Should AI Optimize Your Code? A Comparative Study of Current Large Language Models Versus Classical Optimizing Compilers

Miguel Romero Rosas*
Miguel Torres Sanchez*
miguelro@udel.edu
mgltorsa@udel.edu
University of Delaware
Newark, Delaware, USA

Rudolf Eigenmann
University of Delaware
Newark, Delaware, USA

Abstract

In the contemporary landscape of computer architecture, the demand for efficient parallel programming persists, needing robust optimization techniques. Traditional optimizing compilers have historically been pivotal in this endeavor, adapting to the evolving complexities of modern software systems. The emergence of Large Language Models (LLMs) raises intriguing questions about the potential for AI-driven approaches to revolutionize code optimization methodologies.

This paper presents a comparative analysis between two state-of-the-art Large Language Models, GPT-4.0 and CodeLlama-70B, and traditional optimizing compilers, assessing their respective abilities and limitations in optimizing code for maximum efficiency. Additionally, we introduce a benchmark suite of challenging optimization patterns and an automatic mechanism for evaluating performance and correctness of the code generated by such tools. We used two different prompting methodologies to assess the performance of the LLMs – Chain of Thought (CoT) and Instruction Prompting (IP). We then compared these results with three traditional optimizing compilers, CETUS, PLUTO and ROSE, across a range of real-world use cases.

A key finding is that while LLMs have the potential to outperform current optimizing compilers, they often generate incorrect code on large code sizes, calling for automated verification methods. Our extensive evaluation across 3 different benchmarks suites shows CodeLlama-70B as the superior optimizer among the two LLMs, capable of achieving speedups of up to **2.1x**. Additionally, CETUS is the best among the optimizing compilers, achieving a maximum speedup of **1.9x**. We also found no significant difference between the two prompting methods: Chain of Thought (Cot) and Instructing prompting (IP).

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; **Artificial intelligence**; • **Software and its engineering** → **Software verification and validation**.

Keywords: Automatic parallelization, Large Language Models, Compilers

ACM Reference Format:
Miguel Romero Rosas, Miguel Torres Sanchez, and Rudolf Eigenmann. 2024. Should AI Optimize Your Code? A Comparative Study of Current Large Language Models Versus Classical Optimizing Compilers. In *Proceedings of March 01–05, 2025 (The International Symposium on Code Generation and Optimization (CGO)).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The need for developing and optimizing parallel programs is paramount [18]. Optimizing compilers have long been one of the pillars of creating parallel code, evolving continuously to meet the growing complexity of modern programs. However, despite all advancements, current compilers still do not achieve the necessary performance to be true alternatives to manual parallelization [8].

The dawn of Large Language Models (LLMs), such as GPT and CodeLlama raises a fundamental question: **Can AI-driven models revolutionize the way we approach code optimization?** These models paint an appealing horizon for code optimization and parallel code generation [30]. Among several LLMs, we chose to use GPT4.0 from OpenAI [28] and CodeLlama-70B [23] from META, as these represent the

*Both authors contributed equally to this research.

The International Symposium on Code Generation and Optimization (CGO), Las Vegas, NV,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

state-of-the-art and thus offer the promise of efficient code generation [22].

Our study compares three state-of-the-art optimizing compilers, CETUS [4], PLUTO [10] and ROSE [33] against the two LLMs for parallel code optimization. We evaluate their capacity to serve as Automatic Optimization Tools (AOTs).

One of the challenges in using LLMs for code generation is the lack of correctness guarantee. To address this critical gap, we developed a mechanism for automatic validation of the generated programs. The mechanism enables LLMs to be used as AOTs and facilitates the comprehensive evaluation of these models. The mechanism can also serve as a general instrument facilitating the use of unsafe optimizations. To the best of our knowledge this is the first such mechanism, enabling the automatic use of optimizations that do not ensure correctness.

In our evaluation, we are specifically interested in code patterns that challenge current automatic optimization tools and in the ability of LLM-based AOTs to overcome these challenges. To this end, we have developed a benchmark suite of such patterns. Next to a comprehensive evaluation, the suite also allows us to identify areas for improvement in parallel code optimization.

This paper makes the following contributions:

- We present a comparative study of LLMs and traditional optimizing compilers, utilizing study cases from real-world applications, such as the NAS Parallel Benchmarks Suite (NPB) v3.3 [5], the POLYBENCHMARK (PB) Suite v4.2 [37], and other relevant programs.
- We introduce an automated mechanism for validating the correctness and performance of AOT-generated code.
- We present a Parallel Computing Challenge Benchmark suite (PCB) v1.0, comprising 20 use cases for evaluating the capabilities of AOTs.

This paper is organized as follows: Section 2 explores strengths and limitations of our chosen optimizing compilers and LLMs. Section 3 describes our environment for evaluating automatic optimization tools and presents the Parallel Computing Challenge Benchmark suite (PCB). Section 4 evaluates the capabilities of our LLMs and optimizing compilers on challenging code patterns, followed by the discussion of related work in Section 5 and conclusions 6.

2 LLMs vs Traditional Compilers, Strengths and Limitations

Current optimizing compilers exhibit a mix of challenges and opportunities [19, 26, 38]. They include limitations of static analysis, hampering the identification of optimization opportunities, and the complexity of deciding on best optimization patterns for maximum program performance [7]. It is in this context that the power of LLMs emerges, offering new potential ways to approach program optimization.

Large Language Models have garnered significant attention in recent years owing to their statistical understanding of programs, offering suggestions for code generation, and facilitating software creation [20] [21] [1] [13]. While LLM-generated software has demonstrated success in many applications, recent studies have highlighted the limitations in capturing crucial aspects of linguistic meaning and understanding semantic properties of the user's prompt [2].

Despite these limitations, recent studies indicate that Large Language Models excel in programming tasks such as code generation, issue fixing, and code completion [35]. This proficiency is largely attributed to the vast amount of data available on platforms such as GitHub, enabling effective training. However, even though LLMs can generate effective code, the lack of correctness guarantees make them unsuitable for many code optimization approaches [12, 17, 27, 34].

In pursuit of the question asked in the Introduction **Can AI-driven models revolutionize the way we approach code optimization?**, this paper investigates in what situations Large Language Models perform well and where not. To this end, we have developed an environment for comparing the optimization capabilities of LLMs with those of classical optimizing compilers, including the creation of an automated code verification mechanism and a benchmark suite of code challenge patterns. The next section describes this environment.

3 An Environment for Evaluating Automatic Optimization Tools

Recall that the lack of a methods for verifying the correctness of LLM-generated code make it difficult to compare such tools with other AOTs. Section 3.1 introduces a novel method and environment called PCAOT (Performance and Correctness Evaluation of Automatic Optimization Tools) that overcomes this limitation. Additionally, we have created the Parallel Challenge Patterns Benchmark (PCB) suite v1.0 with 20 distinct use cases. The suite and the challenge patterns are discussed in Section 3.2.

3.1 Evaluation and Verification

Our approach entails selecting two Large Language Models (LLMs), GPT-4.0 and CodeLlama-70B, and evaluating their code optimization capabilities. The results are then compared with three different optimizing compilers, CETUS, PLUTO and ROSE. For engaging the LLMs, we use two prompting strategies, descibed next.

3.1.1 Prompting Strategies. We explored two different prompts associated with utilizing Large Language Models for High-Performance Computing optimization tasks. Initially, we employed instruction prompting (IP) [25] [16]. Additionally, we employed Chain of Thought (CoT) within these

Large Language Models. Recent studies indicate that Chain of Thought prompting enhances performance across a spectrum of arithmetic, commonsense, and symbolic reasoning tasks, making it well suited for assessing domain-specific knowledge such as Parallel Computing [36].

CETUS, PLUTO, and ROSE translate C programs to equivalent C code annotated with OpenMP parallel directives. Incorporating OpenMP into the prompts facilitates the comparison of the code generated by LLMs and these compilers. We chose the following prompts:

- Instructing (IP) - "Given the program below, improve its performance using OpenMP".
- CoT - "Given the C program below, think of a way how to optimize its performance using OpenMP"

The distinction between these two types of prompts lies in their approach to guiding the model's reasoning process. Instructing prompts provide a direct and clear directive, resulting in the immediate production of an optimized program version. In contrast, Chain of Thought prompts enable the model to break down the task into multiple steps, allowing for iterative refinement before generating an optimized program version. Instances of guidance the model in this process are: **"Think of a way..."**, **"propose..."**, **"give a potential strategy..."** [34].

For the optimizing compilers CETUS, PLUTO, and ROSE, we used the default set of parameters, representing the known, safe optimization methods. Note that, while experimenting with optimization options could potentially yield improved performance, doing so is beyond the scope of this paper. Our environment for evaluating the five AOTs comprises three phases: **Preparation, Optimization**, and **Validation**. Figure 1 illustrates this architecture.

3.1.2 Preparation Phase. LLMs have shown to have limitations to process programs with large code sizes [29] [24]. Hence, our LLM evaluation focuses on section-level optimizations. The preparation phase identifies the sections within the source code to experiment with. Each such section is manually enclosed with **#pragma experimental section start** and **#pragma experimental section end**, as illustrated in Figure 2. The impact of code size on the performance of the LLMs will be discussed in Section 4.2.

```
1: #pragma experimental start
2: for (k=0; k<=(grid_points[2]-1); k ++ )
3:  {
4:    for (j=0; j<=(grid_points[1]-1); j ++ )
5:    {
6:        for (i=0; i<=(grid_points[0]-1); i ++ )
7:        {
8:            for (m=0; m<5; m ++ )
9:            {
10:                rhs[k][j][i][m]=forcing[k][j][i][m];
11:            }
12:        }
13:    }
14:}
15: #pragma experimental end
```

Figure 2. Experimental section instrumentation. Example is from subroutine Compute_rhs of the NPB Application CG.

For our experiments we made used of a tool called CaRV (Capture, Replay, and Validate), which enables users to experiment with sections of large applications. It facilitates the comparison of individual program sections before and after optimizations, assessing their efficiency and accuracy [6]. CaRV does so using a selective checkpointing method, based on the identified program states at the beginning and end of an experimental section.

The preparation phase generates a new program version that is instrumented in a way that allows the optimized program section to be run independently and compared against the captured end state of the original program. Both the original and instrumented program versions are passed on to the optimization phase.

3.1.3 Optimization Phase. This phase sends the experimental section with the prompts to the two LLMs to obtain the optimized versions. The resulting code is then inserted into the instrumented program. The original code is also sent to the three compilers, CETUS, PLUTO and ROSE. This process generates a total of 16 program versions, for the five AOTs (two LLMS, two different prompts, three requests per prompt, three optimizing compilers plus the serial program) The next phase executes and validates these versions.

3.1.4 Validation Phase. The Validation phase executes each version of the experimental section individually. The LLM-generated versions are then compared with the original, serial program versions, using the CaRV checkpoints. This process verifies the values of all relevant, "live" variables at the end of the LLM-optimized program section against the corresponding values in the original program. The execution time is measured as well and compared to that of the original program section.

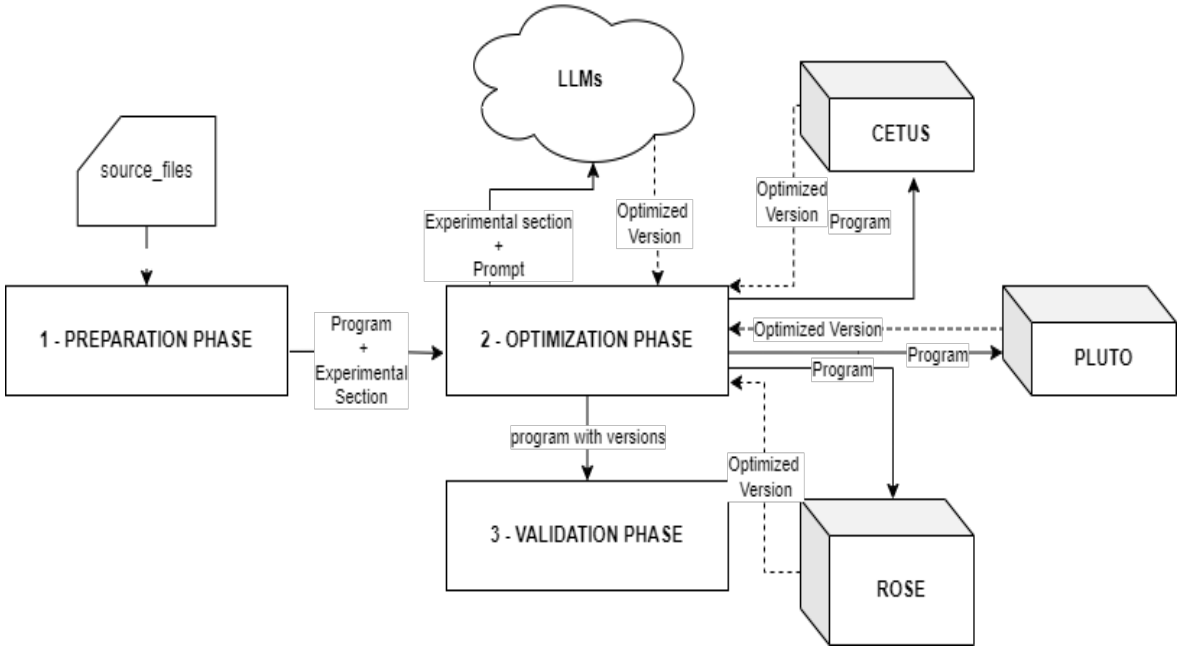


Figure 1. Architecture of the PCAOT Environment.

3.2 Parallel Computing Challenge Pattern Benchmark Suite v1.0 (PCB)

This new Benchmark suite comprises 20 distinct use cases, each presenting one of six challenging scenarios encountered by optimizing compilers [7]. A number of patterns are included in the PCB suite, testing the optimizers in their ability to

- parallelize outermost loops (PO),
- parallelize loops with function calls (PF),
- form parallel regions enclosing multiple parallel loops (PR),
- avoiding load imbalance through dynamic scheduling (DS),
- parallelizing array reductions (PA),
- eliminate barrier synchronizations using NOWAIT clause (NW).

Given the challenges traditional optimizing compilers face in handling these patterns [7] [32], an important question is: **Can LLMs identify these patterns and perform the requisite optimizations?** The following paragraphs explain the involved program patterns:

Parallelizing Outermost Loops. This pattern involves identifying and parallelizing the outermost loops in a program section, which often represent significant computational tasks. Parallelizing these loops can improve performance by distributing the number of iterations across multiple processors or threads. Running the outer loop in parallel typically results in lower overhead compared to parallelizing the inner loop, as inner parallelism increases the number of times of starting and ending parallel activities. This pattern poses a challenge because real-world scenarios often contain loops with complex expressions that prove too difficult for compilers' static analysis capabilities. Automatic optimizers often success in parallelizing the inner loops only. Figure 3 illustrates this situation.

```
1: int sum
2: for (int i = 0; i < size; ++i) {
3:   for (int j = 0; j < size; ++j) {
4:     for (int k = 0; k < size; ++k) {
5:       #pragma omp parallel for
6:       for (int l = 0; l < size; ++l) {
7:         int position =i*size * size * size + j * size ;
8:         int index = position  * size + k * size + l;
9:         sum += data[indices[index]];
10:        }
11:     }
12:   }
13: }
```

Figure 3. Parallelizing the outermost loop pattern in an example loop from the PCB v1.0. Compilers tend to parallelize the inner-most loop, due to the irregular data accesses in code line #9. Doing so incurs the overhead of starting the parallel region $size^3$. Parallelizing the outermost loop would eliminate this overhead.

Parallelizing Loops with Function Calls. Parallelizing loops with function calls presents a significant challenge for optimizing compilers, as many optimization techniques do not operate interprocedurally. A special situation is the

presence of calls to side-effect-free functions (functions that only modify their parameters), which only require the optimizer to determine the independence of the variables passed as function parameters. Loops with such functions can be parallelized. Figure 4 illustrates this situation.

```
1: int size = 1000;
2: int result[size][size];
3: for (int i = 0; i < size; i++) {
4:    for (int j = 0; j < size; j++) {
5:        for (k = 0; k < size; k++){
6:            result[i][j][k] = compute(i, j, k);
8:            }
9:    }
10: }
   .
   .
   .
13: /* Side effect free fuction call*/
14: int compute(int x, int y, int k) {
15:    int result = 0;
16:       for (int i = 0; i < x; i++) {
17:        result += (x + y + k) * i;
20:        return result;
21:    }
22: }
```

Figure 4. Parallelizing Loops with Function Calls pattern in an example loop from the PCB v1.0. Function ***compute*** is side effect free. The loop in line #3 can be parallelized.

Parallel Regions Enclosing Multiple Parallel Loops. Current optimizing compilers are limited to optimizing loops individually and cannot apply optimizations across a set of loops within a parallel region. By enclosing multiple parallel loops within a single parallel region, the overhead associated with starting and ending each parallel loop separately is reduced. Figure 5 illustrates this pattern.

```
1: int array[1000];
2: int result[1000];
3:  // Initialize array
4:    for (int i = 0; i < NUM_ELEMENTS; i++) {
5:        array[i] = i;
6:    }
7:
8:   #pragma omp parallel
9: {
10:    #pragma omp parallel for
11:    for (int i = 0; i < NUM_ELEMENTS; i++) {
12:        result[i] = array[i] * 2;
13:    }
14:
15:    #pragma omp parallel for
16:    for (int i = 0; i < NUM_ELEMENTS; i++) {
17:       result[i] *= result[i];
18:    }
19:
20:    #pragma omp parallel for
21:    for (int i = 0; i < NUM_ELEMENTS; i++) {
22:       result[i] += 10;
23:    }
24: }
```

Figure 5. Parallel Regions Enclosing Multiple Parallel Loops pattern in an example from the PCB v1.0. The parallel region reduces the Fork/join overhead caused by parallelizing each loop individually.

Avoiding Load Imbalance Through Dynamic Scheduling. Load imbalance occurs when the computational workload is unevenly distributed among the available processing units. Load imbalance is a significant source of performance degradation and limited scalability in parallel computing. Dynamic scheduling in OpenMP can mitigate this effect by assigning computational tasks to processing units at runtime based on their load and availability. Figure 6 illustrates this situation by showcasing a *triangular* loop. The default OpenMP scheduler (static) will divide the iterations of the parallel loop evenly, causing load imbalance.

```
1: #pragma omp parallel for schedule(dynamic, 1)
2: //Every thread will process one iteration of the loop,
3: // and it will be used as it is available
4:  for (int i = 0; i < N; i++) {
5:    for (int j = 0; j < N - i; j++) {
6:      // With static scheduling: Thread1->N-i,
7:        // Thread2->N-i-1...
8:      // ThreadN->1
9:       array[i][j] = i + j;
10:    }
11:  }
```

Figure 6. Avoiding Load Imbalance Through Dynamic Scheduling pattern in an example from the PCB v1.0. Inserting the OpenMP clause for dynamic scheduling resolves the load imbalance.

Parallelizing Array Reductions. Array reductions perform operations, such as sum, product, min, or max, on elements of an array. Parallelizing these operations requires careful consideration of data dependencies and synchronization to ensure correct results while maximizing parallel execution. Algorithms to perform reductions in parallel are well known [31] but applying them correctly and beneficially when the result is an array itself can be non trivial [32]. Figure 7 illustrates this situation.

```
1:  #pragma omp parallel for reduction(+:result)
2:  for (i = 0; i < ARRAY_SIZE; i++) {
3:       for (j = 0; j < INNER_SIZE; j++) {
5:          result[j] += array[i][j];
6:       }
7:    }
8:
```

Figure 7. Parallelizing Array Reduction pattern in an example from the PCB v1.0. Most compilers can recognize scalar reductions. The analysis is more difficult, if the reduction variable is an array, as `result[]` in this example.

NOWAIT – Eliminating Barrier Synchronizations. Barriers are essential for maintaining correctness and synchronization in parallel programs but introduce overhead when some threads complete their work before others. NOWAIT constructs can be used to eliminate barrier synchronizations and improve parallel performance by allowing threads to continue executing. AOTs need to determine the situations where doing so is legal. Figure 8 illustrates this pattern.

```
1: #pragma omp parallel
2: {
3:    double local_sum = 0.0;
4:    for (int i = 0; i < ARRAY_SIZE; i++) {
5:        for (int j = 0; j < INNER_SIZE; j++) {
6:             for (int k = 0; k < INNER_SIZE; k++) {
7:                result[i][j][k]+=matrix_A[i][j][k]
8:                * matrix_B[i][j][k];
9:             }
10:        }
11:   }
12:       #pragma omp for nowait
13:       for (i = 0; i < ARRAY_SIZE; i++) {
14:         for (j = 0; j < INNER_SIZE; j++) {
15:             for (k = 0; k < INNER_SIZE; k++) {
16:                 local_sum += array[i][j][k];
17:             }
18:         }
19:     }
20:    #pragma omp atomic
21:     sum += local_sum;
22:   }
```

Figure 8. Eliminating Barrier Synchronizations pattern in an example from the PCB v1.0. An OpenMP NOWAIT clause is inserted to eliminate the barrier synchronization and improve parallel performance.

4 Evaluation: LLMs versus Classical Optimizing Compilers

This section assesses the current capabilities of two Large Language Models, GPT-4.0 and CodeLlama-70B, in comparison with three different automatic compilers, CETUS, PLUTO and ROSE. Section 4.1 outlines the experimental setup, Section 4.2 evaluates the ability of the LLMs to process and correctly optimize large code sizes. Section 4.3 examines the ability of these LLMs to optimize code sections effectively and also assesses their capacity in handling the challenge patterns introduced in Section 3.2. Lastly, Section 4.4 compares the achieved speedup of all five Automatic Optimization Tools (AOTs).

4.1 Experimental setup

To assess our Automatic Optimization Tools, we use the PCAOT environment described in Section 3.1. Our test suite drew from three distinct benchmarks: The NPB v3.3 [5] , the PB suite V4.2 [37], and the PCB benchmark introduced in Section 3.2. For our experiments, we measured the performance of the applications using input Class B for the NPB and the LARGE_DATASET for the PB and PCB suite.

We have selected specific subroutines from the NPB and the PB suites that represent challenge patterns. For the PCB, we developed 20 use cases, with each one representing one of the six challenge scenarios detailed in Section 3.2.

Table 1 displays our test suite. We set the following AOT parameters: For the CETUS, PLUTO and ROSE compilers we chose their default options, selecting the known, safe optimization behavior. For the LLMs, we chose a Temperature of **0.2** and a Top_p value of **0.1**, increasing deterministic behavior for the code generation tasks.

We measured execution times on a compute node featuring a 4-core Intel Xeon Gold 6230 processor configuration in dual sockets. Each processor operates at a base frequency of 2.1 GHz, with a 27.5MB cache, and was supported by up to 1 GB of DDR4 memory. Application codes were compiled using GCC v4.8.5 with -O3 optimization on CentOS v7.4.1708. Reported values represent the median of three application runs, each utilizing one thread per core.

4.2 Impact of Code Size

Table 2, shows the capacity of the Large Language Models to process programs with large code sizes. This table displays the number of lines of code per experimental section versus the successful cases after sending the two different prompts described in Section 3.1.1, 3 times each to the two chosen LLMs; GTP4.0 and Codellama-70B, with a total of 192 attempts.

These results illustrate the capability of the LLMs in handling large code sizes. While smaller code sections tend to be transformed more accurately, they are not guaranteed to

Table 1. Testing Dataset: Experimental Section, Impact of the experimental section on the overall application, Pattern evaluated, and number of loops we are testing with the AOTs

Benchmark	Application Name	Experimental Section	Impact %	Pattern evaluated	#Loops
NAS	BT	initialize	0.2%	PF, PR	11
NAS	BT	rhs_norm	0.5%	PR, PO, NW	1
NAS	BT	compute_rhs	10.2%	PR, PO	11
NAS	BT	x_solve	27.7%	PF, PO	1
NAS	SP	initialize	0.3%	PF, PR	11
NAS	SP	rhs_norm	0.2%	PF, PR	1
NAS	SP	y_solve	19.6%	PF, PO	1
NAS	SP	exact_rhs	0.3%	PR, PO, NW	5
NAS	MG	norm2u3	1.1%	PF, PR, NW	1
NAS	MG	zran3	29.3%	PF, PR	3
NAS	CG	conj_grad	98.9%	PR, PA, PO, NW	5
NAS	IS	full_verify	99.8%	DS, PO	5
Poly	Linear Algebra	2mm	91.8%	PR, PO	1
Poly	Linear Algebra	3mm	98.4%	PR, PO	3
Poly	Linear Algebra	Atax	18.2%	PR, PO	5
Poly	Datamining	Correlation	99%	PR, PO	1
PCB	Challenges	PO_Function_Version[1-5]	97%	PO	5
PCB	Challenges	PF_Function_Version[1-3]	93%	PF	4
PCB	Challenges	PR_Function_Version[1-3]	98%	PR	9
PCB	Challenges	DS_Function_Version[1-3]	96.3%	DS	3
PCB	Challenges	PA_Function_Version[1-3]	95%	PA	4
PCB	Challenges	NW_Function_Version[1-3]	97.2%	NW	3

be 100% correct. As the code size increases, the likelihood of inaccuracies also increases.

Program sections of approximately 20 lines often produce correct output, whereas no section above 200 lines was transformed successfully (performance will be evaluated later). In between, the LLMs exhibit non-deterministic behavior, with some runs producing useful results while others with the same input fail. GPT4.0 and Codellama-70B return successfully transformed code in 14.06% of the cases.

These results further justify our method of employing LLM capabilities on the basis of individual program sections.

4.3 LLM Optimization Capabilities

Figure 9 shows the success rates of generating correctly optimized code by the tested LLMs on specific challenge patterns, using the two prompts described in Section 3.1.1. The results are classified into four distinct categories: **Pattern correctly applied, Pattern not correctly applied but code compiles, Compilation error, Incorrect Result** and **Runtime error**.

Our primary finding from this information is that LLMs are not yet in a position where they can serve as automatic optimizers. Only in three of the analyzed optimization patterns, both LLMs produce partly incorrect results. A key demand on automatic optimizers is that they be 100% correct; tools that require software engineers to engage in possibly lengthy debug phases after use are not acceptable. The partial successes in two of the patterns (Array Reduction and Parallelizing Outermost Loop) indicate that LLMs may be making progress toward the goal of becoming useful AOTs. These two patterns represent some of the most important parallelization capabilities and likely have many code examples serving as LLM training data, which may explain the partial success. The NOWAIT and Parallel-Region patterns tend to involve large code regions, which challenge LLMs, as discussed in Section 4.2. The two LLMs completely fail on Dynamic Scheduling patterns. We attribute this behavior to the more complex code exhibited by such test cases. Dynamic scheduling tends to be needed in loops that involve irregular code and data structures, which are more difficult to comprehend.

The results show no significant difference between the two prompting methods we used. While other researchers have reported more success with Chain-of-Thought (CoT) prompting, this method performed marginally worse in some cases than the more direct Instruction Prompting (IP) method. Finally, comparing the two LLMs shows insignificant differences as well. CodeLLama is optimized for program code synthesis, as well as infilling/completion [14], hence we expected it to perform well. However, ChatGPT has a larger

Table 2. Capacity of Large Language Models to process large programs

Large Language Model							
GPT4.0				CodeLlama-70B			
Benchmark	**Experimental Section**	**#Lines code**	**Succesfull Cases**	**Benchmark**	**Experimental Section**	**#Lines code**	**Sucessfull cases**
BT	rhs_norm	18	3	BT	rhs_norm	18	2
Linear Algebra	Atax	21	5	Linear Algebra	Atax	21	5
Datamining	Correlation	21	5	Datamining	Correlation	21	5
SP	rhs_norm	26	0	SP	rhs_norm	26	0
MG	norm2u3	28	2	MG	norm2u3	28	0
Linear Algebra	2mm	29	6	Linear Algebra	2mm	29	6
Linear Algebra	3mm	34	6	Linear Algebra	3mm	34	6
IS	full_verify	41	0	IS	full_verify	41	0
BT	Initialize	155	0	BT	Initialize	155	0
SP	Initialize	161	0	SP	Initialize	161	0
CG	conj_grand	183	0	CG	conj_grand	183	3
MG	zran3	187	0	MG	zran3	187	0
SP	y_solve	250	0	SP	y_solve	250	0
SP	exact_rhs	313	0	SP	exact_rhs	313	0
BT	x_solve	346	0	BT	x_solve	346	0
BT	compute_rhs	382	0	BT	compute_rhs	382	0

number of AI model parameters, which may make up for the fact that it is a general model.

4.4 Speedup Evaluation

Figure 10 shows the maximum average speedups for correct and incorrect code obtained from the five distinct AOTs across our three benchmark suites against each benchmark's baseline (Serial code) and its Hand-optimized version.

Our findings demonstrate the efficacy of CodeLlama-70B as the superior optimizer among Language Model-based (LLM) solutions, capable of achieving speedups of up to **2.1x** compared to the original program, measuring correct code only. When considering incorrectly optimized code as well, the achieved speedup is approximately **2.7x**.

Conversely, we can observe a noticeable difference between all the AOTs and PLUTO in terms of speedup achieved within the PolyBenchmark suite, where PLUTO outperformed all of the AOTs with a remarkable **7x** speedup. This superior performance is attributed to PLUTO's specialization on the Polyhedral model [11]. The specialization may also explain why PLUTO was unable to process the other benchmark suites.

Overall, CETUS emerged as the best among the optimizing compilers for processing a variety of programs within these suites, achieving a maximum speedup of **1.9x**. The ROSE optimizer achieved a speedup across all benchmarks of **1.2x**, remaining below the other AOTs in terms of efficacy.

While GPT4 has demonstrated success across various tasks, it exhibited the lowest performance for code optimization across all the benchmarks and among the LLMs. It achieved a maximum speedup of **1.6x**, falling short of CodeLlama-70B.

5 Related work

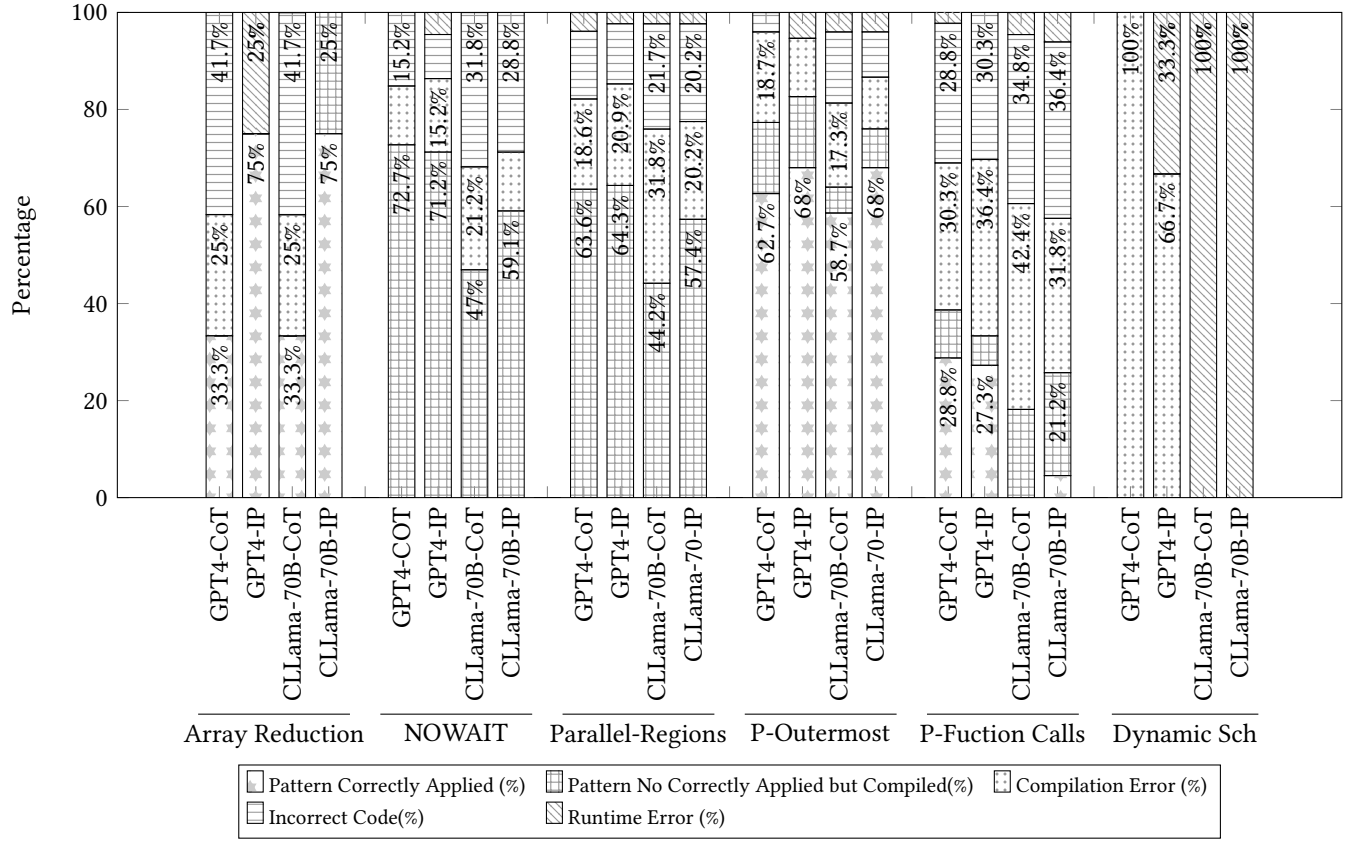
The primary objective of PCAOT is to automatically validate the correctness and performance of optimizations generated by Automatic Optimization Tools (AOTs). To the best of our knowledge, no other research has directly addressed this specific objective. However, several studies have created benchmarks to evaluate the capabilities of Large Language Models (LLMs) in optimization tasks.

For instance, HumanEval [9] is a benchmark designed to assess the performance of LLMs in generating correct and efficient code. Additionally, there has been a study evaluating the ability of LLMs to synthesize short programs from natural language descriptions, demonstrating the potential of these models in code generation tasks [3].

Recent studies have also evaluated the performance of LLMs in specific High-Performance Computing (HPC) optimization tasks; such as OpenMP pragma prediction [27] highlighting the application of LLMs in parallel computing optimizations. Additionally, approaches to code completion [35] and issue fixing [35] show the breadth of LLM applications in software development and maintenance.

Another significant contribution in this domain is the novel paradigm introduced by researchers using Large Language Models with compiler feedback to optimize the code size of LLVM assembly. Their model takes unoptimized

Figure 9. Correctness Evaluation for the following challenging patterns: Array Reduction, NOWAIT, Parallel-Regions, Parallelization at the Outermost level (P-Outermost), Parallelization for function calls (P-Function Calls), Dynamic Scheduling (Dynamic Sch)



LLVM Intermediate Representation (IR) as input and produces optimized IR, optimization passes, and instruction counts for both unoptimized and optimized IRs [15].

Despite these advancements, the automatic validation of both the correctness and performance of optimizations remains underexplored. Our work aims to fill this gap by providing a comprehensive framework for evaluating and validating the outputs of AOTs, ensuring that the generated optimizations meet the desired standards of correctness. We also proposed a novel Parallel Computing Challenge Benchmark suite (PCB) V1.0 containing six challenge patterns for evaluating AOT code optimization capabilities.

6 Conclusions

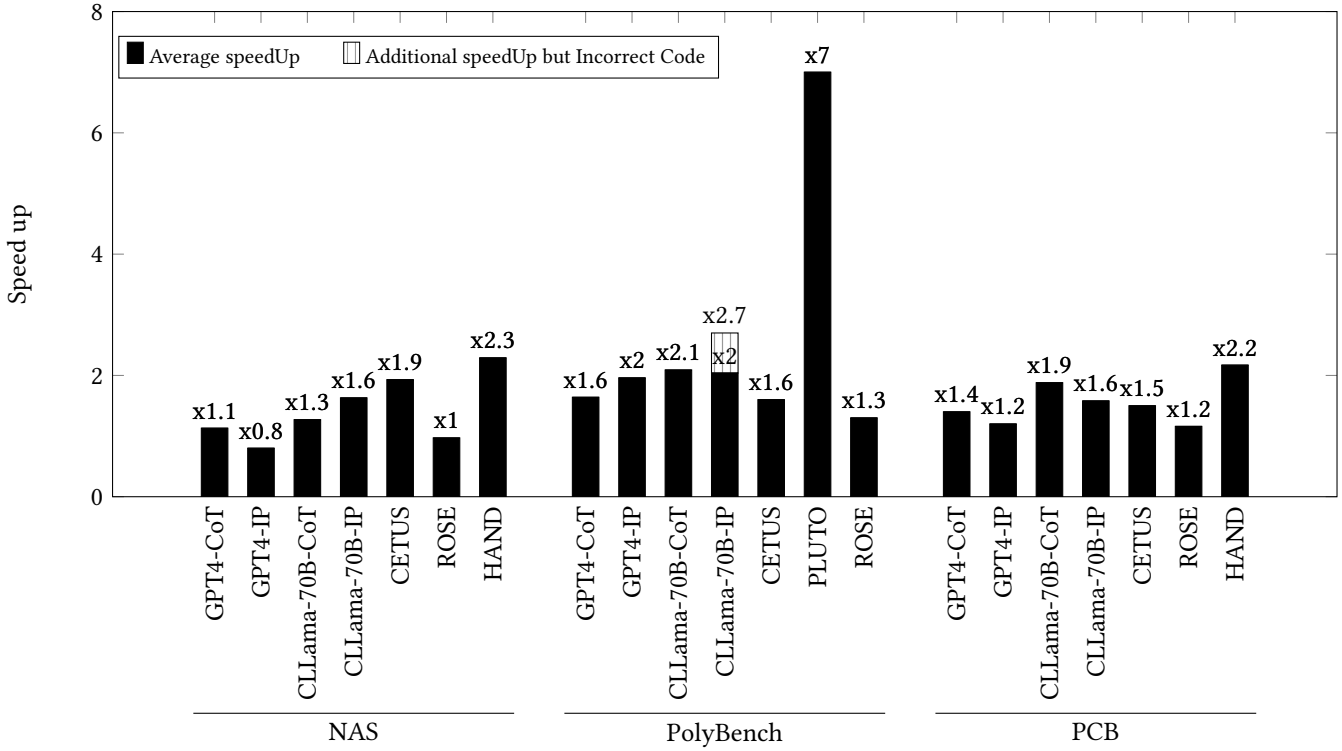
This paper introduced a novel mechanism called PCAOT, which enables the automatic validation of the correctness and performance of Automatic Optimization Tools-generated code. While we have applied the mechanism to verify LLM-generated code, it also enables unsafe compiler optimization to be tested. Additionally, we have presented the Parallel Computing Challenge Benchmark suite (PCB) v1.0, comprising 20 use cases for evaluating the capabilities of Automatic Optimization Tools. To the best of our knowledge, this is the first such benchmark.

Our results show that LLMs only succeed in optimizing very small programs of approximately 20 lines. Given that real-world software often contains thousands of lines of code, these models are not yet ready to serve as automatic optimizers.

Our results also show that LLMs perform only marginally better than current optimizing compilers, in terms of correctness. The chosen LLMs can correctly identify three out of the six different challenge patterns, such as Array Reduction, Parallelizing Outermost Loops, and Loops Containing Function Calls, with a success rate higher than **50%** in most cases. However, they often fail on other patterns, including eliminating barrier synchronization using the NOWAIT clause, parallel regions enclosing multiple parallel loops, and avoiding load imbalance through dynamic scheduling.

In terms of speedup, the best compiler and the best LLM performed within 10%. We also found no significant difference between the two prompting methods we employed,

Figure 10. AOTs - Speedup Evaluation of each AOT using a specific configuration: Chain-of-Thought and Instruction prompting for LLMs and the default configuration for the optimizing compilers.



with Chain-of-Thought (CoT) performing slightly worse in some cases than the more straightforward Instruction Prompting (IP) method.

Finally, addressing the main question asked in the introduction: **Can AI-driven models revolutionize the way we approach code optimization?** We find that, with their current capabilities, the LLMs cannot yet suitably serve as Automatic Optimization Tools. Even though the potential exists, future LLMs will need significant improvements in terms of performance and correctness.

.

7 ACKNOWLEDGEMENTS

This work was supported in part by the University of Delaware and by the National Science Foundation under awards 2112606, 2125703, and 1931339.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Nicholas Asher, Swarnadeep Bhar, Akshay Chaturvedi, Julie Hunter, and Soumya Paul. 2023. Limits for learning with language models. *arXiv preprint arXiv:2306.12213* (2023).

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P Midkiff. 2013. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming* 41 (2013), 753–767.

[5] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 158–165.

[6] Parinaz Barakhshan and Rudolf Eigenmann. 2023. CaRV–Accelerating Program Optimization through Capture, Replay, Validate. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 654–662.

[7] Parinaz Barakhshan and Rudolf Eigenmann. 2023. Learning from Automatically Versus Manually Parallelized NAS Benchmarks. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 29–46. https://link.springer.com/chapter/10.1007/978-3-031-31445-2_3.

[8] Akshay Bhosale, Parinaz Barakhshan, Miguel Romero Rosas, and Rudolf Eigenmann. 2022. Automatic and interactive program parallelization using the Cetus source to source compiler infrastructure v2. 0. *Electronics* 11, 5 (2022), 809.

[9] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive

language model. *arXiv preprint arXiv:2204.06745* (2022).

[10] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[11] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–146.

[12] Le Chen, Pei-Hung Lin, Tristan Vanderbruggen, Chunhua Liao, Murali Emani, and Bronis de Supinski. 2023. Lm4hpc: Towards effective language model application in high-performance computing. In *International Workshop on OpenMP*. Springer, 18–33.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[14] Wenhan Xiong Grattafiori, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[15] Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. 2024. Compiler generated feedback for Large Language Models. *arXiv preprint arXiv:2403.14714* (2024).

[16] Prakhar Gupta, Cathy Jiao, Yi-Ting Yeh, Shikib Mehri, Maxine Eskenazi, and Jeffrey P Bigham. 2022. InstructDial: Improving zero and few-shot generalization in dialogue through instruction tuning. *arXiv preprint arXiv:2205.12673* (2022).

[17] Tal Kadosh, Nadav Schneider, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. Advising openmp parallelization via a graph-based approach with transformers. In *International Workshop on OpenMP*. Springer, 3–17.

[18] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744. https://doi.org/10.1126/science.aam9744 arXiv:https://www.science.org/doi/pdf/10.1126/science.aam9744

[19] Fengqian Li, Feilong Tang, and Yao Shen. 2014. Feature mining for machine learning based compilation optimization. In *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE, 207–214.

[20] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[21] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[22] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE]

[23] META. [n. d.]. *CodeLlama B-70*. https://huggingface.co/chat/

[24] META. [n. d.]. *CodeLlama B-70 tokens*. https://huggingface.co/codellama/CodeLlama-70b-Instruct-hf

[25] Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. 2021. Reframing Instructional Prompts to GPTk's Language. *arXiv preprint arXiv:2109.07830* (2021).

[26] Antoine Monsifrot, François Bodin, and Rene Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, and Applications: 10th International Conference, AIMSA 2002 Varna, Bulgaria, September 4–6, 2002 Proceedings 10*. Springer, 41–50.

[27] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2023. Modeling parallel programs using large language models. *arXiv preprint arXiv:2306.17281* (2023).

[28] OpenAI. [n. d.]. *ChatGPT*. https://chat.openai.com/

[29] OpenAI. [n. d.]. *GPT4*. https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4

[30] Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, and Hajimu Iida. 2024. Parallelizable Loop Detection using Pre-trained Transformer Models for Code Understanding. In *Parallel and Distributed Computing, Applications and Technologies*, Ji Su Park, Hiroyuki Takizawa, Hong Shen, and James J. Park (Eds.). Springer Nature Singapore, Singapore, 32–42.

[31] Bill Pottenger and Rudolf Eigenmann. 1995. Idiom Recognition in the Polaris Parallelizing Compiler. In *The 9th ACM International Conference on Supercomputing (ICS'95)*. ACM Press, 444–448. https://doi.org/10.1145/224538.224655

[32] S Prema, R Jehadeesan, and BK Panigrahi. 2017. Identifying pitfalls in automatic parallelization of NAS parallel benchmarks. In *2017 National Conference on Parallel Computing Technologies (PARCOMPTECH)*. IEEE, 1–6.

[33] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[34] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867* (2023).

[35] Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. 2023. Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation. *arXiv preprint arXiv:2309.07103* (2023).

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[37] T. Yuki and L.N. Pouchet. [n. d.]. *PolyBenchC-4.2.1*. Accessed on 2021-01-05.

[38] Min Zhao, Bruce R Childers, and Mary Lou Soffa. 2005. A model-based framework: an approach for profit-driven optimization. In *International Symposium on Code Generation and Optimization*. IEEE, 317–327.

Received 30 May 2024; Under revision