PerfTuner
Developer Documentation

Thomas Gantz Alexander Goebel Soumili Samanta

March 20, 2024

Abstract

The goal of the PerfTuner project is to exploit LLMs (in our case we use ChatGPT from OpenAI) to optimize C++ functions by using advanced vector extensions (AVX). AVX intrinsics are C++ functions that use single instruction multiple data (SIMD) functionalities included in AMD and Intel CPUs.

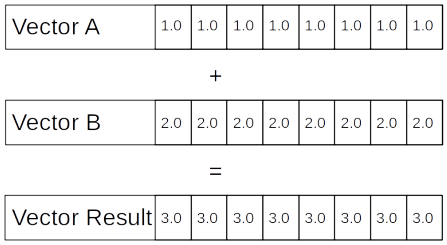

Figure 1: Example for AVX. 8 floating point numbers can be added with a single instruction.

1 Introduction

In this document we explain the main ideas of our implementation. In chapter 2 we describe the structure of our repository. The main program flow can be observed in chapter 3. All important files are described in chapters 4, 5 and 6. In chapter 7 we describe some of the experiences we made during the project and in chapter 8 we suggest some possible steps for future improvements of this project.

2 Repository

The repository can be found under `https://github.com/pvs-hd-tea/23ws-PerfTuner`. The root directory consists of the two folders *Product* and *docs*. The folder *Product* contains all software related files whereas the folder *docs* contains the documentation, including this file.

In *Product* you can find all needed functions. They are explained in detail in chapter 4. Furthermore it contains the folders *Problems* (5), *Snippets* (6) and *Statistics*. All examples for the automatic evaluation (4.4) can be found in *Problems*. Code snippets used for guiding the LLM are in the folder *Snippets*. Finally *Statistics* contains graphs from the automatic evaluation loop.

3 Program Flow

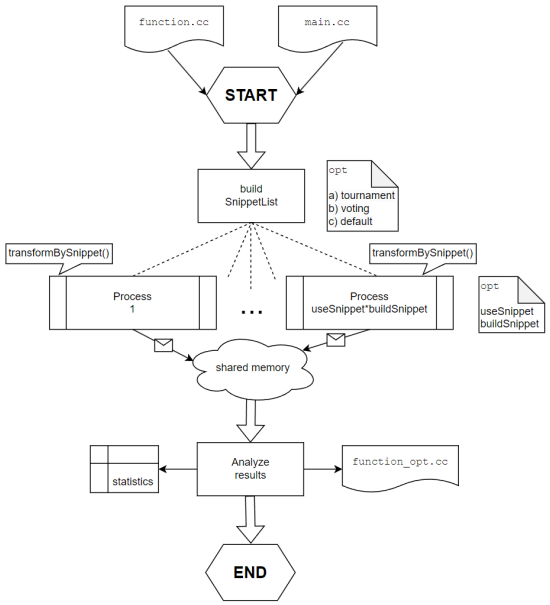

Figure 2: Flowchart illustrating the architecture

4 Files

4.1 perftuner

This function is the entry point for the user. It can be called in a terminal by *python3 perftuner.py*. The user is asked to supply the path of the relevant files, i.e. a file called *function.cc* which contains the code to be optimized and a file called *main.cc* containing the testing environment as explained in 4.7. The user is also asked how many snippets (see 4.3) and how many attempts by the LLM should be tried for the optimization. Finally, he is asked to choose the method (voting (4.5), tournament (4.6) or default) for finding the right snippet.

4.2 PerfTunerClass

The `PerfTuner` class is the core of the program. In its constructor, the settings of a transformation are configured when a `PerfTuner` is instantiated, and the invocation of its `do` method initiates an actual transformation, returning its statistics in an array.

It is crucial to specify the paths for locating the function to be transformed (*function.cc*) and the testing environment (*main.cc*). Additionally, the user can determine the methodology for constructing a `SnippetList` that ranks the snippets by relevance to the problem. Options include tournament, voting, or the default, where snippet 10 (LU decomposition) is selected, typically yielding optimal results for most problems, although an alternative can be specified. The user can also specify the number of snippets from the list to use in `runs_useSnippet` and the frequency of attempts for each snippet in `runs_buildSnippet`.

Upon completion, the results of all processes are presented to the user, including the success rate per snippet and the `transformationQualityAverage`, which represents the averaged sum of the status of all transformations per snippet, serving as an index of the effectiveness of a snippet. Finally, the best *function_opt.cc* is copied to a readily accessible location for the user within the `PerfTuner` class.

The `do` method initially constructs a `SnippetList` based on the specified settings and then executes `runs_useSnippet * runs_buildSnippet` transformation processes accordingly. It's noteworthy that each process generates its own output files. Communication occurs via the shared memory `jobsStatusArray`. Crucial information is conveyed through status codes, with the following interpretations:

- `-99`: The process wasn't active
- `-4`: FAIL: *function.cc* didn't compile
- `-3`: FAIL: *function_opt.cc* didn't compile
- `-2`: FAIL: The results are not identical
- `-1`: FAIL: *function_opt.cc* was slower
- `0`: SUCCESS

4.3 transformBySnippet

This function does the actual transformation and contains the main meta strategy of our approach. The parameters are the function to be optimized and a C++ snippet chosen by 4.5, 4.6 or a default snippet to speed up the process.

As a first step the LLM is trained by the following mechanism: We ask the LLM to optimize the given snippet with AVX. However, we also directly supply the answer with a optimized version that we have written ourselves. We then ask the LLM to explain this optimization. By this trick we guide the LLM on how to transform the function in the last step.

As a second and third step we use the LLM to identify all main sub tasks of the function and to determine which sub tasks can be optimized with AVX.

In a final step we use the list of sub tasks and the guiding information of step one to optimize the function.

Since the OpenAI API does not remember the last conversations we always have to supply the results of the former conversation.

4.4 AutomaticEvaluationLoop

`AutomaticEvaluationLoop.py` aims to evaluate the product proficiencies and intricacies. It iterates over a set of predefined problem classes and their corresponding problems, attempting judge the results of producing optimized code. The process involves invoking the `PerfTunerClass` , which performs transformations and provides statistics on the optimization attempt.

For each problem, collection and storing of appropriate parameters, and optimization statistics are done.

`perftuner.do()` from `PerfTunerClass.py` stores all the statistical values required in list format. These statistics help in producing the following tables and graphs:

- `Table1`: Metrics determining problems solved in class-wise manner.
- `Table2`: Provides problem wise statistics.
- `Table3 Snippet success rate`: Provides percentage of times a particular snippet transformed orginal to optimized code.
- `Graph1 Solved status`: Provides graph determining problems solved and unsolved by our product.
- `Graph2 Speedup`: Provides bar graph representing execution time of original code and optimized code.
- `Graph3 Snippet chosen`: Provides a graph showing which snippet was chosen by which Problem.

4.5 findSnippetList

The function `transformBySnippet` requires a C++ code snippet closely related to *function.cc*. For this purpose, we have compiled a snippet library (6) containing various operations. This function is one of the methods for constructing the hierarchical `SnippetList`, employing a voting strategy to determine the most suitable snippet for the given function. Additionally, a crucial aspect of the function involves returning the `SnippetList` in a specific format, achieved by splitting the output string from ChatGPT in an array of strings (using a stack functionality) and subsequently employing pattern matching.

4.6 findSnippetListByTournament

Just like 4.5 this function finds the best snippet suited for the given function. However, in contrast to the voting strategy a tournament strategy is used. Two snippets of the library are taken at random and with the LLM it is determined which snippet is better suited for the given function. The winner is kept and the looser is discarded.

This procedure is repeated in a tournament style. An overall winner, the runner up and the two remaining final 4 candidates are returned as as snippet list.

4.7 test

This script is designed to assess the compilation and runtime performance of C++ code compared to its AVX-optimized version.

It begins by compiling both the original C++ code and the AVX-optimized version using "g++". If either compilation fails, it returns an error code to signify the failure. Subsequently, it executes both compiled programs and measures their respective runtime using the 'time' module. After execution, it compares the outputs of both programs. If there's a discrepancy between the outputs, it flags this as an issue.

Throughout the process, the script handles platform differences by adjusting the commands for Windows and non-Windows environments accordingly.

4.8 askChatGPT

The `askChatGPT` function is designed to interact with the GPT-3.5 model from OpenAI, utilizing its chat completion capabilities. It takes input file names, an output file name, and a message as arguments.

If input files are provided, it reads the content of each file and appends it to the prompt string. It then creates an instance of the OpenAI class and requests completions from the GPT-3.5 model using the provided message and the constructed prompt. After receiving a response from the model, it extracts the generated text and checks if an output file name is provided.It constructs the output file, writes the generated text to the file, and closes any opened input files.

4.9 TaskCode

This file contains the task code that is performed by every process instantiated in the `PerfTuner` class to do a transformation. The actual transformation is performed by `transformBySnippet`. This is the wrapper for the transforming processes. Note the parameters `i` and `j` that identify a process and are essential for communication with the main process of the `PerfTuner` class over the shared memory `jobsStatusArray`. Also note the different file paths since every process has its own output files for its transformation.

4.10 .env

This file should contain a valid key (token) for using OpenAI. The format is: OPENAI_API_KEY=xxx where xxx is the corresponding key.

5 Problems

All examples for the automated evaluation are in the folder *Problems*. They are divided into three classes of difficulty. For each example there is a sub folder containing the C++ files *function.cc* and *main.cc*. As described in 4.7 *function.cc* is the code that needs to be optimized and *main.cc* contains the test environment.

6 Snippets

As described in 4.3 we use code snippets to guide the LLM in the optimization process. Currently we have 11 such snippets and the corresponding (by hand) optimized versions. The library contains all snippets.

7 Our Experiences

Throughout our project, we've explored various approaches and gained diverse experiences, with many proving unsuccessful.

7.1 Plain ChatGPT

Initially, we attempted to prompt immediate transformations by experimenting with different prompts, many of which were already highly technical or tailored to the specific problem. However, this approach seldom yielded satisfactory results. We've since learned that it's necessary to engage ChatGPT in a more nuanced manner, aiming to put it in the right mindset for productive interactions.

7.2 Pseudo Code Strategy

As a different meta strategy we ask the LLM to translate the C++ snippet and the function into pseudo code. Consequently, we asked to re translate the pseudo code into C++ code with AVX intrinsics. However, the results obtained by this approach were not useful and did not produce compilable code.

7.3 Using Google for Transformations

The code was created in order to make transformation easier with the support of Google search. The code started by reading C++ code from a file named *function.cc*. Then, it utilized OpenAI's ChatGPT to generate a concise Google search query based on the code's functionality. After performing a Google search and extracting relevant content, the script interacted with ChatGPT again to obtain AVX2 code snippets tailored for optimizing the C++ code. Once received, these snippets were integrated into a runnable program, which was then stored in files for further use. Overall, the script automated the process of optimizing C++ code using AI-driven understanding and web scraping techniques. However, the results did not match up to the benchmark by our current product.

7.4 Issues with OpenAI

There's a possibility of exceeding the token limit per minute imposed by OpenAI during the transformation process, especially when utilizing multiple cores (though even with just 4 cores, it can occur). At times, OpenAI's response times were exceptionally slow, leading to no answers for several minutes. In such cases, the program may hang indefinitely. Another issue we've encountered is the inconsistency in the output format. For instance, requesting a list may result in formats like "1. 2. 3." or "- - -". While we've addressed this by crafting prompts that usually yield the desired output, we've also implemented a "while true-try" approach to catch instances where the desired format isn't achieved.

8 Future Steps

If the project is continued, we would suggest the following steps:

1. Transform whole programs.
2. Speed-up: multiprocessing (in particular for finding the snippet).
3. Expand the library (use floats, doubles and integers; more complex problems, e.g. for Fourier Transformation).
4. Experiment with different prompts.
5. Use user feedback and snippets to improve the results (open the possibility for the user to supply a snippet).
6. Generate a *main.cc* automatically.
7. Train the model with the snippets (model fine-tuning).
8. Try more coding specific LLMs.
9. Implement iterative refinement (if there is a working result: make it even better).