ChatBLAS: The First AI-Generated and Portable BLAS Library

Pedro Valero-Lara
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
valerolarap@ornl.gov

William F. Godoy
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
godoywf@ornl.gov

Keita Teranishi
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
teranishik@ornl.gov

Prasanna Balaprakash
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
pbalapra@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
vetter@ornl.gov

***Abstract*—We present ChatBLAS, the first AI-generated and portable Basic Linear Algebra Subprograms (BLAS) library on different CPU/GPU configurations. The purpose of this study is (i) to evaluate the capabilities of current large language models (LLMs) to generate a portable and HPC library for BLAS operations and (ii) to define the fundamental practices and criteria to interact with LLMs for HPC targets to elevate the trustworthiness and performance levels of the AI-generated HPC codes. The generated C/C++ codes must be highly optimized using device-specific solutions to reach high levels of performance. Additionally, these codes are very algorithm-dependent, thereby adding an extra dimension of complexity to this study. We used OpenAI's LLM ChatGPT and focused on vector-vector BLAS level-1 operations. ChatBLAS can generate functional and correct codes, achieving high-trustworthiness levels, and can compete or even provide better performance against vendor libraries.**

***Index Terms*—Julia, JACC, metaprogramming, performance portability, high-bandwidth on-chip memory**

I. ChatBLAS

ChatBLAS [1] is the first AI-generated and portable Basic Linear Algebra Subprograms (BLAS) library that can be deployed in different CPU and GPU configurations, and the C/C++ codes implementing the different routines that compose the BLAS standard specification are generated by AI large language models (LLMs).

ChatBLAS provides a unified standard-based BLAS API on top of different back ends, thereby enabling the deployment of the same API on different hardware configurations without changing one line of code. ChatBLAS interacts with LLMs via libraries that provide the necessary capabilities to communicate with these models by using prompts as input and collecting the responses from the models in the form of text (e.g., programming language codes) as output. Once the codes are collected, they can be compiled, analyzed, and run by ChatBLAS. This study focuses on the use of OpenAI's LLM ChatGPT; however, other LLMs can also be used if a library with which to interact is provided.

We used the Julia [1] language and ecosystem to implement the ChatBLAS library (or a Julia package—ChatBLAS.jl—using the Julia terminology). Julia is a dynamic and just-in-time (JIT) compiled front end to the LLVM programming language that was designed to provide a powerful unifying strategy to close the gaps between scientific computing and data science. We implemented ChatBLAS in Julia to benefit from a relatively easy-to-use syntax similar to that of Fortran and from its JIT capabilities that make testing, packaging, support for different back ends, and other processes extremely easy. Additionally, Julia provides support for reference BLAS, which can be used to easily evaluate the correctness of the AI-generated BLAS codes, everything interactively and efficiently. Furthermore, any library implemented in any language (e.g., C or Fortran) can be compiled and used from a Julia terminal.

ChatBLAS comprises different back ends according to programming languages (e.g., C/C++, Fortran, Julia) or the programming model to be used (e.g., OpenMP, OpenACC, CUDA, HIP). ChatBLAS interacts with LLMs that generate C/C++ codes. LLMs can also generate codes implemented in other programming languages. For clarity, this study focuses on C/C++ codes and OpenMP, CUDA, and HIP models. ChatBLAS specializes in the AI generation of BLAS codes, and in this initial study, we focus on BLAS level-1 routine (*axpy*, *scal*, *swap*, *copy*, *dot*, *sdsdot*, *asum*, *nrm2*, and *isamax*), a set of highly specialized operations focused on vector-vector operations, which are popularly used in multiple HPC applications [2]. ChatBLAS is an extraordinary platform

[1]https://github.com/pedrovalerolara/ChatBLAS

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00010

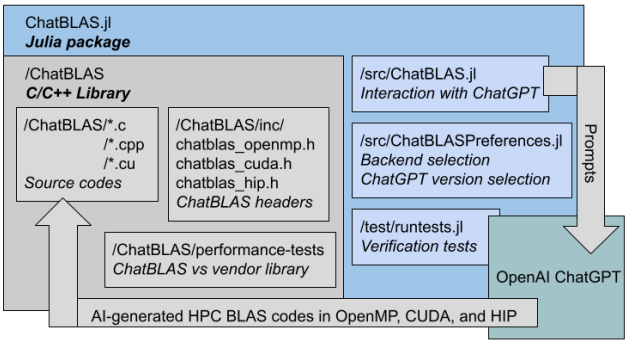

Fig. 1. ChatBLAS main components.

to evaluate and create novel LLMs' capabilities and LLM-based techniques such as prompt engineering or fine-tuning to generate highly optimized BLAS codes on different hardware platforms.

Figure 1 shows the main components of ChatBLAS. ChatBLAS is a Julia package composed of three main components: (i) those files related to the Julia package (files indicated by blue text in Figure 1), (ii) those files related to the AI-generated C/C++ HPC and portable library (files indicated by gray text in Figure 1), and (iii) the LLM model used to generate the source codes that implement the different BLAS routines.

ChatBLAS.jl manages the interaction with OpenAI ChatGPT to send the prompts to the LLM model and collect from that model the C/C++ codes that implement the different BLAS level-1 routines by using a specific programming model for parallelization (OpenMP, CUDA, or HIP). This is implemented in */src/ChatBLAS.jl* via Julia functions, and one function per BLAS routine was implemented. The codes collected are stored in the ChatBLAS library in */ChatBLAS/*.

The use of ChatBLAS functions is very simple with the JIT Julia environment. Entering *ChatBLAS.saxpy()* in a Julia terminal will run the Julia code that sends the prompt to LLM for this to generate an HPC code that implements the *axpy* BLAS operation for a specific target (e.g. C++ and CUDA). Julia provides reference BLAS support, which is used for the testing. Julia packages are created with a default testing support that can be used to evaluate the functionality. In our case, the testing consists of comparing the results of running the Julia reference BLAS and ChatBLAS libraries. The library must have been previously compiled or created and can use any language or programming model. The compilation can be performed from a Julia terminal too, and it is part of ChatBLAS.jl functions (*ChatBLAS.compilation()*).

Additionally, we provide the HPC and portable library (gray files in Figure 1). ChatBLAS stores the AI-generated codes collected from the OpenAI ChatGPT LLM model. We can find three extensions: *.c* for OpenMP codes, *.cu* for CUDA codes, and *.cpp* for HIP codes. ChatBLAS provides a separate header file and makefile corresponding to the different extension files and targets (CPUs, NVIDIA GPUs, and AMD GPUs). The headers and makefiles were not generated by AI. The performance of ChatBLAS vs. that of the vendor libraries can be evaluated in *ChatBLAS.jl/ChatBLAS/performance-tests/*, and the necessary scripts to compile and run the test codes are provided.

The last component is the LLM model. In this study, we use two generations of the OpenAI LLM ChatGPT: 3.5-Turbo and 4o. Although ChatGPT 3.5-Turbo is not the last version of the OpenAI's Generative AI model, it can be used to train the model (fine-tuning), whereas ChatGPT 4o cannot. We use this model to generate the HPC codes that implement the BLAS level-1 operations by using C/C++ codes and OpenMP, CUDA, and HIP models. Additionally, this model allows users to interact with it via different LLM-based techniques, such as prompt engineering and fine-tuning (only on the 3.5-Turbo version).

II. Analysis

In this section, we evaluate the capability of the AI generative model ChatGPT to generate BLAS level-1 codes for different architectures. For this analysis, we used two versions of ChatGPT: 3.5-Turbo and 4o. The prompt used depends on the programming model or ChatBLAS back end: one for OpenMP, one for CUDA, one for HIP, and the BLAS level-1 routine. The prompts used for the CUDA and HIP models are very similar. The operations are made by using single precision.

A. Correctness

The metric used to evaluate the correctness of the AI-generated codes consists of the number of times that the required codes generated by ChatGPT were correct divided by the total number of times that the codes were generated. We ran the tests for 10 iterations. We define correct codes as those that use the programming language (e.g., C/C++) and the programming model (e.g., OpenMP, CUDA, HIP) required and specified in the prompts, that can be compiled and run, and that provide a correct result.

First, we evaluate the correctness of the codes generated by using very basic prompts that do not include any detail about the implementation or parallelization, providing only a brief description of the problem (BLAS level-1 operations) and the programming language and model (e.g., C and OpenMP) that we want the codes to use. Next, we include an example of the prompts used for the *saxpy* operation for the CUDA back end:

> *Give me a function code only that computes a multiplication of a vector x by a constant a and the result is added to a vector y. Vectors x and y are length n, use C and CUDA to compute in parallel include the next line in the code, and use the next function name and parameters void chatblas_saxpy(int n, float a, float *x, float *y). Include the next line at the beginning #include chatblas_cuda.h*

As we observe in Table I, for OpenMP codes, ChatGPT can provide a 93% efficiency (93% of the codes generated are correct) by using the 3.5-Turbo version of the ChatGPT model.

This result is elevated up to 100% by using the same prompt but on the more modern 4o version of ChatGPT. For CUDA codes, both ChatGPT versions, 3.5-Turbo and 4o, provide a 66% efficiency. For HIP codes, we report an efficiency of 75% and 82% when using the 3.5 and 4o versions, respectively.

Routine	OpenMP		CUDA		HIP	
ChatGPT Version	3.5	4o	3.5	4o	3.5	4o
saxpy	1.0	1.0	0.8	0.8	1.0	1.0
sscal	0.6	1.0	0.8	0.6	1.0	0.8
sswap	1.0	1.0	0.4	0.4	0.8	1.0
scopy	1.0	1.0	0.6	0.8	1.0	0.8
sdot	1.0	1.0	1.0	0.6	1.0	0.8
sdsdot	1.0	1.0	0.2	0.2	0.6	0.2
sasum	0.8	1.0	0.6	1.0	0.4	1.0
snrm2	1.0	1.0	0.8	0.6	0.6	0.8
isamax	1.0	1.0	0.6	0.8	0.4	1.0

TABLE I
BLAS LEVEL-1 CORRECTNESS OF CHATBLAS USING BASIC PROMPTS.

The deficiencies found in the code generated by ChatGPT can be very diverse. Most of these deficiencies are found in GPU codes (e.g., CUDA and HIP codes), and these deficiencies include the use of CUDA syntax in HIP codes, GPU codes that do not include CPU-GPU communication primitives, or wrong use for the thread index or block/grid of block size, among others. It is particularly challenging to generate those operations that require compute reductions, such as *sdot*, *sasum*, or *sdsdot*. In these cases, the kernels may be required to use relatively advanced techniques, such as atomic operations, the use of shared memory, or synchronization. One challenging case corresponds to the *sdsdot* operation. This is a very particular case, in which the operations are internally transformed from single to double in the kernel, and the result must be transformed back to single precision before returning it.

1) Prompt Engineering: To increment the efficiency of the AI-generated codes, we study the use of the so-called *"prompt engineering"*. This consists of complementing the prompts with more information regarding the operations that will be computed by the codes to minimize the deficiencies found in the codes (e.g., wrong use of CUDA indexes or the lack of CPU-GPU communication). Next, we include an example of the prompts used for CUDA:

> *Give me a kernel and a function only that computes a multiplication of a vector x by a constant a and the result is added to a vector y. Do not give a main function. Vectors x and y are length n, use C and CUDA to compute in parallel, allocate and free the GPU vectors, and make the CPU - GPU memory transfers in the function. The size of blocks of threads and the number of blocks must be defined. Use the next function name and parameters for the kernel __global__ void saxpy_kernel(int n, float a, float *x, float *y), and the next function name and parameters for the function void chatblas_saxpy(int n, float a, float *x, float *y). Include the next line at the beginning of the code #include "chatblas_cuda.h".*

Routine	OpenMP		CUDA		HIP	
ChatGPT Version	3.5	4o	3.5	4o	3.5	4o
saxpy	1.0	1.0	0.8	1.0	1.0	0.8
sscal	1.0	1.0	1.0	1.0	1.0	1.0
sswap	1.0	1.0	1.0	1.0	1.0	1.0
scopy	1.0	1.0	1.0	0.8	0.8	1.0
sdot	1.0	1.0	0.6	0.8	1.0	0.8
sdsdot	1.0	1.0	0.4	0.8	0.4	0.2
sasum	1.0	1.0	0.8	0.8	1.0	0.4
snrm2	1.0	1.0	1.0	0.8	0.8	0.8
isamax	1.0	1.0	0.6	0.6	0.8	0.6

TABLE II
BLAS LEVEL-1 CORRECTNESS OF CHATBLAS USING PROMPT ENGINEERING.

By performing prompt engineering, we incremented the efficiency of all the AI-generated codes (see Table II) except for HIP codes by using the 4o ChatGPT version. For OpenMP codes, prompt engineering provides 100% efficiency when using both models. For CUDA codes, the efficiency is incremented from 66% to 80% and from 66% to 84% when using the 3.5-Turbo and 4o versions of ChatGPT, respectively. For HIP codes, we achieved better results (from 75% to 86%) by using the 3.5-Turbo version; however, the results were worse when using the 4o version (from 82% to 73%).

2) Fine-Tuning and Training: Although we elevated the efficiency of the AI-generated codes via prompt engineering, this is not enough to achieve the highest level of correctness. To do so, we used *"fine-tuning"*. This consists of training the AI generative models with the necessary information to reduce the deficiencies of the codes generated by AI. This effort is made on only the 3.5-Turbo version of ChatGPT because OpenAI does not allow users to do so on the 4o version.

These techniques require the use of very specific file formats (e.g., JSON). These files contain two blocks: (i) a prompt used as an example that describes the information required by the users and (ii) a valid output. In our case, we used as prompts the prompts detailed in the previous analysis and used as output the codes correctly generated by ChatGPT.

By fine-tuning, we could elevate the correctness to the desired levels, empowering ChatBLAS to reach an efficiency of 100% for the generation of BLAS level-1 operations using OpenMP, CUDA, and HIP programming models. This is most likely due to overfitting because the loss ratio dropped to zero. In our scenario, this is a positive outcome because we want the model to provide a correct and optimized code. We also experimented with different configurations for the hyperparameters with very similar results.

B. Performance

In this section, we focus on the performance analysis of the codes generated by the AI generative model ChatGPT 3.5-Turbo and 4o. We used the vendor libraries as references (e.g., MKL library for Intel CPUs, AOCL for AMD CPUs, cuBLAS/hipBLAS for NVIDIA/AMD GPUs). We used four architectures: two CPU architectures (Intel Xeon E5-2698 v4 Broadwell 20-core and AMD EPYC 7742 Rome 64-core) and two GPU architectures (NVIDIA Ampere A100 and AMD

Mi100). For this analysis, we focused on five of the most representative BLAS level-1 routines by using single-precision (*saxpy*, *sdot*, *sasum*, *sscal*, and *sswap*). The size of the vectors used in this analysis is equal to 500M.

To help ChatGPT generate codes that perform well on GPUs, we added some hints to the prompts in those routines that are more susceptible to an increment in performance, such as *sasum*, *sdot*, or *sdsdot*. These hints corresponded to very high-level details about the efficient use of GPU resources. Examples of these are *"use shared memory to accelerate the computation"* and *"minimize the use of atomic operations."*

First, we evaluated the performance of ChatBLAS using OpenMP vs. the Intel MKL library (see Figure 2). The performance provided by the vendor-specific Intel CPU library and ChatBLAS is very similar. Notably, we used the multi-threading version of the Intel library, and we set the number of OpenMP threads to the number of cores available for the ChatBLAS library.

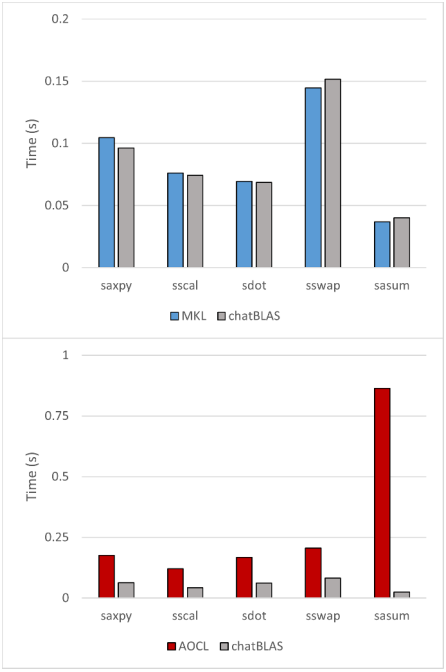

Fig. 2. Intel MKL performance (top) and AMD AOCL performance (bottom) vs. ChatBLAS-OpenMP performance.

Unlike in the comparison with Intel's library, we observe better performance when using ChatBLAS compared with the AMD AOCL library. This library is based on the open-source BLIS library, and it does not provide parallel codes for BLAS level-1 operations (see Figure 2). This is an example of the impact that tools automatically generated by LLMs models have by covering those deficiencies or performance gaps found in vendor libraries.

Next, we focus on NVIDIA GPUs. The implementation of CUDA codes is considerably more complicated than using high-level and directive-based solutions like OpenMP. Figure 5-top shows the performance reached by ChatBLAS

```
__global__ void sdot_kernel(int n, float *x, float
↪  *y, float *res) {
    int index = threadIdx.x + blockIdx.x *
    ↪  blockDim.x;
    float result = 0.0;
    if (index < n) {
        result = x[index] * y[index];
    }
    atomicAdd(res, result);
}
```

Fig. 3. Example of the nontrained CUDA kernel code for single-precision dot product.

codes against the NVIDIA cuBLAS library. The performance is competitive for operations like *saxpy*, *sscal*, or *sswap*, where the performance of ChatBLAS is about 80% concerning the cuBLAS performance. However, we see an important performance gap in operations like *sdot* or *sasum*, where ChatBLAS is performing much worse than cuBLAS. This is mainly due to the algorithms used by ChatBLAS codes for these operations, which are not optimal to perform well on NVIDIA GPUs. In many cases, this low performance is caused by excessive use of costly operations like atomics or the lack of use of high-bandwidth memories, such as shared memory. An example of this is shown in Figure 3, which illustrates the *sdot* code generated by ChatBLAS originally. This code, although functional and correct, makes use of an algorithm that excessively uses atomics, which causes an important fall in performance. Unlike this algorithm, the code/algorithm illustrated by Figure 4 is more efficient. By using shared memory to make part of the reductions, it is possible to reduce the number of atomic operations considerably.

```
__global__ void sdot_kernel(int n, float *x, float
↪  *y, float *res) {
    __shared__ float products[256];
    int index = threadIdx.x + blockIdx.x *
    ↪  blockDim.x;
    float product = x[index] * y[index];
    products[threadIdx.x] = product;
    __syncthreads();
    if (threadIdx.x == 0) {
        float result = 0.0;
        for (int i = 0; i < blockDim.x; i++) {
            result += products[i];
        }
        atomicAdd(res, result);
    }
}
```

Fig. 4. Example of the trained CUDA kernel code for single-precision dot product.

Unlike in the previous section, in which we focus on correctness only, here we can use the same training or fine-tuning techniques to improve performance for *sdot* and *sasum* operations of ChatBLAS when using CUDA. As a result, performance is improved for these operations (Figure 5, bottom), thereby reaching acceleration of about $4\times$ and providing similar performance to those reported by the other BLAS level-1 routines (e.g., *axpy*, *sscal*, or *sswap*).

As illustrated in Figure 6, we observe a different trend on the AMD GPU. In this case, ChatBLAS codes can provide

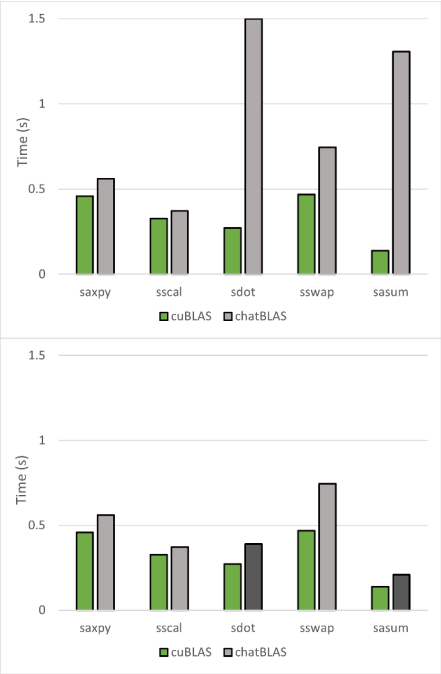

Fig. 5. The cuBLAS performance vs. nontrained (top) and trained (bottom) ChatBLAS-CUDA performance.

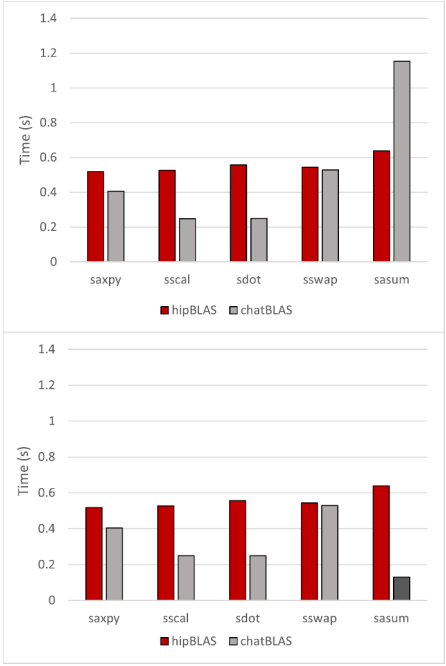

Fig. 6. The hipBLAS performance vs. nontrained (top) and trained (bottom) ChatBLAS-CUDA performance.

```
__global__ void sasum_kernel(int n, float *x, float
↪  *sum) {
    int idx = blockIdx.x * blockDim.x +
    ↪  threadIdx.x;
    if (idx < n) {
        atomicAdd(sum, fabsf(x[idx]));
    }
}
```

Fig. 7. Example of the nontrained HIP kernel code for the single-precision *asum* routine.

better performance than the equivalent codes in hipBLAS, reaching accelerations of about 2× in some cases, except for the *sasum* routine, in which ChatBLAS performs worse than hipBLAS. Similar to CUDA ChatBLAS codes for the *sdot* routine, this performance issue is due to the use of a nonoptimized algorithm (see Figure 7) that uses atomics massively. To improve performance for *sasum* operations, we use as part of the training (fine-tuning) the code/algorithm generated by ChatBLAS for the *sdot* routine and adapt it to *sasum* singularities (see Figure 8). The ChatBLAS code for *sdot* uses advanced techniques to avoid atomic operations on the GPU side by computing the reduction on the CPU side and maximizes the use of high-bandwidth memories in the GPU. As shown (see Figure 6, bottom), the performance for *sasum* is improved considerably, providing accelerations close to 3× compared with hipBLAS.

III. Related Work

Recently, several works related to the application of LLMs to HPC domain-specific tasks have been published. Foundational LLMs driven by the needs of the industry include GPT-3/3.5/4.0, Llama [3], and Bert [4]. Ding et al. [5] introduce HPC-GPT, which is a trained model leveraging the open-source Llama-13B foundational model. HPC-GPT was successfully applied to manage AI models and datasets and for data race detection. Kadosh et al. [6] explore the use of a domain-specific LLM, Tokompiler, which demonstrates better results for code completion and semantics than a GPT-3 based model for Fortran, C, and C++ code. Chen et al. [7] introduce LM4HPC, which is a language model specifically designed for HPC that demonstrates success in specific tasks such as code similarity analysis, parallelism detection, and OpenMP question answering. Similarly, Nichols et al. [8] present HPC-Coder, which is a trained model based on the DeepSpeed [9] deep learning model. They found a varying degree of success in tasks such as code completion, including OpenMP pragmas and Message Passing Interface calls, in their work targeting HPC code. Munley et al. [10] provide LLM4VV, a fine-tuned model that explores the capabilities of GPT-4 and Llama 2 to generate tests for the compiler functionality of the directives-based OpenACC parallel programming model that shows a high degree of success. Godoy et al. [11] and the follow-up work including Llama 2, Valero-Lara et al. [12] evaluate the current LLMs capabilities for generating correct and functional representative HPC kernels on modern HPC programming models.

IV. Conclusions

In this work, we present ChatBLAS, the first AI-generated and portable HPC library for BLAS operations. We present an exhaustive study about how to use disruptive AI-based

```
__global__ void sasum_kernel(int n, float *x, float
↪  *sum) {
    __shared__ float cache[512];
    int tid = threadIdx.x + blockIdx.x *
    ↪  blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < n) {
        temp += abs(x[tid]);
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp;
    __syncthreads();
    int i = blockDim.x / 2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex +
            ↪  i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0) sum[blockIdx.x] =
    ↪  cache[0];
}
```

Fig. 8. Example of the trained HIP code for the single-precision *asum* routine.

technologies, such as large language models, to generate HPC codes using different programming models such as OpenMP, CUDA, or HIP and define the fundamental practices and criteria to interact with LLMs for HPC targets. We also provide a deep analysis of the impact of LLM-based techniques, such as prompt engineering and fine-tuning to improve the trustworthiness and the performance of the AI-generated codes. When the performance of ChatBLAS is compared with that of highly optimized vendor libraries, ChatBLAS codes can provide either competitive performance vs. Intel's MKL and NVIDIA's cuBLAS libraries and better performance vs. AMD's libraries AOCL (on CPUs) or hipBLAS (on GPUs).

As future work, we plan to extend this study to cover the other BLAS levels (e.g. level-2 matrix-vector and level-3 matrix-matrix) operations, as well as use other AI techniques that can help generate highly optimized HPC codes for BLAS and other important HPC operations.

Acknowledgment

This research used resources from the Experimental Computing Laboratory at Oak Ridge National Laboratory, which is supported by the Office of Science of the US Department of Energy (DOE) under contract DE-AC05-00OR22725. This research was funded in part by the ASCR Stewardship for Programming Systems and Tools (S4PST) project, part of the Next Generation of Scientific Software Technologies (NGSST). This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, and Scientific Discovery through Advanced Computing (SciDAC) program. This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with DOE. The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017.

[2] S. Catalán, X. Martorell, J. Labarta, T. Usui, L. A. T. Díaz, and P. Valero-Lara, "Accelerating conjugate gradient using ompss," in *20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5-7, 2019*. IEEE, 2019, pp. 121–126. [Online]. Available: https://doi.org/10.1109/PDCAT46702.2019.00033

[3] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[4] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1, 2019, p. 2.

[5] X. Ding, L. Chen, M. Emani, C. Liao, P.-H. Lin, T. Vanderbruggen, Z. Xie, A. Cerpa, and W. Du, "Hpc-gpt: Integrating large language model for high-performance computing," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023, p. 951–960. [Online]. Available: https://doi.org/10.1145/3624062.3624172

[6] T. Kadosh, N. Hasabnis, V. A. Vo, N. Schneider, N. Krien, A. Wasay, N. Ahmed, T. Willke, G. Tamir, Y. Pinter, T. Mattson, and G. Oren, "Scope is all you need: Transforming llms for hpc code," 2023.

[7] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. de Supinski, "Lm4hpc: Towards effective language model application in high-performance computing," in *OpenMP: Advanced Task-Based, Device and Compiler Programming*, S. McIntosh-Smith, M. Klemm, B. R. de Supinski, T. Deakin, and J. Klinkenberg, Eds. Cham: Springer Nature Switzerland, 2023, pp. 18–33.

[8] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "Modeling parallel programs using large language models," 2023.

[9] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: https://doi.org/10.1145/3394486.3406703

[10] C. Munley, A. Jarmusch, and S. Chandrasekaran, "Llm4vv: Developing llm-driven testsuite for compiler validation," 2023.

[11] W. Godoy, P. Valero-Lara, K. Teranishi, P. Balaprakash, and J. Vetter, "Evaluation of openai codex for hpc parallel programming models kernel generation," in *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ser. ICPP Workshops '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 136–144. [Online]. Available: https://doi.org/10.1145/3605731.3605886

[12] P. Valero-Lara, A. Huante, M. A. Lail, W. F. Godoy, K. Teranishi, P. Balaprakash, and J. S. Vetter, "Comparing llama-2 and gpt-3 llms for hpc kernels generation," 2023.