Universiteit Leiden
The Netherlands

Bachelor Computer Science & Data Science and Artificial Intelligence

SuperCode

A Step Towards Sustainable Algorithms Using SLMs

Pieter Stevens

Supervisors:
Rob V. van Nieuwpoort & P. Chris Broekema

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

21/05/2025

Abstract

This research investigates whether SLMs (Small Language Models) can be used to aid in co-design of sustainable algorithms in a sustainable way. In particular, five open source SLMs, including normal and coder variants, are tested on their ability to fit on 8GB of VRAM (Video Random Access Memory), and on metrics (error-to-line ratio, edit distance from a baseline program, and the BLEU score from multiple baselines) related to the general quality of generated code, specifically of correlator algorithms in 64-bit RISC-V assembly. Findings show that 4-bit and 8-bit quantised SLMs can generally fit on 8GB of VRAM, and that higher quantisation is worth the loss of precision as it allows the usage of larger models. The results also seem to indicate a possible correlation between base VRAM usage and memory growth with increasing context length. Additionally, special coder models seem to outperform general text-generation models, with one of the coder SLMs even getting close in code quality metrics to state-of-the-art LLMs. On top of this, a simple error feedback loop was shown to improve output across the board. These insights suggest the possibility of a step towards more sustainable algorithms using SLMs.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Research questions . . . 2

1.3 Thesis overview . . . 2

2 Background **2**

2.1 Radio astronomy . . . 2

2.2 RISC-V . . . 3

2.3 Edit distance . . . 3

2.4 BLEU score . . . 4

2.5 LLM code generation . . . 4

2.6 Quantisation . . . 5

3 Methods **5**

3.1 Experimental setup . . . 5

3.2 Pre-processing . . . 7

3.3 Experiment 1 . . . 7

3.4 Experiment 2 . . . 9

3.5 Experiment 3 . . . 9

3.6 Experiment 4 . . . 9

4 Results **9**

4.1 Memory usage . . . 9

4.2 Error ratio . . . 12

4.3 Edit distance . . . 16

4.4 BLEU score . . . 21

5 Discussion **23**

6 Related Work **25**

7 Conclusions and Further Research **29**

References **31**

A Prompts **31**
A.1 Experiment 2 31
A.2 Experiment 3 33
A.3 Experiment 4 37

1 Introduction

1.1 Motivation

SuperCode (Sustainability PER AI-driven CO-DEsign) is an approach proposed by P. Chris Broekema and Rob V. van Nieuwpoort that aims to prioritise sustainability in the development of algorithms through co-design of hardware and software using LLMs (Large Language Models) [2]. Modern data-intensive science has enabled vast discoveries, but at the cost of enormous, possibly unconstrained, compute resource usage, which can no longer be tolerated given the current climate crisis. Because of this, the SuperCode initiative argues that scientific gains must be weighed against their environmental footprint [2]. The aforementioned usage of LLMs can be extremely beneficial for this use case, as integrating them into this co-design loop, iterating on generated code with both man and machine, can greatly accelerate the evaluation of new hardware and software combinations for these data-intensive applications, such as radio astronomy algorithms (e.g. correlators and polyphase filters) [2].

However, a challenge with this approach is that state-of-the-art LLMs are extremely resource-hungry. The training of models is often the most intensive step, but even inference, in this case running a trained model to generate code, can consume vast amounts of energy. Recent studies show that 60% of Google's AI-related energy usage is caused by inference, and that the inference of OpenAI's ChatGPT consumes hundreds of MWh per day [19]. Nevertheless, while larger models often require more resources, they also generally yield better results, leading to a trade-off between performance and environmental cost [6]. Additionally, while 16GB of VRAM (Video Random Access Memory) is recommended for day-to-day use, most consumer GPUs only have half of that. This means that of these state-of-the-art LLMs, often only those on the smaller side can run entirely locally. This shifts the focus to state-of-the-art SLMs (Small Language Models) that fit within this 8GB limit. An added benefit of this focus on SLMs is that smaller models have a reduced inference cost resulting in a smaller carbon footprint [6].

Another challenge is that of emergent architectures. There is quite a lot of Python, C, and x86 assembly code available online. This means that most LLMs are bound to be knowledgeable in these languages. However, new hardware could bring entirely new instruction sets that are unknown to most models. RISC-V, for example, is an open standard ISA (Instruction Set Architecture) that features a reduced instruction set, as opposed to a "complex" instruction set seen in CISC. RISC-V only recently (2024) made the inclusion of V extensions mandatory, which are extensions that allow for vectorised arithmetic, the added efficiency being of utmost importance for radio astronomy algorithms such as correlators [2, 15], but the recency could cause a gap regarding these extensions in the knowledge base of SLMs.

In summary, our research explores whether state-of-the-art SLMs can generate/translate correlator algorithms in/to meaningful RISC-V assembly. For this purpose, the focus will be on five SLMs of varying nature and size, exploring which of these can be used to best apply SuperCode, and also how they can be used optimally for this purpose.

1.2 Research questions

RQ1 Can state-of-the-art SLMs serve as a more sustainable alternative to LLMs in the SuperCode initiative?

- RQ 1.1 How do recent SLMs of differing size, domain (normal/coder), architecture, and quantisation compare in the amount of VRAM they require when generating correlators in RISC-V 64-bit assembly?
- RQ 1.2 How do recent SLMs of differing size, domain (normal/coder), architecture, and quantisation compare in code quality metrics such as error ratio, edit distance, and BLEU score when generating correlators in RISC-V 64-bit assembly?
- RQ 1.3 To what end are recent SLMs capable of one-shot generation of correlators in RISC-V 64-bit assembly?

The results show that 4-bit and 8-bit quantised SLMs can generally fit on 8GB of VRAM, as long as the models are less than or equal to eight billion parameters and context length is bounded to $\sim$4k tokens, and that higher quantisation is worth the loss of precision as it allows the usage of larger models, which come with a larger knowledge base. However, VRAM usage varies significantly with architecture as well. Additionally, special coder models seem to outperform general text-generation models in our chosen code quality metrics.

Lastly, although they were close, the SLMs were not really capable of producing a correct correlator in one-shot generation, but this was also the case with state-of-the-art LLMs. A simple error feedback iteration was shown to improve the output of SLMs across all code quality metrics.

These insights suggest the possibility of a step towards more sustainable algorithms using SLMs.

1.3 Thesis overview

This section provides an overview of the thesis; Section 2 includes relevant background information; Section 3 formulates research setup and methodology, and various experiments; Section 4 describes the outcome of these experiments; Section 5 interprets these results; Section 6 discusses other work in this field and how it relates to this research, and Section 7 draws conclusions, as well as discussing possible future research. This bachelor thesis was carried out at the Leiden Institute of Advanced Computer Science (LIACS) under the supervision of Rob V. van Nieuwpoort and P. Chris Broekens, authors of the SuperCode intitiative.

2 Background

2.1 Radio astronomy

Modern radio telescopes consist of stations, which are arrays of many separate antennas in order to increase the resolution of incoming signals, such as ASTRON's Low-Frequency Array (LOFAR) [18]. In order to extract images from these incoming data at this higher resolution, a signal processing unit is used that computes correlation among antenna pairs (baselines) that either belong to the same station or entirely different stations. This signal processing unit is known as a correlator, and it multiplies and accumulates these signals to form visibilities for each baseline [18]. This captures their spatial Fourier components, which describe the signal using trigonometric functions, essentially allowing for the combination and analysis of signals from multiple antennas, thereby increasing the resolution of the data.

Naturally, these higher resolution data come with the caveat that a vast amount of data needs to be processed, so optimisation of correlator code is key for signal processing [2]. Correlators perform many simple 1×1 (autocorrelation) or 2×1 (cross-correlation) multiplications followed by summation across antenna pairs, and these repeated instructions across multiple values lend themselves well to vector instructions, such as V extensions in RISC-V [18].

2.2 RISC-V

As mentioned in Section 1.1, RISC-V is an open standard ISA that features a reduced instruction set, which only recently mandates the presence of V extensions [15]. Because of its simpler design, RISC-V's individual instructions are often more efficient, especially power-wise, than those in more complex architectures, which is very useful for low-power embedded devices and when the focus lies on power efficiency in general [22]. Additionally, the open nature of RISC-V promotes adoption and research. Its V extensions follow the principle of Single Instruction Multiple Data (SIMD), providing homogeneous arrays on which a single instruction can be efficiently mapped. Additionally, RISC-V's vectors are dynamic instead of being fixed like most architectures, allowing for more versatility and portability as the related registers can be resized to fit requirements [1].

The most important vector instructions provided by V extensions in the case of correlators are fadd.s (single-precision floating point addition of two vector registers), fmul.s (single-precision floating point multiplication of two vector registers), and fmadd.s (fused multiply-add of three vector registers) which wraps both into a single instruction [15]. As mentioned, correlators involve cross-correlation, which is akin to a lot of additions and multiplications, so these instructions severely speed up this data-intensive computation.

2.3 Edit distance

When evaluating generated code, one common metric is the edit distance [3]. This is the minimum number of character edits needed to transform two arbitrarily sized strings into each other, and so it incorporates the difference between two strings, but also the difference between their lengths. This distance is often taken to be relative to a desired or baseline string. So a small distance indicates that only a small number of changes need to be made to transform a string into the other, regardless of whether the issue is length or content.

Edit distance is particularly useful for code quality evaluation because it mitigates the impact of superficial differences (e.g. naming or formatting), while still penalising larger deviations [3].

2.4 BLEU score

Another common metric is the BLEU score, which is specifically devised to evaluate the quality of machine translations [3, 14]. It is a single real number between zero and one describing the quality of the translation, also known as the hypothesis, in comparison to given (human) reference translations; e.g. the BLEU score of a generated correlator compared to a dataset of correlators would be closer to one the more correct the correlator is according to the dataset. BLEU does this through comparing matches between chunks of words (n-grams) across references, adjusted to take order and repetition into account, as well as the total number of words in the generation [14]. This is quite different from the edit distance, which works on the level of characters and only takes one baseline into consideration.

A popular stabilised implementation of the BLEU metric is that of SacreBLEU, with a major benefit aside from stability being a reduced need for pre-processing as a result of lowering the influence of minor changes, such as casing and unnecessary token splits, on the resulting score [10].

2.5 LLM code generation

The way modern LLMs generate text is by taking a context and appending their own tokens based on the likelihood of those tokens appearing at that point in the context. Naturally, code generation works the exact same way.

LLMs are often transformer networks with many layers and parameters, which allows the processing of sequential tokens (typically words, but the tokenizer of a model decides this) in parallel, allowing them to better harness the full power of GPUs [20]. Each layer of these transformers computes self-attention, where each token generates keys and values that are used to give weight to all the tokens in the context. These weights can be looked up using the keys, allowing for fast callback at any point in the input [20]. This allows the model to consider multiple different sections of the input tokens before picking an output token. The process of choosing a single output token involves aforementioned sampling of an output distribution, meaning that LLMs are essentially sophisticated word predictors which can be applied to generating code as well. Each decoding step in an LLM involves sampling the next token from the model's output distribution, but the sampling strategy applied can make a big difference in output quality [11].

The temperature setting in LLMs dictates the randomness of this decision, with temperatures closer to zero being more coherent while high temperatures allow for more creative generations at the cost of coherence. A low temperature is therefore desirable in LLMs optimised for code generation, but this could lead to repetition in the output, with the LLM potentially getting stuck into forever selecting the same tokens [11]. Min-p sampling is a way to mitigate this, while also allowing for creative, but coherent outputs [13]. It does this by scaling the decision threshold according to the probability of the most likely token, so an LLM will consider more tokens when it is less confident in this top choice. But despite this increased coherence, hallucinations in the output can still be problematic.

Decoding by Contrasting Layers (DoLa) is a decoding strategy that aims to reduce hallucinations in pre-trained LLMs [4]. It does this by allowing LLMs to either improve performance in short answer tasks when prioritising the later layers, or long answer tasks when prioritising the earlier layers, using the differences between these layers to adjust the output distribution to be more factually correct [4]. This is because the earlier layers often primarily encode general facts, which are important for long answers, while later layers capture more specific facts for concise answers.

2.6 Quantisation

The parameters used by LLMs are generally 16-bit floating-point numbers, but this can cost a lot of VRAM when a model has many parameters. In this case, the parameters can be quantised to use 8-bit floats, or even 4-bit floats, essentially cutting the memory usage in half in the 8-bit case at the cost of precision [6]. However, this cost can be mitigated through NormalFloats (NF4s), which are designed to be optimal for describing weights that follow a normal distribution, allowing for the usage of larger models with minimal accuracy loss [5].

3 Methods

3.1 Experimental setup

To explore the capabilities of SLMs with respect to correlators in RISC-V assembly, we picked five recent open source SLMs, mostly from Eugene Yan's OpenLLMs repository [21]. All models picked are instruct models, because code generation involves generating answers to questions, not simply extending text. The chosen models can be seen in Table 1, and they were chosen with the distinctions in mind of (1) small and larger, with Phi-3 Mini and Phi-3 Small; (2) normal and coder, with Qwen2.5 and Qwen2.5 Coder; according to the research questions. And of course comparing different large models and different coder models. Incidentally, because the models are almost all a different size (∼7B), it will be interesting to see whether that has any influence on results.

Additionally, because the ∼7B parameters models would not fit within 8GB of VRAM, they are all quantised to use 4-bit parameters, specifically NF4 parameters to mitigate precision cost of quantisation, while Phi-3 Mini was quantised to use 8-bit parameters, so there is also the distinction of (3) smaller, but less quantised, and larger, but more quantised, as well as a different datatype being used for 4-bit quantisation. Regarding DoLa, priority is given to the earlier layers in order to improve performance on long answer tasks, given that assembly output is often very long. This and applying a min-p sampling of 0.05 allows us to keep the temperature low (0.2) to prioritise coherence over creativity because assembly is often very strict and specific. The seed of PyTorch is set to zero to ensure reproducibility.

For this research, we conducted four experiments that evaluate the one-shot code generation performance of the five models on writing correlators in 64-bit RISC-V assembly from scratch, and translating existing C implementations to 64-bit RISC-V assembly, with the context growing throughout the experiments. The metrics that the models will be evaluated on for each experiment are (1) their memory usage, (2) the number of errors present in the generated assembly per line, and (3) the edit distance between the generated assembly and the compiled assembly of a baseline correlator, of which the results are in Section 4.1, Section 4.2, and Section 4.3 respectively.

All assembling and compilation will be performed on the banana PI BPI-F3, which is actual 64-bit RISC-V hardware. A promising backup is the open source Spike RISC-V ISA Simulator, which supposedly supports the full instruction set, including V extensions [16].

Name	Abbreviation	Parameters	Quantisation
Phi-3-Mini-4K-Instruct	P3M	3.8B	8
Phi-3-Small-8K-Instruct	P3S	7.39B	4
Qwen2.5-7B-Instruct	Q25	7.62B	4
Qwen2.5-Coder-7B-Instruct	Q25C	7.62B	4
Deepseek-Coder-7B-Instruct-v1.5	DC15	6.91B	4

Table 1: Chosen SLMs. The abbreviations are the names used in all figures, parameters refers to the number of parameters the model has, with B denoting "billion", and quantisation refers to the number of bits the floating-point parameters of the models were reduced to.

Memory usage

This refers to the peak VRAM usage measured during the experiment. This can be measured by running "nvidia-smi" in the terminal after pausing the program once the model and tokens are loaded into VRAM and the model has produced output tokens. Note that there is idle usage of 69MiB (mebibytes), which means there is actually only 8119MiB available out of 8188MiB, so these 69MiB are subtracted from both sides for all memory measurements.

Error ratio

The error count is measured by running "clang -c <filename>.s 2> tmp.txt; wc -l tmp.txt" in the terminal, which prompts Clang to assemble the program, after which the assembler outputs its errors (if any) into "tmp.txt", of which the lines are counted by "wc -l". Clang is used instead of GCC, as the LLVM assembler would report more errors than the GNU assembler in one of Qwen2.5's outputs that incorrectly mixed assembly with parts of the original C code. The LLVM assembler always uses three lines to report each error, so the number of errors has to be divided by three. Additionally, an extra error is added because the ratio of errors across lines is a more interesting metric than the plain error count, and this cannot be visualised in the case of no errors.

Edit distance

As explained in Section 2.3, the edit distance is a common metric of similarity between two strings, which can be applied to the produced outputs by measuring the distance to a desired or baseline output. For this we use the edit_distance function from the nltk Python package. These baselines are compilations of C source code, from Rob V. van Nieuwpoort's many-core-correlators repository, compiled with "clang -S -c -Oz <filename>.cc", to output assembly optimised for size [17]. Note that the file extension "cc" denotes a C++ source file, not C, but while the codebase utilises C++ features, the correlators themselves adhere to the C subset. Clang is used instead of GCC because Clang produces less irrelevant information in the assembly. While edit distance should mitigate the impact of superficial differences, all output from the models and these compiled correlators are put through a pre-processing step as seen in Section 3.2.

BLEU score

Similar to the edit distance, we will use the BLEU score to compare the produced outputs with baseline outputs. However, BLEU has the ability to take multiple references into consideration (Section 2.4). So, instead of taking a different baseline correlator per experiment, we take all the correlators from the same many-core-correlators repository in common (1x1, 2x1, 2x2, and "reference") as the references list for all score calculations [17]. These references are obviously not human-written like intended for BLEU, but it is close enough as the llvm-project is maintained by humans, although machine learning is already being used for optimisation purposes [12]. The corpus_bleu function from the sacrebleu Python package is then used to calculate the scores.

The same command is used for compilation of outputs, and pre-processing is also applied (Section 3.2), but the outputs are also split into lines. This is because every line of assembly can be treated as a sentence, which BLEU works best on in this case. Splitting the file based on assembly tokens was too strict on the output, resulting in unusable scores ($\sim 1e-231$) as few n-grams would actually match to anything.

3.2 Pre-processing

To normalise assembly outputs with the aim of more meaningful results, a small number of simple rules are applied to assembly output as a pre-processing step. The rules are as follows:

1. Remove comments and unnecessary directives, and strip unnecessary whitespace
2. Rename .global to .globl. and .section .<name> to .<name> (.rodata should also become .data). Section .data should be above .text
3. Rename all defined data and the usage to .D<i>, all functions to .F<i>, and all labels to .L<i>, in order of declaration. Symbols that are not defined are not renamed
4. Change brackets to parentheses and convert numbers to decimal
5. Fill in constructs like "sizeof(float)"

The before and after of this pre-processing can be seen for a simple program in Figure 1 and Figure 2 respectively.

3.3 Experiment 1

The first experiment prompts the models to write a 1x1 correlator in 64-bit RISC-V assembly from scratch, without any context. The full prompt can be seen in Figure 3. The baseline correlator is the compiled and preprocessed 1x1 correlator from "common/cpu_correlator_1x1.cc" [17].

```
.section .text
.global hello
hello:
    addi sp, sp, -16
    sd   ra, 8(sp)
    sd   s0, 0(sp)
    addi s0, sp, 16

    lui  a5, %hi(msg)
    addi a0, a5, %lo(msg)
    call puts

    li   a5, 0
    mv   a0, a5
    ld   ra, 8(sp)
    ld   s0, 0(sp)
    addi sp, sp, 16
    jr   ra
.section .rodata
msg:
    .string "Hello, world!"
```

Figure 1: A simple "Hello, world!" function in 64-bit RISC-V assembly

```
.data
.D0:.string "Hello, world!"
.text
.globl .F0
.F0:addi sp,sp,-16
sd ra,8(sp)
sd s0,0(sp)
addi s0,sp,16
lui a5,%hi(.D0)
addi a0,a5,%lo(.D0)
call puts
li a5,0
mv a0,a5
ld ra,8(sp)
ld s0,0(sp)
addi sp,sp,16
jr ra
```

Figure 2: The same simple "Hello, world!" function seen in Figure 1, but with the pre-processing steps specified in Section 3.2 applied

```
[
    {"role": "system", "content": "You are an expert writer of correlators in 64-bit
        RISC-V assembly. You explain everything and give a whole program."},
    {"role": "user", "content": "Write a 1x1 correlator in 64-bit RISC-V assembly."},
]
```

Figure 3: Prompt for experiment 1. Messages from the "system" role act like a screenwriter, dictating what the "assistant" role should be and do. Messages with the "user" role are questions from the user

3.4 Experiment 2

This experiment tasks the models with translating a 1x1 correlator in C to 64-bit RISC-V assembly, still without context. The full prompt can be seen in Appendix A.1. The baseline correlator is the same as in experiment 1.

3.5 Experiment 3

Similarly to the first experiment, the models are tasked with writing a correlator from scratch. However, the models are provided with the assembly of the 1x1 correlator and are tasked to write a 2x1 correlator from scratch. The full prompt can be seen in Appendix A.2. The baseline correlator is the compiled and preprocessed 2x1 correlator from "common/cpu_correlator_2x1.cc" [17].

3.6 Experiment 4

This experiment again tasks the translation capabilities of the SLMs, like in experiment 2. However, the models are provided the translation of the baseline 1x1 correlator, so the original source code and its assembly, and are tasked with translating a 2x1 correlator. The full prompt can be seen in Appendix A.3. The baseline correlator is the same as in experiment 3.

4 Results

4.1 Memory usage

Before any experiments are conducted, it is vital to know how much space each model takes up in VRAM without tokens and cache. This baseline can be seen in Figure 4. Interestingly, Qwen2.5 and Qwen2.5 Coder already take up ∼75% of the available VRAM, which could prove problematic as all tokens passed to and appended by the models will also be allocated to this memory for efficient access. Apart from that, memory usage is about as expected with the amount of parameters each model has (Table 1). The only outlier is that DeepSeek Coder v1.5 takes up 0.3% more memory than Phi-3 Small while it has 0.48B less parameters and they are quantised the exact same (Q8), so the number of parameters does not paint the entire picture.

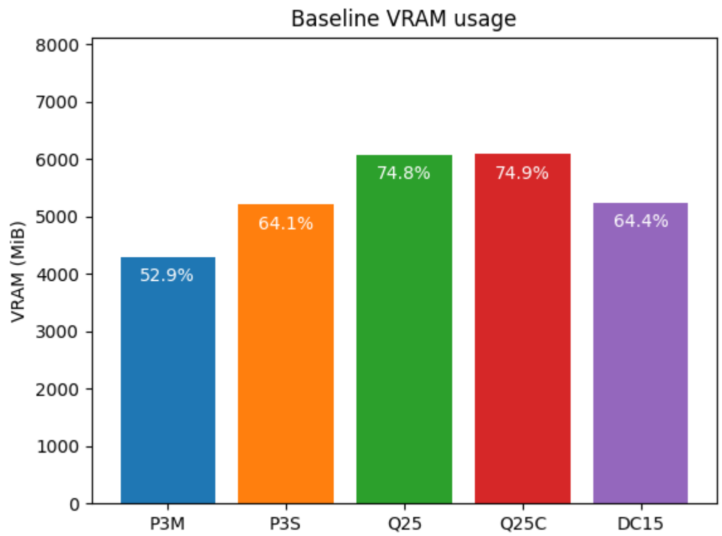

Figure 4: Baseline VRAM usage in mebibytes of all the models

Experiment 1

Now that there is an actual passing of context and question, every model has an increase in VRAM usage, as is visible in Figure 5, which is expected as tokens are passed and produced. There is not much to say yet about the memory usage of these models, except that Qwen2.5 and Qwen2.5 Coder have an equivalent increase, while also seeming to have the slowest increase of memory usage with context length.

Experiment 2

The slower increase in memory usage of Qwen2.5 and Qwen2.5 Coder becomes more apparent during experiment 2, with every other model taking up more than 70%, while Qwen2.5 and Qwen2.5 Coder featured a base memory usage around this range, yet they still have not exceeded 80%, as is visible in Figure 6.

Experiment 3

This experiment shows an even larger increase in memory usage, again especially from Phi-3 Mini, which has the highest increase at 40.2% over the baseline, Phi-3 Small and DeepSeek Coder v1.5. At this point, these models even surpass Qwen2.5 and Qwen2.5 Coder in memory usage, as seen in Figure 7. This is interesting as their baseline usage started far above the other models. Note that Qwen2.5 appears to have a larger increase in memory usage than Qwen2.5 Coder, but that is only because Qwen2.5 would fall into repetition and thus needed extra tokens to ask the model to explain its decisions.

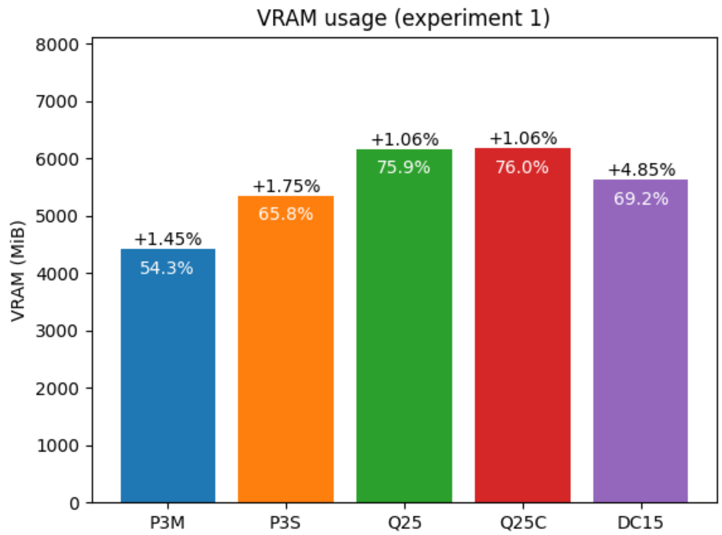

Figure 5: VRAM usage in mebibytes of all the models during experiment 1. The percentages in black are the increases in usage compared to their respective baselines in Figure 4

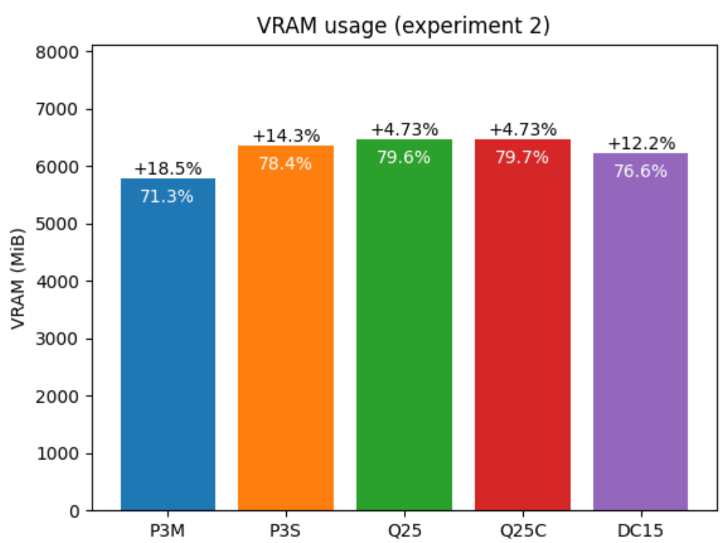

Figure 6: VRAM usage in mebibytes of all the models during experiment 2. The percentages in black are the increases in usage compared to their respective baselines in Figure 4

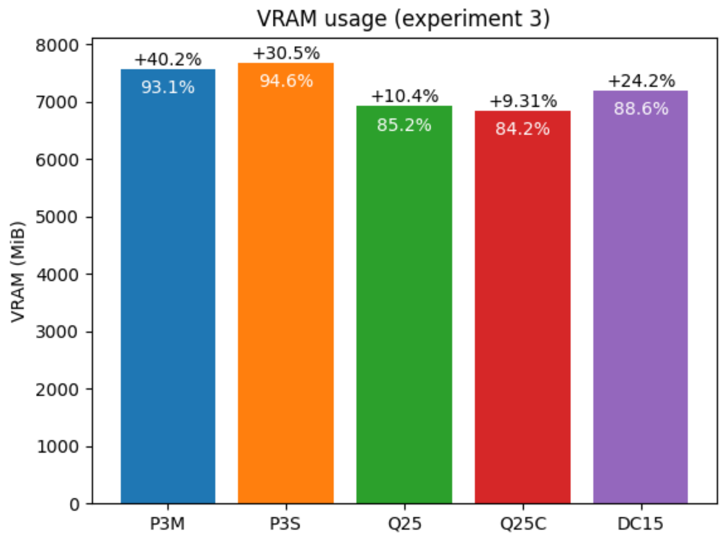

Figure 7: VRAM usage in mebibytes of all the models during experiment 3. The percentages in black are the increases in usage compared to their respective baselines in Figure 4

Experiment 4

This experiment proved to be too much for the 8GB of VRAM as all models exhausted available memory, requiring left-truncation of the input tokens to 4k total, which is the smallest window size among the models. However, even with truncation applied Phi-3 Mini, Phi-3 Small, and DeepSeek Coder v1.5 continued to exhaust available VRAM. Qwen2.5 and Qwen2.5 Coder did not run out of memory and so they are still usable on 8GB of VRAM with 4k tokens.

This slower memory curve of Qwen2.5 and Qwen2.5 Coder is especially apparent in Figure 9, which shows the trend of VRAM usage of the models over the experiments. Qwen2.5 and Qwen2.5 Coder expectedly have a curve that is less steep than those of the other models, with DeepSeek Coder v1.5, Phi-3 Small, and Phi-3 Mini following in that order. Interestingly, it seems like the models that have a lower baseline memory usage seem to have a steeper curve than the models with a higher baseline.

4.2 Error ratio

Experiment 1

When asked to write a 1x1 correlator from scratch, the number of errors the models produce is relatively low, but the models also produce <= 20 lines of assembly while the baseline features 138 lines. This can be seen in Figure 10. So none of the models came close to writing a full 1x1 correlator from scratch in 64-bit RISC-V assembly, which is to be expected. Phi-3 Mini delivers the best performance in this metric, as it produced the largest output that assembled.

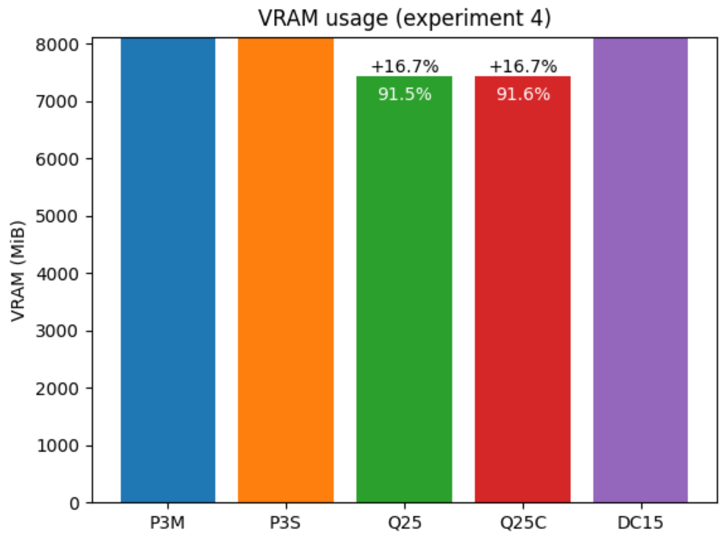

Figure 8: VRAM usage in mebibytes of all the models during experiment 4. P3M, P3S, and DC15 ran out of memory even when left-truncated to 4k tokens, which is the maximum context length of P3M and DC15's window. The bars for these models are still included for illustration purposes. The percentages in black are the increases in usage compared to their respective baselines in Figure 4

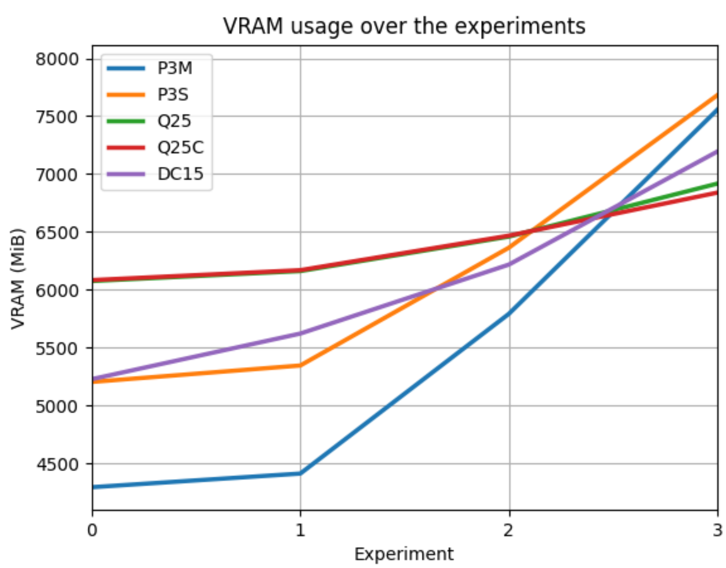

Figure 9: The progression of VRAM usage over the experiments, which steadily increase context length. Experiment 0 refers to the baseline, and experiment 4 is not featured as all models ran out of memory

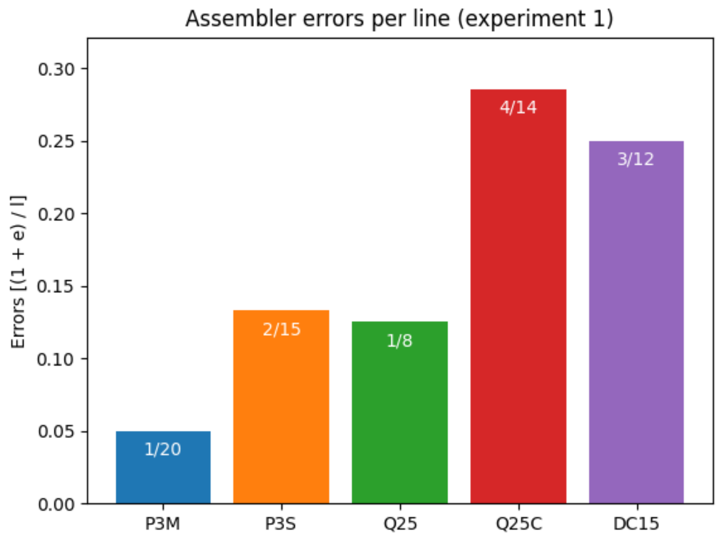

Figure 10: The amount of errors (+1 to account for no errors) per line when assembling the output of experiment 1. The baseline 1x1 correlator has 138 lines of assembly

Experiment 2

Translating a 1x1 correlator from C to 64-bit RISC-V assembly instead of writing from scratch appears to tremendously increase the number of lines in the output, which can be seen in Figure 11. The outlier is DeepSeek Coder v1.5, which instead shows a decrease from 12 to seven (Figure 10). With this increase in output length, the number of errors also shows a significant increase. Interestingly, Phi-3 Small features the highest error-to-line ratio, while Phi-3 Mini again has the lowest error-to-line ratio. It is only surpassed by DeepSeek Coder v1.5 in this category, which produced output that assembled, but with only seven lines of output it is far from an actual correlator.

Experiment 3

Another experiment where the models have to write a correlator from scratch, which is clear from the results in Figure 12: < 50 lines while the baseline is at 232, but a low error count across the board. Although Phi-3 Mini appears to once again have the best performance, Qwen2.5 is not far behind. More importantly, Qwen2.5 Coder might have a worse error-to-line ratio, but the output also contains significantly more lines, which is, of course, also a very important cornerstone of code quality.

Experiment 4

This experiment shows a drastic difference in results. Once again, Figure 13 shows a general increase in the number of lines, which of course comes with the increase in the number of errors as seen in the other translation task of experiment 2. Interestingly, Qwen2.5 Coder produced a staggering 145 lines of assembly without a single error. However, this experiment features a context in which the

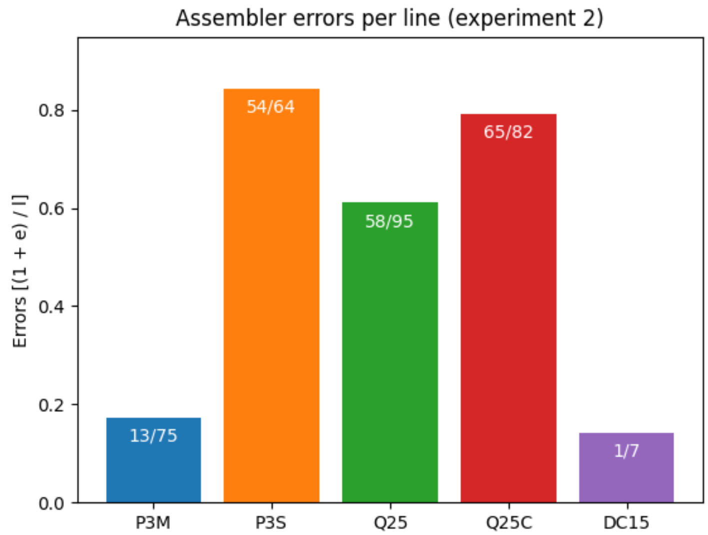

Figure 11: The amount of errors (+1 to account for no errors) per line when assembling the output of experiment 2. The baseline 1x1 correlator has 138 lines of assembly

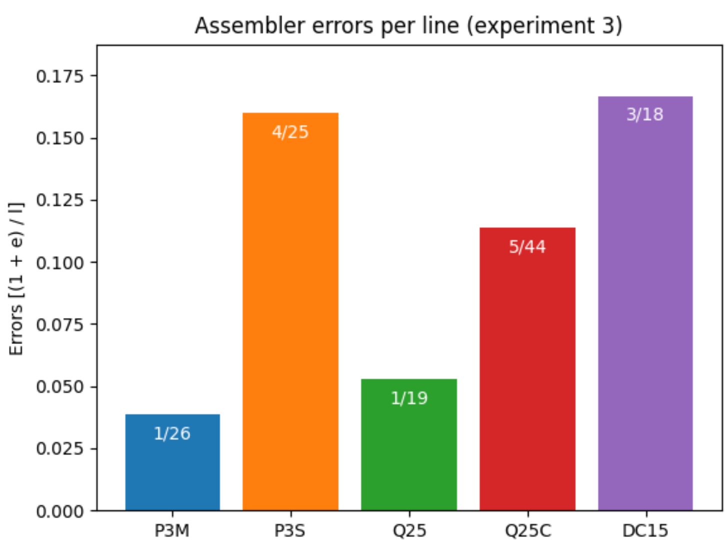

Figure 12: The amount of errors (+1 to account for no errors) per line when assembling the output of experiment 3. The baseline 2x1 correlator has 232 lines of assembly

translation of a 1x1 correlator is already given, and the number of lines corresponds more to that of the 1x1 correlator (138 lines) than the 2x1, and it seems like Qwen2.5 Coder generally did copy the 1x1 correlator's translation. Nevertheless, it actually made additions to the code and other changes without introducing errors, so it is still a good result, as the 2x1 correlator is quite similar in nature. DeepSeek Coder v1.5 also gives a notable performance, just with significantly less lines.

Figure 14 shows the total sum of errors and lines, leading to the average error-to-line ratios of each model. Although DeepSeek Coder v1.5 produced relatively few errors, it also produced the smallest number of lines. Phi-3 Small's ratio is especially poor, having produced 70 errors and only 118 lines in total. In contrast, Phi-3 Mini produced more lines, but significantly less errors. But most importantly, although Qwen2.5 and Qwen2.5 Coder produced the most errors, they also produced double the number of lines of Phi-3 Mini.

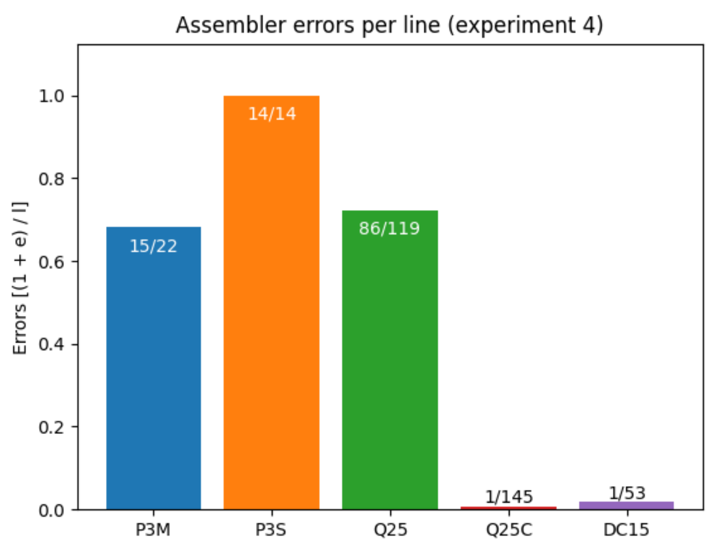

Figure 13: The amount of errors (+1 to account for no errors) per line when assembling the output of experiment 4. The baseline 2x1 correlator has 232 lines of assembly

4.3 Edit distance

Experiment 1

Figure 15 shows the edit distance between the output of the models and the baseline correlator. The average distance is rather high, and every model is almost equally distant, which is to be expected since the task involves writing a correlator from scratch without any additional context.

Experiment 2

In contrast to Figure 15, the results in Figure 16 instead show very different edit distances from the baseline correlator between model outputs. DeepSeek Coder v1.5 shows the largest distance

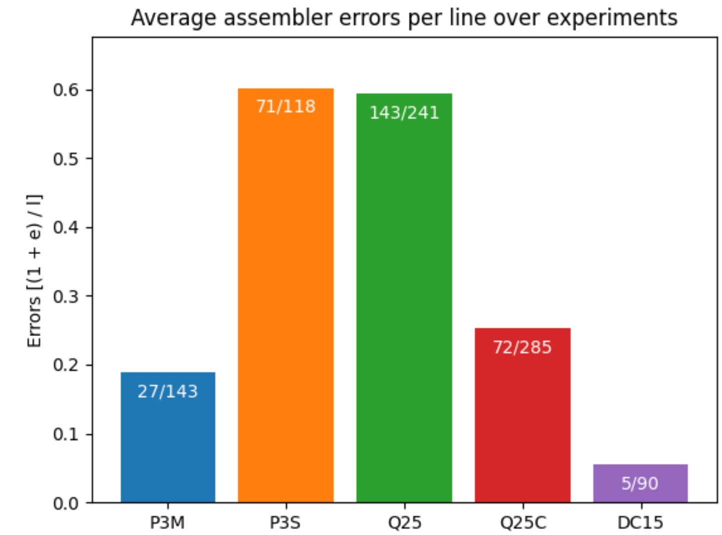

Figure 14: The average amount of errors (+1 to account for no errors) per line over the experiments

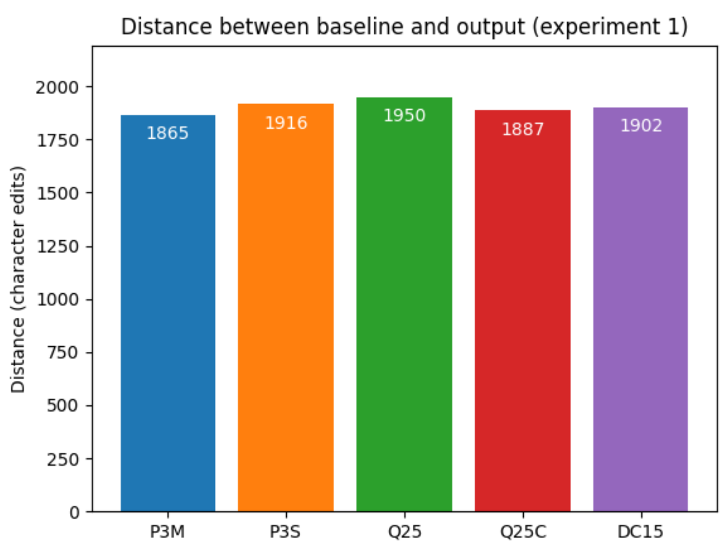

Figure 15: The edit distance between the baseline 1x1 correlator and the output of experiment 1

seen so far, which is not surprising since the output consisted only of seven lines (Figure 11), and Qwen2.5 shows the smallest distance seen so far, which could be because its output is the longest, but the assembly produced, despite errors, is not that far removed from the essence of a correlator. Qwen2.5 Coder has the second best result in this metric, possibly again because of output length. However, it does have multiple instances of the fmadd.s instruction, and it is actually the only model that produced this instruction throughout all the experiments.

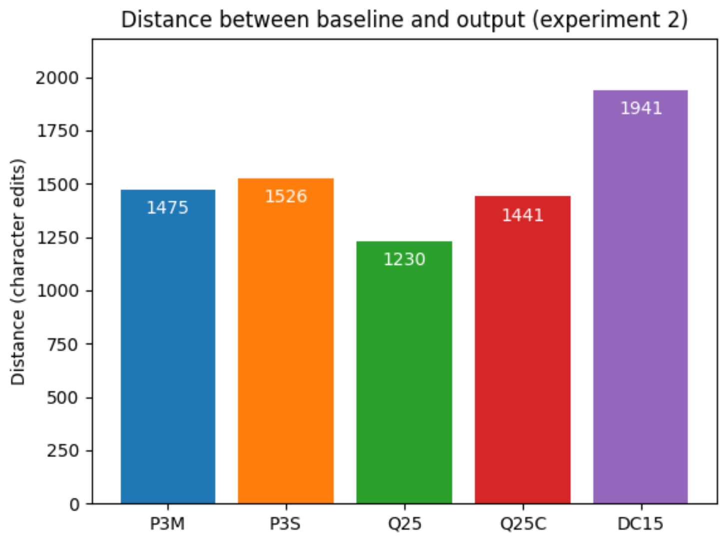

Figure 16: The edit distance between the baseline 1x1 correlator and the output of experiment 2

Experiment 3

Just like in experiment 1, every model is very distant from the desired output, which can be seen in Figure 17. The reasons for this are the same. However, because this writing-from-scratch task has actual context, the outputs are not equally distant. For instance, Qwen2.5 Coder's output is noticeably less distant than the output of the other models, which is likely again in part due to it being the longest output (Figure 12). However, the output performs vectorised addition and multiplication, and looks to be a shortened and simplified correlator with relatively few errors at that.

Experiment 4

Just like in experiment 2, the outputs of Qwen2.5 and Qwen2.5 Coder are most similar to the baseline correlator, as can be seen in Figure 18. Most importantly, Qwen2.5 Coder's output is still distant from the actual baseline correlator, which means it is highly unlikely that it produced a copy of the 1x1 correlator, which was theorised in Section 4.2. Figure 19 depicts the distances to the baseline 1x1 correlator as opposed to the 2x1 correlator, which shows for certain that Qwen2.5 Coder's output is far from a one-to-one copy of the 1x1 correlator.

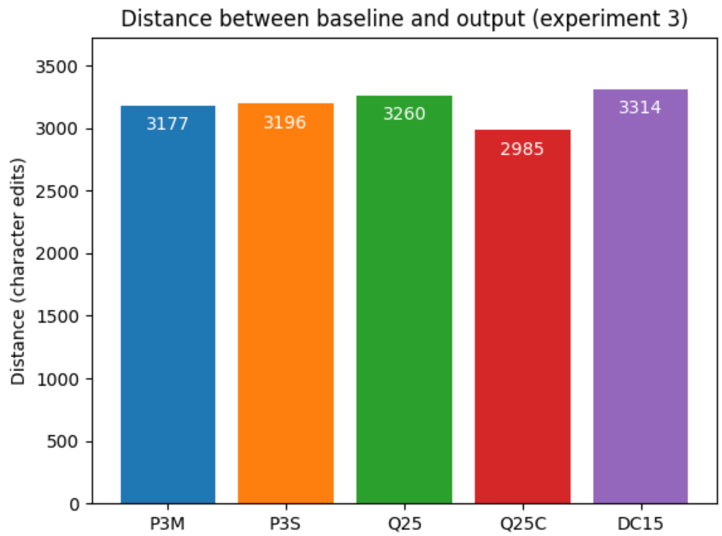

Figure 17: The edit distance between the baseline 2x1 correlator and the output of experiment 3

Figure 20 shows the average edit distance between the model outputs and their respective baseline correlators over the experiments, which shows roughly the same results seen before: the outputs of Qwen2.5 and Qwen2.5 Coder are generally much closer to the desired output.

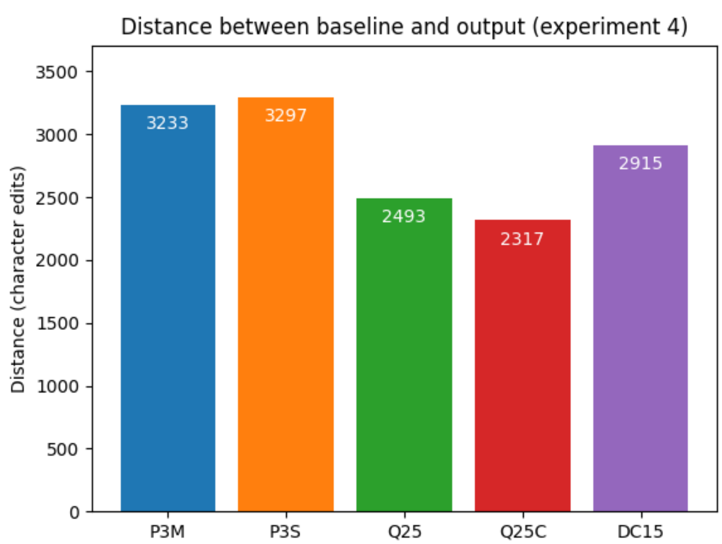

Figure 18: The edit distance between the baseline 2x1 correlator and the output of experiment 4

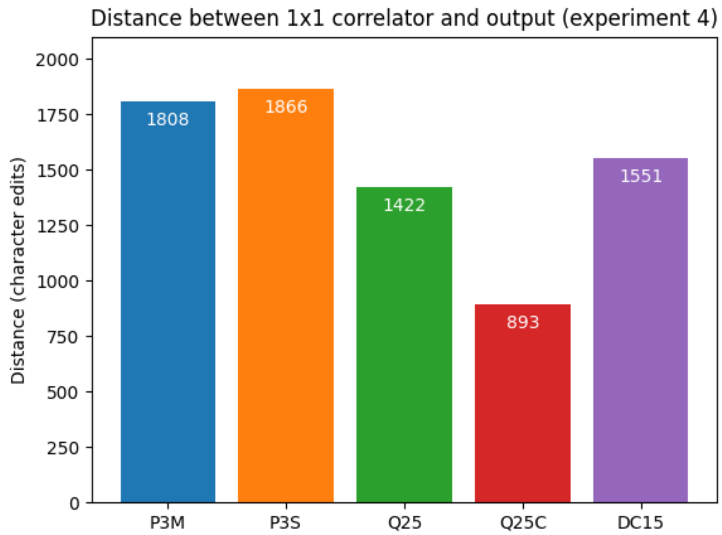

Figure 19: The edit distance between the baseline 1x1 correlator and the output of experiment 4

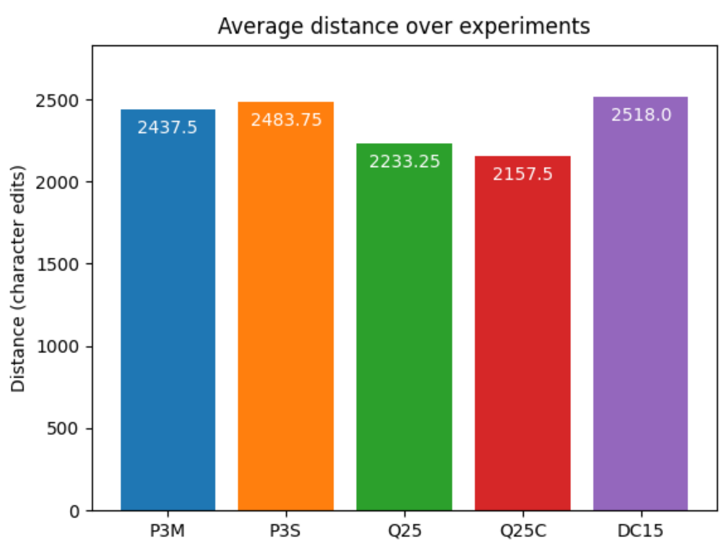

Figure 20: The average edit distance between the baseline correlators and the output over the experiments

4.4 BLEU score

Experiment 1

Figure 21 shows the BLEU score of every model's output in experiment 1 with respect to all baseline correlators. Although no output is of particularly high quality according to BLEU, the score of Qwen2.5's output is significantly higher than the rest. This is to be expected as the output actually compiled, and it is more similar to the references than Phi-3 Mini's output, even though it has far fewer lines (Figure 10). This is in contrast to Qwen2.5 having the largest distance to the baseline correlator in Figure 15.

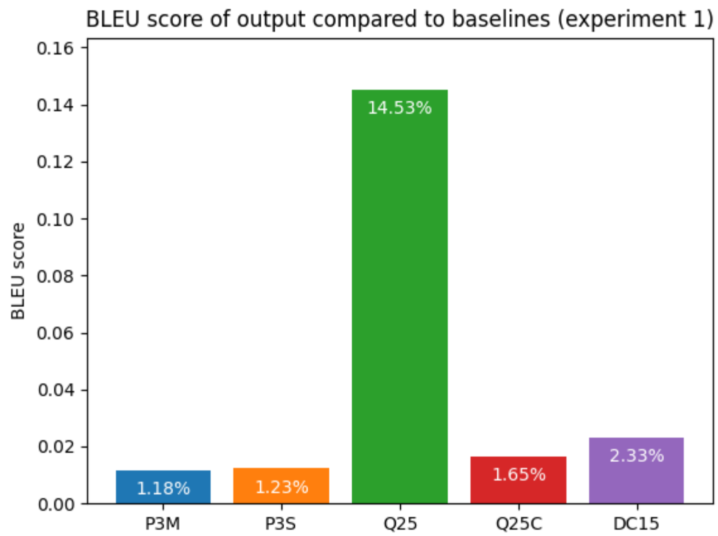

Figure 21: The BLEU score of the output of experiment 1 as hypothesis when given all baseline correlators as reference

Experiment 2

Figure 22 shows quite the difference from Figure 21, in that Qwen2.5's output is among the lower quality outputs according to BLEU, albeit being similar to that of Qwen2.5 Coder. In particular, Phi-3 Mini and DeepSeek Coder v1.5 take the lead, although they are still half of Qwen2.5's BLEU score in experiment 1. This could be due to experiment 2 being a translation task as opposed to writing from scratch, where the original C code and the rigid task of translating it causes the assembly output to deviate from the baselines.

Experiment 3

Experiment 3 continues the trend perceived in experiment 1 and 2, with Figure 23 showing higher BLEU scores than Figure 22. This suggests there is a possible connection between the nature of the task and the magnitude of the scores, which is interesting as the BLEU score is meant for evaluating the quality of translations, not generations that are written from scratch. However, this

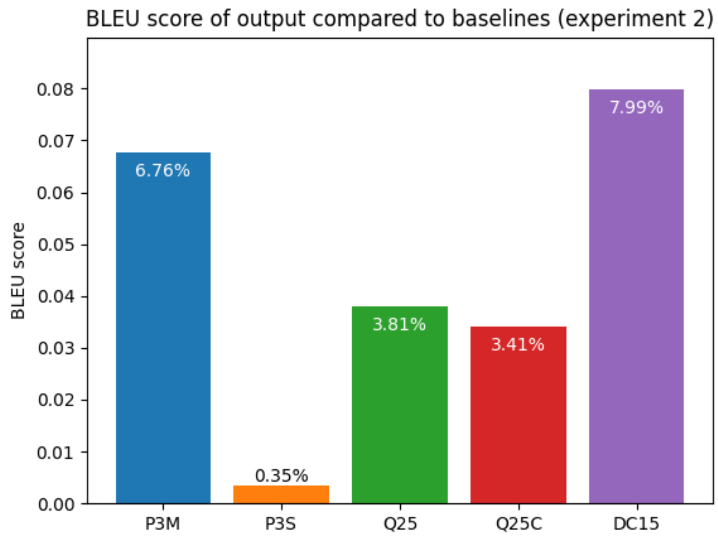

Figure 22: The BLEU score of the output of experiment 2 as hypothesis when given all baseline correlators as reference

experiment does show an increase over experiment 1's scores (Figure 21), and the only difference is an increase in context. So writing from scratch with relevant context potentially allows the model to perform a better "translation" compared to being forced to follow the original C code.

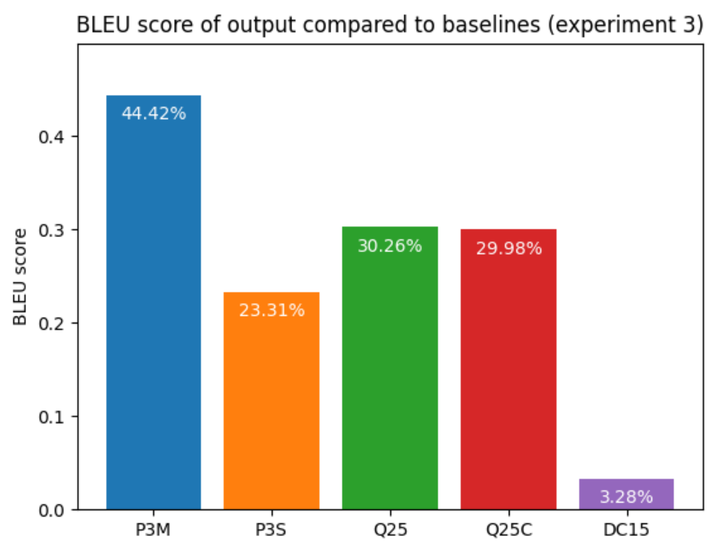

Figure 23: The BLEU score of the output of experiment 3 as hypothesis when given all baseline correlators as reference

Experiment 4

Although Figure 24 does not show the highest BLEU scores seen so far, Phi-3 Mini and especially Qwen2.5 Coder exhibit relatively high scores compared to Figure 22, even though this experiment is a translation task. This could mean that the trend we observed is a coincidence, but the rest of the models again feature lower scores.

Alternatively, it could indicate that Qwen2.5 Coder with its steadily increasing BLEU score over the experiments gets better at generation with increasing context regardless of the nature of the task, while Phi-3 Mini performs solidly throughout (Figures 21– 24). This is especially shown in Figure 25, where Phi-3 Mini has the highest average BLEU score of $\sim 20\%$ with Qwen2.5 Coder not far behind. Although this average score of Phi-3 Mini is less than half the highest score seen ($\sim 44\%$), indicating that the BLEU score fluctuates a lot.

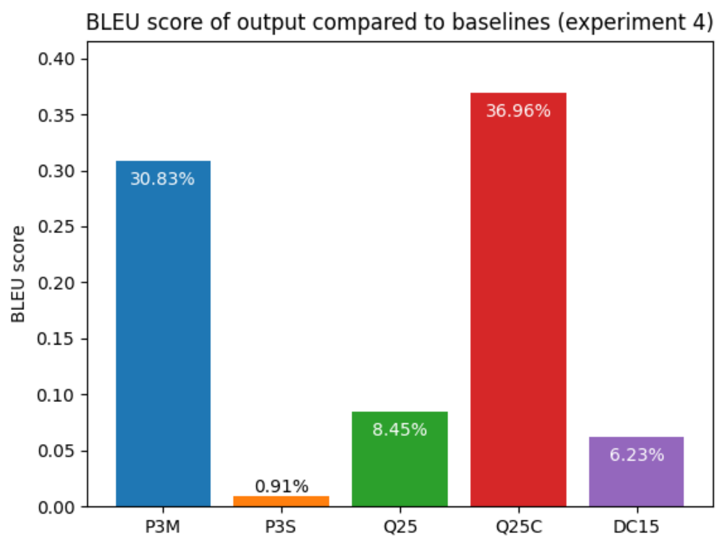

Figure 24: The BLEU score of the output of experiment 4 as hypothesis when given all baseline correlators as reference

5 Discussion

As mentioned in Section 4, Figure 9 shows differing memory usage for all the models over the experiments. Results suggest that Qwen2.5 and Qwen2.5 Coder have a slower memory curve than the other models while starting off with a higher base memory usage, even being surpassed in usage by smaller models. This could be because of the specific architecture of these models, as they both exhibit this same behaviour, but it could also be that the larger the model's base memory usage, the slower the memory usage increases with context size. Figure 9 shows this exact trend, with the models having a difference in the growth of VRAM usage according to the order of their starting sizes in memory. However, to fully conclude this, more in depth research will probably have to be conducted.

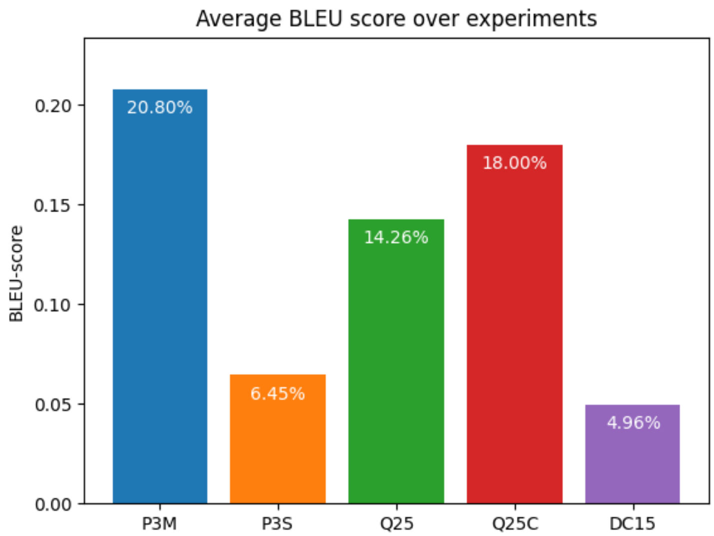

Figure 25: The averaged BLEU score of the output over the experiments as hypothesis when given all baseline correlators as reference

Regarding the other metrics, Phi-3 Mini has the smallest number of errors and a relatively large number of lines (Figure 14), so it is potentially better for from scratch writing (no context) and smaller programs, as exhibited across results, while Qwen2.5 and Qwen2.5 Coder produced more errors, but also a lot more lines of code, and so they might be preferable for translation as the output is generally more similar to the baseline (Figure 20), and with the better memory curve comes the added benefit of being able to take a larger context into consideration, allowing it to potentially be of better use in the co-design of algorithms. In this way, Qwen2.5 and Qwen2.5 Coder were the best performers, indicating a better performance with larger, more quantised models over smaller, less quantised models. Of the two, Qwen2.5 Coder's output looked to be the most correct in general, also often being more similar to the baseline and references (Figures 20, 25), indicating that coder models might be preferable for code generation, which is to be expected as these models exist with that specific task in mind.

Regarding the output, the trade-off of using SLMs instead of state-of-the-art LLMs seems clear, as none of the models produced a one-shot output that approached a functioning correlator, except for Qwen2.5 Coder in experiment 4 (Figure 13), but that was likely due to the context containing a successful translation. Nevertheless, it can be said that larger models, with their more vast knowledge base, contain more successful translations that might aid similarly. And while SLMs seem to not be up to par for one-shot generation, the results are still reasonably promising, as Figures 26– 28 show a comparison of performance between Qwen2.5 Coder, and OpenAI's o4-mini and DeepSeek's R1, which are state-of-the-art reasoning models. Qwen2.5 Coder is not too far off in edit distance to the baseline, competing with o4-mini in terms of error ratio, and even obtaining a higher average BLEU score across the experiments than both o4-mini and DeepSeek-R1 (although it is questionable whether this is actually a meaningful result due to the more lenient approach we took for calculating the BLEU score).

Of course, DeepSeek-R1's output is still a lot better as it features only two errors across 342 lines of assembly, but that is to be expected as it is a reasoning model with $\sim$ 90 times more parameters. A reasoning model can iterate upon its own output, while Qwen2.5 Coder's performance was truly one-shot. Of course, an interesting question would then be how much Qwen2.5 Coder could improve if it also iterated upon its output. A start to this can be seen in Figures 29– 31, where four iterations are run with only the errors outputted by Clang supplied as feedback in between. The first iteration's prompt is that of experiment 2 (Appendix A.1), but no pre-processing is ever applied, and temperature is set to 0.3 as well as a repetition penalty of 1.1 being applied as Qwen2.5 Coder would fall into repetition despite the measures taken in our research. This is obviously too simple of a feedback loop, and the code metrics plateau after this fourth iteration, but it shows promising results as the upper bound of Qwen2.5 Coder's performance has definitely not been reached. All of this indicates that SLMs could definitely have their use in SuperCode.

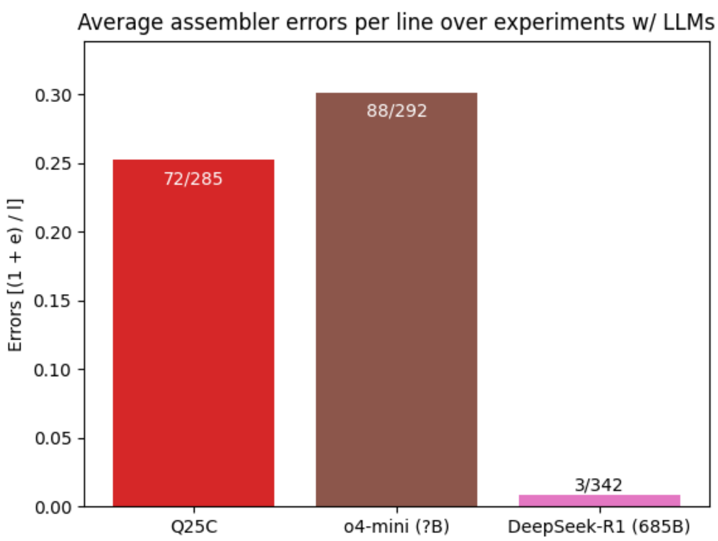

Figure 26: The average amount of errors (+1 to account for no errors) per line over the experiments of Qwen2.5 Coder and example LLMs. The exact number of parameters of o4-mini is unspecified, but the theorised range is 8B to 60B

6 Related Work

Jiang et al. provide a survey of LLMs specifically for code generation [9]. This work discusses data preparation, model architectures, evaluation metrics, and applications. The evaluation metrics discussed are the most important aspect of this survey. It mentions measuring progress using HumanEval and MBPP which are datasets of Python code that generated code can be compared

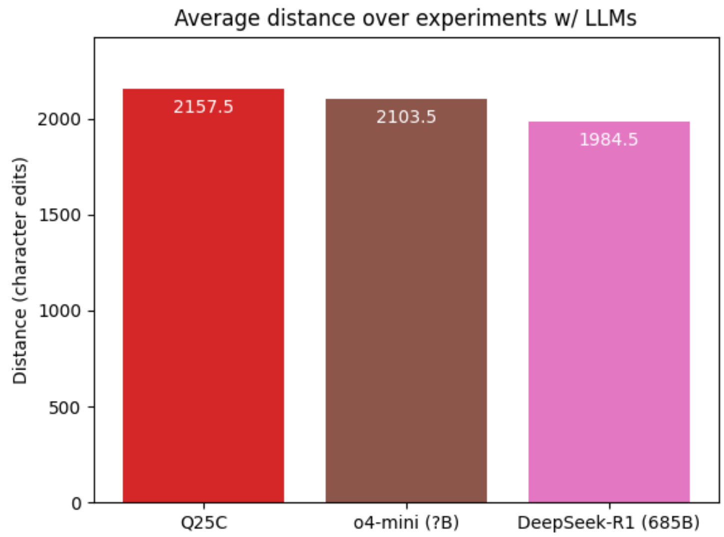

Figure 27: The average edit distance between the baseline correlators and the output over the experiments of Qwen2.5 Coder and example LLMs. The exact number of parameters of o4-mini is unspecified, but the theorised range is 8B to 60B

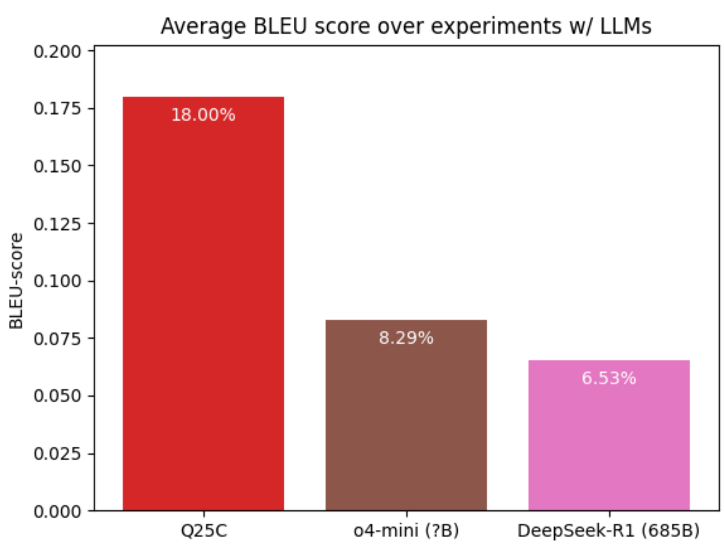

Figure 28: The averaged BLEU score of the output of Qwen2.5 Coder and example LLMs over the experiments as hypotheses when given all baseline correlators as reference. The exact number of parameters of o4-mini is unspecified, but the theorised range is 8B to 60B

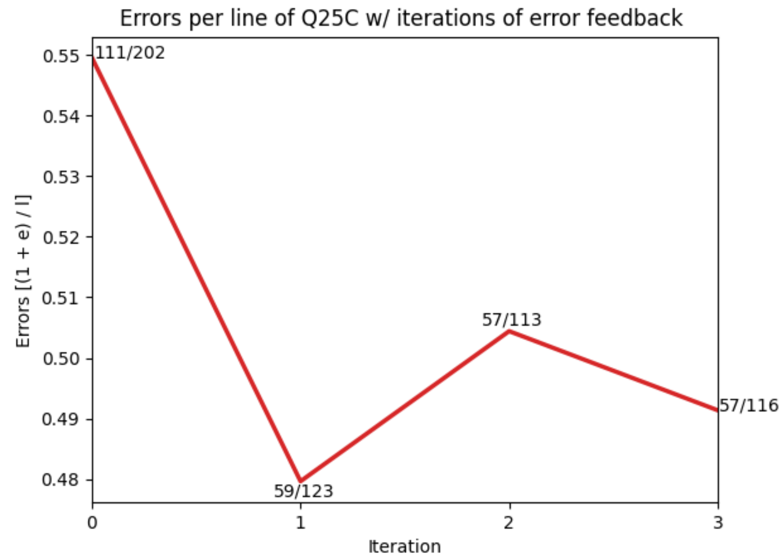

Figure 29: The amount of errors (+1 to account for no errors) per line of Qwen2.5 Coder's output over simple Clang error feedback iterations

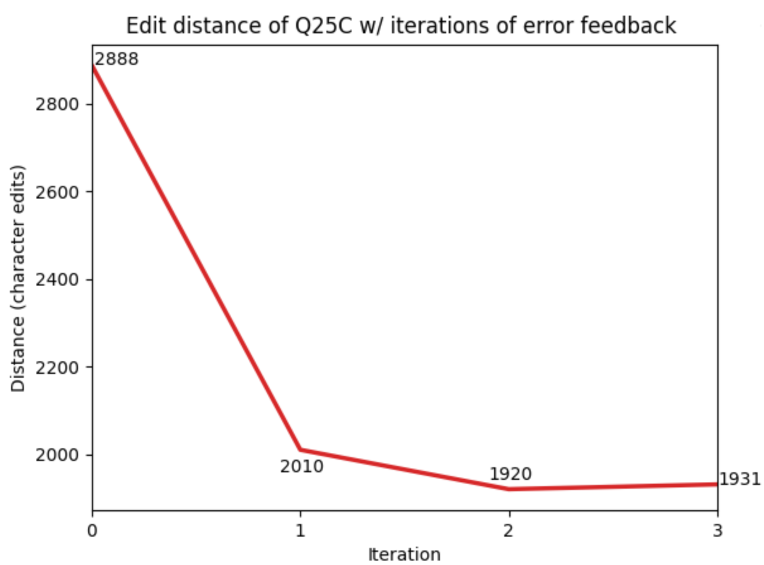

Figure 30: The edit distance between the baseline 1x1 correlator and Qwen2.5 Coder's output over simple Clang error feedback iterations

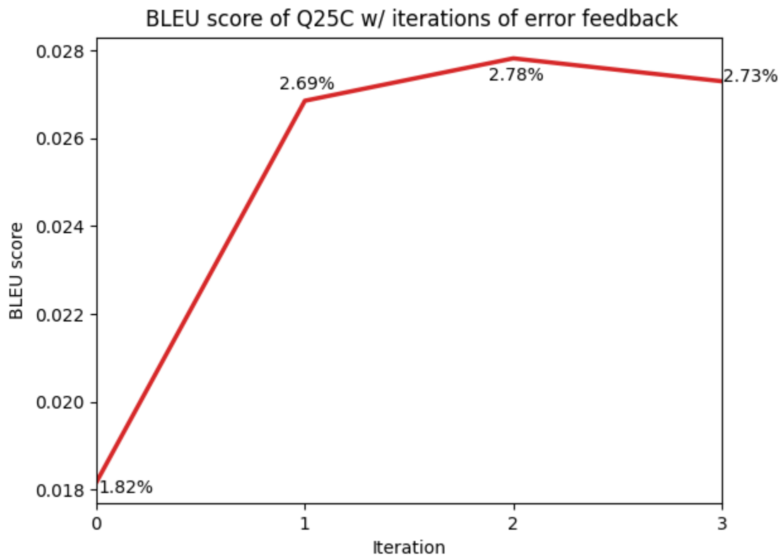

Figure 31: The BLEU score distance with Qwen2.5 Coder's output over simple Clang error feedback iterations as hypotheses when given all baseline correlators as reference

to, functioning as a form of baseline for the output [9]. Our research also uses existing code as a baseline for code generation and uses it in its metrics, which is discussed in Section 3. The survey also discusses the environmental impact, reviewing sustainability considerations with regard to model size, data, and inference costs [9]. This is related to SuperCode in general, as sustainability is a major factor, but especially to our research as it explores the usage of smaller models for code generation, and any trade-offs that may apply. The survey also concludes that LLM code generation capabilities have improved greatly, but that the code is generally not maintainable and robust enough for real software projects [9].

Liu et al. discuss the observation that LLMs tend to generate output code with unnecessary repetition and structural redundancy [11]. This is related to our research as, especially with Phi-3 Mini, which is the smallest model, there was a tendency for the SLMs to get stuck in a loop while generating output. A successful mitigation for this phenomenon in our research proved to be simply asking the SLMs to explain all their decisions. Instead, this work proposes a post-processing technique called "DeRep", which is essentially the removal of code fragments that are deemed unnecessary repetition [11]. DeRep significantly reduced repetition according to their metrics. This technique is not too relevant as repetition in our research was not a problem of output, rather the lack thereof, but it is still a very interesting approach that is relevant for future SuperCode research, as human co-design methods like pre-processing and post-processing are encouraged [2].

Godoy et al. evaluate Codex, OpenAI's Copilot, on fundamental HPC (High-Performance Computing) kernels [7]. They prompt Codex for different languages, including C/C++, and measure proficiency by looking over the top-10 suggested autocompletions. The results show that Codex would score high for common, established paradigms and languages, but much lower for newer, emerging ones, which was slightly mitigated by including specific keywords in the prompts [7]. This foreshadows the fact that SLMs are very limited in their knowledge of RISC-V, in particular its V extensions, but we also used prompts with keywords like "RISC-V", "64-bit", and "assembly" in an effort to remedy this.

Hossain et al. measured the quality of LLM-generated code by looking at the number of errors and iterations needed to get a working RISC processor [8]. Similarly, an exact ground truth is absent as a potentially limitless number of solutions could suffice. Additionally, this work also looks at more general metrics, such as the edit distance to a chosen baseline and the number of errors in the output, but it also mentions more subjective assessments (e.g. readability and maintainability). So, it dives into both aspects of the co-design approach intended for SuperCode [2, 8].

7 Conclusions and Further Research

As mentioned in Section 5, even though the one-shot results shown in our research are not spectacular, the results are also not poor enough to completely disregard the use of SLMs within SuperCode, given the smaller footprint they naturally have (RQ1, RQ 1.3).

Quantisation significantly aids with fitting models into a tight memory budget. To this end, a 4-bit quantised Qwen2.5 and Qwen2.5 Coder allow the usage of a larger context on 8GB of VRAM than the other chosen SLMs, likely due to the difference in architecture, which leads to varying VRAM usage curves across all models (RQ 1.1). There also appears to be a connection between a model's initial size in memory and the rate of increase in memory usage with increasing context length. However, this would have to be more thoroughly examined.

The results make it clear that quantisation brings more benefits than negatives, with a large benefit being the ability to use larger models that naturally have a larger knowledge base, and that specialised coder models should generally be preferred over normal models in the task of code generation according to our chosen code quality metrics, with Qwen2.5 Coder being preferable among chosen SLMs as a result of its longer and more correct output compared to its normal counterpart (RQ 1.2). Qwen2.5 Coder not only performed well among SLMs. It also came close in various metrics to state-of-the-art LLMs, even surpassing them in terms of BLEU score. Additionally, a short, simple feedback loop was shown to already cause an improvement in Qwen2.5 Coder's output across all code quality metrics (RQ1).

Further research should thoroughly examine whether the increased performance of larger models justifies the increased inference footprint, especially because these models may compensate this increased inference footprint by aiding in the development of more efficient algorithms. However, further research could also focus on more thoroughly examing SLMs with a feedback loop, as our research has shown at the very least Qwen2.5 Coder's output having room to grow.

References

[1] Neil Adit and Adrian Sampson. Performance left on the table: An evaluation of compiler autovectorization for risc-v. *IEEE Micro*, 42(5):41–48, 2022. doi:10.1109/MM.2022.3184867.

[2] P. Chris Broekema and Rob V. van Nieuwpoort. Supercode: Sustainability per ai-driven co-design. *arXiv preprint*, 2024. doi:10.48550/arXiv.2412.08490.

[3] Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, and Shikun Zhang. A survey on evaluating large language models in code generation tasks. *arXiv preprint*, 2025. doi:10.48550/arXiv.2408.16498.

[4] Yung-Sung Chuang, Yujia Xie, Hongyin Luo, Yoon Kim, James Glass, and Pengcheng He. Dola: Decoding by contrasting layers improves factuality in large language models. *arXiv preprint*, 2024. doi:10.48550/arXiv.2309.03883.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Advances in Neural Information Processing Systems*, volume 36, pages 10088–10115, 2023. ISBN 9781713899921. URL `https://arxiv.org/abs/2305.14314`.

[6] Alessandro Giagnorio, Antonio Mastropaolo, Saima Afrin, Massimiliano Di Penta, and Gabriele Bavota. Quantizing large language models for code generation: A differentiated replication. *arXiv preprint*, 2025. doi:10.48550/arXiv.2503.07103.

[7] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. Evaluation of openai codex for hpc parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ICPP-W 2023, page 136–144. ACM, 2023. doi:10.1145/3605731.3605886.

[8] Shadeeb Hossain, Aayush Gohil, and Yizhou Wang. Using llm such as chatgpt for designing and implementing a risc processor: Execution, challenges and limitations. *arXiv preprint*, 2024. doi:10.48550/arXiv.2401.10364.

[9] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint*, 2024. doi:10.48550/arXiv.2406.00515.

[10] Ahrii Kim and Jinhyeon Kim. Vacillating human correlation of SacreBLEU in unprotected languages. In Anya Belz, Maja Popović, Ehud Reiter, and Anastasia Shimorina, editors, *Proceedings of the 2nd Workshop on Human Evaluation of NLP Systems (HumEval)*, pages 1–15, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi:10.18653/v1/2022.humeval-1.1.

[11] Mingwei Liu, Juntao Li, Ying Wang, Xueying Du, Zuoyu Ou, Qiuyuan Chen, Bingxu An, Zhao Wei, Yong Xu, Fangming Zou, Xin Peng, and Yiling Lou. Code copycat conundrum: Demystifying repetition in llm-based code generation. *arXiv preprint*, 2025. doi:10.48550/arXiv.2504.12608.

[12] Andy Newell and Sergey Pupyrev. Improved basic block reordering. *IEEE Transactions on Computers*, 69(12):1784–1794, 2020. doi:10.1109/TC.2020.2982888.

[13] Minh Nhat Nguyen, Andrew Baker, Clement Neo, Allen Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. Turning up the heat: Min-p sampling for creative and coherent llm outputs. *arXiv preprint*, 2025. doi:10.48550/arXiv.2407.01082.

[14] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics. doi:10.3115/1073083.1073135.

[15] RISC-V. Rva23u64 mandatory base. `https://github.com/riscv/riscv-profiles/blob/main/src/rva23-profile.adoc#rva23u64-mandatory-base`, 2024.

[16] RISC-V Software. Spike risc-v isa simulator. `https://github.com/riscv-software-src/riscv-isa-sim`, 2021.

[17] Rob V. van Nieuwpoort. many-core-correlators. `https://github.com/RvanNieuwpoort/many-core-correlators`, 2024.

[18] Rob V. van Nieuwpoort and John W. Romein. Building correlators with many-core hardware. *IEEE Signal Processing Magazine*, 27(2):108–117, 2010. doi:10.1109/MSP.2009.935385.

[19] Tina Vartziotis, Maximilian Schmidt, George Dasoulas, Ippolyti Dellatolas, Stefano Attademo, Viet Dung Le, Anke Wiechmann, Tim Hoffmann, Michael Keckeisen, and Sotirios Kotsopoulos. Carbon footprint evaluation of code generation through llm as a service. In *International Stuttgart Symposium*, pages 230–241. Springer, 2024. doi:10.1007/978-3-658-45010-6_15.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5999—-6009. Curran Associates, Inc., 2017. ISBN 9781510860964. URL `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[21] Eugene Yan. Open llms. `https://github.com/eugeneyan/open-llms`, 2025.

[22] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019. doi:10.1109/TVLSI.2019.2926114.

A Prompts

A.1 Experiment 2

```
[
   {"role": "system", "content": "You are an expert translator of correlators from C
      to 64-bit RISC-V assembly. You explain everything and give a whole program."},
```

```
{"role": "user", "content": """Translate this 1x1 correlator in C to 64-bit RISC-V
    assembly:
#define BASELINE(STATION_1, STATION_2) ((STATION_2) * ((STATION_2) + 1) / 2 +
  (STATION_1))

#define SAMPLE_INDEX(STATION, CHANNEL, TIME, POLARIZATION, REAL_OR_IMAG) (((((CHANNEL)
  * nrStations + (STATION)) * nrTimesWidth + (TIME)) * 2 + (POLARIZATION))*2 +
  (REAL_OR_IMAG))

#define VISIBILITIES_INDEX(BASELINE, CHANNEL, POLARIZATION_1, POLARIZATION_2,
  REAL_OR_IMAG) (((((BASELINE) * nrChannels + (CHANNEL)) * 2 + (POLARIZATION_1)) * 2
  + (POLARIZATION_2)) * 2 + (REAL_OR_IMAG))

unsigned long long cpuCorrelator_1x1(const float* __restrict__ samples, float*
  __restrict__ visibilities,
                                   const unsigned nrTimes, const unsigned nrTimesWidth,
                                   const unsigned nrStations, const unsigned nrChannels,
                                   unsigned long long* bytesLoaded, unsigned long long*
                                      bytesStored)
{
  const unsigned nrBaselines = nrStations * (nrStations + 1) / 2;

  for (unsigned channel = 0; channel < nrChannels; channel ++) {
        for (unsigned int stationY = 0; stationY < nrStations; stationY++) {
        for (unsigned int stationX = 0; stationX <= stationY; stationX++) {
            float xxr = 0, xxi = 0, xyr = 0, xyi = 0, yxr = 0, yxi = 0, yyr = 0,
               yyi = 0;
            size_t index1 = SAMPLE_INDEX(stationX, channel, 0, 0, 0);
            size_t index2 = SAMPLE_INDEX(stationY, channel, 0, 0, 0);

            for (unsigned time = 0; time < nrTimes; time ++) {
               float sample1xr = samples[index1+0];
               float sample1xi = samples[index1+1];
               float sample1yr = samples[index1+2];
               float sample1yi = samples[index1+3];
               float sample2xr = samples[index2+0];
               float sample2xi = samples[index2+1];
               float sample2yr = samples[index2+2];
               float sample2yi = samples[index2+3];

               xxr += sample1xr * sample2xr + sample1xi * sample2xi;
               xxi += sample1xi * sample2xr - sample1xr * sample2xi;

               xyr += sample1xr * sample2yr + sample1xi * sample2yi;
               xyi += sample1xi * sample2yr - sample1xr * sample2yi;

               yxr += sample1yr * sample2xr + sample1yi * sample2xi;
               yxi += sample1yi * sample2xr - sample1yr * sample2xi;
```

```
                yyr += sample1yr * sample2yr + sample1yi * sample2yi;
                yyi += sample1yi * sample2yr - sample1yr * sample2yi;

                index1 += 4;
                index2 += 4;
            }
            unsigned baseline = BASELINE(stationX, stationY);
            if (baseline < nrBaselines) {
                const size_t vis_index = VISIBILITIES_INDEX(baseline, channel, 0, 0,
                    0);
                visibilities[vis_index+0] = xxr;
                visibilities[vis_index+1] = xxi;
                visibilities[vis_index+2] = xyr;
                visibilities[vis_index+3] = xyi;
                visibilities[vis_index+4] = yxr;
                visibilities[vis_index+5] = yxi;
                visibilities[vis_index+6] = yyr;
                visibilities[vis_index+7] = yyi;
            }
        }
        }
    }

    *bytesLoaded = nrChannels * nrBaselines * nrTimes * 8L * sizeof(float); // samples
    *bytesStored = nrChannels * nrBaselines * 8L * sizeof(float); // visibilities

    return nrChannels * nrBaselines * nrTimes * 16L * 2L; // #operations
}"""},
]
```

Correlator source code is from Rob V. van Nieuwpoort's many-core-correlators repository [17].

A.2 Experiment 3

```
[
    {"role": "system", "content": "You are an expert writer of correlators in 64-bit
        RISC-V assembly. You explain everything and give a whole program."},
    {"role": "user", "content": "Write a 1x1 correlator in 64-bit RISC-V assembly."},
    {"role": "assistant", "content": """
        .text
        .globl cpuCorrelator_1x1
cpuCorrelator_1x1:
# %bb.0:
        addi   sp, sp, -80
        sd     s0, 72(sp)                # 8-byte Folded Spill
        sd     s1, 64(sp)                # 8-byte Folded Spill
```

```
        sd      s2, 56(sp)                      # 8-byte Folded Spill
        sd      s3, 48(sp)                      # 8-byte Folded Spill
        sd      s4, 40(sp)                      # 8-byte Folded Spill
        sd      s5, 32(sp)                      # 8-byte Folded Spill
        sd      s6, 24(sp)                      # 8-byte Folded Spill
        sd      s7, 16(sp)                      # 8-byte Folded Spill
        sd      s8, 8(sp)                       # 8-byte Folded Spill
        li      t0, 0
        li      t1, 0
        addi    s1, a4, 1
        mul     s1, s1, a4
        srliw   t2, s1, 1
        slli    t3, a3, 2
        mul     t4, a4, a3
        slli    a4, a4, 32
        srli    t5, a4, 32
        addi    s5, a0, 8
        slli    t4, t4, 2
        fmv.w.x fa5, zero
        beqz    a5, .LBB0_10
.LBB0_1:                                # =>This Loop Header: Depth=1
                                        #     Child Loop BB0_2 Depth 2
                                        #       Child Loop BB0_3 Depth 3
                                        #         Child Loop BB0_4 Depth 4
        li      a0, 0
        li      t6, 0
        li      s3, 1
        mv      s4, t0
        beq     zero, t5, .LBB0_9
.LBB0_2:                                #   Parent Loop BB0_1 Depth=1
                                        # =>  This Loop Header: Depth=2
                                        #       Child Loop BB0_3 Depth 3
                                        #         Child Loop BB0_4 Depth 4
        li      s1, 0
        slli    a3, s4, 32
        srli    a3, a3, 30
        add     s6, s5, a3
        addi    s2, a0, 1
        addi    t6, t6, 1
        mul     a0, t6, a0
        srliw   s7, a0, 1
        slli    a0, s3, 32
        srli    s8, a0, 32
        mv      s0, t0
        beq     zero, s8, .LBB0_8
.LBB0_3:                                #   Parent Loop BB0_1 Depth=1
                                        #     Parent Loop BB0_2 Depth=2
                                        # =>    This Loop Header: Depth=3
```

```
                                        #       Child Loop BB0_4 Depth 4
        slli   a0, s0, 32
        srli   a0, a0, 30
        add    a0, a0, s5
        mv     a3, s6
        mv     a4, a2
        fmv.s  fa4, fa5
        fmv.s  fa3, fa5
        fmv.s  fa2, fa5
        fmv.s  fa1, fa5
        fmv.s  fa0, fa5
        fmv.s  ft0, fa5
        fmv.s  ft1, fa5
        fmv.s  ft2, fa5
        beqz   a2, .LBB0_5
.LBB0_4:                                #   Parent Loop BB0_1 Depth=1
                                        #     Parent Loop BB0_2 Depth=2
                                        #       Parent Loop BB0_3 Depth=3
                                        # =>     This Inner Loop Header: Depth=4
        flw    ft3, -8(a0)
        flw    ft4, -4(a0)
        flw    ft5, 0(a0)
        flw    ft6, 4(a0)
        flw    ft7, -4(a3)
        flw    fa6, -8(a3)
        flw    fa7, 0(a3)
        flw    ft8, 4(a3)
        fmul.s ft9, ft4, ft7
        fmadd.s ft9, ft3, fa6, ft9
        fadd.s ft2, ft2, ft9
        fneg.s ft9, ft3
        fmul.s ft10, ft7, ft9
        fmadd.s ft10, ft4, fa6, ft10
        fadd.s ft1, ft1, ft10
        fmul.s ft10, ft4, ft8
        fmadd.s ft3, ft3, fa7, ft10
        fadd.s ft0, ft0, ft3
        fmul.s ft3, ft8, ft9
        fmadd.s ft3, ft4, fa7, ft3
        fadd.s fa0, fa0, ft3
        fmul.s ft3, ft6, ft7
        fmadd.s ft3, ft5, fa6, ft3
        fadd.s fa1, fa1, ft3
        fneg.s ft3, ft5
        fmul.s ft4, ft7, ft3
        fmadd.s ft4, ft6, fa6, ft4
        fadd.s fa2, fa2, ft4
        fmul.s ft4, ft6, ft8
```

```
        fmadd.s ft4, ft5, fa7, ft4
        fadd.s fa4, fa4, ft4
        fmul.s ft3, ft8, ft3
        fmadd.s ft3, ft6, fa7, ft3
        fadd.s fa3, fa3, ft3
        addiw   a4, a4, -1
        addi    a3, a3, 16
        addi    a0, a0, 16
        bnez    a4, .LBB0_4
.LBB0_5:                                # in Loop: Header=BB0_3 Depth=3
        addw    a0, s7, s1
        bgeu    a0, t2, .LBB0_7
# %bb.6:                                # in Loop: Header=BB0_3 Depth=3
        mul     a0, a0, a5
        add     a0, a0, t1
        slli    a0, a0, 35
        srli    a0, a0, 30
        add     a0, a0, a1
        fsw     ft2, 0(a0)
        fsw     ft1, 4(a0)
        fsw     ft0, 8(a0)
        fsw     fa0, 12(a0)
        fsw     fa1, 16(a0)
        fsw     fa2, 20(a0)
        fsw     fa4, 24(a0)
        fsw     fa3, 28(a0)
.LBB0_7:                                # in Loop: Header=BB0_3 Depth=3
        addi    s1, s1, 1
        add     s0, s0, t3
        bne     s1, s8, .LBB0_3
.LBB0_8:                                # in Loop: Header=BB0_2 Depth=2
        addi    s3, s3, 1
        add     s4, s4, t3
        mv      a0, s2
        bne     s2, t5, .LBB0_2
.LBB0_9:                                # in Loop: Header=BB0_1 Depth=1
        addiw   t1, t1, 1
        add     t0, t0, t4
        bne     t1, a5, .LBB0_1
.LBB0_10:
        mul     a1, t2, a5
        mul     a0, a1, a2
        slli    a0, a0, 32
        srli    a0, a0, 27
        sd      a0, 0(a6)
        slli    a1, a1, 32
        srli    a1, a1, 27
        sd      a1, 0(a7)
```

```
        ld      s0, 72(sp)                      # 8-byte Folded Reload
        ld      s1, 64(sp)                      # 8-byte Folded Reload
        ld      s2, 56(sp)                      # 8-byte Folded Reload
        ld      s3, 48(sp)                      # 8-byte Folded Reload
        ld      s4, 40(sp)                      # 8-byte Folded Reload
        ld      s5, 32(sp)                      # 8-byte Folded Reload
        ld      s6, 24(sp)                      # 8-byte Folded Reload
        ld      s7, 16(sp)                      # 8-byte Folded Reload
        ld      s8, 8(sp)                       # 8-byte Folded Reload
        addi    sp, sp, 80
        ret"""},
    {"role": "user", "content": "Write a 2x1 correlator in 64-bit RISC-V assembly."},
]
```

A.3 Experiment 4

```
[
    {"role": "system", "content": "You are an expert translator of correlators from C
        to 64-bit RISC-V assembly. You explain everything and give a whole program."},
    {"role": "user", "content": """Translate this 1x1 correlator in C to 64-bit RISC-V
        assembly:
#define BASELINE(STATION_1, STATION_2) ((STATION_2) * ((STATION_2) + 1) / 2 +
    (STATION_1))

#define SAMPLE_INDEX(STATION, CHANNEL, TIME, POLARIZATION, REAL_OR_IMAG) (((((CHANNEL)
    * nrStations + (STATION)) * nrTimesWidth + (TIME)) * 2 + (POLARIZATION))*2 +
    (REAL_OR_IMAG))

#define VISIBILITIES_INDEX(BASELINE, CHANNEL, POLARIZATION_1, POLARIZATION_2,
    REAL_OR_IMAG) (((((BASELINE) * nrChannels + (CHANNEL)) * 2 + (POLARIZATION_1)) * 2
    + (POLARIZATION_2)) * 2 + (REAL_OR_IMAG))

unsigned long long cpuCorrelator_1x1(const float* __restrict__ samples, float*
    __restrict__ visibilities,
                                  const unsigned nrTimes, const unsigned nrTimesWidth,
                                  const unsigned nrStations, const unsigned nrChannels,
                                  unsigned long long* bytesLoaded, unsigned long long*
                                      bytesStored)
{
    const unsigned nrBaselines = nrStations * (nrStations + 1) / 2;

    for (unsigned channel = 0; channel < nrChannels; channel ++) {
            for (unsigned int stationY = 0; stationY < nrStations; stationY++) {
            for (unsigned int stationX = 0; stationX <= stationY; stationX++) {
                float xxr = 0, xxi = 0, xyr = 0, xyi = 0, yxr = 0, yxi = 0, yyr = 0,
                    yyi = 0;
```

```
            size_t index1 = SAMPLE_INDEX(stationX, channel, 0, 0, 0);
            size_t index2 = SAMPLE_INDEX(stationY, channel, 0, 0, 0);

            for (unsigned time = 0; time < nrTimes; time ++) {
                float sample1xr = samples[index1+0];
                float sample1xi = samples[index1+1];
                float sample1yr = samples[index1+2];
                float sample1yi = samples[index1+3];
                float sample2xr = samples[index2+0];
                float sample2xi = samples[index2+1];
                float sample2yr = samples[index2+2];
                float sample2yi = samples[index2+3];

                xxr += sample1xr * sample2xr + sample1xi * sample2xi;
                xxi += sample1xi * sample2xr - sample1xr * sample2xi;

                xyr += sample1xr * sample2yr + sample1xi * sample2yi;
                xyi += sample1xi * sample2yr - sample1xr * sample2yi;

                yxr += sample1yr * sample2xr + sample1yi * sample2xi;
                yxi += sample1yi * sample2xr - sample1yr * sample2xi;

                yyr += sample1yr * sample2yr + sample1yi * sample2yi;
                yyi += sample1yi * sample2yr - sample1yr * sample2yi;

                index1 += 4;
                index2 += 4;
            }
            unsigned baseline = BASELINE(stationX, stationY);
            if (baseline < nrBaselines) {
                const size_t vis_index = VISIBILITIES_INDEX(baseline, channel, 0, 0,
                    0);
                visibilities[vis_index+0] = xxr;
                visibilities[vis_index+1] = xxi;
                visibilities[vis_index+2] = xyr;
                visibilities[vis_index+3] = xyi;
                visibilities[vis_index+4] = yxr;
                visibilities[vis_index+5] = yxi;
                visibilities[vis_index+6] = yyr;
                visibilities[vis_index+7] = yyi;
            }
        }
        }
}

*bytesLoaded = nrChannels * nrBaselines * nrTimes * 8L * sizeof(float); // samples
*bytesStored = nrChannels * nrBaselines * 8L * sizeof(float); // visibilities
```

```
    return nrChannels * nrBaselines * nrTimes * 16L * 2L; // #operations
}"""},
    {"role": "assistant", "content": """
        .text
        .globl cpuCorrelator_1x1
cpuCorrelator_1x1:
# %bb.0:
        addi   sp, sp, -80
        sd     s0, 72(sp)                   # 8-byte Folded Spill
        sd     s1, 64(sp)                   # 8-byte Folded Spill
        sd     s2, 56(sp)                   # 8-byte Folded Spill
        sd     s3, 48(sp)                   # 8-byte Folded Spill
        sd     s4, 40(sp)                   # 8-byte Folded Spill
        sd     s5, 32(sp)                   # 8-byte Folded Spill
        sd     s6, 24(sp)                   # 8-byte Folded Spill
        sd     s7, 16(sp)                   # 8-byte Folded Spill
        sd     s8, 8(sp)                    # 8-byte Folded Spill
        li     t0, 0
        li     t1, 0
        addi   s1, a4, 1
        mul    s1, s1, a4
        srliw  t2, s1, 1
        slli   t3, a3, 2
        mul    t4, a4, a3
        slli   a4, a4, 32
        srli   t5, a4, 32
        addi   s5, a0, 8
        slli   t4, t4, 2
        fmv.w.x fa5, zero
        beqz   a5, .LBB0_10
.LBB0_1:                                # =>This Loop Header: Depth=1
                                        #     Child Loop BB0_2 Depth 2
                                        #       Child Loop BB0_3 Depth 3
                                        #         Child Loop BB0_4 Depth 4
        li     a0, 0
        li     t6, 0
        li     s3, 1
        mv     s4, t0
        beq    zero, t5, .LBB0_9
.LBB0_2:                                #   Parent Loop BB0_1 Depth=1
                                        # =>  This Loop Header: Depth=2
                                        #       Child Loop BB0_3 Depth 3
                                        #         Child Loop BB0_4 Depth 4
        li     s1, 0
        slli   a3, s4, 32
        srli   a3, a3, 30
        add    s6, s5, a3
        addi   s2, a0, 1
```

```
        addi    t6, t6, 1
        mul     a0, t6, a0
        srliw   s7, a0, 1
        slli    a0, s3, 32
        srli    s8, a0, 32
        mv      s0, t0
        beq     zero, s8, .LBB0_8
.LBB0_3:                                # Parent Loop BB0_1 Depth=1
                                        #   Parent Loop BB0_2 Depth=2
                                        # =>  This Loop Header: Depth=3
                                        #       Child Loop BB0_4 Depth 4
        slli    a0, s0, 32
        srli    a0, a0, 30
        add     a0, a0, s5
        mv      a3, s6
        mv      a4, a2
        fmv.s   fa4, fa5
        fmv.s   fa3, fa5
        fmv.s   fa2, fa5
        fmv.s   fa1, fa5
        fmv.s   fa0, fa5
        fmv.s   ft0, fa5
        fmv.s   ft1, fa5
        fmv.s   ft2, fa5
        beqz    a2, .LBB0_5
.LBB0_4:                                # Parent Loop BB0_1 Depth=1
                                        #   Parent Loop BB0_2 Depth=2
                                        #     Parent Loop BB0_3 Depth=3
                                        # =>    This Inner Loop Header: Depth=4
        flw     ft3, -8(a0)
        flw     ft4, -4(a0)
        flw     ft5, 0(a0)
        flw     ft6, 4(a0)
        flw     ft7, -4(a3)
        flw     fa6, -8(a3)
        flw     fa7, 0(a3)
        flw     ft8, 4(a3)
        fmul.s ft9, ft4, ft7
        fmadd.s ft9, ft3, fa6, ft9
        fadd.s ft2, ft2, ft9
        fneg.s ft9, ft3
        fmul.s ft10, ft7, ft9
        fmadd.s ft10, ft4, fa6, ft10
        fadd.s ft1, ft1, ft10
        fmul.s ft10, ft4, ft8
        fmadd.s ft3, ft3, fa7, ft10
        fadd.s ft0, ft0, ft3
        fmul.s ft3, ft8, ft9
```

```
        fmadd.s ft3, ft4, fa7, ft3
        fadd.s fa0, fa0, ft3
        fmul.s ft3, ft6, ft7
        fmadd.s ft3, ft5, fa6, ft3
        fadd.s fa1, fa1, ft3
        fneg.s ft3, ft5
        fmul.s ft4, ft7, ft3
        fmadd.s ft4, ft6, fa6, ft4
        fadd.s fa2, fa2, ft4
        fmul.s ft4, ft6, ft8
        fmadd.s ft4, ft5, fa7, ft4
        fadd.s fa4, fa4, ft4
        fmul.s ft3, ft8, ft3
        fmadd.s ft3, ft6, fa7, ft3
        fadd.s fa3, fa3, ft3
        addiw   a4, a4, -1
        addi    a3, a3, 16
        addi    a0, a0, 16
        bnez    a4, .LBB0_4
.LBB0_5:                                #   in Loop: Header=BB0_3 Depth=3
        addw    a0, s7, s1
        bgeu    a0, t2, .LBB0_7
# %bb.6:                                #   in Loop: Header=BB0_3 Depth=3
        mul     a0, a0, a5
        add     a0, a0, t1
        slli    a0, a0, 35
        srli    a0, a0, 30
        add     a0, a0, a1
        fsw     ft2, 0(a0)
        fsw     ft1, 4(a0)
        fsw     ft0, 8(a0)
        fsw     fa0, 12(a0)
        fsw     fa1, 16(a0)
        fsw     fa2, 20(a0)
        fsw     fa4, 24(a0)
        fsw     fa3, 28(a0)
.LBB0_7:                                #   in Loop: Header=BB0_3 Depth=3
        addi    s1, s1, 1
        add     s0, s0, t3
        bne     s1, s8, .LBB0_3
.LBB0_8:                                #   in Loop: Header=BB0_2 Depth=2
        addi    s3, s3, 1
        add     s4, s4, t3
        mv      a0, s2
        bne     s2, t5, .LBB0_2
.LBB0_9:                                #   in Loop: Header=BB0_1 Depth=1
        addiw   t1, t1, 1
        add     t0, t0, t4
```

```
        bne     t1, a5, .LBB0_1
.LBB0_10:
        mul     a1, t2, a5
        mul     a0, a1, a2
        slli    a0, a0, 32
        srli    a0, a0, 27
        sd      a0, 0(a6)
        slli    a1, a1, 32
        srli    a1, a1, 27
        sd      a1, 0(a7)
        ld      s0, 72(sp)                      # 8-byte Folded Reload
        ld      s1, 64(sp)                      # 8-byte Folded Reload
        ld      s2, 56(sp)                      # 8-byte Folded Reload
        ld      s3, 48(sp)                      # 8-byte Folded Reload
        ld      s4, 40(sp)                      # 8-byte Folded Reload
        ld      s5, 32(sp)                      # 8-byte Folded Reload
        ld      s6, 24(sp)                      # 8-byte Folded Reload
        ld      s7, 16(sp)                      # 8-byte Folded Reload
        ld      s8, 8(sp)                       # 8-byte Folded Reload
        addi    sp, sp, 80
        ret"""},
        {"role": "user", "content": """Translate this 2x1 correlator in C to 64-bit
            RISC-V assembly:
#define BASELINE(STATION_1, STATION_2) ((STATION_2) * ((STATION_2) + 1) / 2 +
    (STATION_1))

#define SAMPLE_INDEX(STATION, CHANNEL, TIME, POLARIZATION, REAL_OR_IMAG) (((((CHANNEL)
    * nrStations + (STATION)) * nrTimesWidth + (TIME)) * 2 + (POLARIZATION))*2 +
    (REAL_OR_IMAG))

#define VISIBILITIES_INDEX(BASELINE, CHANNEL, POLARIZATION_1, POLARIZATION_2,
    REAL_OR_IMAG) (((((BASELINE) * nrChannels + (CHANNEL)) * 2 + (POLARIZATION_1)) * 2
    + (POLARIZATION_2)) * 2 + (REAL_OR_IMAG))

unsigned long long cpuCorrelator_2x1(const float* __restrict__ samples, float*
    __restrict__ visibilities,
                                    const unsigned nrTimes, const unsigned
                                        nrTimesWidth,
                                    const unsigned nrStations, const unsigned
                                        nrChannels,
                                    unsigned long long* bytesLoaded, unsigned long
                                        long* bytesStored)
{
    const unsigned nrBaselines = nrStations * (nrStations + 1) / 2;
    bool missedBaselines[nrBaselines];
    for(unsigned baseline=0; baseline < nrBaselines; baseline++) {
        missedBaselines[baseline] = true;
    }
```

```
for (unsigned channel = 0; channel < nrChannels; channel++) {
   for (unsigned stationY = 0; stationY < nrStations; stationY++) {
      for (unsigned stationX = 0; stationX + 2 <= stationY; stationX += 2) {

         missedBaselines[BASELINE(stationX, stationY)] = false;
         missedBaselines[BASELINE(stationX+1, stationY)] = false;

         float s0xxr = 0.0f, s0xxi = 0.0f;
         float s0xyr = 0.0f, s0xyi = 0.0f;
         float s0yxr = 0.0f, s0yxi = 0.0f;
         float s0yyr = 0.0f, s0yyi = 0;

         float s1xxr = 0.0f, s1xxi = 0.0f;
         float s1xyr = 0.0f, s1xyi = 0.0f;
         float s1yxr = 0.0f, s1yxi = 0.0f;
         float s1yyr = 0.0f, s1yyi = 0.0f;

         size_t index0 = SAMPLE_INDEX(stationX, channel, 0, 0, 0);
         size_t index1 = SAMPLE_INDEX(stationX+1, channel, 0, 0, 0);

         for (unsigned time = 0; time < nrTimes; time ++) {
            float sample0xr = samples[index0+0];
            float sample0xi = samples[index0+1];
            float sample0yr = samples[index0+2];
            float sample0yi = samples[index0+3];

            float sample1xr = samples[index1+0];
            float sample1xi = samples[index1+1];
            float sample1yr = samples[index1+2];
            float sample1yi = samples[index1+3];

            s0xxr += sample0xr * sample1xr;
            s0xxi += sample0xi * sample1xr;
            s0xyr += sample0xr * sample1yr;
            s0xyi += sample0xi * sample1yr;
            s0yxr += sample0yr * sample1xr;
            s0yxi += sample0yi * sample1xr;
            s0yyr += sample0yr * sample1yr;
            s0yyi += sample0yi * sample1yr;
            s0xxr += sample0xi * sample1xi;
            s0xxi -= sample0xr * sample1xi;
            s0xyr += sample0xi * sample1yi;
            s0xyi -= sample0xr * sample1yi;
            s0yxr += sample0yi * sample1xi;
            s0yxi -= sample0yr * sample1xi;
            s0yyr += sample0yi * sample1yi;
            s0yyi -= sample0yr * sample1yi;
```

```
                s1xxr += sample1xr * sample1xr;
                s1xxi += sample1xi * sample1xr;
                s1xyr += sample1xr * sample1yr;
                s1xyi += sample1xi * sample1yr;
                s1yxr += sample1yr * sample1xr;
                s1yxi += sample1yi * sample1xr;
                s1yyr += sample1yr * sample1yr;
                s1yyi += sample1yi * sample1yr;
                s1xxr += sample1xi * sample1xi;
                s1xxi -= sample1xr * sample1xi;
                s1xyr += sample1xi * sample1yi;
                s1xyi -= sample1xr * sample1yi;
                s1yxr += sample1yi * sample1xi;
                s1yxi -= sample1yr * sample1xi;
                s1yyr += sample1yi * sample1yi;
                s1yyi -= sample1yr * sample1yi;

                index0 += 4;
                index1 += 4;
            }

            unsigned baseline = BASELINE(stationX, stationY);
            size_t vis_index = VISIBILITIES_INDEX(baseline, channel, 0, 0, 0);
            visibilities[vis_index+0] = s0xxr;
            visibilities[vis_index+1] = s0xxi;
            visibilities[vis_index+2] = s0xyr;
            visibilities[vis_index+3] = s0xyi;
            visibilities[vis_index+4] = s0yxr;
            visibilities[vis_index+5] = s0yxi;
            visibilities[vis_index+6] = s0yyr;
            visibilities[vis_index+7] = s0yyi;

            baseline = BASELINE(stationX+1, stationY);
            vis_index = VISIBILITIES_INDEX(baseline, channel, 0, 0, 0);
            visibilities[vis_index+0] = s1xxr;
            visibilities[vis_index+1] = s1xxi;
            visibilities[vis_index+2] = s1xyr;
            visibilities[vis_index+3] = s1xyi;
            visibilities[vis_index+4] = s1yxr;
            visibilities[vis_index+5] = s1yxi;
            visibilities[vis_index+6] = s1yyr;
            visibilities[vis_index+7] = s1yyi;
        }
    }
}

unsigned long long missedBytesLoaded, missedBytesStored;
```

```
    unsigned long long missedOps = computeMissedBaselines(samples, visibilities,
        missedBaselines,
                                                            nrTimes, nrTimesWidth,
                                                                nrStations, nrChannels,
                                                            &missedBytesLoaded,
                                                                &missedBytesStored);

    unsigned nrCells = calcNrCells(2, 1, nrStations);
    *bytesLoaded = missedBytesLoaded + nrChannels * nrCells * nrTimes * 8L *
        sizeof(float); // samples
    *bytesStored = missedBytesStored + nrChannels * nrCells * 8L * 2L * sizeof(float);
        // visibilities
    return missedOps + nrChannels * nrCells * nrTimes * 16L * 2L * 2L;
}"""},
]
```

Correlator source code is from Rob V. van Nieuwpoort's many-core-correlators repository [17].