State of The Art LLM Tools to Detect and Classify Performance Related Issues in HPC Code

Group Members:

Can Korkmaz 28068
Cem Kaya 26440
Yağız Tüfek 28412

Date: 07/06/2023

Introduction

The sudden appearance and inclusion of artificial intelligence, and specifically LLM (Large Language Models) into the common life is partly in debt to its relative accuracy in answering and solving specific problems in wide area of subjects. One of the pioneers and early successes was the OpenAI's ChatGPT tool that utilized GPT-3.5 api behind the scene, which were widely praised for its ability to write and review code while being language agnostic. This early success made ChatGPT to be widely used by developers. In this project, we tested LLM tools for the purpose of detection, classification and performance improvement capabilities of performance related issues in HPC code. One flexibility of the LLM models can be attributed to the ease of adapting differing requests through natural language, which was the case for our trial. Our trials with multiple tools suggested us the usability of currently state-of-the-art LLM models for code analysis in HPC code, although the shortcomings of these models are severe enough to limit the tools effectiveness, mostly rooting from the limited context size and lack of effective context management system.

Methodology

Part 1: Finding And Classification of HPC Performance Bugs

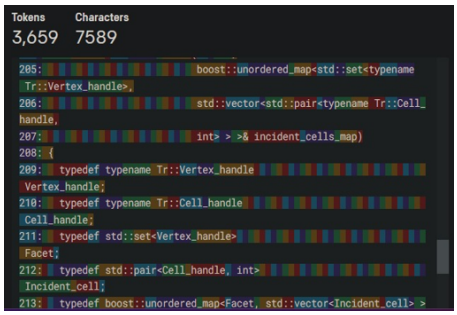
We branched off to test different tools and methodologies for the purpose of detection and classification of HPC code. The models we used included ChatGPT with GPT-4 and GPT-3.5, GPT-3.5 API, LaMMa-7b (4-bit quantized), Alpaca, OpenAssistant/oasst-sft-6-llama-30b with HuggingChat. Although GPT-3.5 based tools (chat interface, API) were modestly successful, and LaMMa-7b model were not able to detect most of performance issues, the most successful tool was ChatGPT-4, which is able to point out performance issues with comparable accuracy to the manual analysis, without extensive access to or fine-tuning with source code of the project (3, 4). The performance issues detected and

classified with ChatGPT-4 were highly accurate with the manual classification of the RQ1 database sheet. One shortcoming of the ChatGPT-4, which is further exacerbated by the comparatively smaller context sizes of the other mentioned models, is the context size limitation and the management of the context storage throughout the conversation. Although with the first few commits ChatGPT-4 tool successfully classified most of the performance related problems, the later prompts with codes didn't meet with the same quality of responses. We hypothesize that this is mainly due to the inefficient use of context size. This shortcoming led us to use shorter conversations with the same initial prompt engineering, such as introduction of categories and purpose of the project, and the format of the answer that we wanted for.However due to GPT-4 api being in closed beta we could not automate the task thus we have utilized GPT3.5 api for analysis of the models capabilities.

Results of Part 1

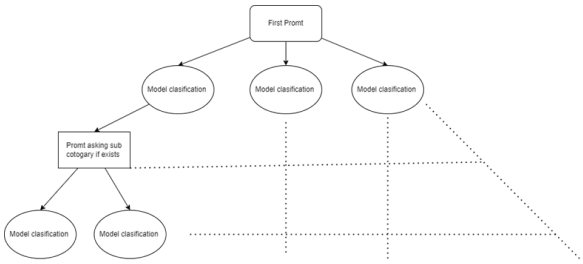
Out of the tested LLM tools, the models from OpenAI were the most promising in terms of finding mistakes, classifying and suggesting changes. GPT3.5 was able to find the cache locality performance bug with the first commit of RQ1 which manual analyses also suggested that there was a cache locality problem. However GPT4 was not only able to find cache locality issues but also other performance bugs such as missing parallelism and frequent function calls (4). It seems like the issues GPT3.5 finds are enough for our purpose and since we have an api access to GPT3.5 we are able to automate the process of finding and labeling the issue. That is why we plan to continue using GPT3.5 api for the second phase. Unless we can obtain access to GPT-4 api especially the 32k context size version or a similarly advanced model. The use of a standard prompt gives us the opportunity to pivot to differing LLMs with minimal adjustments.

We have used the GitHub API to get all the data from the commits mentioned in the paper, sourced directly from GitHub. And then processed the patch diff to obtain the old version of the code to use in the prompting of the models via an api. We made recursive prompts for sub-sub category classification. This approach was inspired by *autogpt*. We have done the work for depth-1 category classification and depth 2 classification. However after the detailed analyses of the data we determined this was not a productive avenue to investigate even further.



While processing the codes from the github repos to make the maximal utilization from the limited context size of the model we have removed the unnecessary chars from the code.

As previously noted, we evaluated various models for the subsequent phases of the project, primarily by comparing their performance on basic HPC issues discussed in CS_406. Both Chat GPT-4 and GPT-3.5 were capable of identifying the problem and proposing multiple solutions, while LaMMa struggled to comprehend the question. Consequently, we concluded that without additional fine-tuning, these models are unsuitable for our specific use case. Nevertheless, the option to refine them using instruction-following demonstrations remains open. Additionally, to enhance the performance of weaker models, the inputs and outputs from stronger models can be utilized. Alternatively, as suggested in the WizardLM paper, more robust models can generate examples to improve the performance of less powerful models. However such goals are currently outside of our scope.



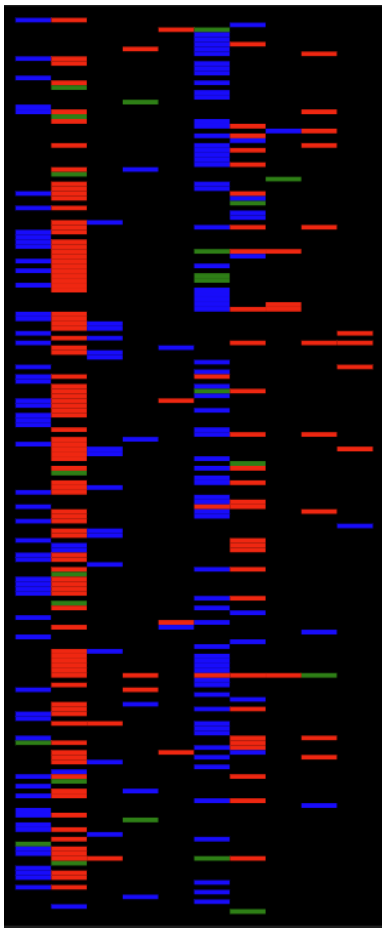
For the prompting style we have investigated a few different styles of prompting with manually selected examples. At the end we used a combination of Role Prompting, zero shot Prompting and One Shot Prompting. We have tried to add chain of thought Prompting strategies which usually result in higher model capabilities however due to context size limits this proved more error prone than the other tried combinations. On the other hand, the new tree of thought prompting style seems promising for such tasks under similar constraints, however we didn't have time to test it and the use of multiple models to search for consistent and higher probability decision states is exponentially higher then the basic prompting strategies we have utilized.

This system prompt is designed to convince the model that it is good at performance engineering and gives it a specific mission to identify and classify potential performance issues in a given piece of code patch by giving it the role of an assistant which does so. By explicitly specifying the model's role and its expected capabilities, the prompt provides a

clear framework for the AI's response, potentially increasing the quality and relevance of its output.

The specificity of the task also helps to reduce ambiguity in the model's responses. By identifying particular categories of performance issues which were taken from the given example project, the model is guided to not only find potential problems but to consider them within the context of these specific categories. This hopefully pushes the model to go beyond simple error detection and into the realm of detailed, structured analysis. This multi-faceted approach encourages the model to examine code performance from a holistic perspective, ensuring a comprehensive analysis that doesn't miss potential issues hidden in the complexities of modern software development.This prompt was designed to harness the model's capabilities in a highly structured, focused manner, which could yield more useful and targeted outputs.

The actual question prompts were “Please categorize the following patches into a single one of the listed categories ? {patch}.” the patch was the parts of the code which were visible at the commit from the github. We have tried just giving the removed parts, giving the removed and unchanged parts, just giving added parts, giving unchanged parts and newly added parts and many other combinations. However, these did not affect the result significantly if the patch type contained enough code.



This picture shows the classification predictions, the ground truth and if the model predicted correctly or not for the initial category of the patches performance bug type. There are horizontal and vertical lines. Every vertical line denotes a performance bug type : Inefficient coding for target micro-architecure, Missing parallelism, Parallelization overhead/inefficiency, Inefficient Concurrency control and synchronization, Unnecessary process communication, Inefficient algorithm /data-structure and their implementation, Inefficient memory management, I/O inefficiency, Unintentional Programming logic error, Inefficiency due to new compiler version,

Every horizontal line denotes a code patch.

A black line segment denotes no bug and no bug prediction.

A green line segment denotes a bug and bug prediction.

A Red line segment denotes the model predicted no bug while there was a performance bug.

A blue line denotes a model predicted a bug while there was no such bug.

Class	Ground Truth: 0	Ground Truth: 1
Model Predicts: 0	1522 (Both agree)	163 (Model incorrect)
Model Predicts: 1	152 (Model incorrect)	23 (Both agree)

This gives Accuracy: 0.83 Precision: 0.13 Recall: 0.12 F1 Score: 0.12 False Negative Rate: 0.87. (The numbers are scaled between 0 and 1)

Accuracy - Our model correctly predicted the outcome 83.06% of the time. This might seem impressive at first glance. However, accuracy can sometimes give us a distorted picture if our data is skewed or imbalanced, meaning one class has far more instances than the other. This is statistically significantly higher than a naive classifier for this problem but other metrics give more insight to performance of the model on this task. Which is due to unbalance in the task.

Precision - This measure tells us how often our model is right when it predicts the positive class. At 13.14%, our model's precision is relatively low, indicating that our positive predictions are incorrect a significant amount of the time.

Recall - This metric gives us the percentage of actual positive cases our model managed to capture through its predictions. Here, our model detected only 12.37% of all positive cases. This low recall suggests that our model is missing a large number of positive cases—it's not finding all the instances it should be finding.

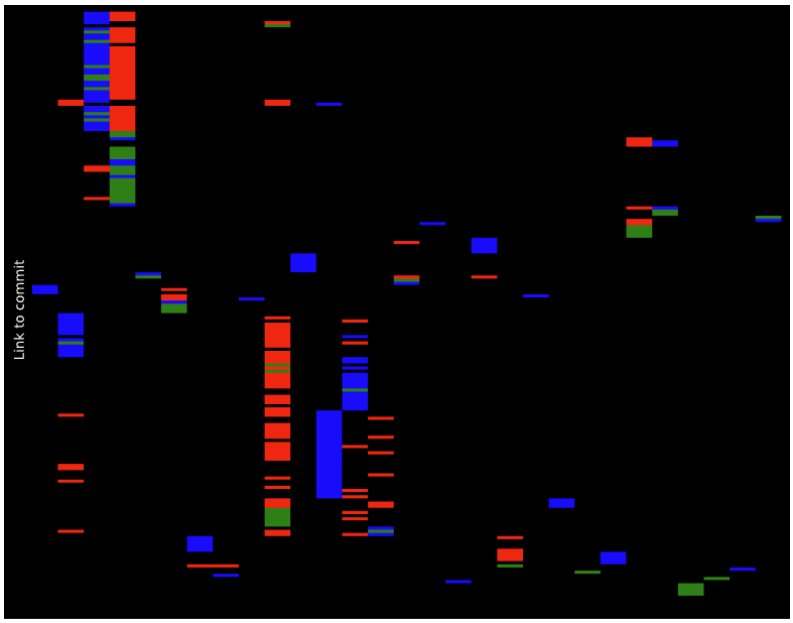
F1 Score - The F1 Score is a blend of precision and recall, giving us a single measure that tries to balance the two. Ideally, we want an F1 score close to 1. Our model's F1 score is just 12.74%, which is rather low, indicating our model's performance is not balanced when it comes to precision and recall.

False Negative Rate - This measure tells us how often our model wrongly predicted a positive case as negative. With a high rate of 87.63%, our model is frequently predicting positive instances as negative—this is a substantial issue.

In summary, while our model shows a reasonable accuracy rate, other metrics suggest substantial areas for improvement. The model often fails to identify positive instances and frequently mislabels them—a problem indicated by the low precision, low recall, and high false negative rate. Future steps could include exploring different models, refining feature engineering and more advanced prompt engineering, or balancing our dataset if it is found to be skewed.

In light of the above we have further investigated the effectiveness of adding more details to the prompt for more advanced classification this refers to the depth 2 classification of the system. However this means the model takes the ground truth classification of code as an input and tries to predict the subcategory of the code. This time the system prompt was "As a software high performance engineering assistant with exceptional intelligence, your task is to analyze a given piece of code and identify any performance issues. Specifically it is known that this patch solves a problem about {category}, you are expected to classify any performance problem and the given solution you detect into only a single one of the following categories: {sub_cat_str}".

The main prompt was "If you think this solves a {category} problem please categorize the following patches into a single one of the listed categories ? {patch} ? " . The if statement in this second prompt gave the model more flexibility on the categorization since even in the ground truth there were commits with no sub^n--categories. This resulted in the following results.



Every horizontal line denotes a code patch.

A black line segment denotes no bug and no bug prediction.

A green line segment denotes a bug and bug prediction.

A Red line segment denotes the model predicted no bug while there was a performance bug.

A blue line denotes a model predicted a bug while there was no such bug.

The classes were : Unncessary locks, Unnecessary operation/traversal/function call, Memory/Data locality, Micro-architectural inefficiency, Under-parallelization, Unnecessary synchronization, memory leak, unnecessary data copy, Lock management overhead, Inefficient data-structure library, Inefficeint thread mapping / inefficient block size / Load imbalance, Expensive operation, Redundant operation, Usage of improper data type, Over-Parallelization, Instruction level parallelism, over parallelization, small parallel region, Redundant memory allocation, Unncessary strong memory consistency, Frequent function call, Insufficient memory, repreated memory allocation, Task parallelism, Vector/SIMD parallelism, boundary condition check, sequential I/O operation, Slower memory allocation library call, GPU parallelism.

With this version of the task the model performed:

Class	Ground Truth: 0	Ground Truth: 1
Model Predicts: 0	5202 (Both agree)	130 (Model incorrect)
Model Predicts: 1	122 (Model incorrect)	56 (Both agree)

This gives us:

Accuracy: 0.954, Precision: 0.314, Recall: 0.301 F1 Score: 0.307, False Negative Rate: 0.698.

This was significantly better than the previous Version of the task, thus giving more information to the model increases the model capabilities. However this version of the problem has more constraints over it and when a naive classifier is consider in this version of the task the models performance improvements appear less significant. Since with these constraints a model which always classifieds the same output would perform better compared to the previous task. If we take such a model as a baseline there was no significant improvement on the model's capabilities. In summary, despite a high accuracy, the model's performance seems inadequate, particularly in its correct identification of positive instances. Evidenced by the high number of false negatives (missed positives) and a considerable number of false positives (low precision), the model requires improvement.

Conclusion to Part 1

Although the model showed reasonable accuracy, other metrics indicated significant room for improvement, particularly in the model's ability to correctly identify positive instances and frequent mislabelling, as highlighted by the low precision, low recall, and high false negative rate. Future considerations could include exploring different models, refining feature engineering, advancing prompt engineering, or balancing the dataset if skewness is detected.

In conclusion, while the current version of the GPT3.5 model may not be sufficient to supplant performance engineers, it exhibits enough capability to serve as a heuristic for an engineer. The model's output should be viewed as suggestions, and the possibility of the model's error should always be taken into account when utilizing the model. Nevertheless, the performance of this model doesn't necessarily set a benchmark for the potential of future models. As shown in the GPT-4 white paper, advances in model capabilities can exhibit characteristics of phase transitions rather than a gradual progression, indicating that significant leaps in performance are possible with the next iteration.

Part 2: Fixing The Categorized HPC Performance Bugs Through The Use OF Large-Langue-Models (LLM) Systems

We studied the effectiveness of the GPT-4 model for the purpose of optimizing the given HPC code with known and specified performance bugs. On this objective, we first selected commits that are categorized in our reference project and got code from the course material[1]. From the hand selected commits, we extracted the code snippets that are specifically related to our categorization and our reference's categorization, and either changed the code snippets to be adapted for micro-benchmarking suits or written from the scratch small code sections that still possessed the specific code bug. This is where we used the OpenAI GPT models for fixing the performance issues. As the prompt to the ChatGPT client, we included the specification of the problem in varying detail, that is the depth of the categorization of the problem as categorized in our reference study. Although the accuracy of the GPT3.5-turbo model which we have used for classification of the performance bugs were near 83%, the precision and recall metrics have shown that the model wasn't fit for relying for the second part of this project, thus we used the hand-classifications of the RQ1 database so that we could isolate and assess the performance of the GPT4 - and less often GPT3.5 - models for fixing the bug given the categorization of the bug.

We then created micro-benchmark suits for the prepared code snippets two be able to benchmark and compare the performances of both the code which possessed the performance bug and the version that is fixed by the ChatGPT-3.5, or ChatGPT-4 models. Although the OpenAI API for Python was used as the primary driver for the classification of the performance bugs given the categorization, for the second part of the project, the most time complex part was creating the test benchmarks and the few number of test benchmarks made it less feasible and unnecessary to consume the GPT-3.5 and GPT-4 models through the OpenAI API. Aside from the few number of test cases, the limitation of the OpenAI API's in our particular scenario was that it only provided the GPT3.5-turbo model and lacked the GPT-4 model exposure for testing, thus we opted to use OpenAI's Chat client for this part of the project. Although the *24 May ChatGPT3.5-turbo* model showcased comparable results to *24 May ChatGPT-4 model*, we primarily used *ChatGPT-4* client to create the performance bug fixed version of the given code snippets, and the below experiments and results are processed with this model.

---

[1] Azad, Md Abul Kalam. (2023, March 7). An Empirical Study of High Performance Computing (HPC) Performance Bugs. The 20th International Conference on Mining Software Repositories 2023 (MSR 2023), Melbourne, Australia. https://doi.org/10.5281/zenodo.7731544

The study was split into three sub-parts, that are each focusing individually on the HPC performance bugs in one of the categories of Architectural, OpenMP and CUDA computation codes. Below are the experiments for divided into 3 main categories and sub-divided for specific performance bug issue.

Experiments & Tests

Architectural speed ups

- Cache associativity
- Branch prediction
- False sharing
- Prefetch
- Simd
- Spatial locality
- Loop unrolling

Architectural speedups by making small changes to code can come from a number of strategies. These optimizations are largely dependent on the specific architecture at hand , different architectures have different strengths and weaknesses and differing ways to utilize them to obtain speedups.

Number of test have been performed to determine the models performance on fixing such performance bugs:

Bug type	Could the model fix the bug	Details
Cache associativity	Partial	Can identify the bug and change the computation to solve it.
Branch prediction	NO	Can sort the data to make it more friendly to the cpus branch predictor.
False sharing	Yes	Uses padding to solve.
Prefetch	NO	Trying to use intrinsics makes the code even slower.
Simd	YES	Can use AVX and other SSE instructions to make the code faster.
Spatial locality	YES	Can turn a row major code to column major and even use more advanced techniques to make matrix matrix multiplication faster.
Loop unrolling	YES	Can unroll the loops. However, it does not produce the optimal amount of unroling.

The used prompt had this signature:
“
This code has a {bug category } performance bug
{CODE}
Can you fix it ?
”

For example:

“

```
This code has a performance bug
void multiply (vector<vector<int>>& A, vector<vector<int>>& B, vector<vector<int>>& C) {
   for(int i = 0; i < SIZE; ++i) {
      for(int j = 0; j < SIZE; ++j) {
         for(int k = 0; k < SIZE; ++k) {
            C[i][j] += A[i][k] * B[k][j];  } }}}
can you fix it ?
```

“

From testing 10 functions it improved 7.5 of them. Cache associativity was partially incorrect in which the computation and the end result was altered. However since the tested code was a dummy micro benchmark it was not fully wrong. There were more than one class of tests for some of the problem types. These were SIMD and Spatial locality.

For more details please check out the part2 directory in the repo.

OpenMP

- Race Conditions
  - Synchronization
  - Reduction
  - Padding
  - Critical section
- Tasks
- Nested Loops
- Vectorization
- Numa

OpenMP (Open Multi-Processing) is an industry-standard API (Application Programming Interface) for shared-memory parallel programming. It provides a set of directives, runtime library routines, and environment variables that enable developers to parallelize their code and harness the computational power of modern multicore processors. OpenMP abstracts many low-level details of

thread management, synchronization, and data sharing, making it easier for programmers to exploit parallelism in their applications without diving into complex parallel programming concepts.

A race condition is a phenomenon that occurs in concurrent programming when multiple threads or processes access shared data or resources without proper synchronization, so OpenMP is there to aid developers with such synchronization methods such as reductions and barriers.When it was asked GPT3.5 to find issues with a code that has a missing critical section it failed to determine the issue until it was pointed out by the prompter.

In OpenMP, tasks provide a way to express fine-grained parallelism by dynamically creating and scheduling independent units of work. Unlike the traditional parallel for loop construct, which divides work among a fixed number of threads, tasks allow for more flexibility in distributing work among threads at runtime. It lets developers divide code into chunks that can be executed in parallel. However some tasks might depend on other tasks output or other variables. It is essentially a constraint on how the tasks should be executed. When inputted a task dependency code asked to make the parallelization better GPT3.5 was able to give the correct result on the first try without needing any other prompts.

GPT3.5 was successful in the parallelization of the nested loop.

Vectorization is a technique used in computer programming and optimization to enhance performance by exploiting the capabilities of vector processors or SIMD units within modern processors. It involves executing multiple operations simultaneously on multiple data elements (vectors) using a single instruction. Vectorizable operations typically involve performing the same operation on multiple data elements, such as addition, multiplication, or other arithmetic/logical operations. For efficient vectorization, the data accessed by the vector instructions should be properly aligned in memory. Padding or reorganization of the data layout to ensure proper alignment can be necessary. When inputted a vectorizable code and asked "can this be vectorized" GPT3.5 was able to give a vectorized code in the first try without needing any other prompts.

NUMA (Non-Uniform Memory Access) is a memory architecture used in multiprocessor systems where each processor has its own local memory. NUMA systems are designed to improve memory performance in large-scale servers and supercomputers by reducing memory bottlenecks. By distributing the memory across multiple nodes, it allows each processor to access its own local memory faster. When dealing with NUMA architectures, OpenMP can be utilized to optimize data placement and thread assignment. OpenMP provides mechanisms to control thread affinity, which is the mapping of threads to

specific processing units (cores) in a system. By aligning thread affinity with NUMA nodes, you can minimize remote memory accesses and improve overall performance. When asked GPT3.5 to make a code numa compatible it did on the first try without needing any other prompts while giving additional information like “in this modified code, each thread is manually assigned to a specific core using OpenMP's affinity settings (omp_set_affinity). The number of threads per core is calculated based on the total number of threads and the number of available cores. The computations are performed within each thread using the specified affinity. It's important to note that thread affinity settings may not guarantee strict NUMA locality, as the operating system's scheduler can still move threads across cores. ”

After asking 10 diverse OpenMP inquiries to GPT3.5 regarding related concerns, it impressively resolved the issues flawlessly in 9 instances, while in one particular problem it reached the solution with the assistance from the prompter. Also the GPT3.5 failed to detect the issue with the critical so it was asked the same question(missing critical section) with another piece of code and it was successful on the first attempt.

main Issues	sub-clauses	total asked	total correct
Race_conditions			
	omp task	1	1
	critical section removed	2	1
	padding	1	1
	atomic	1	1
	reduction	1	1
Nested_Loops			
	parallelize	1	1
Tasks			
	privates	1	1
	depend clauses	1	1
Numa			
	numa_compatible	1	1
Vectorization			
	simd_loop	1	1

CUDA

a. Data Type Optimization

In this example, the effective bandwidth utilization of the kernel with 32 bit indexing and array fetching and insertion appears to be 2.5 times of the performance bugged kernel's effective bandwidth utilization. The straightforward reasoning can be made that because the unsigned long long takes 8 bytes instead of 4 bytes that can, a warp needs to do approximately half as many memory access when inserting into the array. It's unclear how much of the performance gain necessitates front he calculation of the index and usage for the array indexing of unsigned int time compared to unsigned long long time, but cuda threads known to exhibit better performance with 32 bit integer types rather than 64 bit integer types.

```
//code benchmark file: spatial_locality_suite.cu
64bit unsigned long long data type indexing performance
Time to calculate results on GPU: 0.090048 ms
Effective bandwidth: 444.207533 GB/s

32 bit unsigned integer indexing performance
ime to calculate results on GPU: 0.075520 ms
Effective bandwidth: 1059.322015 GB/s
```

b. Reducing Usage of Constant Memory Section of Global Memory

In this example, we created a test suite to test the execution time differences between two kernels both of which accessed the particular location with their threadIDx and blockIdx'es, but one accessing the global memory directly and the other to the constant section of the global memory. The commit we investigated had found that in their code, the tradeoff between reducing the usage of constant memory space in the place of non-constant area of the global memory affect the performance of the program negligibly, or not at all[2]. Although that could've been the case for long runnign tasks, our benchmark suit showed speedups of 4 for the kernel accessing to the constant section of the global memory. Although using constant memory section may indeed lead to performance gains, it's limited in size and the compiler

[2] See Commit d8d8c439c8d0a43c0f92b11fd06133be80754ab8 in the ArrayFire GitHub repository for detailed changes.

gave warnings for using two much constant space allocation for the matrix sizes of 2048x2048, 4096x4096, and larger.

```
//code benchmark file: memacess.cu
For 4096x4096 matrix access.
Time to calculate results on GPU: 0.023040 ms
Time to calculate results on GPU: 0.006208 ms
```

c. Loop Level Unrolling in Cuda Kernels

For this performance bug in CUDA C++ code, we created a micro test benchmark suite with a kernel that can be unrolled to improve performance, though even with the category sub-sub and sub-sub-sub category specification, ChatGPT-4 didn't change the code in the same way as they did in the commit[3], using #pragma unroll, but unrolled the loop for four iterations at a time and reduced the number of for loop iterations by 4 times. Although this is could've improved the performance in some different computational context, in this micro benchmark suite, the execution time wasn't improved and instead got slower.[4]

```
Execution time Unrolled: 0.00645669 seconds
Execution time Unrolled: 0.00753641 seconds
```

d. Spatial Locality Exploitation

A very important concept for performance optimization in CUDA programming is exploatiotation of the spatial locality so that the effective bandwidth utilization is increased and overall performance gain is obtained through fewer memory accesses, which often constitute big parts of the total execution time. Like the other experiments on CUDA performance bug fixing through the use of ChatGPT-4 model, we've first fed the performance bugged code to the ChatGPT-4 with the categorization of the performance bug. ChatGPT-4 adeptly solved the performance bug in the micro benchmark test suite and the ChatGPT-4 optimization resulted in speedup of two.

---

[3] See Commit 928e77aed1db65680f9b6bfbfa4d7791bdb32511in the ArrayFire GitHub repository for detailed changes.

[4] Linkto a conversation with ChatGPT-4 containing the promp and the answer: link

```
//code benchmark file: spatial_locality_suite.cu
For Performance Bugged Kernel: Execution time: 0.000043
For ChatGPT-4 Locality Optimized Kernel: Execution time: 0.0000243
```

References

1. Yao, Shunyu & Yu, Dian & Zhao, Jeffrey & Shafran, Izhak & Griffiths, Thomas & Cao, Yuan & Narasimhan, Karthik. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models.

Appendix

For further information, our GitHub repository contains our tools to process HPC code commits and our experiments with OpenAI APIs:

https://github.com/Cem-Kaya/Promt_Engineering_for_Performance_Engineering_with_AI