High Performance OpenCL-Based GEMM Kernel Auto-Tuned by Bayesian Optimization

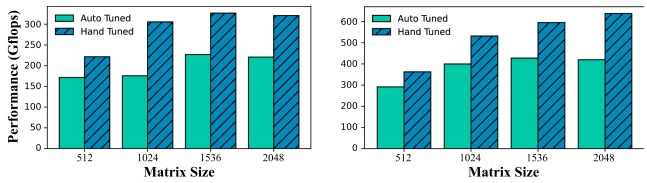
Shengle Lin, Guoqing Xiao, *Member, IEEE*, Haotian Wang, Wangdong Yang, Kenli Li, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—OpenCL has become the favored framework for emerging heterogeneous devices and FPGAs, owing to its versatility and portability. However, OpenCL-based math libraries still face challenges in fully leveraging device performance. When deploying high-performance arithmetic applications on these devices, the most important hot function is General Matrix-matrix Multiplication (GEMM). This study presents a meticulously optimized OpenCL GEMM kernel. Our enhanced GEMM kernel emphasizes two key improvements: 1) a three-level double buffer pipeline that efficiently overlaps data fetching with floating-point computations; 2) a fine-grained prefetching strategy of private memory to increase device occupancy by optimizing register unit utilization. Furthermore, this work presents a Bayesian Optimization (BO) tuner for kernel auto-tuning. Experimental results demonstrate considerable optimization improvement and performance advantages achieved on diverse OpenCL devices. Additionally, the BO tuner demonstrates superior efficiency and robustness, outperforming contemporary tuning methods.**

***Index Terms*—OpenCL, GEMM, auto-tuning, Bayesian optimization, high-performance computing.**

I. Introduction

In the realm of parallel computing, heterogeneous architectures are becoming increasingly popular due to their cost efficiency, flexibility, and potential for development. For deploying high-performance arithmetic applications across these platforms, the most important routine is General Matrix-matrix Multiplication (GEMM) [1], [2]. Nevertheless, achieving high performance and portability in GEMM kernel, especially when dealing with varying data patterns on devices with diverse architectures, is a formidable challenge.



(a) CLBlast SGEMM on JM 9230. (b) CLBlast SGEMM on Intel FPGA.

Fig. 1. Unstable performance of auto-tuned CLBlast on lightweight devices.

While several high-performance variants of Netlib BLAS exist for CPUs [3], [4], [5], options for heterogeneous platforms consisting of GPUs, General Purpose Digital Signal Processors (GPDSPs), field-programmable gate arrays (FPGAs), and other processors or accelerators remain limited. For example, cuBLAS [6] is renowned for its high performance and proprietary API, yet its reliance on CUDA restricts it to NVIDIA devices. Similarly, AMD's ROCBLAS [7] is tailored for AMD GPUs that support ROCm. The Open Computing Language (OpenCL) addresses this gap by offering a unified, cross-platform framework that efficiently executes high-performance computing tasks across diverse hardware architectures, making it essential in areas requiring heterogeneous computational power. Moreover, Intel *SYCL* and Huawei *AscendCL* further simplify the development process while still providing processing power on heterogeneous platforms. The OpenCL-based libraries like CLBLAS [8] and CLBlast [9] can deliver good arithmetic performance from CPUs to GPUs and beyond. Note that CLBlast further leverages parameterized kernels and auto-tuning techniques to offer a superset of BLAS routines with greater performance portability.

However, in the case of emerging lightweight accelerators, our empirical analysis has revealed the suboptimal performance of GEMM kernels when benchmarking these OpenCL-based libraries. Specifically, matrix data prefetching is limited to local memory, leading to inefficient utilization of vector register units. Moreover, current auto-tuning strategies typically rely on random search or heuristic algorithms [9], [10], resulting in considerable search overheads and unsatisfactory configurations. As demonstrated in Fig. 1, our observation shows that the auto-tuned configurations of CLBlast SGEMM kernels achieve

Received 16 April 2024; revised 22 December 2024; accepted 7 July 2025. Date of publication 10 July 2025; date of current version 31 July 2025. The work was supported in part by the National Key R&D Program of China under Grant 2023YFB3002702, in part by the Key Program of National Natural Science Foundation of China under Grant U21A20461, Grant 92055213, and Grant 62227808, in part by the Natural Science Foundation of Hunan Province, China under Grant 2023GK2002 and Grant 2024JJ2026, and in part by the Natural Science Foundation of Chongqing under Grant CSTB2022NSCQ-MSX1213. Recommended for acceptance by P. D'Ambra. *(Corresponding author: Guoqing Xiao.)*

Shengle Lin, Haotian Wang, Wangdong Yang, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China, and also with the National Supercomputing Center in Changsha, Hunan 410082, China (e-mail: lsl036@hnu.edu.cn; wanghaotian@hnu.edu.cn; yangwangdong@hnu.edu.cn; lkl@hnu.edu.cn).

Guoqing Xiao is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China, and also with the Research Institute of Hunan University, Chongqing 401135, China (e-mail: xiaoguoqing@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/TPDS.2025.3587673

TABLE I
MAINSTREAM OPENCL PLATFORM INFORMATION

Computing platforms	Supported OpenCL version	Heterogeneous resources
Mesa	1.1 ∼ 1.2	AMD, NVIDIA GPU
ROCm	2.0	AMD CPU and GPU
CUDA	2.0	NVIDIA GPU
FPGA	1.2 ∼ 2.0	Intel Emulation Platform

sub-optimal performance on Jingjia Micro 9230 and Intel FPGA Emulation Platform for OpenCL, because it accidentally disabled the usage of local memory.

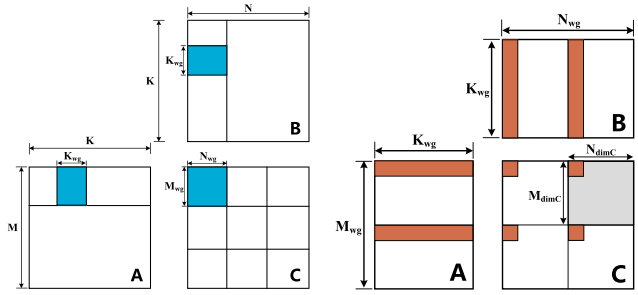
In this context, we design a highly optimized OpenCL-based GEMM kernel developed based on the CLBlast. Utilizing its extensive compatibility with a variety of devices, we have implemented a double-buffer pipeline optimization and a fine-grained prefetching strategy. Furthermore, we introduce an adaptive tuning component based on Bayesian Optimization (BO), facilitating rapid identification of optimal parameter configurations for all kernels within a minimal number of iterations. Our experimental results show SGEMM performance improvements of 32.89% on Intel FPGA Emulation, 30.82% on Jingjia Micro, 12.39% on AMD RX550, 13.76% on NVIDIA A100-40 GB, and 75.58% on Intel Xeon 5120 CPU compared to CLBlast. Additionally, our Bayesian optimization (BO) tuner can find parameters obtaining 94.74% (near-optimal) of the best-known performance configuration while incurring a mere 10.35% search overhead compared to random search. In general, these results prove that our carefully designed GEMM is highly competitive on diverse devices, and the BO tuner can effectively improve configuration efficiency and accuracy. In the future, these critical optimization technologies have the prospect of being extended to other parallel programming frameworks akin to OpenCL, such as Intel *SYCL* and Huawei *AscendCL*.

II. BACKGROUND & MOTIVATION

A. OpenCL Basics

The OpenCL is recognized as a framework facilitating heterogeneous computing [11]. As a generic open standard, it enables general-purpose parallel programming across diverse heterogeneous platforms encompassing various processors such as CPUs, GPUs, DSPs, and FPGAs [12], [13], [14]. In stark contrast to CUDA and ROCm, which are constrained to NVIDIA's and AMD's GPUs respectively, OpenCL boasts unique compatibility across diverse accelerators from a multitude of vendors—including AMD, NVIDIA and Intel, as listed in Table I. This feature markedly enhances the portability of OpenCL-based kernels in comparison to other vendor-specific programming frameworks.

An OpenCL platform typically operates under the governance of a host-end, which controls one or more OpenCL devices. The device-end generally comprises numerous *compute units* (CUs), each housing multiple *process elements* (PEs). Upon kernel execution, the host-end defines an N-dimensional space, called *NDRange*, determining the distribution of multiple *work-items*



(a) First tiling for work-groups in global-memory. (b) Further tiling for work-items in local-memory and strided access.

Fig. 2. Multi-level GEMM Tiling implementation based on OpenCL.

for task execution. Each work-item, bearing a unique worker ID, is systematically assigned into a *work-group*. During the execution, the computational tasks of a work-group are scheduled to a CU, where all PEs concurrently engage in computation. The *execution unit* is the minimum unit being executed by CUs in parallel, whose size depends on the architecture of the particular hardware (usually 32 or 64 work-items), and is called *wavefront* or *warp* on AMD or NVIDIA GPUs, respectively.

Within an OpenCL kernel, memory access by work-items is structured across three levels of abstract hierarchical memory. The *global memory* serves as a shared domain, accessible for read/write operations by all work-items. Within each compute unit, OpenCL defines *local memory* dedicated to a particular work-group, enabling data sharing amongst all work-items within that group. The final tier, *private memory*, typically correlates with the register files in most devices, ensuring data privacy such that the private memory of each work-item remains invisible to others.

B. Parameterized GEMM on OpenCL

General matrix-matrix multiplication (GEMM) is a fundamental but crucial routine in BLAS, and its standard operation is defined as follows:

$$C = \alpha \cdot op(A) \times op(B) + \beta \cdot C, \tag{1}$$

where α and β are scalar values, and $op(A), op(B)$ and C are matrices of dimensions $M \times K, K \times N$ and $M \times N$, respectively. The $op(\cdot)$ operator specifies whether the matrix should be transposed(T) or not(N), resulting in four types of matrix multiplications.

Matrix tiling, also called blocking, is an indispensable technique for achieving high-performance computations across various architectures. Since matrix multiplication requires $O(N^3)$ floating-point multiply-add operations on $O(N^2)$ data, matrix tiling efficiently leverages the hierarchical memory structures of devices, thereby enhancing the data reuse ratio and overall performance. In the context of OpenCL, the classic implementation of the GEMM employs a two-level tiling strategy [15], [16], as shown in Fig. 2. The first tiling level focuses on global memory and local memory within a work-group computation. The second level further divides the matrix tiles, distributing

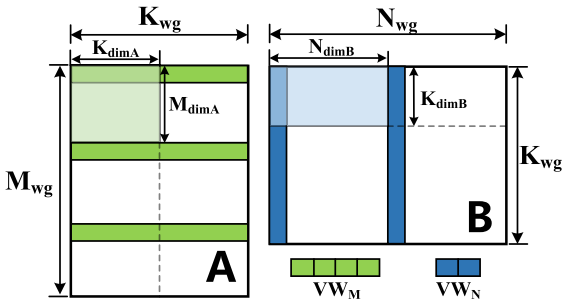


Fig. 3. Extra tuning parameters for loading tiles and vectorization.

matrix elements to work-items to alleviate the access overhead between local memory and registers.

As shown in Fig. 2(a), a work-group computes the results for an $M_{wg} \times N_{wg}$ tile of matrix C. To perform the multiplication, each work-group loads an $M_{wg} \times K_{wg}$ tile of A and a $K_{wg} \times N_{wg}$ tile of B. Here, K_{wg} acts as the tiling factor along the K-dimension, ensuring that both tile A and tile B fit into local memory. Fig. 2(b) illustrates the further tiling strategy. The gray region represents the $M_{dimC} \times N_{dimC}$ work-group grid and the orange area denotes the elements computed by a single work-item. Each work-item accumulates results for an $M_{wi} \times N_{wi}$ sub-block of C, where $M_{wi} = M_{wg}/M_{dimC}$ and $N_{wi} = N_{wg}/N_{dimC}$. In this example, the work-item uses strided memory access to mitigate potential bank conflicts.

To achieve high performance across devices, CLBlast further offers a highly parameterized GEMM kernel that can be easily tuned. As illustrated in Fig. 3, CLBlast incorporates extra parameters such as M_{dimA}, N_{dimB} for reshape of work-group to optimize data loading patterns. Moreover, vector widths VW_M and VW_N for different devices, and loop unrolling factor K_{wi} are also taken into account.

C. Contributions

Although CLBlast sets a large number of tunable parameters to achieve performance portability, there is still room for further optimization of its GEMM implementation. First, it solely relies on the OpenCL compiler for low-level optimizations, which leads to data access and computation being difficult to cover efficiently. Second, register reuse is not finely considered, resulting in low device occupancy. Furthermore, our observations (Fig. 1) show that there is an urgent need for a reliable and stable tuning method to improve the performance of kernels on different devices. In light of these shortcomings, the primary contributions of this work are outlined as follows:

- Design a pipelined GEMM kernel that significantly alleviates the overhead associated with data access between hierarchical memory structures.
- Implement a fine-grained prefetching strategy to enhance the reuse ratio of register units, thereby improving efficiency and occupancy.
- Retain 11 tunable parameters and propose a Bayesian optimization tuner to achieve adaptive parameters tuning across devices with high accuracy and efficiency.

III. OpenCL GEMM Optimization Scheme

Our optimized kernel retains the 11 critical parameters to ensure high-performance portability. These parameters encompass several aspects as follows:

- M_{wg}, N_{wg}, K_{wg}: determine the matrix tiling size to explore the utilization of local memory;
- M_{dimC}, N_{dimC}: determine the size of work-group for work-item level parallelism;
- M_{dimA}, N_{dimB}: define the data loading reshape to leverage memory bandwidth;
- STR_M, STR_N: define whether or not to strided memory access (Fig. 2(b)) to mitigate possible bank conflict;
- VW_M, VW_N: define vector width (SIMD) of private memory to utilize instruction level parallelism.

Based on these parameter settings, we implement a three-level pipelined double-buffer algorithm and a fine-grained register prefetching optimization for the GEMM kernel under the assumption that local memory is available.

A. Three-Level Double-Buffer Policy

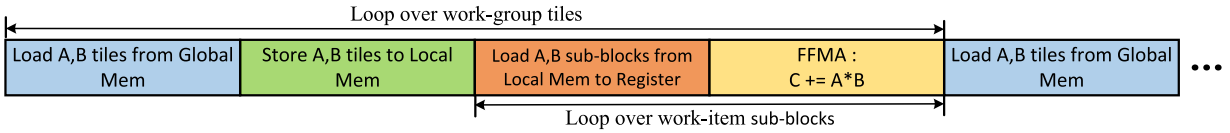
The execution of a GEMM operation on a hierarchical memory device typically involves the following four steps:

1) *Loading Tiles:* The $M_{wg} \times K_{wg}$ tile of matrix A and the $K_{wg} \times N_{wg}$ tile of matrix B are loaded from global memory into temporary registers.
2) *Storing Tiles:* These loaded tiles are transferred from the temporary registers to local memory.
3) *Sub-block Fetching:* Within each inner loop, each work-item loads an $M_{wi} \times K_{wi}$ sub-block of A and a $K_{wi} \times N_{wi}$ sub-block of B into private memory.
4) *Arithmetic Computing:* Floating-point Fused Multiply Add (FFMA) operations are performed to update the corresponding $M_{wi} \times N_{wi}$ sub-block of C.

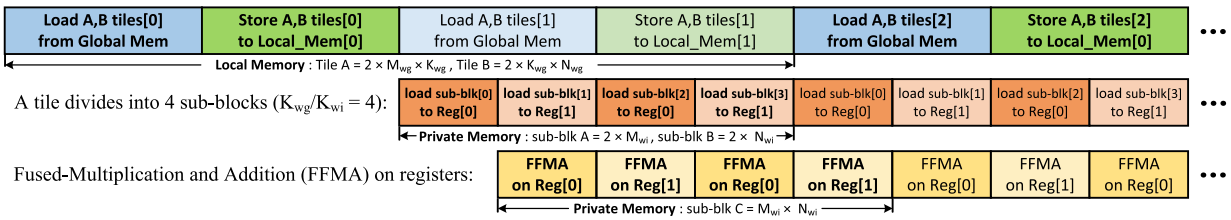
These four steps, which depend on three levels of memory hierarchies (global, local and private) and different instruction components, are typically executed sequentially in OpenCL BLAS libraries, as shown in Fig. 4(a). Each memory-accessing step, characterized by high instruction latency, necessitates synchronization operations to ensure data consistency.

In OpenCL, two common strategies are employed to mitigate the high latency associated with memory access: work-item level parallelism and instruction level parallelism. The GEMM kernel inevitably employs the outer-product algorithm to improve arithmetic intensity for addressing the issue of bandwidth bottleneck [1], [3]. However, the substantial register consumption by outer-product results in low occupancy on computing devices, which makes it challenging for groups of work-items (namely warps or wavefronts) parallelism to overlap the latency effectively. Consequently, to improve the parallelism of the instruction level within a single execution unit, it becomes essential to implement a pipeline for the four steps of the GEMM kernel.

In Fig. 4(b), we present a three-level pipeline design for the GEMM kernel, leveraging a double-buffer strategy at each memory hierarchy to overlap access latency. Specifically, local memory is partitioned into two buffers to accommodate double



(a) A four-step straightforward serial implementation of GEMM kernel.



(b) Three-level double buffer pipelined optimization of GEMM kernel. Shades of colors represent two different buffers.

Fig. 4. GEMM kernel pipelining optimization overview.

tiles in alternating iterations. While the previous tile undergoes computation in internal loops, the next tile is prefetched into the other buffer. Similarly, private memory is divided into two parts to enable sub-block prefetching. To better mask the latency in the double-buffer kernel, the loop unrolling factor K_{wi} is fixed to 1. Assuming that $K_{wg} = 4$, each work-item executes 4 loops to accumulate the partial results for sub-block C. Fig. 4(b) illustrates the memory space required for each level of double buffering. Our approach expects to maximize the utilization of hardware resources by ensuring that data access at odd iterations overlaps as much as possible with FFMA computations at even iterations. Therefore, this three-level double buffer policy effectively eliminates data dependencies and mitigates data transfer overhead across different memory hierarchies.

B. Analysis Model for Pipeline

To address the challenge of determining the optimal sizes for tiles and sub-blocks for pipelining, we rely on a simple but useful model to quantitatively evaluate the data transfer cost and computation time. In a practical scenario, the ideal situation is to attain a theoretical equilibrium where one buffer is loading data for the next iteration while another buffer is engaged in FFMA computation.

Consider an OpenCL-supported device with a global memory bandwidth of BW_G and a local memory bandwidth of BW_L. For simplicity, we assume that the loading and writing bandwidths are equivalent. The register access latency is typically about a single clock cycle, which is negligible for our analysis. Suppose that matrix A has dimension $M \times K$, matrix B has dimension $K \times N$, and matrix C is $M \times N$. Each tile of matrix A is defined as $Tile_A = M_{wg} \times K_{wg}$, and the tile size of B is $Tile_B = K_{wg} \times N_{wg}$. Thus, the time required for loading tile A and tile B from global memory to local memory can be expressed as:

$$T_{A,g2l} = \frac{Tile_A \times sizeof(data)}{BW_G} + \frac{Tile_A \times sizeof(data)}{BW_L}, \tag{2}$$

$$T_{B,g2l} = \frac{Tile_B \times sizeof(data)}{BW_G} + \frac{Tile_B \times sizeof(data)}{BW_L}, \tag{3}$$

where the two terms of the above equations correspond to the processes of loading and storing a tile (blue and green blocks in Fig. 4) through intermediate registers, respectively.

Subsequently, each tile is further partitioned into sub-blocks, which are loaded into the private memory by individual work-items. Assume that the dimensions of sub-block A and B are $subBLK_A = M_{wi} \times K_{wi}$ and $subBLK_B = K_{wi} \times N_{wi}$, respectively. So the time of loading A, B sub-blocks from local memory to private memory can be described as:

$$T_{A,l2p} = \frac{subBLK_A \times sizeof(data)}{BW_L}, \tag{4}$$

$$T_{B,l2p} = \frac{subBLK_B \times sizeof(data)}{BW_L}. \tag{5}$$

Given that the 2D work-group grids parallelize at the M and N dimensions, the number of sub-blocks that each work-item needs to process is $K_n = K_{wg}/K_{wi}$. Let's consider that each FFMA operation takes C_{FMA} instruction cycles to complete, the frequency of computing device is denoted as f, and each work-item can compute V elements in parallel through SIMD processing. Thus the computation time of sub-block C could be estimated as:

$$T_{c,workitem} = \frac{M_{wi} \cdot K_{wi} \cdot N_{wi}}{.} C_{FMA} V \cdot f. \tag{6}$$

Correspondingly, the time required to compute a tile C within a work-group can be expressed as:

$$T_{c,workgroup} = \frac{M_{wi} \cdot K_{wg} \cdot N_{wi}}{.} C_{FMA} V \cdot f. \tag{7}$$

In order to maximize the utilization of computational resources while minimizing idle time, it's crucial to take into account the three-level memory hierarchies and ensure that data accessing and computing times overlap as much as possible. Based on the analysis above, the theoretical double-buffer

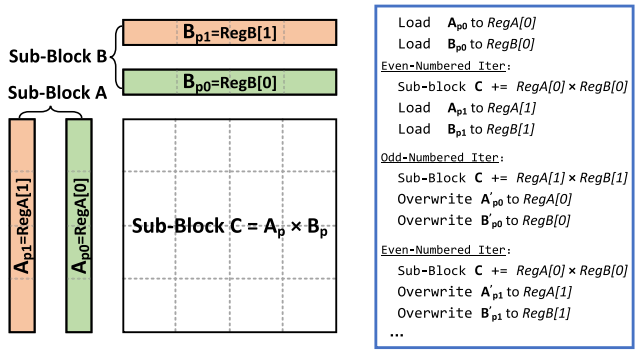


Fig. 5. The general prefetching strategy of a work-item in private memory for a 4×4 sub-block accumulation.

pipelining optimization should be organized as follows:

$$\begin{cases} T_{c,workitem} &= T_{A,l2p} + T_{B,l2p} \quad (local\ memory) \\ T_{c,workgroup} &= T_{A,g2l} + T_{B,g2l} \quad (global\ memory). \end{cases} \tag{8}$$

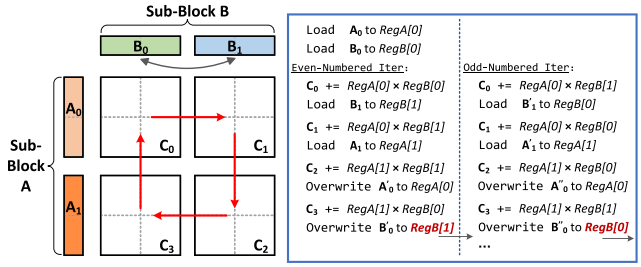
As illustrated in Fig. 4(b), the objective is not only to ensure that the computation of each sub-block overlaps with the prefetching of the next sub-block (*local memory* level) but also to balance the cost of computing and buffering within a tile (*global memory* level). Due to the differences in architectural parameters across diverse devices, we consider utilizing these pipeline analysis formulas in the auto-tuning module (Section IV) to speed up the tuning process.

C. Fine-Grained Prefetching Strategy

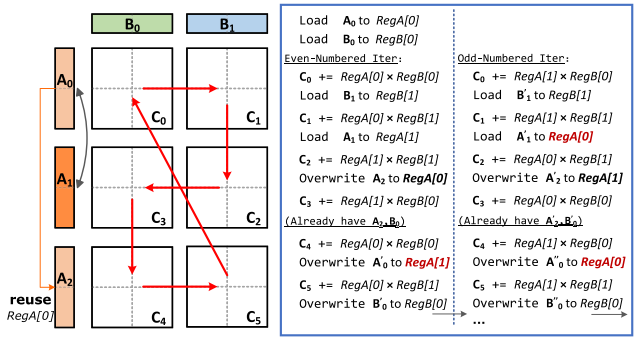
In the context of Fig. 2, each work-item is responsible for executing FFMA operations within private memory. Typically, it's essential to fully utilize the vector register unit's capacity to enhance instruction-level parallelism. However, the rareness of private memory leads to the fact that its utilization directly impacts the parallelism among execution units (warps or wavefronts) [17], [18]. In other words, the additional consumption by register double buffering affects work-item level parallelism, which in turn reduces the hardware resource occupancy. Recent studies [19], [20] have shown performance enhancements by efficiently utilizing register units on GPUs. Therefore, we propose a fine-grained vector register prefetching strategy to strike a balance between work-item level parallelism and instruction-level parallelism.

Considering a 4×4 sub-block C, we analyze the general double buffer scheme in Fig. 5. This approach generally results in an extra private memory overhead of about 50% while loading the A_{p1} and B_{p1}. The increase is attributable to the use of 4×4 vector registers for accumulating sub-block C, coupled with the allocation of an additional $2 \times (4 + 4)$ register units for the double-buffer prefetching.

We design the following strategy to optimize private memory reuse while maintaining the tiling hierarchy intact for work-item parallelism. As illustrated in Fig. 6(a), a 4×4 sub-block of C is further partitioned into four 2×2 blocks C_0, C_1, C_2 and C_3. Initially, A_0 and B_0 are loaded from local memory into private memory. In iteration 0, $A_0 \times B_0$ is accumulated to subblock



(a) Fine-grained prefetching micro-kernel for a 4×4 sub-block.



(b) Fine-grained prefetching micro-kernel for a 6×4 sub-block.

Fig. 6. The fine-grained prefetching optimization with different sub-block sizes in private memory.

C_0 with concurrent prefetching B_1 to $RegB[1]$. Subsequently, $RegA[0]$ is reused to compute $C_1 \mathrel{+}= A_0 \times B_1$ while prefetching A_1 to $RegA[1]$. Upon completing this computation, $RegA[0]$ becomes available to prefetch A'_0 for the next iteration. In the following step, the accumulation $C_2 \mathrel{+}= A_1 \times B_1$ effectively reuses $RegB[1]$, while $RegA[0]$ is simultaneously overwritten by A'_0. Next, $RegB[1]$ will be overwritten by B'_0, while accumulating $C_3 \mathrel{+}= A_1 \times B_0$ with the reuse of $RegA[1]$. In subsequent iterations, we continue to accumulate this sub-block C with a similar register reuse logic. The primary difference lies in the alternating overwriting of vector register B.

Our fine-grained prefetching strategy efficiently utilizes private memory, employing only a total of $(4 + 4)$ vector register units for accumulating sub-block C. In contrast to the general method, our approach enables GEMM to conserve 8 register units (50%) for double-buffering, which can improve the overall occupancy of OpenCL devices. During sub-block arithmetic, our carefully designed computation order will augment the reuse of registers, which is of benefit to reducing instruction latency and enhancing computational throughput for the kernel. Furthermore, our fine-grained prefetching optimization can be easily adapted to various sub-block sizes, like $4 \times 6, 4 \times 8, 6 \times 8$, and so on. As depicted in Fig. 6(b), we provide an example for a 6×4 sub-block kernel. Note that the reuse of $RegA[0]$ within an iteration allows us to load both A_0 and A_2, thereby further enhancing private memory utilization. Since it still only requires $(4 + 4)$ units of private memory for prefetching, the savings in vector register units are more significant (150%) than for the 4×4 case.

In summary, the fine-grained prefetching strategy for private memory can be viewed as a delicate trade-off between Single Instruction Multiple Data (SIMD) parallelism and device occupancy. Since different matrix sizes need to be handled, and each requires distinct group-thread partitions tailored to the specific OpenCL devices, we provide a range of template implementations for fine-grained micro-kernels.

IV. Auto-Tuning Kernels by BO

Identifying the optimal parameter setting for the GEMM kernel is complicated due to the intricate interplay among double-buffer overlap, local and private memory utilization, and the balance between occupancy and instruction-level parallelism. Additionally, the cross-device characteristic of OpenCL compounds more challenges, as optimal settings vary with diverse matrix sizes and distinct architectures. Thus, with an emphasis on Gflops as a straightforward and intuitive performance metric, we consider introducing an auto-tuning technology to assess the impact of parameter selection.

Some existing OpenCL-based libraries use random search and heuristic algorithms for parameter tuning. However, the instability of these tuning methods often leads the kernel to get a terrible parameter configuration, as shown in Fig. 1. In response to these challenges and to harmonize our two optimizations across diverse OpenCL devices, we consider utilizing a machine learning algorithm for auto-tuning. Summarily, we provide an efficient, robust, and reliable adaptive parameter tuner based on Bayesian Optimization (*BO*).

A. Bayesian Optimization Analysis

Bayesian Optimization *(BO)* [21], [22] is an efficient algorithm designed for optimizing complex black-box objective functions that are expensive to evaluate. Its utility is pronounced in situations with extensive search space that happens to be a feature of our kernel. The overview of the Bayesian Optimization algorithm is as follows:

- *Modeling the black-box function:* BO relies on a *Surrogate Probabilistic Model* to approximate the behavior of the objective function within the expansive search space. It will be iteratively updated as new samples are collected, allowing it to converge towards the optimal solution.
- *Acquisition Function:* BO uses an *Acquisition Function* to direct the search process. It dictates the next sampling locations, adeptly balancing exploration (sampling in uncertain regions) and exploitation (where the model predicts high performance).
- *Iterative updating:* Once a new sample is determined, BO will run the real object function to evaluate its performance. The corresponding result will be used to back-update the surrogate probabilistic model, refining its understanding of the objective function's behavior.
- *Convergence:* The iterative process persists until a predefined stopping criterion is satisfied, either when the results converge to an optimum or when the maximum number of iterations has been completed.

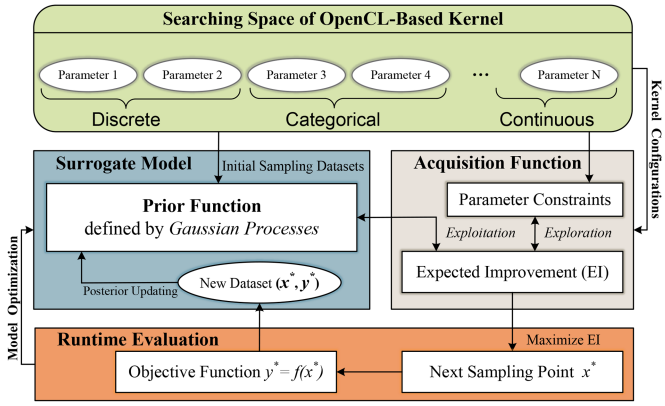


Fig. 7. An overview of BO-based adaptive tuner.

Compared to stochastic search methods and heuristic algorithms, BO at least has three significant advantages. First, BO leverages a probabilistic model that can incorporate prior knowledge into the search process. This model can be continually refined with new sample points, enabling it to make well-informed decisions aimed at achieving improved performance. Second, BO is well-suited for handling noisy objective functions [23] because the surrogate model can account for noise during its predictions. This enhances robustness and provides greater confidence when evaluating the performance of the OpenCL kernels. Finally, BO aims to find the best configuration with as few evaluations as possible, which is meaningful in the case of large parameter dimensions.

B. BO-Based Adaptive Tuner

Fig. 7 illustrates the structure of our BO-based adaptive tuner for generic OpenCL kernels. The first component is the search space for a specific kernel, referred to as the *Bayesian domain*. This domain categorizes parameters into various types such as discrete, categorical, and continuous, thereby facilitating the involvement of diverse variable types in the tuning process. For our GEMM kernel, all parameters are discrete, dictating aspects like tile shapes, work-group grids, or the vector width of the device.

Initially, the BO tuner begins by randomly sampling t parameter combinations $\mathbf{X} = \{x_1, x_2, \ldots, x_t\}$ and recording their actual performances by benchmarking. This process generates the initial sampling dataset $D_{1:t} = \{(x_1, y_1), \ldots, (x_t, y_t)\}$. From these samples, the BO tuner constructs a *Surrogate Model* employing Gaussian processes (GPs). The *Gaussian Process* [24] is a renowned non-parametric method for inferring distributions over black-box functions. Specifically, it is a random process where any point x_i has its random variable $f(x_i) = y_i$ and where the joint distribution of a finite number of variables $p(f(x_1), \ldots, f(x_N))$ is itself Gaussian:

$$p(f \mid \mathbf{X}) = \mathcal{N}(f \mid \mu, \mathbf{K}), \tag{9}$$

where μ is the mean function and $\mathbf{K}_{ij} = \kappa(x_i, xj)$ is a positive definite function called *Gaussian Kernel*. In BO tuner we use common setting $\mu = 0$ and the squared exponential kernel $\mathbf{K}$,

also known as Gaussian kernel:

$$\kappa\left(x_i, xj\right) = \sigma^2 exp\left(-\frac{1}{2l^2}\left(x_i - x_j\right)^T\left(x_i - x_j\right)\right), \quad (10)$$

where σ is the variance that controls vertical variation, l is length scale that controls smoothness of function f. When a new sample point (x^*, y^*) arrives, the GP surrogate model will be updated by GP posterior $p(f_* = y^* \mid x^*, \mathbf{X}, f)$:

$$\begin{pmatrix} \mathbf{f} \\ f_* \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix}\right), \quad (11)$$

where $\mathbf{K}_* = \kappa(\mathbf{X}, x^*) \in \mathbb{R}^{N\times 1}$ and $\mathbf{K}_{**} = \kappa(x^*, x^*) \in \mathbb{R}^{1\times 1}$. In general, evaluating the GP posterior is computationally efficient and serves as the foundation for approximating a surrogate model to the objective function. The surrogate model aids in parameter tuning within a huge search space, helping identify configurations that are likely to result in better performances.

The *acquisition function* will determine the location of the next sampling x_{t+1}, where $x_{t+1} = argmax_x \mathbf{aq}(x \mid D_{1:t})$. In BO tuner we employ the expected improvement (EI) function [25] for $\mathbf{aq}$, which can be described as:

$$EI(x_{t+1}) = \mathbb{E}\left[\max\left(f(x_{t+1}) - f\left(x_t^+\right), 0\right)\right], \quad (12)$$

where x_t^+ is the location of best result in $D_{1:t}$ so far. To balance the exploitation and exploration, as the GP model is updated, function $\mathbb{E}$ is also adapted based on its mean and variance.

It should be noted that specific constraints can affect the exploration, as shown in Fig. 7. These constraints encompass factors like divisible relations for tiling and ensuring that tiling sizes are compatible with the buffer. Moreover, our BO tuner further incorporates the tiling size limit and (8) of double-buffering into the constraints, which could significantly improve the efficiency of the tuning process.

After determining the next sample x^*, the *Runtime Evaluation* module configures the GEMM kernel by new parameters and subsequently measures the average floating-point performance over five rounds, denoted as y^*. Subsequently, the GP model will be refined based on the new dataset (x^*, y^*), leading to the next round of iterations. Based on this framework, we have built an efficient, lightweight, and reliable tuner for OpenCL-based generic kernels.

V. Experimental Results

A. Experimental Setup

To demonstrate the effectiveness of our GEMM, all tests run across various typical OpenCL devices listed in Table II. *Jingjia Microelectronics* is an emerging mobile GPU maker, and the JM9 series is the latest generation of products supporting OpenCL 2.0. *AMD RX550* belongs to the Polaris series and offers comparable theoretical peak performance to JM9230. *NVIDIA A100* is a powerful accelerator card that can be used as a representative of NVIDIA's high-performance GPUs. In addition to GPUs, our performance evaluations also run on the Intel CPU and the FPGA Emulation Platform to assess the impact of each optimization strategy. The benchmarks involve a warm-up round for devices, followed by 10 repeated runs to count the final performance results. First, we primarily focus on presenting the optimization effects of each strategy on FPGA in Section V-B. In Section V-C, the optimized GEMM kernel implementation will be compared against:

1) *CLBLAS v2.12 [8]:* The first CL-based BLAS library, capable of auto-generating and enqueuing optimized OpenCL kernels for three levels of BLAS routines.
2) *CLBlast v1.6.1 [9]:* A high-performance and parameterized OpenCL BLAS library that allows users to tune parameter configurations for different devices.
3) *NVIDIA cuBLAS 11.7 [6]:* An integral component of the CUDA runtime environment, which is widely regarded as the gold standard on NVIDIA GPUs.
4) *Intel MKL 23.2.0 [5]:* A highly optimized library for mathematical computations, designed to maximize performance on the Intel architectures.

TABLE II
Overview of the Tested OpenCL Platforms

Vendor and device name	archi-tecture	library and SDK	Peak FLOPS
Jingjia Micro. JM9230	JM9 Series	OpenCL 2.0 V1.2	1228.8G
AMD Radeon RX550	Polaris	OpenCL 1.1 Mesa	1211.4G
NVIDIA A100-40GB	Ampere	CUDA 11.3	19.5T
Intel FPGA Emulation Platform for OpenCL	Xeon 5120	OpenCL 1.2	1230.2G

B. Ablation Study of Optimization Effect

We use the *Intel FPGA Emulation Platform for OpenCL* on Intel Xeon(R) Gold 5120 CPU. Table II shows the theoretical single-precision peak performance for these OpenCL platforms. The benchmark is the latest CLBlast, which currently offers superior performance across OpenCL devices. Subsequently, each optimization strategy is activated to assess its influence on the performance for different matrix sizes.

Fig. 8 illustrates the effectiveness of our double-buffer pipelining and fine-grained prefetching strategies for the benchmark GEMM kernel. Here we compare the best achievable performance for each kernel to show improvements, without incorporating the auto-tuning component. The adoption of fine-grained prefetching achieves an average speedup of 6.5%, whereas a three-level pipeline implementation facilitates an average speedup of 19.3%. For matrices of size less than 512, the effectiveness of pipelining is typically reduced, largely because size constraints hinder the full exploitation of double-buffer advantages. Significantly, the concurrent implementation of both optimizations usually results in greater performance gains (32.89% on average), surpassing the sum of their individual contributions. This effect is attributed to the fine-grained control over register units, which facilitates a more rational exploration of tiling parameters for the double-buffer pipeline, leading to markedly improved performance.

TABLE III
BEST PARAMETERS TUNED BY BO TUNER AT A FIXED SIZE OF 1024 FOR DIFFERENT PRECISIONS

Parameters	JM9230		AMD RX550		Intel Xeon 5120		NVIDIA A100	
	half	float	half	float	float	double	float	double
M_{wg}, N_{wg}, K_{wg}	128, 128, 32	128, 128, 16	64, 128, 16	32, 64, 32	128, 64, 16	64, 32, 16	32, 64, 16	64, 32, 16
M_{dimC}, N_{dimC}	16, 16	16, 16	8, 16	8, 8	8, 8	4, 8	8, 16	32, 8
M_{dimA}, N_{dimB}	16, 16	16, 16	8, 32	8, 8	16, 8	16, 4	8, 16	32, 8
$STRM, STRN$	0, 0	1, 1	0, 0	0, 0	1, 0	1, 0	1, 1	1, 1
VWM, VWN	8, 8	4, 4	8, 4	4, 2	8, 8	4, 4	4, 4	2, 2

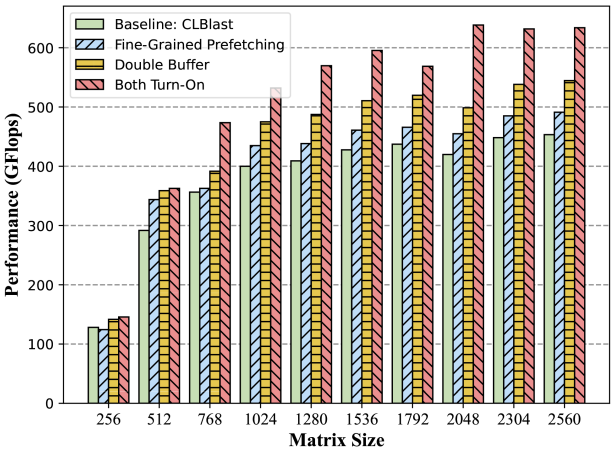


Fig. 8. The Optimization effect of GEMM kernels on Intel FPGA Emulation Platform for OpenCL. All kernels are tuned at their best performance.

C. Performance Results and Analysis

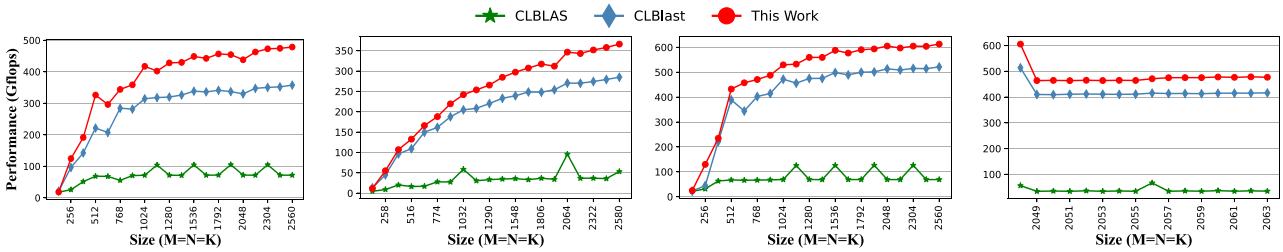
1) Performance Portability: To demonstrate the portable high-performance of our GEMM kernel, we conducted the performance comparisons of different precisions across four distinct devices. On each device, our kernel is tuned by BO tuner at a given size M=N=K=1024, which is the same as the CLBlast tuning process. Other libraries employ their default fixed-tuning strategies for configurations. Table III shows the tuning results of our GEMM kernel across these platforms.

The performance comparisons are shown in Figs. 9, 10, 11 and 12, covering three representative scenarios: (1) matrix sizes are multiples of 128, (2) matrix sizes are multiples of $(128 + 1)$, and (3) matrix sizes around 2048. Kernel performance at ideal sizes, without preprocessing overhead, is demonstrated by multiples of 128. In contrast, multiples of irregular odd numbers highlight the performance involving preprocessing. Matrix sizes around 2048 represent a common use case for the GEMM kernel. The analysis of these results yields several critical conclusions:

- *Jingjia Micro 9230:* For both precisions, our fine-grained GEMM kernel shows superior performance over CLBLAS and CLBlast. CLBLAS exhibits poor performance on JM9230, primarily due to its AMD-centric targeting. CLBlast achieves commendable performance yet is surpassed by our kernel. On average, our kernel obtains performance improvement of 30.82% and 16.76% in single- and half-precision, respectively.
- *AMD RX550:* For single precision, our GEMM kernel outperforms other libraries, achieving the best performance overall. In half-precision experiments, CLBLAS shows an advantage for small-scale matrices but struggles with irregular matrix sizes. On average, our kernel can achieve a speedup of 12.39% and 21.93% compared to CLBlast in single- and half-precision, respectively.
- *NVIDIA A100:* NVIDIA's cuBLAS remains superior to all OpenCL-based libraries in most cases, because of CUDA's highly optimized intrinsics control. When matrix sizes are multiples of 128, cuBLAS significantly outperforms other libraries, while our kernel achieves the second-best performance. Compared with CLBlast, our kernel gets a performance gain of 17.78% and 13.76% in double- and single-precision. When dealing with sizes around 2048, our kernel achieves performance comparable to cuBLAS, reaching 89.93% on average.
- *Intel 5120 CPU:* Intel oneMKL consistently outperforms OpenCL-based GEMM implementations on the CPU, with our kernel achieving up to 64.06% of MKL's performance in the best-case scenario. Nevertheless, compared to the optimal OpenCL-based library CLBlast, our GEMM kernel achieves an average speedup of 55.66% and 75.58% in double- and single-precision, respectively.

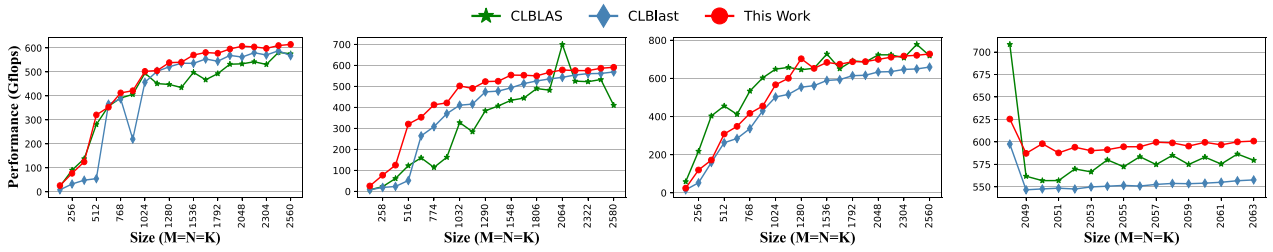
2) Performance Analysis: Note that there are performance fluctuations in all test routines on A100 GPU, we analyzed it by *NVIDIA Nsight*. The cuBLAS library employs different micro-kernels as the matrix size increases during the test, such as 64×32, 128×32, and 128×64, leading to noticeable performance jumps. In contrast, OpenCL libraries typically rely on a single parameterized kernel, and high performance can only be achieved when the matrix exceeds a specific scale. To eliminate the impact of compiler optimization between CUDA and OpenCL, we utilized *NVRTC* (NVIDIA Runtime Compilation) to recompile our kernel, ensuring a consistent runtime. As shown in Fig. 11, the results demonstrate that the runtime environment and compilation optimizations account for less than 5% of the performance difference, indicating that compilers are not the primary cause of the observed performance gap. Therefore, We attribute the performance disparity to hardware-specific optimizations in cuBLAS, such as PTX-level and assembly-level tuning. In contrast, OpenCL-based kernels are unable to fully exploit these hardware properties, which limits their performance potential.

A similar performance gap is observed on the Intel Xeon 5120, where MKL leverages highly optimized assembly code tailored to Intel CPUs, making it difficult for generic OpenCL kernels to achieve comparable performance. Table IV presents the profiling



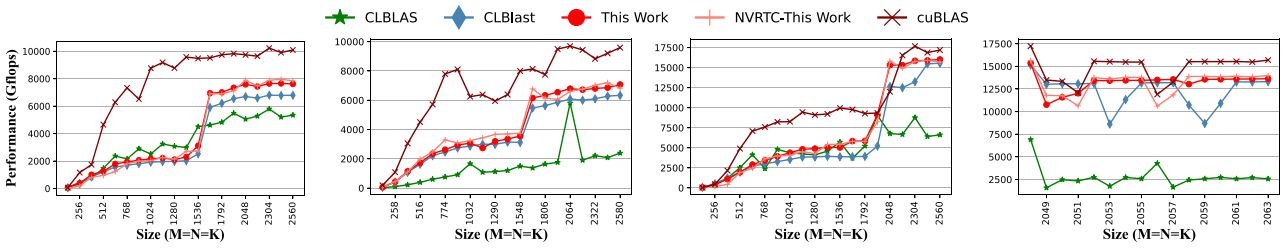
(a) SGEMM multiples of 128. (b) SGEMM multiples of (128+1). (c) HGEMM multiples of 128. (d) HGEMM sizes around 2048.

Fig. 9. Performance comparison of single- and half-precision GEMM against CLBLAS and CLBlast on Jingjia Micro 9230.



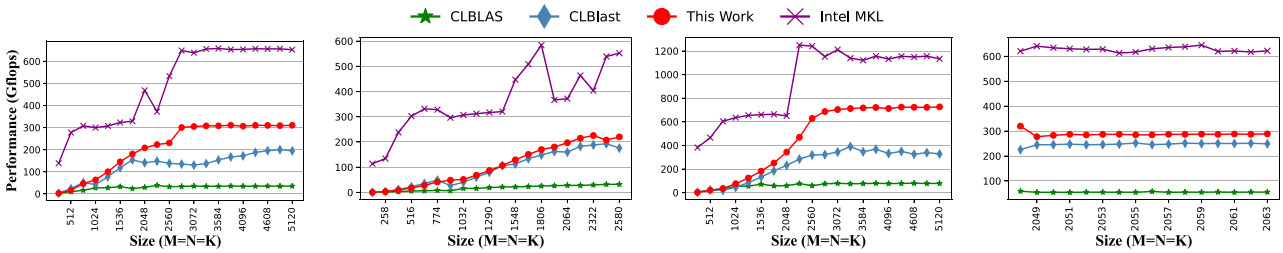
(a) SGEMM multiples of 128. (b) SGEMM multiples of (128+1). (c) HGEMM multiples of 128. (d) HGEMM sizes around 2048.

Fig. 10. Performance comparison of single- and half-precision GEMM against CLBLAS and CLBlast on AMD RX550.



(a) DGEMM multiples of 128. (b) DGEMM multiples of (128+1). (c) SGEMM multiples of 128. (d) SGEMM sizes around 2048.

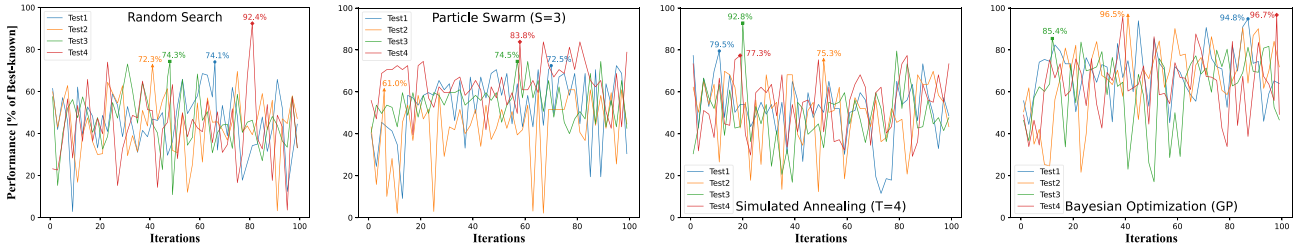
Fig. 11. Performance comparison of double- and single-precision GEMM against CLBLAS, CLBlast and cuBLAS on NVIDIA A100.



(a) DGEMM multiples of 256. (b) DGEMM multiples of (128+1). (c) SGEMM multiples of 256. (d) SGEMM sizes around 2048.

Fig. 12. Performance comparison of double- and single-precision GEMM against CLBLAS, CLBlast and MKL on Intel Xeon 5120 CPU.

results of CLBlast, cuBLAS, MKL, and our GEMM kernel. On the NVIDIA A100, the SM throughput calculates the percentage of cycles where the FMA pipe is active, and our kernel can reach a peak of 91.81%. Furthermore, the L2 Cache throughput is $1.83\times$ higher than CLBlast, demonstrating the efficiency of our double-buffer optimization. On the Intel Xeon 5120, the improvements are even more pronounced. The core utilization and IPC reach $1.29\times$ and $4.06\times$ of CLBlast respectively, while the number of L2 misses is reduced by 79.31%. cuBLAS and MKL, as the optimal kernels on their respective platforms, both



(a) Random Search Process (b) Particle Swarm Optimization (c) Simulated Annealing Process (d) Bayesian Optimization with GPs

Fig. 13. Comparison of tuning processes of different algorithms for SGEMM performance on NVIDIA A100.

TABLE IV
HARDWARE PROFILING RESULTS FOR 4096-SIZED SGEMM KERNELS

NVIDIA A100 GPU profiled by NVIDIA Nsight Compute			
Metrics	CLBlast	Our Kernel	cuBLAS
Compute (SM) Throughput	87.13%	91.81%	97.96%
Warp Instructions Executed	52.71%	60.19%	56.80%
L2 Cache Throughput	17.80%	32.55%	43.12%
Intel Xeon 5120 CPU profiled by Intel PCM			
Metrics	CLBlast	Our Kernel	MKL
Core Utilization	56.88%	73.83%	98.56%
IPC (Instructions Per Cycle)	0.31	1.26	1.67
L2 MPI (Misses Per Instruction)	0.0029	0.0006	0.0001

achieve nearly 100% of compute unit utilization with minimal L2 cache misses.

In conclusion, our optimized GEMM kernel demonstrates high-performance portability, attributed to the effectiveness of double-buffer pipelining and fine-grained prefetching of private memory. These optimizations enhance core and memory utilization, resulting in substantial performance improvements. Although it may not match the performance of kernels fine-tuned for specific hardware, our approach highlights the benefits of generalization. Overall, our optimized GEMM kernel achieves average speedups of $1.31\times$, $1.12\times$, $1.14\times$, and $1.76\times$ over CLBlast SGEMM on these devices, underscoring its portability and efficiency for heterogeneous platforms.

D. Bayesian Optimization Tuner Study

In this section, we compare our BO tuner against other widely used auto-tuning methods, including Random search, Particle Swarm Optimization (PSO) with settings $g = 0.4, l = 0, r = 0.4$, and Simulated Annealing (SA) with $T = 4$ in CLTune. For the matrices, we maintain fixed dimensions of 1024, generated with identical random seeds for consistency.

Fig. 13 illustrates the search process over four rounds for each method, highlighting the best results in each round. For clarity, each round consists of 100 iterations, capturing fluctuations during searching. In Fig. 13(a), the random search strategy exhibits erratic behavior, achieving $74.1\%, 72.3\%, 74.3\%$, and 92.4% of the best attainable performance. The PSO method's performance, as shown in Fig. 13(b), demonstrates substantial fluctuations but exhibits a trend of gradual convergence to more favorable positions. The Simulated Annealing achieves

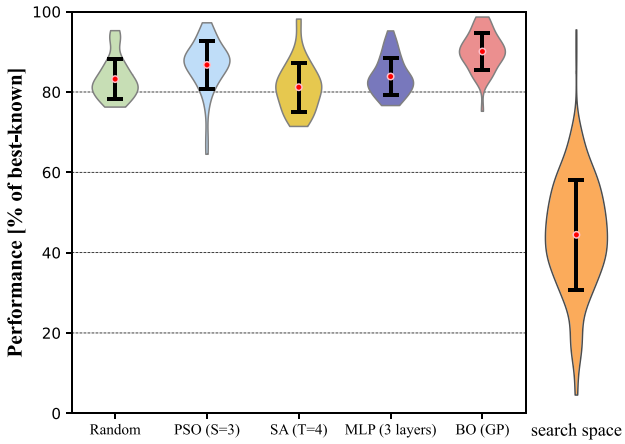


Fig. 14. The violin plot shows the statistical properties of 100 rounds of different tuning algorithms on A100. The sub-figure on the right shows the performance distribution across the search space ($160,000+$ configurations).

$79.5\%, 75.3\%, 92.8\%$, and 77.3% of the best performance. Our BO tuner, as illustrated in Fig. 13(d), shows more stable and progressive patterns throughout the iterations. It attains $94.8\%, 96.5\%, 85.4\%$, and 96.7% of the best-known performance in each round.

To minimize the impact of stochastic variables in each method, we executed each search method 100 times, performing 100 iterations in every round. Additionally, we compared the neural network tuning strategy, which employs a three-layer MLP to establish a mapping between parameters and performance, and subsequently predicts the results for unexplored configurations. The optimal performance for each round was recorded, and their distributions are presented in Fig. 14 by a violin plot. In Fig. 14, each violin plot comprises three elements: 1) a T-line indicating the standard deviation error bar, 2) a red point denoting the average searching result, and 3) a rotated density graph representing the data distribution.

The statistical results reveal unique distribution patterns for the performance distributions of various tuning methods. The random search and SA methods display a vase-like shape, with a majority clustered at the middle or lower performance levels. Note that due to the randomness of the training set partitioning, the neural network can be viewed as an optimized strategy for random search and exhibits a similar vase-like distribution. In

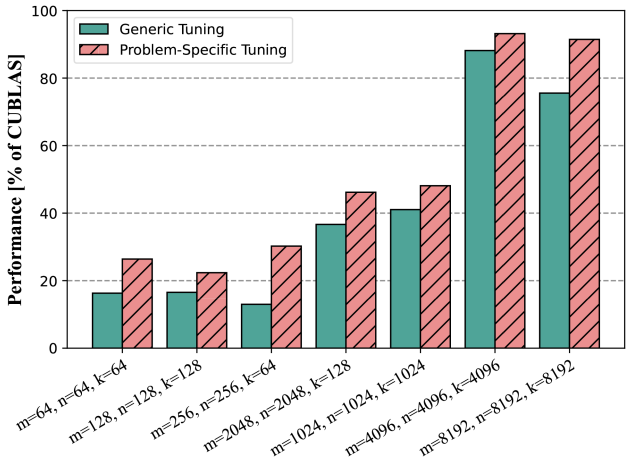


Fig. 15. The problem-specific tuning of our SGEMM compared to a fixed size (1024) tuning of CLBlast on NVIDIA A100.

contrast, the PSO and BO tuners display a butterfly-shaped distribution, indicating a higher proportion of rounds attaining better performance. Notably, the results of BO tuner are more concentrated compared to PSO, with all configurations surpassing 75% performance after 50 iterations. On average, the BO tuner attains a performance level of 90.08% on A100, while the other methods yield performances of $83.21\%, 86.75\%$, 81.19% and 83.85% respectively. Furthermore, the BO tuner demonstrates superior stability and reliability, evidenced by its low standard deviation of 4.66%, compared to $5.01\%, 6.17\%$, 6.06% and 4.62% for the other methods. On the Intel 5120 CPU, running 100 iterations of the BO tuner averages approximately 210 seconds, with a maximum memory overhead of 0.88 GB, without requiring extra storage. Such tuning overhead is deemed acceptable for both JM9230 and RX550 devices. In many cases, BO tuning takes only 30 to 50 iterations to find a near-optimal parameter configuration. In conclusion, our BO tuner can discover a configuration with 94.74% of the best-known performance in fewer iterations, with just a 10.35% time overhead compared to the default exhaustive or random methods.

E. Scalability and Extensibility

In practice, the BO tuner exhibits notable scalability and extensibility, evident in two aspects. The first aspect pertains to matrix size scalability, a crucial factor for deep-learning models, where kernel sizes are tailored to specific applications. Fig. 15 displays the performance achieved in general tuning (at a fixed size of 1024) and size-specific tuning compared to cuBLAS. The BO tuner's rapid convergence and robust statistical properties enable precise tuning of GEMM kernels in various sizes, with an average speedup of 42%. Moreover, the problem-specific tuning has the potential to extend to other frameworks beyond OpenCL. The second aspect is the BO tuner's extensibility. It is readily adaptable for tuning other arithmetic kernels involving multiple parameters. The process involves defining an objective function and establishing some constraints. We have successfully utilized the BO tuner for GEMV, original GEMM, and double-buffer GEMM kernels, thereby demonstrating its extensibility and potential for tuning various critical kernels.

VI. Related Work

As for the OpenCL-based BLAS study, AMD's CLBLAS [8] is the first open-source implementation. However, the performance of CLBLAS GEMM is not portable for less common devices, and its kernels are auto-generated, leading to unreadable and unmaintainable. NVIDIA's cuBLAS GEMM [6] is close-sourced and must rely on CUDA, which is not generic for other devices. CLBlast [9] provides performance portability and general auto-tuning for kernels. This library and other works [10], [16], [26] have demonstrated that parameterized kernels can facilitate the efficiency of large-scale and complex applications. In both CPU and GPU architectures, double-buffer pipelining is a latency-hiding technique that is effective [15], [27], [28], [29], but no OpenCL kernels consider introducing it. Moreover, balancing instruction-level parallelism and device occupancy on both AMD and NVIDIA GPUs has demonstrated acceleration potential in recent studies [19], [20]. Some recent GEMM implementations [30], [31], [32] consider utilizing FPGAs to reconfigure circuits and redesign pipelined architectures for acceleration. Sun [33] finds better HLS directives by a multi-objective nonlinear optimization algorithm. However, those hardware-level and directive-level improvements cannot be utilized in OpenCL. Therefore we integrate double-buffer pipelining and fine-grained register control techniques into parameterized GEMM, which is beneficial for exploiting better performance.

As for auto-tuning, the exhaustive search strategy, while theoretically comprehensive, suffers from the curse of dimensionality [34], leading to an impractical number of configuration options to explore. To achieve efficient auto-tuning of parameters, researchers usually employed random search or heuristic algorithms during their search process [10], [35], [36], [37], [38]. Recently some machine learning or deep learning technologies were applied for performance modeling [39], [40], [41], [42]. Some research [43], [44] has demonstrated that profiling machine learning models for an HPC application can be highly resource-intensive. For instance, modeling the GalaxSee kernel incurs 1521.2% time overhead and demands 5164 KB of storage per round for 357 features. Consequently, such intricate modeling overhead is often unnecessary and unaffordable for lightweight OpenCL devices. Some recent tuners leverage Bayesian Optimization for auto-tuning configurations. GPTune [45] focuses on tuning parallel tasks in distributed-memory systems. PrBO [46] augments BO searching with expert knowledge, but it's infeasible to find an expert who is well-versed in all types of OpenCL devices. B_{LISS}[47] sets multiple BO models to interact with runtime environments and applications, which is slightly redundant for one single kernel tuning in this work. Willemsen's work [48] also used BO to tune various kernels on GPU, but it neglected some extra constraints on the parameter space based on the tiling algorithm. In contrast, our BO tuner is much simpler and more efficient compared with those "enormous" models, while also being able to discover

near-optimal parameter configuration without training and expert knowledge guide.

VII. Conclusion & Future Work

In summary, this study conducted further optimizations of the GEMM kernel on CLBlast, employing double-buffer pipelining and fine-grained prefetching strategies. Experimental results revealed substantial performance improvements across four different OpenCL devices. Furthermore, we introduced a Bayesian Optimization-based tuner (BO tuner) for automated tuning of parameter configurations. Compared to state-of-the-art tuning methods, the BO tuner demonstrates superior performance in both exploration and exploitation, attaining higher efficiency without relying on a complex model. The performance jump points on A100 prompted us to consider designing multiple microkernel implementations like CUDA in the future. The structured and efficient search methodology of the BO tuner not only facilitates future problem-specific tuning but also extends its applicability to parameterized kernels beyond the OpenCL framework. In addition, the double-buffer technology is expected to extend to other general mathematical kernels like SpGEMM and this work has been released to https://github.com/lsl036/CL-DB-GEMM.

References

[1] B. Kågström, P. Ling, and C. Van Loan, "GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, 1998.

[2] K. Goto, "Gotoblas," 2007. [Online]. Available: http://www.tacc.utexas.edu/resources/software/

[3] F. G. Van Zee and R. A. Van De Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 1–33, 2015.

[4] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 1–12.

[5] E. Wang et al., "Intel math kernel library," in *High-Performance Computing on the Intel Xeon Phi™: How to Fully Exploit MIC Architectures*. Berlin, Germany: Springer, 2014, pp. 167–188.

[6] N. Corporation, "cuBLAS: A GPU-accelerated library for linear algebra operations," 2023, Accessed: Dec. 25, 2023. [Online]. Available: https://developer.nvidia.com/cublas

[7] A. Corporation, "rocBLAS documentation," 2023, Accessed: Dec. 25, 2023. [Online]. Available: https://rocblas.readthedocs.io/en/rocm-5.7.1/

[8] clMathLibraries, "clblas repository," 2017, Accessed: Dec. 25, 2023. [Online]. Available: https://github.com/clMathLibraries/clBLAS

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

[10] C. Nugteren and V. Codreanu, "CLTune: A generic auto-tuner for OpenCL kernels," in *Proc. IEEE 9th Int. Symp. Embedded Multicore/Many-Core Syst.-On-Chip*, 2015, pp. 195–202.

[11] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing With OpenCL 2.0*. San Mateo, CA, USA: Morgan Kaufmann, 2015.

[12] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 341–352.

[13] B.-Y. Su and K. Keutzer, "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 353–364.

[14] N. Contini et al., "Enabling reconfigurable HPC through MPI-based inter-FPGA communication," in *Proc. 37th Int. Conf. Supercomputing*, 2023, pp. 477–487.

[15] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the fermi GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2045–2057, Nov. 2012.

[16] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs," in *Proc. SC Companion: High Perform. Comput., Netw. Storage Anal.*, 2012, pp. 396–405.

[17] Y. Liang, Z. Cui, K. Rupnow, and D. Chen, "Register and thread structure optimization for GPUs," in *Proc. 18th Asia South Pacific Des. Automat. Conf.*, 2013, pp. 461–466.

[18] X. Xie et al., "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 395–406.

[19] G. Shobaki, A. Kerbow, and S. Mekhanoshin, "Optimizing occupancy and ILP on the GPU using a combinatorial approach," in *Proc. 18th ACM/IEEE Int. Symp. Code Gener. Optim.*, 2020, pp. 133–144.

[20] J. Li, H. Ye, S. Tian, X. Li, and J. Zhang, "A fine-grained prefetching scheme for DGEMM kernels on GPU with auto-tuning compatibility," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 863–874.

[21] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," 2010, *arXiv:1012.2599*.

[22] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," in *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.

[23] J. Mockus, "Application of Bayesian approach to numerical methods of global and stochastic optimization," *J. Glob. Optim.*, vol. 4, pp. 347–365, 1994.

[24] C. E. Rasmussen et al., *Gaussian Processes for Machine Learning*, vol. 1. Berlin, Germany: Springer, 2006.

[25] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. Glob. Optim.*, vol. 13, pp. 455–492, 1998.

[26] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, T. Grasser, and A. Jüngel, "Performance portability study of linear algebra kernels in OpenCL," in *Proc. Int. Workshop OpenCL 2013 2014*, 2014, pp. 1–11.

[27] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Comput.*, vol. 38, no. 8, pp. 391–407, 2012.

[28] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, 2019, pp. 229–241.

[29] H. Wang, W. Yang, R. Hu, R. Ouyang, K. Li, and K. Li, "IAP-SpTV: An input-aware adaptive pipeline SpTV via GCN on CPU-GPU," *J. Parallel Distrib. Comput.*, vol. 181, 2023, Art. no. 104741.

[30] M. Meyer, T. Kenter, and C. Plessl, "Evaluating FPGA accelerator performance with a parameterized OpenCL adaptation of selected benchmarks of the hpcchallenge benchmark suite," in *Proc. IEEE/ACM Int. Workshop Heterogeneous High- Perform. Reconfigurable Comput.*, 2020, pp. 10–18.

[31] J. Liu, A.-A. Kafi, X. Shen, and H. Zhou, "MKPipe: A compiler framework for optimizing multi-kernel workloads in OpenCL for FPGA," in *Proc. 34th ACM Int. Conf. Supercomputing*, 2020, pp. 1–12.

[32] P. Colangelo, S. Sengupta, and M. Margala, "Sparse persistent GEMM accelerator using OpenCL for intel FPGAs," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2020, pp. 1–6.

[33] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, "Correlated multi-objective multi-fidelity optimization for HLS directives design," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 27, no. 4, pp. 1–27, 2022.

[34] R. Bellman et al., *Dynamic Programming and Modern Control Theory*, vol. 81. Princeton, NJ, USA: Citeseer, 1965.

[35] J. Dongarra and V. Eijkhout, "Self-adapting numerical software and automatic tuning of heuristics," in *Proc. Int. Conf. Comput. Sci.*, Melbourne, Australia and St Petersburg, Russia, Springer, 2003, pp. 759–767.

[36] H. H. Hoos, "Automated algorithm configuration and parameter tuning," in *Autonomous Search*. Berlin, Germany: Springer, 2012, pp. 37–71.

[37] J. Ansel et al., "Opentuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Architectures Compilation*, 2014, pp. 303–316.

[38] C. Huang, Y. Li, and X. Yao, "A survey of automatic parameter tuning methods for metaheuristics," *IEEE Trans. Evol. Comput.*, vol. 24, no. 2, pp. 201–216, Apr. 2020.

[39] Q. Sun et al., "csTuner: Scalable auto-tuning framework for complex stencil computation on GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2021, pp. 192–203.

[40] X. Zeng, S. Zhang, C. Ren, and T. Shao, "Physics informed neural networks for electric field distribution characteristics analysis," *J. Phys. D: Appl. Phys.*, vol. 56, no. 16, 2023, Art. no. 165202.
[41] X. He et al., "Enabling energy-efficient DNN training on hybrid GPU-FPGA accelerators," in *Proc. ACM Int. Conf. Supercomputing*, 2021, pp. 227–241.
[42] D. Boehme, K. Huck, S. Kale, V. Kale, and V. Surjadidjaja, "Sophisticated tools for performance analysis and auto-tuning of performance portable parallel programming,".
[43] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen, "Automated performance modeling of HPC applications using machine learning," *IEEE Trans. Comput.*, vol. 69, no. 5, pp. 749–763, May 2020.
[44] D. Yokelson, M. R. J. Charest, and Y. W. Li, "HPC application performance prediction with machine learning on new architectures," in *Proc. Perform. EngineeRing, Modelling, Anal., VisualizatiOn Strategy*, 2023, pp. 1–8.
[45] Y. Liu et al., "GPTune: Multitask learning for autotuning exascale applications," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 234–246.
[46] A. Souza, L. Nardi, L. Oliveira, K. Olukotun, M. Lindauer, and F. Hutter, "Prior-guided Bayesian optimization," 2020.
[47] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Bliss: Auto-tuning complex applications using a pool of diverse lightweight learning models," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2021, pp. 1280–1295.
[48] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven, "Bayesian optimization for auto-tuning GPU kernels," in *Proc. Int. Workshop Perform. Model., Benchmarking Simul. High Perform. Comput. Syst.*, 2021, pp. 106–117.

Shengle Lin received the PhD degree in computer science and technology from the College of Information Science and Electronic Engineering, Hunan University, Changsha, China, in 2024. From 2023 to 2024, he was a joint PhD student with the Agency for Science, Technology and Research (A∗STAR), Singapore. He is currently a Postdoctoral Fellow with Hunan University. His research interests include high-performance computing, parallel computing, numerical computation, and artificial intelligence.

Guoqing Xiao (Member, IEEE) received the PhD degree in computer science and technology from the College of Computer Science and Electronic Engineering (CSEE), Hunan University (HNU), China, in 2017. He is currently a Professor with the HNU. He worked as a Postdoctoral Research Fellow with the Data Systems Group of the David R. Cheriton School of Computer Science, University of Waterloo, Canada, during 2017 to 2019. His main research interests are on high-performance computing and AI computing. He has published more than 50 papers in peer-reviewed international journals and conferences, such as ISCA, DAC, ICDE, IEEE TRANSACTIONS ON COMPUTERS/IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS/TKDE/TII, *ACM Transactions on Parallel Computing/ACM Transactions on Recommender Systems/ACM Computing Surveys*, etc.

Haotian Wang received the BS degree from the School of Information Engineering, Nanchang University, China, in 2018, and the PhD degree in computer science from Hunan University, China, in 2023. He is currently working as a Postdoctoral Fellow with Hunan University, China. He previously completed a one-year joint PhD program from Nanyang Technological University, and he is an ACM member. His research interests include parallel computing, tensor compilation, and artificial intelligence.

Wangdong Yang received the PhD degree in computer science from Hunan University, China, and the MS degree in computer science from Central South 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 the IEEE INTERNET OF THINGS JOURNAL.

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a Visiting Scholar with the University of Illinois with Urbana-Champaign (2004–2005). 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 with National Supercomputing Center in Changsha. His major research interests include parallel and distributed processing, high-performance computing, and Big Data management. He has published more than 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 the IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING.

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 current 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 1130 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. Since 2020, he has been among the world's top few most influential scientists in parallel and distributed computing regarding single-year impact (ranked #2) and career-long impact (ranked #4) based on a composite indicator of the Scopus citation database. He is listed in Scilit Top Cited Scholars (2023–2024) and is among the top 0.02% out of over 20 million scholars worldwide based on top-cited publications. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He was a recipient of the 2022–2023 International Science and Technology Cooperation Award and the 2023 Xiaoxiang Friendship Award of Hunan Province, China. He is a Member of the SUNY Distinguished Academy. He is an AAAS Fellow, an AAIA Fellow, an ACIS Fellow, and an AIIA Fellow. He is a Member of the European Academy of Sciences and Arts. He is a Member of Academia Europaea (Academician of the Academy of Europe).