COALA: A Compiler-Assisted Adaptive Library Routines Allocation Framework for Heterogeneous Systems

Qinyun Cai, Guanghua Tan, Wangdong Yang, Xianhao He, Yuwei Yan, Keqin Li, *Fellow, IEEE*, and Kenli Li, *Senior Member, IEEE*

***Abstract*—Experienced developers often leverage well-tuned libraries and allocate their routines for computing tasks to enhance performance when building modern scientific and engineering applications. However, such well-tuned libraries are meticulously customized for specific target architectures or environments. Additionally, the performance of their routines is significantly impacted by the actual input data of computing tasks, which often remains uncertain until runtime. Accordingly, statically allocating these library routines may hinder the adaptability of applications and compromise performance, particularly in the context of heterogeneous systems. To address this issue, we propose the Compiler-Assisted Adaptive Library Routines Allocation (COALA) framework for heterogeneous systems. COALA is a fully automated mechanism that employs compiler assistance for dynamic allocation of the most suitable routine to each computing task on heterogeneous systems. It allows the deployment of varying allocation policies tailored to specific optimization targets. During the application compilation process, COALA reconstructs computing tasks and inserts a probe for each of these tasks. Probes serve the purpose of conveying vital information about the requirements of each task, including its computing objective, data size, and computing flops, to a user-level allocation component at runtime. Subsequently, the allocation component utilizes the probe information along with the allocation policy to assign the most optimal library routine for executing the computing tasks. In our prototype, we further introduce and deploy a performance-oriented allocation policy founded on a machine learning-based performance evaluation method for library routines. Experimental verification and evaluation on two heterogeneous systems reveal that COALA can significantly improve application performance, with gains of up to 4.3x for numerical simulation software and 4.2x for machine learning applications, and enhance system utilization by up to 27.8%.**

***Index Terms*—Compiler assistance, computing task, dynamic allocation, high-performance computing, library routine.**

Manuscript received 10 August 2023; revised 18 March 2024; accepted 30 March 2024. Date of publication 9 April 2024; date of current version 11 June 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022YFF0606302, in part by the Key Program of National Natural Science Foundation of China under Grant U21A20461, Grant 62227808, and Grant 92055213, in part by the major projects of Xiangjiang Laboratory under Grant 22xj01011, and in part by China Scholarship Council (CSC). Recommended for acceptance by C. Li. *(Corresponding authors: Guanghua Tan; Wangdong Yang.)*

Qinyun Cai, Guanghua Tan, Wangdong Yang, Xianhao He, Yuwei Yan, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China, and also with the National Supercomputing Center in Changsha, Hunan 410082, China (e-mail: hnutsai@hnu.edu.cn; guanghuatan@hnu.edu.cn; yangwangdong@hnu.edu.cn; hexianhao@hnu.edu.cn; yanyuwei@hnu.edu.cn; lkl@hnu.edu.cn).

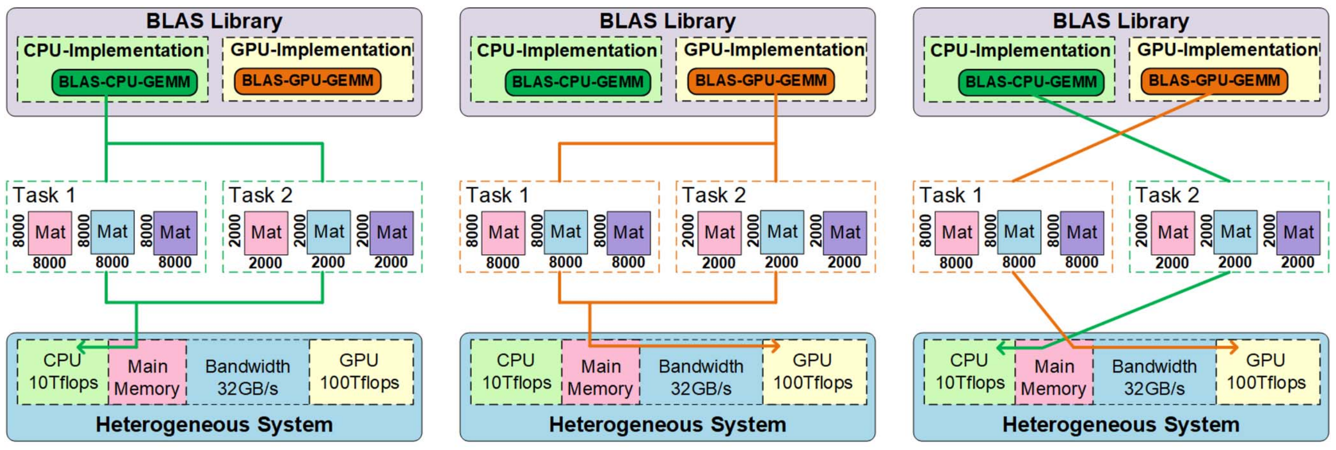
Keqin Li is with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China, also with the National Supercomputing Center in Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TC.2024.3385269

I. Introduction

Effective use of computing libraries, such as BLAS [1], FFT [2], tensor [3], and SPARSE [4], is an important technique for developers to build high-performance scientific or engineering computing applications [5]. Well-crafted computing library implementations of these libraries that are manually tuned by domain experts can deliver much higher performance than code generated by compilers [6]. These libraries are carefully tailored for specific target architectures or environments. For the instance of the popular BLAS (Basic linear algebra subprograms) library, many highly optimized implementations are available for heterogeneous systems: MKL [7], [8] customized for Intel CPUs, OpenBLAS [9] supported by OpenMP [10] or POSIX [11], ATLAS [12] with automatically tuned ability, CuBlas [13] customized for Nvidia GPUs, and ClBLASt [14] supported by OpenCL [15]. Although routines from different BLAS library implementations can perform the same calculation, they may behave a critical difference when handle a dynamic computing task. The dynamic computing task refers to a computing task where the actual computational operations or data are not fully specified or determined until runtime, rather than being predefined statically. Library routines are significantly influenced by the characteristics of the input data of computing tasks, which often remains uncertain until runtime. Accordingly, solely relying on a single library implementation may not be optimal for applications, as dynamic computing tasks could exceed the the optimal performance range of the library's routines at runtime. Therefore, developers should consider multiple library implementations and their corresponding routines during software development to ensure the utilization of the most efficient and effective routines from these libraries.

Efficient utilization of computing resources and enhancement of application performance remains a challenging research problem [16], [17]. Particularly, when addressing the allocation



(a) via BLAS-CPU-GEMM only (b) via BLAS-GPU-GEMM only (c) via optimal routine

Fig. 1. Two computing tasks need to be executed on the heterogeneous system via two alternative library routines, but naively allocating routines cannot bring the optimal performance.

of library routines for dynamic computing tasks, this challenging is further compounded by three critical issues. Firstly, allocating routines from multiple library implementations and assigning an optimal one to each dynamic computing task is inherently difficult. It necessitates not only programmers' extensive experience in utilizing these routines but also accurate prediction of the characteristics of the target computing task. Furthermore, it imposes a substantial workload on programmers, requiring them to manage diverse and complex assignments. Secondly, allocating routines from only one library implementation and assigning an optimal one to each dynamic computing task is easy but hard to bring an overall optimum in applications. There has yet to be a one-for-all library implementation that could perfectly suit various situations that varying input data features and uncertain computing resources. Although most programmers would select an appropriate library implementation to cover all tasks, the application's effect depends on the developers' experience. Lastly, static allocation of routines imposes limitations on the software, input data, and platform. When users intend to execute the software on a new platform, they are compelled to manually modify the code for using another library implementation, entailing a significant workload. Therefore, these issues can be prohibitive for both developers and users, hindering the efficient allocation of library implementation routines and impeding the adoption of software across different platforms.

A. Motivating Example

Fig. 1 illustrates the issue in a heterogeneous system, where the peak theoretical performance of GPU is ten times that of CPU (assuming that 10 Tflops of CPU and 100 Tflops of GPU), and the bandwidth between CPU and GPU is 32 GB/s. It assumes that an application has two dynamic computing tasks; task 1 has a considerable input data size, while task 2 has a small input data size. Both of them are going to perform a General Matrix Multiplication (GEMM) with the help of the BLAS library; Two highly optimized implementations are available for these tasks, each of which can supply a GEMM routine. Nevertheless, the BLAS-CPU-GEMM routine executes tasks on the CPU while the BLAS-GPU-GEMM routine executes tasks on GPU. Fig. 1(a) shows the process of assigning BLAS-CPU-GEMM routine for tasks, while Fig. 1(b) shows the other. We can easily estimate the total time cost for these two situations:

- Fig. 1(a): the total time cost is 104.00 ms, including 102.40 ms computing time cost of task 1 and 1.60 ms computing time cost of task 2;
- Fig. 1(b): the total time cost is 42.06 ms, including 10.24 ms computing time cost and 29.80 ms data transfer time cost of task 1, 0.16 ms computing time cost and 1.86 ms transfer time cost of task 2.

Though the overall performance in the Fig. 1(b) situation is better than the Fig. 1(a) situation, the application needs to achieve optimal performance. The time cost of the task in Fig. 1(b) outnumbers Fig. 1(a), because there needs to be more computation to hide the data movement via PCI-E. A good solution is to assign the optimal routine for each task as Fig. 1(c) shows, task 1 is assigned the BLAS-GPU-GEMM routine and task 2 is assigned the BLAS-CPU-GEMM routine. However, it is impossible to allocate them statically because we need to know their data size in advance, and so we propose a runtime solution in this work.

B. The Proposed Solution

The above example implies a need for a system-level mechanism that can efficiently allocate library routines and dynamically assign the optimal routine for each dynamic computing task. To address this challenge, we first formalize this challenging problem and then design a framework called **COALA**[1] (Compiler-Assisted Adaptive Library Routines Allocation), which provides adaptive allocation of library routines

[1]The source code is available at github.com/e2mcc/coala

for heterogeneous systems. **COALA** offers a uniform management approach for all library implementations and employs compiler assistance to dynamically identify the requirements and data features of dynamic computing tasks. It then dynamically assigns an optimal routine for each dynamic computing task in terms of task features and the capabilities of the computing device within heterogeneous systems. Implementation-wise, **COALA** leverages LLVM [18] (Low-Level Virtual Machine) thereby applies to any language once it is compiled to LLVM-IR. It works without manual effort and does not change any source code. Because it is fully automated and leverages the compiler to reconstruct original dynamic computing tasks. Briefly, a reconstructed dynamic computing task contains an information probe and related memory operations. It contains a whole set of memory operations, including memory allocations and data transmissions, to ensure it can be executed on any computing device without breaking its correctness. The probe is statically inserted into the task code. It gathers and conveys the task's requirements and the data's information (such as task purpose, data size, and requirement of computing flops obtained by the compiler static analysis) to a user-level allocation at runtime before the task is executed. The allocation then dynamically assigns the optimal library routine for the dynamic computing task in terms of its probe and allocation policies. Our **COALA** supports deploying different allocation policies for different optimization targets. In this paper, we focus on the design of the COALA framework and introduce a performance-oriented allocation policy. The allocation policy is based on a machine learning (ML)-based performance evaluation method specifically developed for library routines. The main contributions of this work are summarized as follows:

- Proposal of **COALA**: We propose **COALA**, an adaptive library routines allocation framework to uniformly manage all mainstream library implementations on heterogeneous systems. A prototype of **COALA** is implemented by the LLVM framework support. It enables independent and uncooperative library implementations to execute simultaneously within an application at runtime. Furthermore, **COALA** can dynamically assign optimal routines for each dynamic computing task based on task features and computing device's characteristics in heterogeneous systems.
- Compiler-assisted method: We devise a compiler-assisted method to reconstruct dynamic computing tasks, acquire their requirements, analyze their data features, and insert probes to transmit relevant information to the allocation component at runtime. It is a fully automated method without any manual effort or changes to the application source code.
- Efficient performance-oriented allocation policy: We introduce an efficient performance-oriented allocation policy aimed at enhancing application performance and system utilization. This policy is grounded in the ML-based performance-oriented evaluation method, characterized by using neural network model for prediction.
- Verification and evaluation on heterogeneous systems: We verified and evaluated our work on two distinct types of heterogeneous systems: a personal computer and a HPC server. The experimental evaluation provides evidence: (a) the proposed ML-based performance evaluation method can make a accurately prediction for library routines and the deployed policy in **COALA** verified its effectiveness; (b) **COALA** effectively manages library implementations and makes them cooperate; (c) **COALA** offload computation to the device which is beyond the original code support; (d) **COALA** increase performance and system utilization efficiently, the results indicate that it improves the performance of machine learning application up to 4.2x, and system utilization up to 27.8%, and the performance of numerical simulation software up to 4.3x.

TABLE I
TERMINOLOGY USED IN THIS PAPER

Terminology	Description
CUDA	Compute unified device architecture
OpenCL	Open computing language
BLAS	Basic linear algebra subprograms
MKL	Math kernel library
ATLAS	Automatically tuned linear algebra software
ClBlast	Tunable OpenCL-based implementation of the BLAS
CuBLAS	Nvidia's implementation of the BLAS
FFT	Fast fourier transform
HPC	High-performance computing
COALA	Compiler-assisted adaptive library routines allocation
GEMM	General matrix multiplication
SYR2K	Symmetric rank-2k update
LLVM	Low Level Virtual Machine
IR	Intermediate representation
YOLO	You only look once, a neural network for target detection
DNN	Deep neural network
flops	Floating point operations per second
ML	Machine learning

The rest of paper is organized as follows. In Table I, we provide a comprehensive list of abbreviations and corresponding descriptions for the terminology employed throughout this study. Section II discusses the relevant work. In Section III, we provide a formalized description of the problem. Section IV presents the **COALA** framework, and details the task reconstruction, lazy runtime, the allocation component. In Section V, we introduce the performance-oriented allocation policy, accompanied by its machine learning-based performance evaluation method. Section VI describes the experimental setup and discuss the results. Section VII discussed the potential drawbacks or areas where **COALA** may not be as effective. Section VIII summarizes this paper.

II. RELATED WORK

To the best of our knowledge, this work is the first to address this challenge of adaptively allocating library routines on heterogeneous systems via a fully automated manner with no source code or system changes.

Several unified programming models, such as oneAPI [19] developed by Intel, SYCL [20] developed by the Khronos Group, is proposed to simplify the development of applications across a wide range of computing architectures. oneAPI uses a set of libraries including oneDNN, oneMKL, and oneCCL that help developers optimize and accelerate their applications. It also provides compatibility and portability across different hardware architectures. SYCL stands for "Standard C++ for Parallelism". It allows developers to write code in standard C++ and execute it efficiently on different heterogeneous devices, such as GPUs, CPUs, FPGAs, and DSPs, without requiring device-specific modifications. These programming models primarily focus on addressing the portability of code across different architectures and selecting the optimal code implementation based on the characteristics of each architecture. This approach is more of a static optimization method. However, we are more concerned with finding the optimal implementation for different computing tasks at runtime, which is a dynamic optimization approach.

Several works are proposed by using the compiler-assisted method. In [21], Ghiglio et al. discussed the need for efficient compiler and runtime support for the growing number of platforms that use SYCL as an open-standard API for accelerating C++ software. They contributed an alternate approach that bypasses OpenCL and uses a CPU-directed compilation flow with Whole Function Vectorization to generate optimized host and device code. It was implemented in a specific compiler, ComputeCpp. While our work focuses on the application built with libraries, and it is not limited to a specific programming framework. Neves et al. [22] proposed a compiler-assisted data streaming approach to improve performance for regular code structures instead of complex prefetching schemes. It uses static analysis at compile-time to detect and encode memory access patterns, including indirect accesses, into descriptors. The compiler then transforms array accesses into stream references, reducing instructions. At runtime, a stream controller generates addresses from the descriptors and fetches data ahead into buffers, bypassing caches. However, we are more concerned with the dynamic characteristics of tasks and the input data they handle during runtime. Our research aims to enhance the performance of dynamic computing tasks, ultimately improving the overall application performance. In [6], Carvalho et al. presented an idiom recognizer implemented as a LLVM pass, named KernelFaRer (Kernel Find & Replacer). KernelFaRer is able to identify GEMM (general matrix-matrix multiplication) and SYR2K (symmetric rank-2k update) idioms, and replaced them with corresponding BLAS library routines. This article combined pattern matching and loop information analysis in LLVM Intermidiate Representation (IR) to determine GEMM and SYR2K matrix's access orders through a formulation of an analysis. However, KernelFaRer is limited to only two BLAS routines, GEMM and SYR2K, and restricted to statically replacement. In contrast, COALA applies to most math libraries and dynamically analyze the computing tasks for assigning optimal library routines. Chen et al. [23] presented a fully automated GPU scheduling framework, called CASE, by utilizing the compiler-assisted method. CASE focused on efficiently utilizing multiple high-power GPUs' resources. This contribution has inspired our work. Regrettably, this work has not yet incorporated the consideration of CPU devices, nor has it been extended to heterogeneous platforms other than Nvidia GPUs.

Several works are proposed to optimizing computing task running in a specific situation. In [24], Ayala et al. presented the design and implementation of the heFFTe (Highly Efficient FFT for Exascale) library, which targets exascale supercomputers. The heFFTe library provides highly scalable GPU kernels that achieves more than 40x speedup compared to local kernels from CPU state-of-the-art libraries and over 2x speedup for the whole FFT computation. The library also includes a communication model for parallel FFTs to analyze the bottleneck for large-scale problems. In [25], Springer et al. introduced the open-source C++ library, High-Performance Tensor Transposition (HPTT). HPTT is a high-performance implementation for tensor transpositions that can be used in applications where tensor sizes and permutations are determined at runtime. HPTT incorporates optimizations such as blocking, multi-threading, and explicit vectorization, making it easy to port to different architectures. Yang et al. [26] proposed a probability-based method to partition sparse matrices that provides an effective partitioning technique to accelerate SpMV computing tasks on GPUs and multicore CPUs. These related works extended or created APIs, which requires manual code adjustments by programmers to utilize them. While **COALA** is a system-level framework and can manage independent libraries. Consequently, **COALA** can effectively support these works and facilitate collaboration with other library implementations, leading to broader optimizations for various applications.

III. Problem Formalization

We begin by walking through the motivating example depicted in Fig. 1. we use i to denote the statically predefined computational function and d to represent the actual input data at runtime. As a result, let $x = g(i, d)$ represent the dynamic computing task. Here, g refers to an actual execution of the dynamic computing task at runtime. We use $\mathcal{R}$ to denote the space consisting of all alternative routines. Let t denote a specific target that we primarily care about, such as prioritizing throughput or reducing energy consumption. f is the evaluation function for evaluating the effect of utilizing the routine r to execute a dynamic computing task at this specific target t. We prefer to use a smaller value to represent a better evaluation result. Therefore, we are interested in a $r \in \mathcal{R}$ that can execute the dynamic computing task x to achieve the minimum value of $y = f(x, r, t)$ for the specific target t. For a given tuple of $(i, d, g, r, \mathcal{R}, t, f)$, our problem can be formalized by

$$\arg\min_{r \in \mathcal{R}} f(g(i, d), r, t). \tag{1}$$

Equation (1) has guiding significance for us to design a system-level framework to deal with this challenging problem. According to this equation, our framework needs several parts to be composed of: firstly, our framework needs to be able to manage available library routines; secondly, our framework needs to be able to dynamically recognize the characteristics of dynamic computing tasks, especially those resulting from data input; lastly, our framework needs to be able to deploy different polices, and our framework should automatically allocate the optimal routines based on the evaluation functions matched with these strategies.

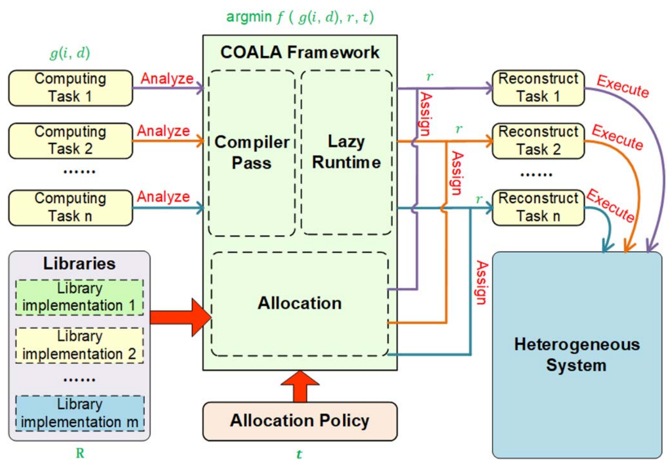


Fig. 2. The overview of COALA framework. The COALA framework provides an automated and adaptive solution for allocating library routines in heterogeneous systems. The diagram incorporates green expressions and letters to demonstrate the correlation between the various components depicted in the diagram and equation (1).

```
1   ...
2   //allocate host memory
3   float * hA = malloc(M*K*sizeof(float));
4   float * hB = malloc(K*N*sizeof(float));
5   float * hC = malloc(M*N*sizeof(float));
6   //input task data
7   hA = loadFromFile("A.mtx");
8   hB = loadFromFile("B.mtx");
9   hC = loadFromFile("C.mtx");
10  //allocate device memory
11  cl_mem dA = clCreateBuffer(..., M*K*sizeof(float), ...);
12  cl_mem dB = clCreateBuffer(..., K*N*sizeof(float), ...);
13  cl_mem dC = clCreateBuffer(..., M*N*sizeof(float), ...);
14  //data transfer
15  clEnqueueWriteBuffer(..., dA, ..., hA, ...);
16  clEnqueueWriteBuffer(..., dB, ..., hB, ...);
17  clEnqueueWriteBuffer(..., dC, ..., hC, ...);
18  //a routine from the specific library implementation
19  CLBlastSgemm(...);
20  //data transfer
21  clEnqueueReadBuffer(..., dC, ..., hC, ...);
22  //release device memory
23  clReleaseMemObject(dA);
24  clReleaseMemObject(dB);
25  clReleaseMemObject(dC);
26  //release host memory
27  free(hA);
28  free(hB);
29  free(hC);
30  ...
```

Fig. 3. An example dynamic computing task consists of an original library routine and related memory operations.

IV. The Design of COALA

The purpose of **COALA**'s design is to implement a system-level mechanism that address the problem we formalized in Section III. As shown in Fig. 2, the green expressions and letters illustrate the correlation between the corresponding parts in the diagram and Equation (1). Our **COALA** has the ability to manage almost all mainstream libraries by identifying their routines. Routines are explicitly implied by calls to computing library routines' API, which are listed in the head file of computing library implementations (e.g., "cblas.h", "cublas.h"). **COALA** consists of three main components: a compiler pass, lazy runtime, and an allocation component. The compiler pass, along with the lazy runtime, reconstructs dynamic computing tasks and inserts a probe in each task. At runtime, the probe conveys the information gathered from tasks and data to the allocation component. The allocation component assigns an optimal library routine for the dynamic computing task based on their probe information and the allocation policy.

A. Dynamic Computing Task

In this work, the "dynamic computing task" is defined as a group operation with a close relationship, containing a computing library routine API as well as a set of host operations and device operations. The calculation function of a dynamic computing task is predefined statically, while the actual data and related computational operations are not fully specified or determined until runtime. Fig. 3 shows an example task, in which the code from lines 3-29 belongs to a dynamic computing task for performing a BLAS GEMM calculation. Line 19 is the GEMM routine API from ClBLASt implementation, which performs the task on the GPU, and the rest is the related memory operation. The host operations allocate (e.g. *malloc*, lines 3-5) and release (e.g. *free*, lines 27-29) memory on the host. Additionally, the host operations load data from the file to the memory (e.g. *loadFromFile*, lines 7-10). The device operations allocate (e.g. *clCreateBuffer*, lines 11-13) and release (e.g. *clReleaseMemObject*, lines 23-25) the memory on the device, such as GPU global memory. Moreover, the device operations transfer data from host to the device (e.g. *clEnqueueWriteBuffer*, lines 15-17) and transfer results from device to the host (e.g. *clEnqueueReadBuffer*, line 21).

B. Task Reconstruction

Our **COALA** leverages a compiler pass, along with the lazy runtime, to reconstruct dynamic computing tasks and gather the information of the data input. It works on the LLVM-IR of applications, consequently, it can support applications programmed with various programming languages supported by LLVM. Essentially, **COALA** reconstructs a dynamic computing task by first searching for its original computing library routine. Then, the compiler pass utilizes the def-use chain to locate the related operations and identifies the memory address of data input. However, in some cases, applications encapsulate data input and related memory operations in separate functions. Alternatively, the data may not enter memory for the first time or be reused from other tasks. The compiler cannot embody such def-use chains among operations on LLVM-IR, thereby the compiler pass cannot follow up. To address this, the compiler will reconstruct dunamic computing tasks by generating an abstract version. It consists unbound operations, a probe and an API linking to the allocation component, the pseudo code is shown in Fig. 4. The "unbound operation" is a static temporary representation. For example, a call to *clCreateBuffer* will be

```
1  ...
2  //allocate host memory
3  hA = host_malc(M, K, "float");
4  hB = host_malc(K, N, "float");
5  hC = host_malc(M, N, "float");
6  //input task data
7  DATA_INPUT(hA, data);
8  DATA_INPUT(hB, data);
9  DATA_INPUT(hC, data);
10 //a probe with tasks information
11 probe(M, N, K, hA, hB, hC, "SGEMM", ...);
12 //device memory allocation
13 dA = dev_malc(M, K, "float");
14 dB = dev_malc(K, N, "float");
15 dC = dev_malc(M, N, "float");
16 //data transfer
17 host2device(hA, dA);
18 host2device(hB, dB);
19 host2device(hC, dC);
20 //the optimal routine assigned by this
21 coala_allocating(ID);
22 //data transfer
23 device2host(dC, hC);
24 //release device memory
25 dev_free(dA);
26 dev_free(dB);
27 dev_free(dC);
28 //release host memory
29 host_free(hA);
30 host_free(hB);
31 host_free(hC);
32 ...
```

Fig. 4. The pseudo code of the reconstructed dynamic computing task generated by COALA.

replaced by the compiler with the *dev_malc*, which will simply assign a unique pseudo address for representing the memory object to be allocated instead of performing the actual allocation. The "unbound operation" will be realized an actual operation eventually and this process of realizing we called "binding".

Generally, host operations remains unchanged in an application running on a platform. Therefore, the unbound host operations will be realized as the original host operations at static analysis stage. For example, the task shown in Fig. 3, *host_malc* at line 3 in Fig. 4 will be bound to *malloc*. However, the bindings of these unbound device operations will be deferred to lazy runtime. If the optimal routine assigned by **COALA** runs on the CPU at the end, these pseudo memory operations would not bind any operation and return immediately. After that, the compiler pass collects and analyzes the task information, which is presented in the form of symbols. In LLVM IR, each symbol has a unique identifier, usually a name starting with a "%" sign or an "@" sign like %42 or @malloc. These symbols are used to represent various tasks, data and operations, such as: functions, function parameters, variables and instructions. Then the compiler pass inserts a probe to convey these symbols to the allocation, as shown in line 11 in Fig. 4. These symbols will be interpreted at lazy runtime to obtain the actual value.

C. Lazy Runtime

The lazy runtime refers to the runtime environment in which certain operations are delayed until they are absolutely necessary. There are primarily three steps to be performed within the lazy runtime. Firstly, the lazy runtime must be enabled to record the actual value of symbols carried by the probe since we cannot get the specific data information, which only exists in the form of symbols in the static analysis stage. Secondly, the **COALA** analysis is used to evaluate the specific data size and floating-point requirements in the lazy runtime. For example, the information about the dynamic computing task purpose carried by the probe is to compute a matrix-vector product (GEMV), and the information about the dimension of the related matrix is m by n. So that **COALA** would analyze the actual float point requirement q by the equation $q = 2mn$ since each matrix element in GEMV would do one float-point multiplication operation and one float-point addition operation. Moreover, the actual data size of the matrix s_{mat} is analyzed by the equation $s_{mat} = mn$ and the actual data size of the vector is s_{vec} is analyzed by the equation $s_{vec} = n$. Following the aforementioned steps, whether to bind those unbound device operations will be determined based on the final evaluation by **COALA**. If **COALA** ultimately determines assigning a CPU-executing routine, those unbound device operations will not bind to any operation and will return immediately. Conversely, if **COALA** determines that a GPU-executing routine should be assigned, the unbound device operations will bind to their corresponding device operations.

D. The Allocation Component

The allocation component is deployed to assign optimal library routines for dynamic computing tasks based on their requirements and input data features. For applications, the allocation exposes a simple API, *coala_allocating*, to replace a task's original library routine, as shown in line 21 in Fig. 4. This API is a synchronized function that can block the process until it returns. It is inserted by the compiler pass and fed with a task ID used by the runtime to identify the task uniquely. From the task ID, the allocation component can find the corresponding task's probe. The *coala_allocating* put this parameters to the allocation and then waits for the response from the allocation component. In return, the allocation component find optimal routine in term of the deployed allocation policy. This allocation component provides a flexible and adaptable foundation for designing and implementing various allocation policies that can be tailored to specific computing environments. For example, in cloud computing environments, prioritizing low-power routines as optimal routines may be a suitable cost-reducing strategy. while in a high-performance computing environment, a policy that focuses on maximizing processing power might be more suitable. Additionally, the deployed allocation policy can be easily adjusted to reflect changing circumstances from a general *getOptimalRoutine()* interface (line 2 at Algorithm 1). Algorithm 1 explains how the *coala_allocating* function works, and it is implemented based on LLVM-IR statements. The following are detailed explanations of Algorithm 1:

1) Line 1 retrieves the probe associated with the task ID t from the probe list P, since all tasks' probe within a LLVM-IR module are stored in a probe list.

Algorithm 1 The pseudo code of the *coala_allocating* function

```
Require:
    $P$: The probe list;
    $t$: The task ID;
Ensure:
    $\hbar$: Execution status.
1: Get the probe from the probe list:
   Probe $p \leftarrow P[t]$;
2: Get optimal routine:
   Routine $r \leftarrow$ getOptimalRoutine($p$);
3: Get required arguments from the probe:
   Args $a \leftarrow$ getRoutineArgs($p$,$r$);
4: Assemble the routine and the arguments together:
   $r(a) \leftarrow r$, $a$;
5: return $\hbar$.
```

2) Line 2 uses the *getOptimalRoutine(p)* interface to find the optimal routine based on the deployed allocation policy. The detailed process is described in Algorithm 2.
3) Line 3 uses the *getRoutineArgs(p, r)* interface to extract the required arguments from the probe's information based on the optimal routine found in line 2.
4) Line 4 assembles the optimal routine and related arguments to form a completed routine.

In this paper, we mainly concentrate on the design of this **COALA** framework, demonstrating its benefits with an efficient and performance-oriented allocation policy.

V. The Performance-Oriented Allocation Policy

An efficient and performance-oriented allocation policy is introduced at this section. It is based on the ML-based (Machine Learning based) performance evaluation method for library routines.

In a given heterogeneous computing system, the peak performance of computing hardware is fixed and can be quantified by the number of floating-point operations executed per unit time. This peak performance is attainable under ideal conditions. Nevertheless, library routines can only utilize a fraction of the computing hardware's peak performance due to various constraints, such as the routine's algorithm, data scale, system data transfer capability, and processor performance, among others. Most of these factors are static, with the sizes of input data being the most crucial dynamic influencing factor. We represent the utilization ratio of the computing hardware's peak performance by a library routine at runtime as $\mu = U(\boldsymbol{s})$, where $\boldsymbol{s}$ is a vector representing the sizes of the input data and U signifies the dynamic impact of input data sizes on μ.

The performance evaluation method involves utilizing a machine learning method, constructing a fully connected neural network model, to predict the utilization ratio μ that the library routine can achieve for given input data sizes. Upon initial deployment of the COALA framework, benchmarks is systematically executed to sample the peak utilization ratios of library routines across diverse data sizes. These samples serve as the dataset for the neural network model, with data sizes $\boldsymbol{s}$ as the

Algorithm 2 The pseudo code of *getOptimalRoutine*() function based on the performance-oriented allocation policy

```
Require:
    $W$: The file recording the system information;
    $R$: The file recording the routines' information;
    $P$: The probe;
Ensure:
    $z$: The optimal routine.
1: Read the host peak performance from the file:
   $\pi_h \leftarrow$ readSysInfo($W$, "peak", "host")
2: Read the device peak performance from the file:
   $\pi_d \leftarrow$ readSysInfo($W$, "peak", "device");
3: Read the Bandwidth from the file:
   $b \leftarrow$ readSysInfo($W$, "Bandwidth", null);
4: Get required routine type from the probe:
   $r \leftarrow$ getRoutineType($P$,$R$);
5: Read the neural network model list of host routines:
   $M_h \leftarrow$ readModels($r$,"host");
6: Read the neural network model list of device routines:
   $M_d \leftarrow$ readModels($r$,"device");
7: Get data size vector from the probe:
   $\boldsymbol{s} \leftarrow$ getDataSizeVec($P$);
8: Get float point requirement from the probe:
   $q \leftarrow$ getFloatPointRequirement($P$);
9: min $\leftarrow$ INF;
10: minIdx $\leftarrow -1$;
11: flag $\leftarrow 0$;
12: for $i$ in $M_h$ do
13:     $\mu \leftarrow$ predictUR($M_h[i]$,$\boldsymbol{s}$);
14:     temp $\leftarrow q/(\mu * \pi_h)$;
15:     if temp < min then
16:         min $\leftarrow$ temp;
17:         minIdx $\leftarrow i$;
18:     end if
19: end for
20: for $j$ in $M_d$ do
21:     $\mu \leftarrow$ predictUR($M_d[i]$,$\boldsymbol{s}$);
22:     temp $\leftarrow q/(M_d[i] * \pi_d)+$ norm2($\boldsymbol{s}$)$/b$;
23:     if temp < min then
24:         min $\leftarrow$ temp;
25:         minIdx $\leftarrow$ j;
26:         flag $\leftarrow$ 1 ;
27:     end if
28: end for
29: $z \leftarrow$ getTheRoutine($R$, $r$, minIdx, flag);
30: return $z$;
```

feature variable and utilization ratio μ as the target variable. The neural network model is trained, utilizing this sampling data to derive a set of weight parameters for each library routine. In the inference stage, the model loads these corresponding weight parameters to accurately forecast the peak utilization ratio of each library routine, adapting to varying data sizes at runtime. The workflow of the ML-based performance evaluation method is shown in Fig. 5. The fully connected neural network model

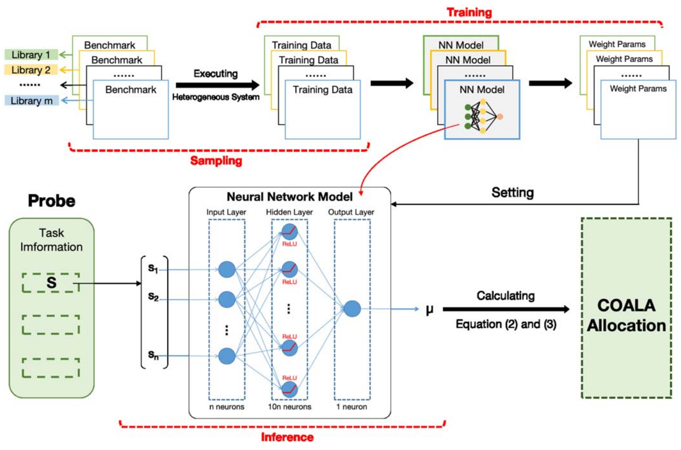


Fig. 5. The workflow of the ML-based performance evaluation method by using a neural network model for prediction.

comprises three layers: an input layer, a hidden layer, and an output layer. The input layer contains n neurons, corresponding to the number of input data and the dimension of the data is denoted as $\boldsymbol{s}$. For example, to perform a GEMV routine ($\boldsymbol{y} = \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$) needs to load 5 input data including variable α and β, matrix $\boldsymbol{A}$, vector $\boldsymbol{x}$ and vector $\boldsymbol{y}$. The dimension of the data is $\boldsymbol{s} = (1, 1, \mathit{sizeof}(\boldsymbol{A}), \mathit{sizeof}(\boldsymbol{x}), \mathit{sizeof}(\boldsymbol{y}))$. Accordingly, the number of the neurons n is 5. The hidden layer consists of $10n$ neurons which is 50 neurons in the example and the ReLU [27] (Rectified Linear Unit) activation function is applied to its outputs. According to our empirical tests, this configuration enables the model to effectively capture intricate relationships and features, resulting in a high predictive capability. The output layer contains 1 neuron, responsible for calculating regression predictions. We adopt the Xavier initialization method to initialize edge weights for enhanced training performance.

Based on the neural network model, our performance-oriented allocation policy is as follows. Assume that there is a heterogeneous system consisting of a CPU with π_c flops peak performance and a GPU with π_g flops peak performance, as well as a b GB/s bandwidth between the CPU and the GPU. The task's float point requirement is q, and its data size vector is $\boldsymbol{s}$. Additionally, there are a set of routines with the same calculation function; the i-th routine is r_i. It assumes that r_i can utilize $\mu_{ic} = U_{ic}(\boldsymbol{s})$ ratio of CPU peak performance, $\mu_{ig} = U_{ig}(\boldsymbol{s})$ ratio of GPU peak performance. Accordingly, a time cost function f is proposed to evaluate the execution time of a dynamic computing task by using the routine r_i. The cost function f of routine r_i can be represented by

$$f(r_i, q, \boldsymbol{s}) = \begin{cases} \dfrac{q}{U_{ic}(\boldsymbol{s})\pi_c}, & r_i \text{ runs on CPU;} \\ \dfrac{q}{U_{ig}(\boldsymbol{s})\pi_g} + \dfrac{\|s\|}{b}, & r_i \text{ runs on GPU.} \end{cases} \tag{2}$$

Thus, the performance-oriented policy for allocating the optimal routine z from n available routines for a computing task can be represented by

$$z = \operatorname{argmin}\{f(r_i, q, \boldsymbol{s})\}, \quad i = 1, 2, \cdots, n \tag{3}$$

Algorithm describes the details about this performance-oriented allocation policy. The following are explanations of the algorithm.

1) Lines 1-3 read the peak performance value and the bandwidth value. It can be easily extract from the static system information file.
2) Line 4 obtains the required routine type of the computing task according to the information carried by the probe.
3) Lines 5-6 retrieve the list of neural network models of corresponding routines.
4) Lines 7-8 extract information of the task requirements carried by the probe.
5) Lines 9-11 initialize temporary variables for further calculations.
6) Lines 12-28 firstly find the minimum theoretical time along with the corresponding index of CPU routines in the first for loop. Then comparing with the theoretical time of GPU routines. At the end, the index of optimal routine by the minimum theoretical time is obtained. At Line 13 and 21, the *predictUR*() function invokes the corresponding pre-trained neural network model and predicts the value of the peak performance utilization ratio based on the data size of the computing task. At line 22, *norm2*() function calculates the level 2 norm of the data size vector.
7) Line 29 returns the optimal routine from *getTheRoutine*() interface.

VI. Evaluation

This section evaluated **COALA** on two independent platforms: a personal computer(PC) and a HPC server. The PC consists of an Intel Core i5-10400 CPU @ 2.90 GHz with 6 cores, an AMD Radeon RX550 GPU, and PCI-E 1.0 with width x8. The server consists of two Intel Xeon Gold 5120 CPUs @ 2.20 GHz with 14 cores/CPU, an NVIDIA A100 GPU, and PCI-E 3.0 with width x16. All experiments in this section are carried out on Ubuntu 20.04.3 LTS. Firstly, we verified the effectiveness of our described ML-based performance evaluation method and the consequent policy. Subsequently, we conducted experiments on deep learning and numerical simulation jobs, considering their widespread adoption and significant importance in the field. This allowed us to further investigate the capabilities of our approach in these specific contexts.

A. Experiments on the Proposed Performance Evaluation Method and Policy

In this subsection, we first show that the ML-based performance evaluation method can accurately predict the peak performance utilization ratio of GEMM routine. Furthermore, we verified the effectiveness of our **COALA** with the deployed performance-oriented allocation policy by testing GEMM routines from multiple BLAS implementations.

GEMM stands for "General Matrix-Matrix Multiplication," and it performs a highly optimized multiplication operation

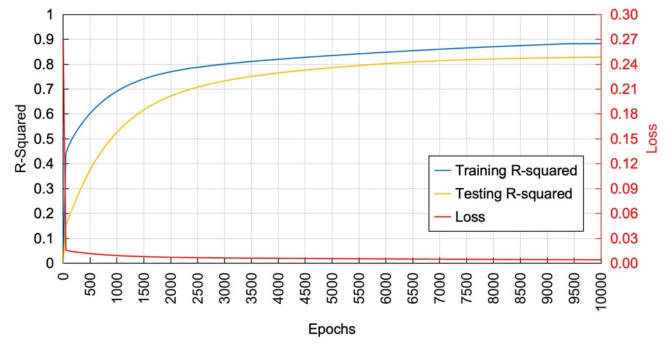


Fig. 6. The results of the neural network model for the OpenBLAS GEMM routine.

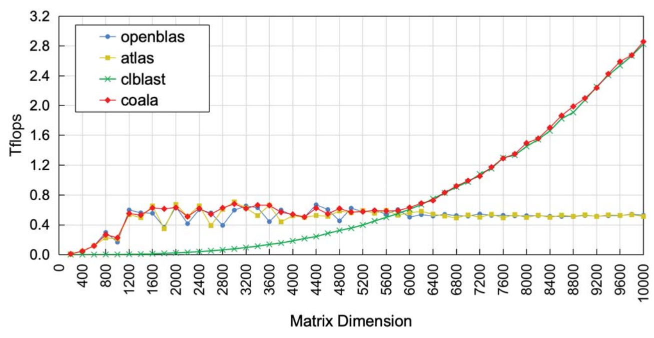


Fig. 7. Comparison on the PC platform (Intel Core i5-10400 CPU and AMD Radeon RX550 GPU). Tflops is calculated by dividing the total number of floating-point operations by the task performing time. The performing time involves the routine computing time and the data migration time.

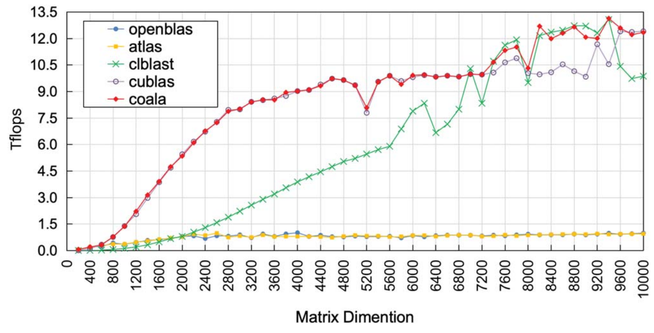


Fig. 8. Comparison on the server platform (Intel Xeon Gold 5120 CPU and NVIDIA A100 GPU). Tflops is calculated by dividing the total number of floating-point operations by the task performing time. The performing time involves the routine computing time and the data migration time.

between two matrices. The GEMM routine in the BLAS library is highly optimized, flexible, and fundamental for General Matrix-Matrix Multiplication. Its significance lies in its efficiency, flexibility, and central role in many linear algebra operations. Applications that benefit significantly from using GEMM include machine learning and deep learning, signal processing, computational physics and engineering, computer vision, and data science. GEMM calculates the product of two matrices A (of size $M \times K$) and B (of size $K \times N$), resulting in a new matrix C (of size $M \times N$). The formal definition of GEMM is

$$C(i,j) = \beta \cdot C(i,j) + \alpha \cdot \sum_{p=1}^{k} A(i,p)B(p,j), \tag{4}$$

where $C_{i,j}$ denotes the element at the i-th row and j-th column of C. Matrix A and B follows the same convention. α and β are any value. The flops (floating point operations per second) of a GEMM can be calculated by

$$\text{flops} = \frac{2MNK}{t}, \tag{5}$$

where t represents the time required to perform the GEMM in seconds. The value of flops represents the performance level of the routine executing the GEMM.

The neural network model is implemented in C++ language for conveniently embedding into **COALA**. It consists of 3 layers including a input layer with 5 neurons for dimensional features of the GEMM input data (matrices A, B, C and coefficients α, β), a hidden layer with 50 neurons by using ReLU activation function for output and a output layer with 1 neuron for outputting the regression result. It use MSE (Mean Square Error) as the loss function and R-squared (the coefficient of determination) as the evaluate function. Using R-squared can provide a quantitative measure of how well the model fits the data. R-squared values range from 0 to 1, with higher values indicating a better fit. We generate 500 random square matrices with a dimension from 100 to 10,000 and use them to test OpenBLAS GEMM routine in the HPC server. The result data of testing the OpenBLAS GEMM function is randomly divided into two parts, with a ratio of 7:3. 70% of the data is used for the model training, and the remaining part is used for testing. The experimental results are as shown in Fig. 6. The training accuracy of the neural network model is shown in the blue curve in Fig. 6, the test accuracy is the yellow curve and the loss is the red curve. When the epochs reach 9250, the loss gradually converges to 0.004, the training R-squared of the model is about to 0.88, and the test R-squared of the model is about to 0.82. The experimental results show that the neural network model can accurately predict the routine's peak performance utilization ratio through the its input data sizes.

Our prototype of **COALA** manages three highly optimized BLAS implenmentations, including OpenBLAS, ATLAS, and ClBLASt, on the PC with an AMD GPU. On the HPC server with an Nvidia GPU, it manages OpenBLAS, ATLAS, ClBLASt, and CuBLAS. We test the benchmark program, which has the GEMM computing task with a dimension of data input from 2,00 to 10,000 (interval of 200). For comparison, this GEMM computing task are performed by using these highly optimized BLAS implementaitons and our **COALA** method, respectively. Fig. 7 shows the results of the program running on the PC platform, and the Fig. 8 shows the results of the program running on the HPC server.

In theory, the execution performance of computing tasks using the **COALA** method should closely approach the performance achieved with manually selected optimal routines. This theoretical expectation is supported by the results depicted in

TABLE II
THREE JOBS AT DARKNET AND THE CORRESPONDING CONFIGURATIONS

Task	Model	Weight	Data
Detection	yolov3 [30]	COCO Dataset [31]	500 images
Prediction	vgg-16 [32]	ILSVRC-2014 [33]	500 images
Training	cifar_small [34]	random initialization	cifar-10 [34]

Fig. 7 and Fig. 8, which align with the theory. The observed consistency between the performance of **COALA** and the optimal routine implies that the proposed allocation policy introduces minimal overhead and does not lead to performance degradation. It is worth noting that certain performance data points in the figures show **COALA** outperforming the theoretical optimum, which is theoretically impossible. This discrepancy can be attributed to unavoidable errors arising from performance fluctuations. Despite this, both Fig. 7 and Fig. 8 confirm the validity of **COALA**, as it consistently maintains the executing performance of computing tasks close to the optimal level, thus affirming the efficacy of **COALA** in optimizing task execution performance.

B. Results With the Darknet Benchmarks

In this subsection, we utilize Darknet [28], an open source neural network framework written in C and CUDA, as the benchmark. It provides several machine learning models, such as YOLO [29] and neural network, for training and inference tasks. In our experiments, we conducted three types of jobs using Darknet: real-time object detection (CNN), prediction for image classification (CNN), and neural network training. For real-time object detection, we used the pre-trained yolov3 [30] architecture and weights from COCO Dataset [31]; the data of detection contains 500 images. For prediction, we used the pre-trained vgg-16 [32] architecture and weights from the ImageNet Large Scale Visual Recognition Challenge 2014 [33]; the data of prediction contains 500 images. For neural network training, we employed the provided CIFAR small architecture offered by Darknet; the data of training is from CIFAR-10 [34] dataset. Table II shows the configurations used for each Darknet task. In our evaluation, all Darknet workloads are ten homogeneous jobs for a given task. This approach ensured consistency and fairness in the assessment process. For comparisons, we complied Darknet source code into three versions, a CPU version (represented by *DN-CPU*) via its original CPU implementation, a GPU version (represented by *DN-GPU*) via its original GPU implementation supported by CUDA, and a **COALA** version (represented by *DN-COALA*) via our **COALA**. Note that, on the PC platforms, we only built Darknet into *DN-CPU* and *DN-COALA* since our PC platform with an AMD GPU cannot support CUDA.

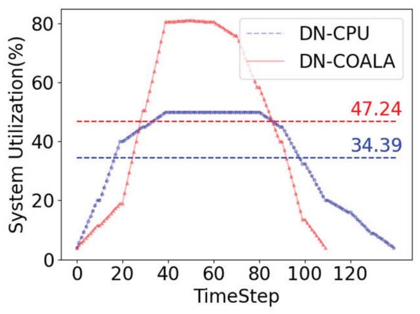
At first, we evaluated the utilization of systems, the results are shown in Fig. 9. Note that overall system utilization ratio U_s is computed by $U_s = (U_c + U_g)/2$, where U_c represents the ratio of CPU utilization ratio and U_g represents the ratio of GPU utilization ratio. If $U_s > 50\%$, it means both CPU and GPU are used. Fig. 9(a), 9(b), and 9(c) show that using **COALA** on the PC platform, the average utilization of the three jobs is improved by 12.3%, 8.0%, and 28.1%, respectively. Furthermore, as shown in Fig. 9(a) and 9(d), the system utilization ratio is more than 50% by introducing **COALA**. This indicates that with the compilation assistance of **COALA**, Darknet can effectively utilized both Intel CPU and AMD GPU, which contributes to improving computing performance. Remember that original Darknet cannot utilize AMD GPU directly because its source code is C and CUDA. It implies that **COALA** enables Darknet to utilize the AMD GPU by adaptively assigning ClBLASt library routines. Fig. 9(d), 9(e), and 9(f) show that using **COALA** on the HPC server platform, the average utilization of the three jobs is improved by 4.3%, 4.6%, and 17.5%, respectively, comparing to *DN-CPU*, and 27.8%, 9.4%, and 27.4%, respectively, comparing to *DN-GPU*.

Then we summarized the performance by the results shown in Fig. 10. Note that the ratio of performance improvement is computed by normalized to the lowest performance, where the ratio in Fig. 10(a) are normalizing to *DN-GPU*, and the ratio in Fig. 10(b) are normalized to *DN-CPU*. Both of them indicate that **COALA** can improve the performance on heterogeneous systems, the performance improvement by over 1.21x and up to 1.32x on the PC platform, and by over 1.78x and up to 4.26x on the HPC server platform.

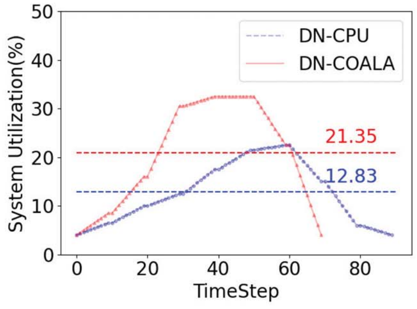
C. Results With the CitcomCu Benchmarks

In this subsection, we utilize CitcomCu (Citcom for Convection Underworld) [35], [36], [37] as the benchmark. CitcomCu is a well-known numerical simulation software used in the field of geodynamics, specifically for modeling mantle convection processes. It uses iterative methods to solve the Stokes equation, which yields the velocity and pressure fields. These fields are then combined with energy conservation equations and geological physical parameters to solve for the temperature field. The primary computational process of CitcomCu involves numerical simulation, which utilizes various computing library routines' API present in its source code. When CitcomCu performing large-scale numerical simulation, numerous dynamic computing tasks will be executed by using library routines. In CitcomCu, an initialization file is a text-based configuration file that serves as the primary input for the software when setting up a numerical simulation. The initialization file contains various parameters and settings that define the specific problem to be modeled, including initial conditions, boundary conditions, iteration steps, and simulation accuracy to the simulation. The number of iterations steps represents the time of the geological changes simulated by CitcomCu, with each step representing one hundred thousand years. The simulation accuracy refers to how closely the CitcomCu simulation represents the real-world mantle processes it aims to model. The higher the simulation accuracy, the larger the data scale generated in the simulation.

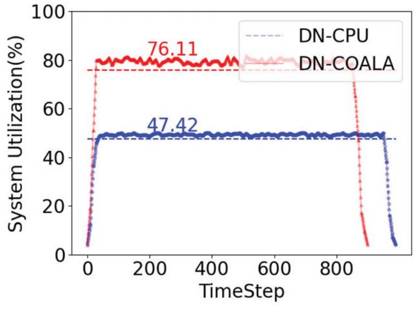
For comparisons, we complied CitcomCu source code into two versions, an original version (represented by *Cit-Ori*) and a **COALA** version (represented by *Cit-COALA*). Note that, in its original source code, CitcomCu only supports libraries only in CPU. We test the time performance of 8 simulation case with



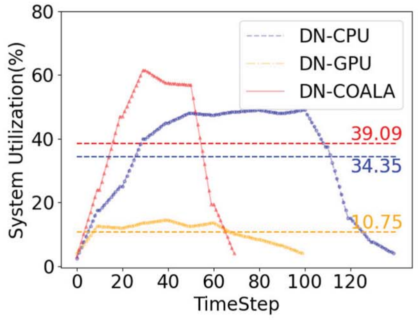
(a) Detection the PC platform



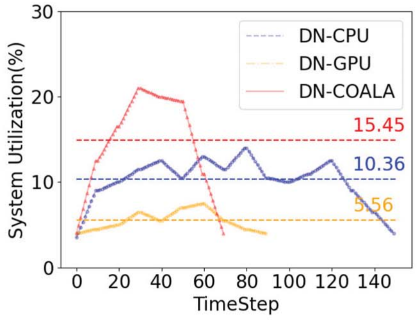
(b) Prediction on the PC platform



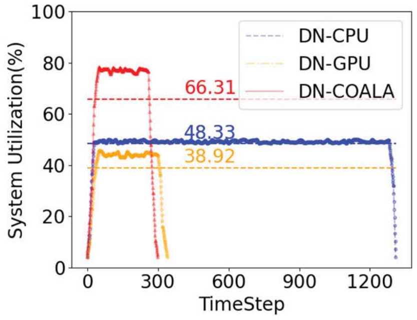
(c) Training on the PC platform



(d) Detection the server platform

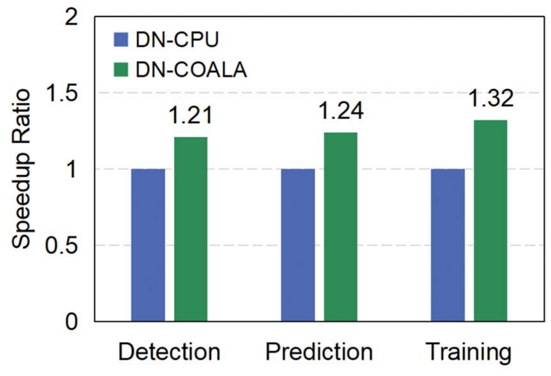


(e) Prediction on the server platform

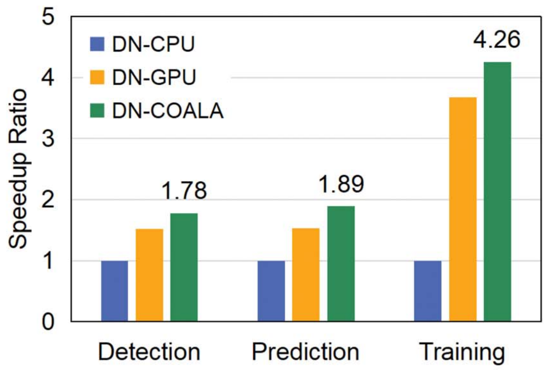


(f) Training on the server platform

Fig. 9. Utilization comparison among *DN-CPU*, *DN-GPU* and *DN-COALA* where dot lines represent the average ratio.



(a) on the PC platform



(b) on the server platform

Fig. 10. Histogramme of speedup ratio by the performance comparison among *DN-CPU*, *DN-GPU* and *DN-COALA*.

8 different initialization files, each case simulating a 20 million year (200 iteration steps) mantle convection process but with different simulation accuracy.

The result is shown in Fig. 11, the simulation accuracy of cases 1 to 8 increases gradually and the speedup ratio is the overall execution time ration of the case running by *Cit-Ori* to the case running by *Cit-COALA*. In Fig. 11, the yellow portion of the dual-color bars represents the ratio of tasks running on the GPU to the total number of tasks, while the green portion represents the ratio of tasks running on the CPU to the total number of tasks. From the Fig. 11, it can be seen that as the simulation accuracy of cases increases, COALA allocates more GPU routines to computing tasks with larger and larger data scales, resulting in an increasingly higher proportion of the yellow part of the bar. Both of Fig. 11(a) and Fig. 11(b) indicate that **COALA** can improve the CitcomCu simulation performance on heterogeneous systems, the performance improvement in our cases by over 1.12x and up to 3.07x on the PC platform, and by over 1.18x and up to 4.36x on the HPC server platform.

VII. Discussion on Limitations

In this section, we discuss the potential drawbacks or areas where the COALA framework may not be optimally effective.

In scenarios of homogeneous systems, homogeneous computing devices exhibit similar performance levels, the efficacy

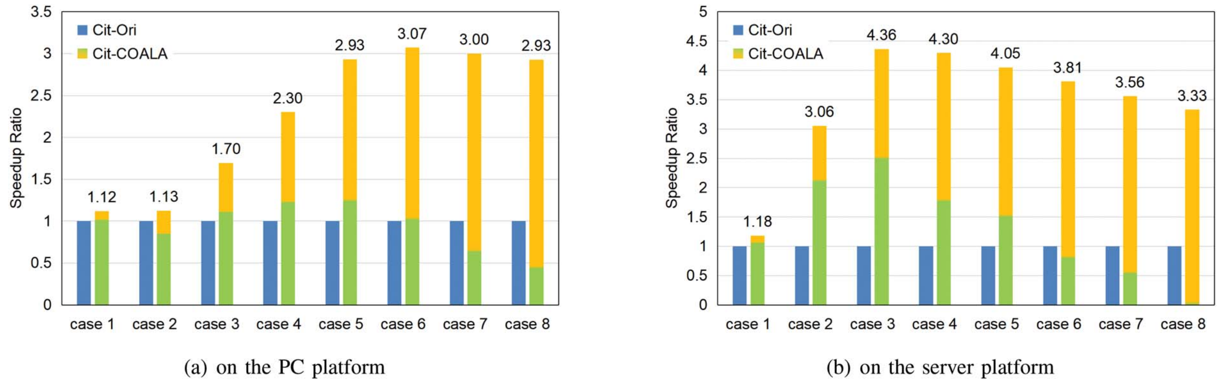


Fig. 11. Histogramme of speedup ratio by the cases of *Cit-Ori* and *Cit-COALA*, where the yellow portion in dual-color bar of *Cit-COALA* represents the ratio of computing tasks running on GPU and the green portion represents the ratio of computing tasks running on CPU.

of the COALA framework in optimizing applications is notably diminished. This limitation stems from the fundamental design of the COALA framework, which predicates its effectiveness on the inherent optimizability of the task execution process. In these systems, the uniformity in computing task characteristics and the architecture of homogeneous computing devices results in a minimal differential impact on task execution. Consequently, in such circumstances, the COALA framework struggles to achieve pronounced optimization effects on the task execution process.

In the realm of dedicated system, dedicated devices are tailored for specific tasks or functions, often demonstrating enhanced efficiency and performance optimization within their respective specialized areas. However, these devices inherently lack the versatility required for general-purpose computing and lack the possibility of fine-tuning the executive process of computing tasks. As a result, the COALA framework, which hinges on the adaptability and tunability of the task execution process, finds its optimization capabilities invalid in such environments.

In scenarios where the primary optimization goal shifts from sheer performance to other goals, current COALA prototype lacks the deployment of tailored optimization strategy. For example, many embedded devices more concern energy consumption, especially high performance per watt. Although the existing optimization strategy deployed in the COALA framework is towards enhancing computing task performance and also reduce overall energy consumption, it does not explicitly target on performance per watt. However, it is important to note that COALA possesses the capability to deploy diverse optimization strategies, involving an energy efficiency strategy. Moving forward, our plan involves the deployment of additional optimization strategies to expand the COALA framework's adaptability to more optimization requirements.

VIII. Conclusion

This paper introduces **COALA** framework, an automated and adaptive library routine allocation framework for heterogeneous systems. **COALA** supports various mainstream library implementations and assigns the most suitable routine for each computing task. It utilizes a compiler-assisted approach to analyze and reconstruct computing tasks by inserting probes to gather relevant information. Additionally, it incorporates a lazy runtime that defers related operations until they are necessary. The runtime involves symbol value recording, data analysis, and binding unbound operations based on the evaluation of **COALA**. **COALA**'s allocation component is responsible for assigning the optimal routine for computing tasks at runtime, taking into account the probe information and the applied allocation policy. Furthermore, the paper introduces a performance-oriented allocation policy founded on the ML-based performance evaluation for library routines. The effectiveness of **COALA** is evaluated on two heterogeneous systems, namely a personal computer and an HPC server. The results from the experiments of Darknet application and Cit-comCu software demonstrate significant performance improvement and system utilization enhancement when utilizing the **COALA** framework.

References

[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for FORTRAN usage," *ACM Trans. Math. Softw. (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[2] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP'98)* (Cat. No. 98CH36181), vol. 3, Piscataway, NJ, USA: IEEE Press, 1998, pp. 1381–1384.

[3] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 61–70.

[4] T. Davis, W. Hager, and I. Duff, "Suitesparse." Accessed: 2014. [Online]. Available: http://faculty.cse.tamu.edu/davis/suitesparse.html

[5] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw. (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.

[6] J. P. De Carvalho et al., "Kernelfarer: replacing native-code idioms with high-performance library calls," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 18, no. 3, pp. 1–22, 2021.

[7] MKL Intel, "Developer reference for Intel® oneAPI math kernel library—C," Intel R, 2021.

[8] M. K. Jaiswal and N. Chandrachoodan, "FPGA-based high-performance and scalable block LU decomposition architecture," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 60–72, Jan. 2012.

[9] X. Zhang, Q. Wang, and Z. Chothia, "OpenBLAS." Accessed: 2012. [Online]. Available: http://xianyi.github.io/OpenBLAS

[10] J. Sun, N. Guan, F. Li, H. Gao, C. Shi, and W. Yi, "Real-time scheduling and analysis of OpenMP DAG tasks supporting nested parallelism," *IEEE Trans. Comput.*, vol. 69, no. 9, pp. 1335–1348, Sep. 2020.

[11] A. Castelló, R. M. Gual, S. Seo, P. Balaji, E. S. Quintana-Orti, and A. J. Pena, "Analysis of threading libraries for high performance computing," *IEEE Trans. Comput.*, vol. 69, no. 9, pp. 1279–1292, Sep. 2020.

[12] R. Whaley and J. Dontarra, "Automatically tuned linear algebra software," in *Proc. ACM/IEEE Conf. Supercomput.*, Piscataway, NJ, USA: IEEE Press, 1998.

[13] C. Nvidia, "CuBLAS library," NVIDIA Corporation, Santa Clara, CA, USA, vol. 15, no. 27, p. 31, 2008.

[14] C. Nugteren, "CLBlast: A tuned OpenCL BLAS library," in *Proc. Int. Workshop OpenCL*, 2018, pp. 1–10.

[15] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–72, 2010.

[16] X. Hou et al., "AlphaR: Learning-powered resource management for irregular, dynamic microservice graph," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 797–806.

[17] Q. Cai, G. Xiao, S. Lin, W. Yang, K. Li, and K. Li, "ABSS: An adaptive batch-stream scheduling module for dynamic task parallelism on chiplet-based multi-chip systems," *ACM Trans. Parallel Comput.*, vol. 11, no. 1, Mar. 2024, Art. no. 6.

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Int. Symp. Code Gener. Optim. (CGO),* Piscataway, NJ, USA: IEEE Press, 2004, pp. 75–86.

[19] "OneAPI specification release 1.2-rev-1." Intel Corporation. [Online]. Available: http://spec.oneapi.io/versions/latest/oneAPI-spec.pdf

[20] R. Keryell, R. Reyes, and L. Howes, "Khronos SYCL for OpenCL: A tutorial," in *Proc. 3rd Int. Workshop OpenCL, IWOCL '15*, New York, NY, USA: Association for Computing Machinery, 2015, Art. no. 24.

[21] P. Ghiglio, U. Dolinsky, M. Goli, and K. Narasimhan, "Improving performance of SYCL applications on CPU architectures using LLVM-directed compilation flow," in *Proc. 30th Int. Workshop Program. Models Appl. Multicores Manycores*, 2022, pp. 1–10.

[22] N. Neves, P. Tomás, and N. Roma, "Compiler-assisted data streaming for regular code structures," *IEEE Trans. Comput.*, vol. 70, no. 3, pp. 483–494, Mar. 2021.

[23] C. Chen, C. Porter, and S. Pande, "CASE: A compiler-assisted scheduling framework for multi-GPU systems," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 17–31.

[24] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, "heFFTe: Highly efficient FFT for exascale," in *Proc. Int. Conf. Comput. Sci.*, New York, NY, USA: Springer-Verlag, 2020, pp. 262–275.

[25] P. Springer, T. Su, and P. Bientinesi, "HPTT: A high-performance tensor transposition C++ library," in *Proc. 4th ACM SIGPLAN Int. Workshop Libraries, Languages, Compilers Array Program.*, 2017, pp. 56–62.

[26] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore cpus," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.

[27] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML-10)*, 2010, pp. 807–814.

[28] J. Redmon, "Darknet: Open source neural networks in C." [Online]. Available: http://pjreddie.com/darknet/

[29] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[30] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*.

[31] T.-Y. Lin et al., "Microsoft COCO: Common objects in context," 2015, *arXiv:1405.0312*.

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[33] O. Russakovsky et al., "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis. (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[34] A. Krizhevsky et al., "Learning multiple layers of features from tiny images," 2009.

[35] M. Kronbichler, T. Heister, and W. Bangerth, "High accuracy mantle convection simulation through modern numerical methods," *Geophys. J. Int.*, vol. 191, no. 1, pp. 12–29, 2012.

[36] G. Morra, "Pythonic geodynamics: Implementations for fast computing on Jupyter notebooks," in *Proc. AGU Fall Meeting Abstr.*, vol. 2019, Dec. 2019, ED53F-0902.

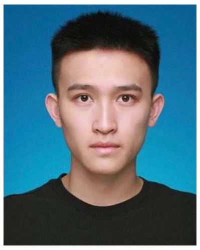
[37] J. Assuncão and V. Sacek, "Benchmark comparison study for mantle thermal convection using the CitcomCU numerical code," in *Proc. 15th Int. Congr. Brazilian Geophys. Soc. EXPOGEF, Rio de Janeiro*, Brazil. Brazilian Geophys. Soc., 2017, pp. 1630–1635.

Qinyun Cai received the B.S. degree from Harbin Institute of Technology, China, in 2014. He is currently working toward the Ph.D. degree with the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include parallel and distributed processing, high-performance computing, compiler optimization, heterogeneous computing systems, and distributed computing systems.

Guanghua Tan received the Ph.D. degree from the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. Currently, he is an Associate Professor with the College of Information science and Engineering, Hunan University, Changsha, China. His research interests include medical image processing, computer vision, 3D modeling, and computer graphics.

Wangdong Yang received the M.S. degree in computer science from Central South University, China, and the Ph.D. degree in computer science from Hunan University, China. He is a Professor of computer science and technology with Hunan University, China. His research interests include modeling and programming for heterogeneous computing systems, parallel and distributed computing, and numerical computation. He has published more than 60 papers in International conferences and journals. He is currently served on the editorial boards of IEEE INTERNET OF THINGS JOURNAL.

Xianhao He received the M.S. degree in computer science from the National University of Defense Technology, in 2021. Currently, he is working toward the Ph.D. degree in computer science and technology with the College of Computer Science and Electronic Engineering, Hunan University. His research interests include high-performance computing, compiler optimization, and machine learning.



Yuwei Yan received the B.S. degree in transportation engineering from Hohai University, in 2021. Currently, he is working toward the M.S. degree in computer science and technology with the College of Computer Science and Electronic Engineering, Hunan University. His research interests include high-performance computing, artificial intelligence, and edge intelligence.

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor of computer science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, Big Data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored more than 860 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently an Associate Editor of *ACM Computing Surveys* and *CCF Transactions on High Performance Computing*. He is a Member of Academia Europaea (Academician of the Academy of Europe).

Kenli Li (Senior Member, IEEE) received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003. He was a Visiting Scholar with the University of Illinois with Urbana-Champaign, from 2004 to 2005. He is currently a Cheung Kong Professor of computer science and technology with Hunan University (HNU), the Vice-President of the HNU, the Dean of the College of Computer Science and Electronic Engineering of HNU, and the Director in the National Supercomputing Center in Changsha. His research interests include parallel and distributed processing, high-performance computing, and Big Data management. He has published over 350 research papers in international conferences/journals. He is a Fellow of the CCF. He is currently serving or has served as an Associate Editor for IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING.