks.
4) Improvements attributed to DAPT with in-domain tasks exhibit a positive correlation with model size, with larger models demonstrating more pronounced enhancements in domain-specific task performance post-DAPT.

C. Training Ablation Studies

For our ablation studies, we conducted multiple rounds of domain adaptive pre-training. We provide brief summaries and refer to the Appendix B for details.

The differences between training with the augmented tokenizer and the original tokenizer appeared to be negligible. We thus primarily attribute the accuracy degradation on academic benchmarks to domain data. Moreover, the removal of the public dataset only slightly regressed on most tasks including academic benchmarks, with the exception of Verilog coding, where we observed a noticeable difference. This suggests that the inclusion of GitHub Verilog data contributed to enhanced Verilog coding capabilities, particularly when the base foundation models lacked sufficient data in this domain.

In our exploration, we experimented with employing a larger learning rate, as in CodeLLaMA [32]. We observed large spikes in training loss at the initial training steps. Although this approach eventually led to improved training and validation loss, we noted substantial degradations across all domain-specific and academic benchmarks, except on coding. We hypothesize that a smaller learning rate played a dual role, facilitating the distillation of domain knowledge through DAPT while maintaining a balance that did not veer too far from the base model, thus preserving general natural language capabilities.

We also explored the application of Parameter Efficient Fine-Tuning (PEFT) in the context of Domain-Adaptive Pretraining (DAPT). In this pursuit, we conducted two experiments involving the incorporation of LoRA adapters [16], introducing additional parameters of 26.4 million (small) and 211.2 million (large) respectively. In both instances, our findings revealed a significant accuracy gap on in-domain tasks when compared to the full-parameter DAPT approach. Furthermore, when contrasting the outcomes between small and large PEFT models, we observed a marginal enhancement on in-domain task accuracy, with large models exhibiting a slight improvement. We posit that this phenomenon may be attributed to the necessity of training a large amount of parameters in order to accommodate a substantial volume of information, and the susceptibility of PEFT models to catastrophic forgetting [33].

D. Training Cost

All models have undergone training using 128 A100 GPUs. We estimate the costs associated with domain adaptive pretraining for ChipNeMo as illustrated in Table IV. It is worth noting that DAPT accounts for less than 1.5% of the overall cost of pretraining a foundational model from scratch.

Model Size	Pretraining	DAPT	SFT
7B	184,320	2,620	90
13B	368,640	4,940	160
70B	1,720,320	-	-

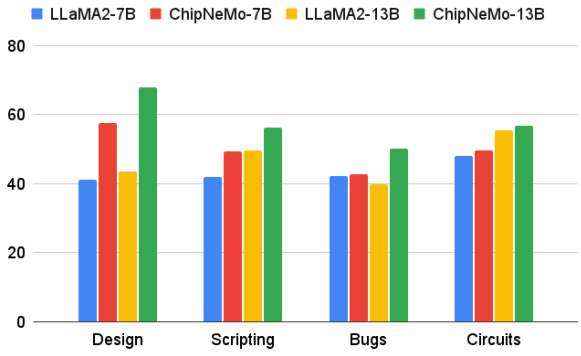
TABLE IV: Training cost of LLaMA2 models in GPU hours. Pretraining cost from [5].

E. RAG and Engineering Assistant Chatbot

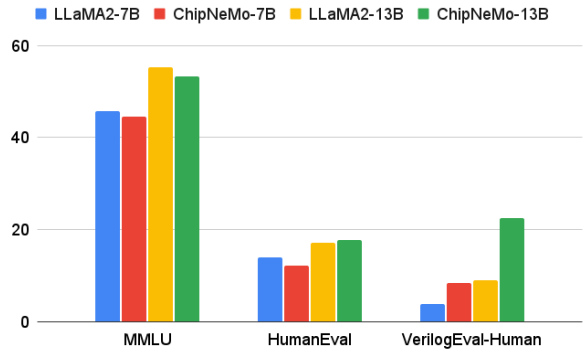
We created a benchmark to evaluate the performance of design chat assistance, which uses the RAG method. This benchmark includes 88 questions in three categories: architecture/design/verification specifications (Specs), testbench regression documentation (Testbench), and build infrastructure documentation (Build). For each question, we specify the golden answer as well as the paragraphs in the design document that contains the relevant knowledge for the answer. These questions are created by designers manually based on a set of design documents as the data store for retrieval. It includes about 1.8K documents, which were segmented into 67K passages, each about 512 characters.

First, we compare our domain adapted retrieval model with Sentence Transformer [34] and *e5_small_unsupervised* [30] on each category. Each model fetches its top 8 passages from the data store.

As shown in Figure 7, our domain-adapted model performed 2x better than the original *e5_small_unsupervised* model and 30% better than sentence transformer.



(a) Chip Design Domain Benchmarks.



(b) Academic Benchmarks.

Fig. 6: AutoEval Benchmark Result for ChipNeMo.

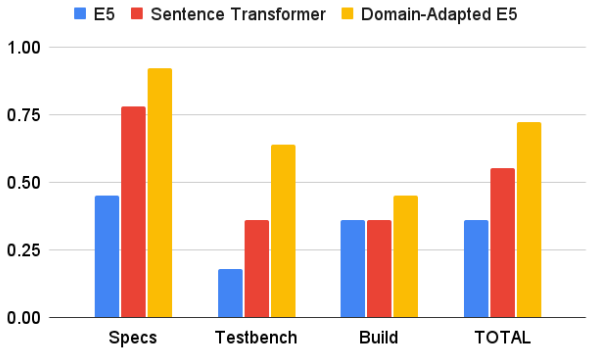


Fig. 7: Retrieval Model Accuracy Comparison

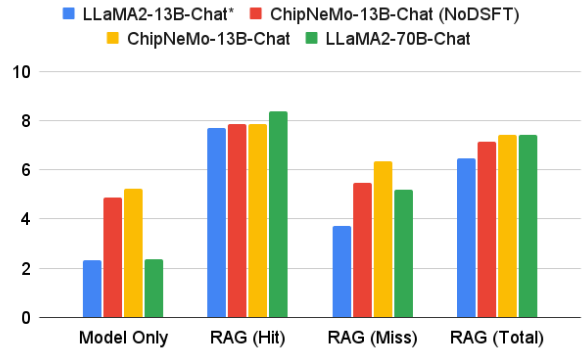


Fig. 8: Human Evaluation of Different Models. Model Only represents results without RAG. RAG (Hit)/(Miss) only include questions whose retrieved passages hit/miss their ideal context, RAG (Total) includes all questions.

The queries in the Specs category are derived directly from passages in the documents, so their answers are often nicely contained in a concise passage and clearly address the query. On the other hand, the queries of the Testbench and Build categories are not directly derived from passages, so their answers were often not as apparent in the fetched passages and required more context (see Appendix C for detailed examples). This significantly contributes to the difference in retrieval quality between the categories.

We conducted evaluation of multiple ChipNeMo models and LLaMA2 models with and without RAG. The results were then scored by human evaluators on a 10 point scale and shown in Figure 8.

We made the following observations:

- RAG significantly boosts human scores. RAG improves the scores of LLaMA2-13B-Chat*, ChipNeMo-13B-Chat, and LLaMA2-70B-Chat by 3.82, 2.19, and 5.05, respectively. Note that, scores are generally higher even with RAG miss, particularly on LLaMA2 models. We hypothesize that the additional in-domain context helps to boost the performance.
- ChipNeMo-13B-Chat outperform similar sized LLaMA2-13B-Chat* in model only and RAG evaluations by 2.88 and 1.25, respectively.
- ChipNeMo-13B-Chat with RAG achieves the same score (7.4) as the 5X larger model LLaMA2-70B-Chat with RAG, where LLaMA2-70B-Chat does better in extracting answers on hits; however, domain adaptation makes up for it on the misses.
- Domain SFT helps improve the performance of ChipNeMo-13B-Chat by 0.28 (with RAG) and 0.33 (without RAG).

The complete evaluation results on all models are shown in Appendix D.

F. EDA Script Generation

In order to evaluate our model on the EDA script generation task, we created two different types of benchmarks. The first is a set of “Easy” and “Medium” difficulty tasks (1-4 line solutions) that can be evaluated without human intervention by comparing with a golden response. Due to the work required to build and evaluate these benchmarks we only have this evaluation set for our Python task. The second set of tasks (“Hard”) come from real use case scenarios that our engineers chose. These tasks are much harder requiring 10’s of lines to solve. Because these are hard to evaluate in an automatic way, we had human engineers judge the correctness between 0% and 100%. The size of these benchmarks are described in Table V.

Work is ongoing to both increase the size and scope for these benchmarks to allow us to further improve these models.

We discovered that our models were unable to answer some of our harder tasks. The tasks required knowledge of many tool APIs and the model seemed to be unable to decide on the proper ones while keeping the control flow properly organized. To mitigate this, we appended a human curated context to the prompt, specific to each question. This context contained explanations of different functions or attributes needed to properly write the desired script. We only provided this for the "Hard with Context" benchmark category. This also allows us to study the possible effect of a retrieval based solution, which we leave to future work.

As can be seen in the ablation results in Figure 9, both DAPT and domain SFT for our problem was important. Without DAPT, the model had little to no understanding of the underlying APIs and performed poorly on automatically evaluated benchmarks. Domain SFT further improved the results. We believe this is because our domain SFT data helps guide the model to present the final script in the most directly applicable fashion.

One interesting result is the LLaMA2-70B pass rate on "Hard with Context" benchmarks. It performs better than most models on the Python tool but poorly on the Tcl tool. This is likely because when provided with the correct context, LLaMA2-70B's superior general Python coding ability is able to solve novel problems it has not been trained on. However, the LLaMA2-70B model is unable to generalize its coding ability to the Tcl tool, likely because it has not been exposed to a large volume of Tcl code. This highlights the benefit of DAPT when it comes to low-volume or proprietary programming languages.

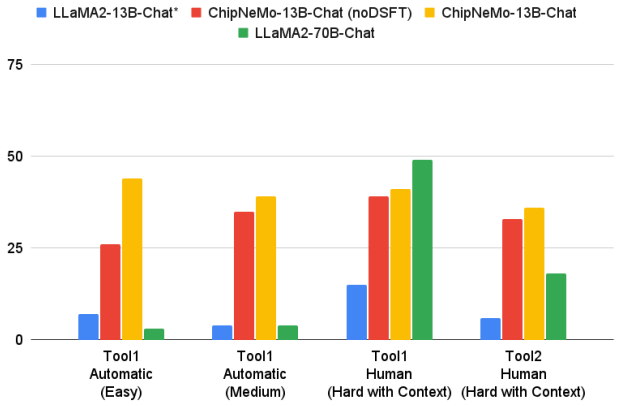


Fig. 9: EDA Script Generation Evaluation Results

G. Bug Summarization and Analysis

To evaluate our models on bug summarization and analysis we have a hold out set of 40 bugs which are ideal candidates for summarization. This includes having a long comment history or other data which makes the bugs hard for a human to quickly summarize. We then ask humans to rate both modes of summarization as well as the bug assignment the LLM suggests. The evaluation metric is based on a 7 point Likert scale. Our results are included in Figure 10.

Evaluation Benchmark Name	Size
Tool1 (Python) - Automatic (Easy)	150
Tool1 (Python) - Automatic (Medium)	30
Tool1 (Python) - Human (Hard with Context)	10
Tool2 (Tcl) - Human (Hard with Context)	10

TABLE V: EDA Script Generation Evaluation Benchmarks

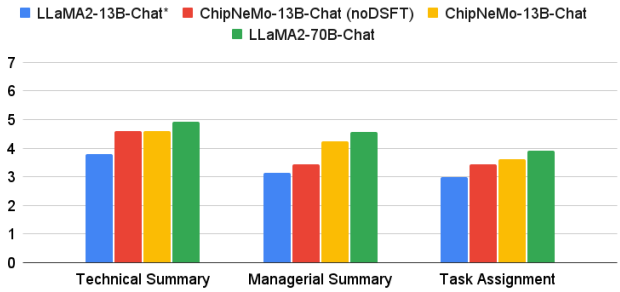


Fig. 10: Bug Summarization and Analysis Evaluation Results

ChipNeMo-13B-Chat models outperform the base LLaMA2-13B-Chat* model for all three tasks, improving the 7 point Likert score by 0.82, 1.09 and 0.61 for technical summary, managerial summary and assignment recommendation, respectively. Domain SFT also significantly improves the performances over without domain SFT on managerial summarization and task assignment.

We hypothesize that contrary to the technical summarization task whose quality and technical content are more dependent on the model's understanding of natural language semantics, managerial summary requires the model to understand how to summarize the input data while retaining key personnel/engineer names. This needs a more careful instruction-based fine-tuning of the LLM.

LLaMA2-70B-Chat model also performs very well on all three tasks, beating ChipNeMo-13B model over all tasks. Note that LLaMA2-70B-Chat model also suffers from long-context challenges with 4096 context size, we believe effective chunk-and-combine schemes (hierarchical and incremental), choice of instructional prompts at various stages of summarization, choice of prompt during task assignment, and raw data formatting/pre-processing help in circumventing the long-context challenge and enable LLaMA2-70B-Chat to achieve high scores even without DAPT and domain SFT.

VI. Discussion

A. Considerations for Domain Adaptation

Although domain-adapted ChipNeMo models achieve significant improvements over their corresponding foundation models, we also observe that the larger LLaMA2 70B can sometimes achieve similar accuracy as ChipNeMo, as seen in Figures 8, 9, and 10. Recent work has leveraged these powerful models to perform chip design tasks.

However, it is important to consider the cost-efficiency benefits gained from the use of a smaller model. Pope et al. demonstrate that inference costs on an 8B model are 8-12x lower than on a 62B model for equal latency targets

[35]. Furthermore, model size reduction can lead to dramatic increases in inference speed by allowing a model to fit within a single GPU or node where it otherwise could not [36]. Our ChipNeMo 13B model can be loaded within the memory of a single A100 GPU without any quantization, unlike the LLaMA2 70B model. This leads to significant inference speed increases under normal GPU operation, which can be traded off for significant inference cost reduction should the GPU be underclocked.

Thus, when deciding between the use of a larger general-purpose model versus a smaller specialized model in a production environment the following criteria must be considered:

- **Training and inference trade-off:** Smaller domain adapted models can match the accuracy of larger general purpose models. While domain adaptation incurs additional up-front costs, the use of smaller models leads to significantly reduced operating costs.
- **Uniqueness of use case:** As can be seen from Figures 6, 9, and 10, domain adapted models show the most improvement on tasks that are rarely present in the public domain, such as writing code in proprietary languages or libraries. Indeed, our data shows that even when they are provided with hand-picked contexts, large general purpose models have difficulty matching the accuracy of domain adapted models in such scenarios.
- **Availability of domain data:** Domain adaption works best when there is large amount of training data, i.e. billions of training tokens. This is often the case for large corporations and projects which have accumulated a large amount of internal documents and code, but not necessarily true for smaller businesses or projects.
- **End use case diversity:** It is possible to fine-tune a general purpose model for a particular task, but domain-adapted models are suited for a diverse set of tasks in a domain. Although we only demonstrate three use cases for ChipNeMo models in this work, it can be readily re-used for other use cases with sufficient SFT data.

B. Performance Gap

Although ChipNeMo achieves impressive results in our selected applications as shown in Appendix E, the evaluation results for all applications still show a considerate gap with human expert performance. We are considering the following approaches to bridge this performance gap:

1) Data Collection: We can expand the DAPT dataset to include more internal proprietary data. In addition, we plan to add more task specific instruction sets for SFT as evidence shown task specific SFT improves the evaluation results meaningfully.

2) Base Model: We expect better and larger base models can improve performance, such as LLaMA2 70B. We can also explore applying DAPT to code-specific base models such as Code LLaMA [32] for code generation tasks.

3) Training: We also plan to conduct reinforcement learning from human feedback (RLHF) [37] over the ChipNeMo chat model to make it more versatile. We plan to leverage pretrained reward models trained over general purpose datasets. We also plan to conduct long-context training [38] to overcome the challenge where long context is needed, e.g. in the bug summarization application. In general, longer context support would help improve retrieval based methods for chat assistance as well as code generation.

4) Retrieval: We will further investigate better RAG methods for both the engineering assistant chatbot and EDA script generation. For the engineering assistant chatbot, we can create different data stores for different application areas. We can also integrate enterprise search engines with RAG to find relevant context for a diverse set of problems. For code generation, we can investigate automated retrieval of context from existing code and documentation.

C. Agent-Based Design Methodologies

The use cases we experimented in this work are straightforward applications of the prompt and response capability of LLMs. *Agents* refer to the use of an LLM to choose a sequence of actions to take, where an LLM is acting as a reasoning engine to drive outside tools. Chip design processes involve many existing EDA tools and methodologies. We believe some of these methodologies can be driven by agents powered by domain-adapted LLMs such as ChipNeMo models. We plan to work on agent-based design methodologies for verification and optimization in the future.

VII. Related Works

Many domains have a significant amount of proprietary data which can be used to train a domain-specific LLM. One approach is to train a domain specific foundation model from scratch, e.g., BloombergGPT [10] for finance, BioMedLLM [11] for biomed, and Galactica [39] for science. These models were usually trained on more than 100B tokens of raw domain data. The second approach is domain-adaptive pretraining (DAPT) [14] which continues to train a pretrained foundation model on additional raw domain data. It shows slight performance boost on domain-specific tasks in domains such as biomedical, computer science publications, news, and reviews. In one example, [40] continued-pretrained a foundation model on technical content datasets and achieved state-of-the-art performance on many quantitative reasoning tasks.

Retrieval Augmented Generation (RAG) helps ground the LLM to generate accurate information and to extract up-to-date information to improve knowledge-intensive NLP tasks [41]. It is observed that smaller models with RAG can outperform larger models without RAG [42]. Retrieval methods include sparse retrieval methods such as TF-IDF or BM25 [43], which analyze word statistic information and find matching documents with a high dimensional sparse vector. Dense retrieval methods such as [44] [45] find matching documents on an embedding space generated by a retrieval model pretrained on a large corpus with or without fine-tuning on a retrieval dataset. The retrieval model can be trained standalone [44] [45] [46] or jointly with language models [47] [42]. In addition, it has been

shown that off-the-shelf general purpose retrievers can improve a baseline language model significantly without further fine-tuning [48]. RAG is also proposed to perform code generation tasks [49] by retrieving from coding documents.

Foundation models are completion models, which have limited chat and instruction following capabilities. Therefore, a model alignment process is applied to the foundation models to train a corresponding chat model. Instruction fine-tuning [20] and reinforcement learning from human feedback (RLHF) [37] are two common model alignment techniques. Instruction fine-tuning further trains a foundation model using instructions datasets. RLHF leverages human feedback to label a dataset to train a reward model and applies reinforcement learning to further improve models given the trained reward model. RLHF is usually more complex and resource hungry than instruction fine-tuning. Therefore, recent studies also propose to reduce this overhead with simpler methods such as DPO [50] and SteerLM [51].

Researchers have started to apply LLM to chip design problems. Early works such as Dave [52] first explored the possibility of generating Verilog from English with a language model (GPT-2). Following that work, [6] showed that fine-tuned open-source LLMs (CodeGen) on Verilog datasets collected from GitHub and Verilog textbooks outperformed state-of-the-art OpenAI models such as *code-davinci-002* on 17 Verilog questions. [12] proposed a benchmark with more than 150 problems and demonstrated that the Verilog code generation capability of pretrained language models could be improved with supervised fine-tuning by bootstrapping with LLM generated synthetic problem-code pairs. Chip-Chat [7] experimented with conversational flows to design and verify a 8-bit accumulator-based microprocessor with GPT-4 and GPT-3.5. Their findings showed that although GPT-4 produced relatively high-quality codes, it still does not perform well enough at understanding and fixing the errors. ChipEDA [8] proposed to use LLMs to generate EDA tools scripts. It also demonstrated that fine-tuned LLaMA2 70B model outperforms GPT-4 model on this task.

VIII. Conclusions

We explored domain-adapted approaches to improve LLM performance for industrial chip design tasks. Our results show that domain-adaptive pretrained models, such as ChipNeMo-13B-Chat, achieve similar or better results than their base models. Closing the gap with much more powerful LLaMA2 70B model on all three use cases: engineering assistant chatbot, EDA scripts generation, and bug summarization and analysis. Our future work will focus on further improving ChipNeMo models and methods to make them ready for production use.

IX. Acknowledgements

The authors would like to thank: NVIDIA IT teams for their support on NVBugs integration; NVIDIA Hardware Security team for their support on security issues; NVIDIA NeMo teams for their support and guidance on training and inference of ChipNeMo models; NVIDIA Infrastructure teams for supporting the GPU training and inference resources for the project; NVIDIA Hardware design teams for their support and insight.

X. Contributions

Mingjie Liu conducted DAPT and SFT model training.

Teo Ene, Robert Kirby developed inference and application evaluation infrastructure.

Chris Cheng developed RAG framework.

Nathaniel Pinckney collected and prepared data sets for training.

Rongjian Liang developed custom tokenizers.

Siddhanth Dhodhi, Ismet Bayraktaroglu, Himyanshu Anand, Eric Hill designed engineering assistant chatbot, provided domain instruction datasets, evaluation benchmarks, and conducted evaluation.

Parikshit Deshpande, Zhengjiang Shao, Kaizhe Xu, Jiashang Hu, Laura Dang, Xiaowei Li, Hao Liu, Ambar Sarkar developed engineering assistant chatbot application.

Sreedhar Pratty, Kishor Kunal, Varun Tej, Sumit Jain, Sujeet Omar, Pratik P Suthar, Hanfei Sun developed EDA scripts generation application, provided domain instruction datasets and evaluation benchmarks.

Bonita Bhaskaran, Arjun Chaudhuri, Sanmitra Banerjee developed bug summarization and analysis application, provided domain instruction datasets and evaluation benchmarks.

Brucek Khailany, Stuart Oberman, Sharon Clay, Sameer Halepete, Bryan Catanzaro, Jonah Alben, Bill Dally advised from AI research and hardware engineering perspectives.

Haoxing Ren designed and led the research.

References

[1] B. Khailany *et al.*, "Accelerating chip design with machine learning," *IEEE Micro*, vol. 40, no. 6, pp. 23–32, 2020.

[2] H. Ren and M. Fojtik, "Invited- nvcell: Standard cell layout in advanced technology nodes with reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[3] R. Roy *et al.*, "PrefixRL: Optimization of parallel prefix circuits using deep reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[4] W.-L. Chiang *et al.*, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," March 2023. [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/

[5] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[6] S. Thakur *et al.*, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[7] J. Blocklove *et al.*, "Chip-chat: Challenges and opportunities in conversational hardware design," 2023.

[8] Z. He *et al.*, "Chateda: A large language model powered autonomous agent for eda," 2023.

[9] S. Bubeck *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," 2023.

[10] S. Wu *et al.*, "Bloomberggpt: A large language model for finance," 2023.

[11] M. LLC. (2022) Biomedlm: a domain-specific large language model for biomedical text. [Online]. Available: https://www.mosaicml.com/blog/introducing-pubmed-gpt

[12] M. Liu *et al.*, "VerilogEval: evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.

[13] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," *ICLR*, 2023.

[14] S. Gururangan *et al.*, "Don't stop pretraining: Adapt language models to domains and tasks," 2020.
[15] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.
[16] E. J. Hu *et al.*, "Lora: Low-rank adaptation of large language models," *CoRR*, vol. abs/2106.09685, 2021. [Online]. Available: https://arxiv.org/abs/2106.09685
[17] L. Gao *et al.*, "The pile: An 800gb dataset of diverse text for language modeling."
[18] D. Kocetkov *et al.*, "The stack: 3 tb of permissively licensed source code," 2022.
[19] A. Köpf *et al.*, "Openassistant conversations – democratizing large language model alignment," 2023.
[20] J. Wei *et al.*, "Finetuned language models are zero-shot learners," 2022.
[21] V. Sanh *et al.*, "Multitask prompted training enables zero-shot task generalization," 2022.
[22] D. Hendrycks *et al.*, "Measuring massive multitask language understanding," 2021.
[23] M. Chen *et al.*, "Evaluating large language models trained on code," 2021.
[24] F. Koto, J. H. Lau, and T. Baldwin, "IndoBERTweet: A pretrained language model for Indonesian Twitter with effective domain-specific vocabulary initialization," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Nov. 2021, pp. 10 660–10 668.
[25] O. Kuchaiev *et al.*, "Nemo: a toolkit for building ai applications using neural modules," 2019.
[26] M. Shoeybi *et al.*, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.
[27] T. Dao *et al.*, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems*, 2022.
[28] A. Chowdhery *et al.*, "Palm: Scaling language modeling with pathways," 2022.
[29] Z. Ji *et al.*, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571730
[30] L. Wang *et al.*, "Text embeddings by weakly-supervised contrastive pre-training," *arXiv preprint arXiv:2212.03533*, 2022.
[31] L. Gao *et al.*, "Tevatron: An efficient and flexible toolkit for dense retrieval," 2022.
[32] B. Rozière *et al.*, "Code llama: Open foundation models for code," 2023.
[33] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.
[34] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: http://arxiv.org/abs/1908.10084
[35] R. Pope *et al.*, "Efficiently scaling transformer inference," 2022.
[36] R. Y. Aminabadi *et al.*, "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," 2022.
[37] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," 2022.
[38] W. Xiong *et al.*, "Effective long-context scaling of foundation models," 2023.
[39] R. Taylor *et al.*, "Galactica: A large language model for science," 2022.
[40] A. Lewkowycz *et al.*, "Solving quantitative reasoning problems with language models," 2022.
[41] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.
[42] S. Borgeaud *et al.*, "Improving language models by retrieving from trillions of tokens," 2022.
[43] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, p. 333–389, apr 2009. [Online]. Available: https://doi.org/10.1561/1500000019
[44] V. Karpukhin *et al.*, "Dense passage retrieval for open-domain question answering," 2020.
[45] G. Izacard *et al.*, "Unsupervised dense information retrieval with contrastive learning," 2022.
[46] W. Shi *et al.*, "Replug: Retrieval-augmented black-box language models," 2023.
[47] G. Izacard *et al.*, "Few-shot Learning with Retrieval Augmented Language Models," 2022. [Online]. Available: http://arxiv.org/abs/2208.03299
[48] O. Ram *et al.*, "In-context retrieval-augmented language models," 2023.
[49] S. Zhou *et al.*, "Docprompting: Generating code by retrieving the docs," 2023.
[50] R. Rafailov *et al.*, "Direct preference optimization: Your language model is secretly a reward model," 2023.
[51] Y. Dong *et al.*, "Steerlm: Attribute conditioned sft as an (user-steerable) alternative to rlhf," 2023.
[52] H. Pearce, B. Tan, and R. Karri, "Dave: Deriving automatically verilog from english," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, ser. MLCAD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 27–32. [Online]. Available: https://doi.org/10.1145/3380446.3430634
[53] "Beautiful Soup," https://www.crummy.com/software/BeautifulSoup/, accessed: 10 Oct 2023.
[54] K. Sakaguchi *et al.*, "Winogrande: An adversarial winograd schema challenge at scale," *arXiv preprint arXiv:1907.10641*, 2019.
[55] R. Zellers *et al.*, "Hellaswag: Can a machine really finish your sentence?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.
[56] P. Clark *et al.*, "Think you have solved question answering? try arc, the ai2 reasoning challenge," 2018.
[57] G. Lai *et al.*, "Race: Large-scale reading comprehension dataset from examinations," 2017.

APPENDIX

A. Data Collection Process

Collection was implemented with a set of shell and Python scripts, designed to identify relevant design data and documentation, convert them to plain text if applicable, filter them using basic quality metrics, compute a checksum for precise file deduplication, and compress them for storage. The collection flow did not use off-the-shelf LLM-specific scraping and collection scripts, as we aimed to minimize space requirements through in-situ data collection of internal data sources (both networked file systems and internal web applications). For file system-based collection, data was kept in-place while being filtered for quality, instead of storing additional sets of raw data locally.

The design and verification data collection encompassed a variety of source files, including Verilog and VHDL (RTL and netlists), C++, Spice, Tcl, various scripting languages, and build-related configuration files. Data from internal web services were gathered through both REST API calls and conventional crawling, with HTML formatting being removed using the open-source BeautifulSoup [53] Python library in both instances to minimize inadvertent removal of coding examples, at the cost of introducing more boiler plate navigation bars and other HTML page elements. Our data collection flow supported conventional documentation formats, including .docx, .pptx, and .pdf, using readily available Python conversion libraries and open-source tools.

As most internal data is believe to be of high quality, minimal filtering was applied: line count filtering was used to ensure that exceedingly large or small files were excluded, and files were sorted into broad categories of manually written versus tool-generated.

B. Domain Adaptive Pretraining (DAPT)

In this section we present detailed results on our domain adaptive pretrained models. We also detail our ablation experiments on domain adaptive pretraining.

DAPT Hyperparameters: Details presented in Table VI.

Hyperparameters	Value
Context Window	4096
Global Batch Size	256 (128)
Optimizer	distributed_fused_adam
Weight Decay	0.01
Betas	0.9, 0.95 (0.9, 0.98)
Learning Rate	$5 \cdot 10^{-6}$
Scheduler	None

TABLE VI: DAPT and SFT hyperparameters, SFT values shown in parenthesis (if differs from DAPT).

Auto Eval Results: We present detailed results on auto evaluation benchmarks in Table VII and Table VIII. For simplicity, in the remainders of the section we present aggregated benchmark results for ablation studies:

- **Chip:** We report average results on in-domain Design, Scripting, Bugs, and Circuits benchmarks from Table III (5-shot).
- **MMLU:** We report the overall results on MMLU (5-shot) [22] a popular aggregated benchmark on a wide variety of subjects.
- **Reasoning:** We report average results on popular public benchmarks on common sense reasoning (0-shot), including Winogrande [54], hellaswag [55], ARC-easy [56], and RACE-High [57].
- **Code:** We report average pass-rate of coding benchmarks with greedy decoding, including HumanEval [23], VerilogEval-Machine [12], and VerilogEval-Human [12].

Tokenizer Augmentation: We experimented with DAPT using the original LLaMA2 tokenizer and the augmented tokenizer as described in Section III-A. Figure 11 depicts smoothed training loss for ChipNeMo with the original unmodified tokenizer. When compared with Figure 2, we observe that an augmented tokenizer has larger training loss upon initialization, due to added tokens never being observed during foundation model pretraining. Similar training loss is achieved for DAPT with 1 epoch.

Table IX presents aggregated auto evaluation benchmark results. We note that careful tokenizer augmentation and weight initialization only slightly impacts model performance on general academic benchmarks. DAPT significantly improved domain benchmarks with any tokenizer, including Verilog coding (no major difference in HumanEval). We conclude that augmenting the tokenizer comes with the benefit of improved tokenizer and training efficiency with no degradation on the models general language and domain capabilities.

Public Datasets Mix-in: As introduced in Section II-A we included public data in DAPT, sampled from commonly-used public datasets for foundation model pre-training. We primarily hoped that mixing in public data such as Wikipedia in DAPT could help "correct" disturbances brought by tokenizer augmentation and improve general natural language capabilities

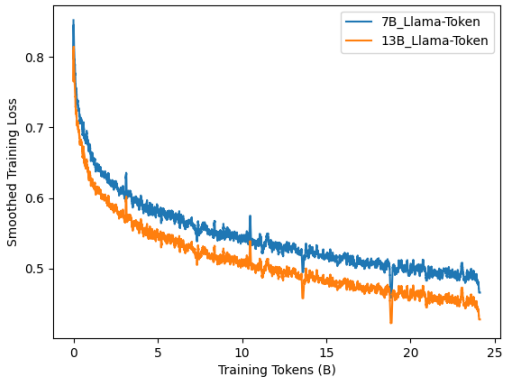


Fig. 11: Smoothed Training Loss with Original LLaMA2 Tokenizer.

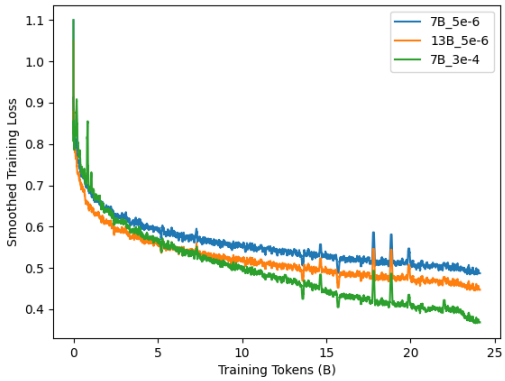


Fig. 12: Smoothed Training Loss with Larger Learning Rate. We include loss curves of suggested hyperparameters for comparison.

of models. We conducted another round of DAPT with tokenizer augmentation using only the domain data, training for the same number of steps equating to roughly 1.1 epoch of the data. We found that public data mix-in slightly improves results. We present detailed results in Table X.

Learning Rate: We experimented with employing a larger learning rate, inspired by the approach used in CodeLLaMA [32]. We use similar training hyperparameters as in Table XI. We use a cosine schedule with 200 warm-up steps, and set the final learning rate to be 1/30th of the peak learning rate of $3 \cdot 10^{-4}$. We use the same batch size and number of training steps as DAPT.

Figure 12 shows the training loss for ChipNeMo-7B with augmented tokenizers including public dataset mix-in. We observed large spikes in training loss at the initial training steps with the final training loss for 7B models to even be better than 13B original DAPT hyperparameters. However, we note substantial degradation across natural language benchmarks as shown in Table XII, including in-domain chip design. Coding capabilities improved as consistent with the findings of [32].

We highlight that our case differs from that in [32]. Although we also conduct "continued pretraining" initializing from pretrained checkpoints, we preferably want the model to maintain high degrees of performance on general capabilities, while

Model	Design	Scripting	Bugs	Circuits	MMLU [22]	Winogrande [54]	hellaswag [55]	ARC-e [56]	RACE-H [57]
LLaMA2-7B	41.1	42.0	42.2	47.9	45.7	68.9	75.6	73.5	46.2
ChipNeMo-7B	57.5	49.3	42.8	49.5	44.6	67.4	76.3	73.7	46.2
LLaMA2-13B	43.6	49.6	39.7	55.5	55.4	72.1	79.3	76.3	46.7
ChipNeMo-13B	67.9	56.3	50.1	56.8	53.4	71.1	80.3	76.7	46.1

TABLE VII: Auto Evaluation Results. We report academic benchmark results for LLaMA2 using proprietary evaluation methods. ChipNeMo models trained with tokenizer augmentation.

Model	HumanEval [23]	VerilogEval-Human [12]	VerilogEval-Machine [12]
LLaMA2-7B	14.0	3.8	24.5
ChipNeMo-7B	12.2	8.3	28.7
LLaMA2-13B	17.1	9.0	30.8
ChipNeMo-13B	17.7	22.4	43.4

TABLE VIII: Coding Evaluation Results. Showing pass-rate with greedy decoding. We report results for LLaMA2 using proprietary evaluation methods. ChipNeMo models trained with tokenizer augmentation.

Model	Tokenizer	DAPT	Chip	MMLU	Reason	Code
7B	Ori.	No	43.4	45.7	66.1	14.1
7B	Aug.	No	42.7	44.6	65.9	13.9
7B	Ori.	Yes	51.2	44.8	65.7	17.6
7B	Aug.	Yes	49.8	44.6	65.8	16.4
13B	Ori.	No	47.1	55.4	68.6	18.9
13B	Aug.	No	46.0	55.1	68.6	18.4
13B	Ori.	Yes	57.7	54.0	68.4	27.2
13B	Aug.	Yes	57.8	53.4	68.5	27.8

TABLE IX: Evaluation Results on ChipNeMo models with Different Tokenizers. `Aug.` indicate augmented tokenizer and `Ori.` indicate using LLaMA2 original tokenizer. Using augmented tokenizer without DAPT corresponds to the model initialization as in Section III-A.

distilling domain dataset information and knowledge (unseen in model pretraining) into model weights. In contrast, [32] use publicly available code data that predominantly lacks natural language elements, emphasizing their primary focus on coding-related tasks. We hypothesize that a smaller learning rate played a dual role for domain adaptation, facilitating the distillation of domain knowledge through DAPT while maintaining a balance that did not veer too far from the base model, thus preserving general natural language capabilities while significantly improving performance on in-domain tasks.

Parameter Efficient Fine-Tuning (PEFT): Parameter efficient fine-tuning freezes the pre-trained model weights and injects trainable parameters in smaller adapter models for efficient fine-tuning of downstream tasks. We explore the use of PEFT in DAPT using Low-Rank Adaptation (LoRA) [16]. Since our transformer layer implementation fuses KQV into a single projection, we add LoRA adapters for a single Low-Rank projection for each self attention layer in combined fashion. We experiment on LLaMA2-13B models with the original LLaMA2 tokenizer, using the same DAPT training setups in Table VI. We ran two experiments, introducing additional trainable parameters of 26.4 million (small) and 211.2 million (large) respectively.

Figure 13 shows the training loss curves of LoRA models and compares with full parameter training. For both LoRA models, the loss quickly converges and stops decreasing beyond a certain point. Table XIII reports the evaluation results on LoRA models. Both LoRA models significantly underperforms full parameter training on in-domain chip design tasks. LoRA models improve in chip design tasks compared to their non-DAPT counterparts, with the larger model exhibiting slightly better (but non significant) results.

Public	Chip	MMLU	Reason	Code
No	56.9	53.0	67.5	24.1
Yes	57.8	53.4	68.5	27.8

TABLE X: Ablation on Public Dataset Mix-in with ChipNeMo-13B. Public data mix-in slightly improves results.

Hyperparameters	Value
Context Window	4096
Global Batch Size	256
Optimizer	distributed_fused_adam
Weight Decay	0.01
Betas	0.9, 0.95
Learning Rate (lr)	$3 \cdot 10^{-4}$
Scheduler	CosineAnnealing
Warmup Steps	200
min_lr	$1 \cdot 10^{-5}$

TABLE XI: Training Hyperparameters with Larger Learning Rate. We adopt similar parameter as to [32].

Based on the results, we hypothesize that the observed phenomenon can be attributed to the imperative need for a sufficiently trainable number of model parameters capable of accommodating the substantial volume of information for DAPT. Additionally, it indicates that PEFT models with limited trainable parameters are susceptible to encountering the challenge of catastrophic forgetting [33].

C. Retrieval Model Training

Manually generating training samples is very effort intensive, so we elected to implement a process to generate them automatically. Since we are using contrastive learning to fine-tune our model, each sample requires a set of both positive passages and negative passages, particularly hard negatives to maximize the accuracy.

1) Dataset Sampling Procedure: Figure 14 describes the steps taken to generate a sample:

- Step 1: Randomly select a passage from the document corpus
- Step 2: Use a language model (Vicuna) to generate a valid query from the passage
- Step 3: Use a pre-existing retrieval model (sentence transformer) to fetch the top-N passages from the document corpus for the query where each passage is a potential hard-negative

Learning Rate	Chip	MMLU	Reason	Code
$5 \cdot 10^{-6}$	49.8	44.6	65.8	16.4
$3 \cdot 10^{-4}$	25.5	26.6	49.8	18.1

TABLE XII: Ablation on Learning Rate with ChipNeMo-7B. A larger learning rate significantly degrades performance on all language related tasks but slightly improves coding.

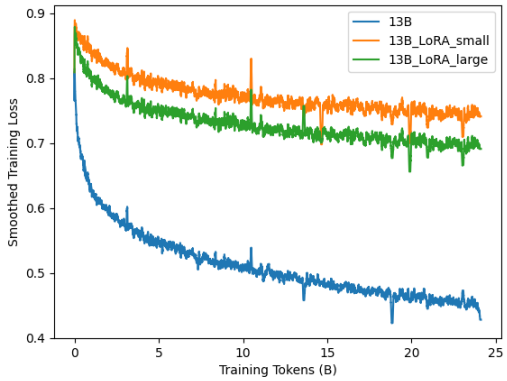


Fig. 13: Smoothed Training Loss of LoRA [16]. 13B corresponds to full parameter DAPT.

- Step 4: It is possible that some of the fetched passages are actually positive, so use the same language model to filter out the positive passages
- Step 5: If there are not enough negative passages after this filtering process, supplement with random passages from the corpus

For our initial research we used Vicuna [4] and Sentence Transformer [34]; however, they can easily be replaced with LLaMA2 [5] and BM25 [43] respectively to produce a retrieval model that is commercially viable.

2) Hit Quality Comparison: Not all hits are created equal. The passage in the Spec example below clearly and completely answers its query. The passage in the Build example contains the answer; however, more context is required to answer the query.

Spec Example: Hit passage clearly answers the query.

Query:
What is the XXX buffer in the YYY used for?

Hit Passage:
Since the memory subsystem is partitioned, YYY may receive read responses whose order is different from their requests. ZZZ requires strict in-order per-thread processing, thus YYY reorders the read responses to the request order before sending them back to ZZZ. YYY re-orders the read responses in a read return reorder buffer (XXX) to guarantee the ordering.

Answer: re-orders the read responses.

Build Example: Additional information is required to fully answer the query. Such as: What is a DL? How do we know Arch-Build-Hotseat-XXX is a DL?

Parameters	Chip	MMLU	Reason	Code
None	47.1	55.4	68.6	18.9
26.4M	49.0	55.0	68.2	13.0
211.2M	49.6	54.2	68.6	15.3
13B	57.7	54.0	68.4	27.2

TABLE XIII: Evaluation Results on LoRA Models. First column indicate number of trainable parameters. None indicates LLaMA2-13B model without DAPT. 13B indicates full parameter training.

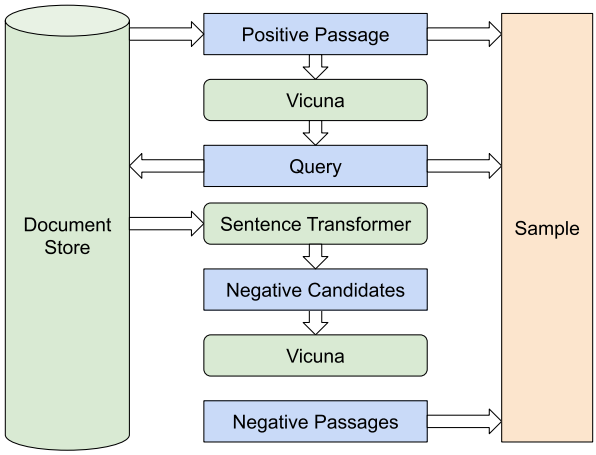


Fig. 14: Sample Generation For Retrieval Model Training

Query:
What is the support DL for XXX build issues?

Hit Passage:
Tree Setup
Working in the XXX Mainline explains initial tree setup and build steps
Build
Arch-Build-Hotseat-XXX - Hotseat support for XXX build issues
YYY build failures

D. Additional Evaluation Data

Table XIV shows the evaluation data for all models on the engineering assistant chatbot application.

Table XV shows our evaluation results for all models on the EDA script generation task.

Table XVI shows our evaluation results for all models on the bug summarization and analysis task.

Model	Domain SFT	Hit	Miss	ALL
LLaMA2-13B-Chat*	No	2.13	2.80	2.33
ChipNemo-13B-Chat	No	4.64	5.40	4.88
ChipNemo-13B-Chat	Yes	4.66	6.44	5.21
LLaMA2-13B-Chat* + RAG	No	7.68	3.72	6.46
ChipNemo-13B-Chat + RAG	No	7.86	5.48	7.12
ChipNemo-13B-Chat + RAG	Yes	7.86	6.36	7.40
LLaMA2-70B-Chat	No	2.36	2.32	2.35
LLaMA2-70B-Chat + RAG	No	8.38	5.20	7.40

TABLE XIV: Engineering Assistant Chatbot Human Evaluation

Model	Tool1 (Python)			Tool2 (Tcl)
	Automatic (Easy)	Automatic (Medium)	Human (Hard with Context)	Human (Hard with Context)
LLaMA2-13B-Chat*	7%	4%	15%	6%
ChipNeMo-13B-Chat (noDSFT)	26%	35%	39%	33%
ChipNeMo-13B-Chat	44%	39%	41%	36%
LLaMA2-70B-Chat	3%	4%	49%	18%

TABLE XV: EDA Script Generation Evaluation.
Scored binary for Automatic Evaluation and 0-100% for Human Evaluation.

Model	Technical Summary	Managerial Summary	Task Assignment
LLaMA2-13B-Chat*	3.79	3.15	3.00
ChipNeMo-13B-Chat (noDSFT)	4.61	3.45	3.43
ChipNeMo-13B-Chat	4.61	4.24	3.61
LLaMA2-70B-Chat	4.92	4.58	3.91

TABLE XVI: Bug Summarization and Analysis Evaluation. Likert scale 1-7.

E. Examples

1) *Engineering Assistant Chatbot:*

Question:
Write a System Verilog assert to check if signal A is true within 5 cycles of signal B being true

Answer:
Sure, here's an example System Verilog assert to check if signal A is true within 5 cycles of signal B being true:

```
// Define the signals
logic a, b;
// Define the assert
assert property (@(posedge clk) a |-> ##[1:5] b);
```

In this example, the assert property is used to check if signal a is true, then signal b must be true within 5 cycles. The assert will fail if b is not true within 5 cycles of a being true.

Question:
why do we add retiming stages?

Answer:
Retiming stages are added to improve the timing of the design and to account for the physical distance between different parts of the chip. By adding retiming stages, we can reduce the amount of time it takes for signals to travel between different parts of the chip, which can help to improve the overall performance of the design. Additionally, retiming stages can also help to reduce the amount of noise and interference that can occur on the signals, which can further improve the performance of the design.

Question:
How do I run multiple seeds of random tests using testgen framework?

Answer:
You can run multiple seeds of random tests using the Testgen framework by using the *rand_tgen* target. This target allows you to specify a variety of options for the stopping criteria of random runs using the same Testgen testlist(s).

```
nvmk -c . rand_tgen TESTLIST=my_testlist MON=1
↪  TGEN.LOOP=50
```

This command will run 50 iterations of the *my_testlist* testlist, each with a different seed. You can specify the number of iterations and the seed mode using the TGEN.LOOP and SEED options, respectively.

2) *EDA Script Generation:* Some function names and commands are obfuscated.

Question:
write a code in $TOOL1$ to print the number of flop cells in the give rectangle (0,0,100,100)

Answer:

```
# Sure, here is a TOOL1 code to print the number of
↪  flop cells in the given rectangle
↪  (0,0,100,100):

# This code gets the top_ref of the design
top_ref = env().get_top_reference()
# This code gets the cells of the rectangle in the
↪  top_ref
cells = env().layout().cells_in_rectangle( top_ref,
↪  rectangle(0, 0, 100, 100), hierarchical)
# This code gets the cells that are flop cells
flop_cells = cells.type("is_flop_cell")
# This code prints the number of flop cells
print(flop_cells.size())
```

Question:
Write a $TOOL2$ tcl proc to find intersecting logic from pins '$pinA' to pin '$pinB'.

Answer:

```
#Sure, here is the code:
proc myfunc {pinA pinB} {
   # Get all the pins in the fanout of the given pin
   set fanout_pins [fanout -from $pinA]
   # Get all the pins in the fanout of the given pin
   set fanin_pins [fanin -to $pinB]
   # Get the intersection of the two collections
   set intersection [common_logic $fanout_pins
   ↪  $fanin_pins]
   # Return the intersection
   return $intersection
}
```

*3) **Bug Summary and Analysis**:* Usernames, chip names and paths are obfuscated.

Prompt:
BugId : 002
Synopsis : Request power/current measurement on fabrics using fabric test module
Module : Silicon - DFT
Description :
Hi all,
We want to evaluate silicon current (power) consumed by fabric macros in a chip. This will help us correlate this power to pre-Si power estimates.
Here are the conditions we want to look at for this data:
Fabric test module activity: 0/25/50/75/100%
Corners:
typical_low_voltage
typical_high_voltage
We want to isolate fabric current (power) as much as possible, so we want to enable only the logic needed for fabric test modules and the fabrics themselves.
If this information can be broken down per module (there are 6 in the chip), that will help.
Severity : 5-Performance
Priority : 5-Normal
Comments :
#1 : Author : User A
Is this a requirement only for current or for future chips as well ?
#2 : Author : User B
There will likely be interest in future chips to perform this exercise as we push the boundaries of fabric power density for better area.
#3 : Author : User A
What is needed in this bug ?
@User B It is not clear.
Can you list the specific needs with timelines and priority

Technical Summary:
In NVIDIA's internal bug management system, a bug has been filed regarding the power and current measurements of the fabrics in the latest chip. The request is to evaluate the silicon current (power) consumed by the fabric macros in the latest chip and to correlate this power to pre-Si power estimates. The conditions for the measurements are the fabric test module activity at 0/25/50/75/100% and the corners are typical_low_voltage and typical_high_voltage. The data needs to be broken down per module (there are 6 in the latest chip). Only the logic needed for fabric test modules and the fabrics themselves need to be enabled.

Managerial Summary:
User A asks User B if the need for evaluating the silicon current of fabric macros in a chip is only for that specific chip or for future ones as well. User B replies that there will likely be interest in future chips to perform this exercise as they push the boundaries of fabric power density for better area. User A then asks what is needed in this bug and asks for specific needs with timelines and priority.

Task Assignment:
User B