moTuner: A Compiler-based Auto-tuning Approach for Mixed-precision Operators

Zewei Mo
mozw5@mail2.sysu.edu.cn
Sun Yat-Sen University
Guangzhou, China

Zejia Lin*
linzj@mail.nwpu.edu.cn
Northwestern
Polytechnical University
Xi'an, China

Xianwei Zhang
zhangxw79@mail.sysu.edu.cn
Sun Yat-Sen University
Guangzhou, China

Yutong Lu
luyutong@mail.sysu.edu.cn
Sun Yat-Sen University
Guangzhou, China

ABSTRACT

Arithmetic operators are now used in a wide spectrum of domains, including artificial intelligence, data analytics and scientific computing. Meanwhile, specialized hardware components to enable low-precision computing are increasingly deployed in GPUs and accelerators. Whereas promising to boost performance, accelerating the operators on the hardware necessitates manually tuning the mixed-precision knobs to balance the performance and accuracy, which can be extremely challenging in real practices.

To address the issue, we present *moTuner* , an automatic framework for efficiently tuning mixed-precision operators. *moTuner* works on compiler-level to automatically enable the mixed-precision computation, without involving any manual modifications of source code and/or the operator library, thus significantly alleviating the programming burden. Owing to be implemented in compilation phase, *moTuner* can be more widely applicable with lessened efforts on the libraries. Further, *moTuner* adopts optimized search strategy in tuning to effectively narrow down the configuration space. The evaluations on GEMM operators and real applications demonstrate that *moTuner* achieves performance improvement up to 3.13x and 1.15x respectively, while guaranteeing considerably high accuracy.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

KEYWORDS

mixed-precision operator, auto-tuning, compiler, performance and accuracy, GPUs

ACM Reference Format:
Zewei Mo, Zejia Lin, Xianwei Zhang, and Yutong Lu. 2022. moTuner: A Compiler-based Auto-tuning Approach for Mixed-precision Operators. In *19th ACM International Conference on Computing Frontiers (CF'22), May 17–19, 2022, Torino, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3528416.3530231

1 INTRODUCTION

The past decade witnesses the ubiquitous development of compute-intensive machine learning, data processing workloads and scientific computing applications, and the popularity of hardware computing devices including GPUs and domain-specific accelerators. The continuous demand of compute horsepower drives the emergence of dedicated operators and low-precision computing to raise the performance levels on resource-constrained systems. Accordingly, dense arithmetic operations like convolution and general matrix multiplication (GEMM) have been greatly invested, leading to various pre-built libraries, such as cuBLAS [33], rocBLAS [2] and MKL [19]. As the operators are overwhelmingly adopted, different types of hardware devices with specialized units have been introduced to efficiently support the operator computations. Recent GPUs are adding extra units like Tensor Cores [34] and Matrix Cores [1], and accelerators including Google TPUs [22] and Cambricon's MLUs [6] are used in multiple domains. To further improve the performance, low-precision computing is being actively explored from both hardware and software perspectives. On hardware, relaxed data types like 16-bit floating point (FP16) and 8-bit integer (INT8) are well supported in almost all recent GPUs and accelerators; on software, quantization and precision refinement [28, 41]were proposed and utilized to support mixed-precision computations.

Naturally, the software and hardware techniques should be combined to unleash the full potential of mixed-precision computing. As such, the application programmers are required to scrutinize the source code to refactor the operator calls, which typically involves multiple parameters to tune the optimal mixed-precision settings with acceptable errors and higher performance. Besides, the aforementioned quantization and precision refinement are usually implemented inside the underlying libraries, thus necessitating the application programmers to revise the library and/or cooperate with the library providers. However, the low-level libraries are often highly-optimized and pre-defined by the hardware vendors, and can even be closed-source for proprietary reasons. Alternatively, the quantization and precision refinement steps can be migrated to application codes instead, thus leaving all changes to the end programmers, which inevitably increases the programming burden and further impedes the mixed-precision usages. For either option, programmers have to tune the knobs to tradeoff the speedup and output accuracy, which commonly incurs non-trivial manual efforts and expertise. Even worse, the prevalent use of heterogeneous computing platforms composed of CPUs, GPUs and accelerators frequently demands the portability of the mixed-precision procedures, rendering recurring engineering efforts.

[*] Work done while interning at Sun Yat-sen Unversity.

CF'22, May 17–19, 2022, Torino, Italy
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9338-6/22/05…$15.00
https://doi.org/10.1145/3528416.3530231

To reduce the programming burden when taking advantage of mixed-precision computation on large programs and decouple applications from unnecessary operator libraries, we devise *moTuner* to efficiently tune mixed-precision settings for operators in programs. It uses an optimized tuning strategy and user-defined error thresholds of the operator to tune mixed-precision settings for operators. When tuning, it applies the compiler to automatically analyze dependencies of operators and collects profiling information to help narrow down the search space of mixed-precision settings for operators. In the end it generates a program with higher performance and acceptable error. Also it targets on operator libraries of GPU and allows self-defined approaches of quantization and precision refinement to be embedded into compilers, which eases usage effort of various libraries when programming on heterogeneous systems.

In summary, the contributions of this paper are:

- we highlight the importance of removing the obstacles to mixed-precision uses, and then propose to consolidate the application- and library-level changes into the compiler, which makes the whole procedure transparent to end users.
- our designed compiler-based framework *moTuner* automatically enables mixed-precision by identifying and wrapping up the operator calls, and meanwhile adaptively and effectively selects the appropriate parameters to well support the desired mixed-precision computations.
- experimental evaluations for multiple GEMMs and real world applications show that the proposed framework is effectively speeding up the execution, while refraining the accuracy loss to low levels.

The rest of the paper is organized as follows. Section 2 introduces the background and motivation. Section 3 elaborates the proposed designs. Section 4 presents the experimental methodology, and Section 5 analyzes our experimental results. Section 6 discusses related work. Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce representative operators together with their mixed-precision computations, and then the basic compilation process. Next, we present the motivation of compiler-based automatic tuning to support mixed-precision operators.

2.1 Operators and Mixed-precision

As the requirement of machine learning and scientific computing applications continuously grows, the high computation demands drive GPUs to become the de facto computing platforms. And, increasing problem complexity and data volume motivate researches to optimize compute resources from both hardware and software perspectives, with examples of dedicated arithmetic units and operator libraries. In recent years, operators are dominating the whole execution of machine learning applications, and are even frequently involved in scientific computing workloads. Among the diverse operators, GEMM, $D = \alpha AB + \beta C$, is a representative one used in many neural networks like CNN [24], ResNet [15] and linear algebra applications like HPL [31], NTChem [35], Laghos [11].

The operator in these applications usually runs in a high-precision floating-point arithmetic (e.g., FP64, FP32). However, as the quantization methods and analysis of error control are getting mature, various efforts focus on taking advantages of low-precision (e.g., FP16, FP21 and INT8) arithmetic to speedup programs [4, 16, 21, 39] and many operators are implemented in mixed-precision like ReLU [40], GEMM [28] and winograd [3]. These low-precision representations with smaller size occupy less resources of GPU than high-precision ones, leading to a more limited value range, coarse-grained precision and higher performance. To further boost the operator efficiency, different levels of floating-point precision are now supported in recent GPUs (e.g., MI100 [1] and A100 [34]) to operate data with reduced sizes. Precision of mixed-precision operators' input and output can be represented like P_l/P_h where the former is the low precision of input and the latter is the high precision of output.

To achieve this, developers need to first transform high-precision data to low-precision, which is called quantization [12]. Then they utilize hardware and software with the capability of exploiting low-precision calculation like Tensor Core and VNNI [18], to produce results. In the end, de-quantization is performed to turn low-precision data produced back to high-precision. This method has already been widely employed in AI application due to their insensitivity to precision [29], greatly reducing dearth of resource and improving performance. But it's difficult to exploit it in scientific computing applications because they are strict with the precision loss.

2.2 Compilation Procedure

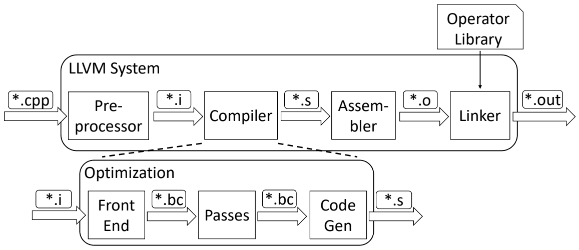
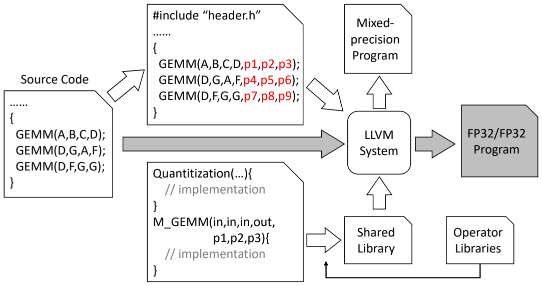


Figure 1: General compilation procedure.

Operator usage requires the assistance from compilers, which bridges the high-level codes to the underlying hardware. Figure 1 shows the compilation procedure of how C++ code using operators is turned into executable file through LLVM [27]. In the beginning, source code is handled by pre-processor to process the included files and macro definitions. Then the pre-processed file is delivered to the compiler. The procedure of compiler is typically partitioned into three phases: front end, middle end and back end. Front end is to check syntax correctness. Then, it turns the pre-processed code into intermmediate representation (IR) file (i.e., bitcode file). IR is the data structure and code used internally by compiler to represent source code, which eases difficulties for applying numerous general optimizations on it. The middle end uses multiple passes to analyze or optimize IR. Each of them performs one specific operation on IR. Next, the back end uses code generator to parse the optimized IR file and generates assembly based on the target hardware platform. Later the assembly file is turned into the re-allocated object file by the following assembler. In the end, linker links the object file with shared libraries including operator libraries provided by hardware vendors to produce the executable file. The shared library is produced by compiling with flag of `-shared` in advance and linked with target file in linker to provide implementations for invoked APIs of operators. Thus, the executable file is able to unleash the computing power of hardware by using operator libraries.

2.3 Motivation

2.3.1 Automate Mixed-precision. To bring mixed-precision operators into play when optimizing programs, developers need to manually modify some parts in Figure 1. In source code, developers are required to write the quantization kernel functions, choose which operators to be in mixed-precision and then replace them with mixed-precision ones. This can be particularly time consuming when working on large programs with thousands lines of code like HPL [31], CFD [32]. Further, to better trade off performance and precision in each operator without sacrificing much generality and convenience, developers implement their own mixed-precision refinement approach as dynamic shared objects based on existing operator libraries. The precision refinement approach utilizes mixed-precision operator from vendors to calculate a result and applies other calculations on it to accomplish the precision refinement job. But the shared objects may need to be rebuilt whenever the dependent operator libraries are updated, which costs a lot of time and efforts to maintain.



```
Source Code
......
{
  GEMM(A,B,C,D);
  GEMM(D,G,A,F);
  GEMM(D,F,G,G);
}
```

```
#include "header.h"
......
{
  GEMM(A,B,C,D,p1,p2,p3);
  GEMM(D,G,A,F,p4,p5,p6);
  GEMM(D,F,G,G,p7,p8,p9);
}
```

```
Quantitization(...){
  // implementation
}
M_GEMM(in,in,in,out,
        p1,p2,p3){
  // implementation
}
```

Figure 2: An example use of mixed-precision GEMM operator.

What's more, using such a framework requires parameter manual tuning to gain the best trade-off between performance and correctness. Figure 2 illustrates the procedure of applying mixed-precision operators in an application. Here `M_GEMM` is a mixed-precision GEMM framework with precision refinement, providing some parameters (e.g., three [28] in the red code) for developers to tune and strike a balance between performance and accuracy. In order to employ it, developers need to replace the original code with the one using `M_GEMM` and set parameters for them. Assuming that each `M_GEMM` has M setting combinations, the tuning space is $O(M^3)$ in this sample. But it can be as huge as $O(M^N)$ when the program has N GEMMs. So, to gain a mixed-precision program with acceptable error and the highest performance, developers have to pay huge efforts when carefully tuning the setting for each operator. Then developers are required to build their own operator libraries based on ones provided by vendors and link them with the compiled file of modified code, which increases the complexity of usage.

2.3.2 Control Error. Due to more limited range and coarse-grained precision that low-precision data has than high-precision one, there exists the emergency of re-designing existing algorithms to harness high performance mixed-precision hardware within acceptable error in multiple domains. This requires a lot of domain-specific knowledge and engineering efforts for developers. For instance, in linear algebra applications running on GPU like cholesky factorization [20] and HPL-AI [17], GMRES [38] are applied to accomplish the precision refinement. Multiple operators provided by GPU vendors captures most computation in them. So unlike what prior works [7, 13, 14, 25] focus on, most error in these applications are brought up or propagated by mixed-precision operators instead of instructions and accumulated in the final results of programs.

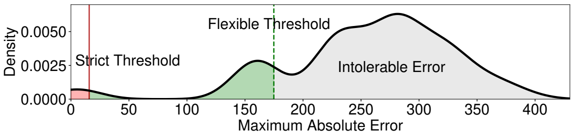


Figure 3: The maximum absolute error distribution of matrix G under all mixed-precision setting combinations.

For example, matrix `G` in Figure 2 is the output of the third mixed-precision `M_GEMM` which is implemented based on [28]. Figure 3 shows the maximum absolute error distribution of matrix `G` under all combinations of precision refinement settings. Density represents the occurrence frequency of an error, which equals to the frequency of settings producing this error. Assuming that a scientific computing program contains the source code shown in Figure 2 and takes matrix `G` as its output. As the maximum tolerable absolute error of its output is marked by the red line, qualified mixed-precision settings of these three `M_GEMM` is few. So it's hard to efficiently control the error of such a program by tuning these parameters. But if the program has a more flexible error threshold (the green line) for its output like neural network, more qualified settings are available which eases the burden of controlling the error. Because the setting tuning for programs containing mixed-precision operators varies according to different error requirements for different applications or input, we need an efficient tool to help controlling error in different scenarios.

3 COMPILER-BASED AUTO-TUNING

We introduce *moTuner* , a novel auto-tuning approach to well support mixed-precision operators to balance performance and accuracy. It analyzes dependency among multiple mixed-precision operators using compiler and then applies an optimized tuning strategy to efficiently determine the appropriate setting of each operator under a given error threshold. In the end, *moTuner* produces a program with mostly optimal performance.

3.1 Design Overview

Figure 4 presents the overall architecture of *moTuner* . The input is a linked IR file of all source code files. The output is a program with improved performance and constrained error.

The flow of *moTuner* can be divided into three parts: `Marker`, `Adjuster`, and `Finalizer`. As the forefront component, `Marker` extends compiler to tag each mixed-precision operator with an unique identifier and insert helper functions in IR which is to dump runtime information in execution. Then `Marker` turns the processed IR file into a new program. It will be executed once to dump information. Next, `Adjuster` comes into play to analyze dependency among executed operators in IR as well as tune mixed-precision setting for

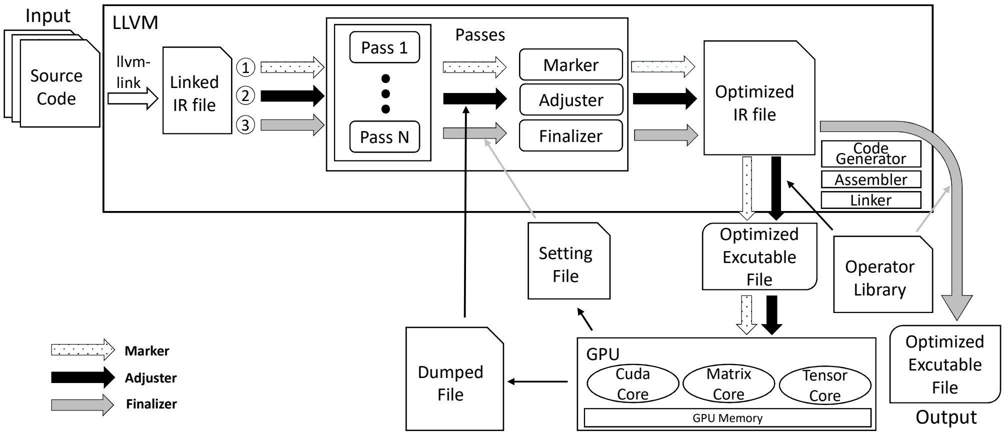
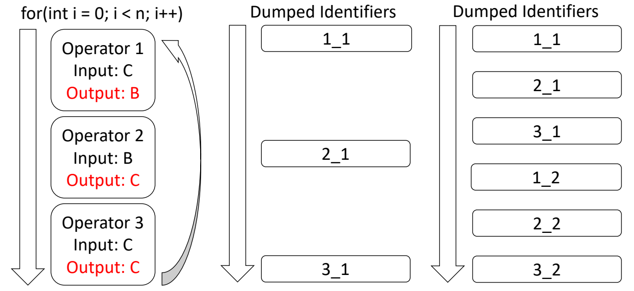


Figure 4: An overview of *moTuner*.

each operator. Allowing customizable error thresholds to control the error of each operator, it finds the mixed-precision setting with high performance and acceptable error for each operator. In the end, it generates a file describing recommended mixed-precision settings for executed operators. The third part is `Finalizer`, which directs the compiler to replace the original operators by mixed-precision ones with recommended settings.

3.1.1 Marker. In this part, `Marker` assigns an unique identifier to every execution of each operator in LLVM IR and generates a new program. The identifier for each operator includes two parts, ordered ID of its first execution and the counter of its execution. Then the new program is executed once to dump identifier and corresponding output under original precision.



(a) A loop with 3 operators. (b) Identifiers when iteration runs once. (c) Identifiers when iteration runs twice.

Figure 5: A loop with three operators and their identifiers in different iteration times.

Here we give an example to examine the identifier. Assuming that the loop body in Figure 5(a) is executed twice, Figure 5(c) shows dumped identifiers of each execution of every operator. Two dumped outputs of operator 1 will have the following identifiers: 1_1 and 1_2. The first number is the ordered ID and the second one is the counter. Considering an operator may accept a different input every time, error produced in every execution can differ a lot. *moTuner* uses it to identify the output of each operator's every execution and obtain every output error of each execution.

3.1.2 Adjuster. In this phase, the original linked IR file and data dumped before are taken as input. Instead of trying all combinations for operators' settings, we design an optimized tuning strategy using dumped output and identifiers of each operator under original precision. It assists *moTuner* to efficiently narrow down the search space and find the appropriate mixed-precision settings for operators. The tuning strategy consists of two main parts. The first one is the related operators analysis and the second one is the optimized adjustment. Details of these two will be discussed in Section 3.2. In the end, `Adjuster` generates a file describing recommended mixed-precision settings for all operators. All mixed-precision settings are guaranteed to satisfy given error threshold and not degrade the performance.

3.1.3 Finalizer. In this part, `Finalizer` takes the setting file generated by `Adjuster` as input and replaces executed operators by mixed-precision ones with corresponding mixed-precision settings in LLVM IR. Then it turns the optimized IR file into an executable file for developers.

3.2 Tuning Optimizations

3.2.1 Related Operators Analysis. Related operators of one are those directly or indirectly propagate error to its outputs (including itself). It means that one's related operators (except itself) should be executed before it and have data dependency with its input. Related operators help to identify which mixed-precision settings of operators should be upgraded if one operator raises unacceptable error even in original precision. To obtain one's related operators in adjuster, we leverage the identifiers dumped in the `Marker` phase. Once an identifier of operator *a* occurs before one of operator *b* in the dumped list and there exists data dependency between *a* and *b* in static code analysis, *a* is considered as a related operator of *b*. Here we assume that LLVM already constructs data dependency graph of values for us.

Figure 5(a) shows a loop body consisting of three operators. In static code analysis, there exists data dependency between any two of them. Assuming that the body of loop runs only once, we can get the identifier list in Figure 5(b) after `Marker` phase. As the directly related one of operator 3, operator 2 produces error in calculation

Table 1: Variable definition.

Notation	Definition
O	Ordered ID set of run operators
T_i	Cost time list of i-th operator under all settings
E_i	Error list of i-th operator under all settings
ET	Error list of each tuned operator in the last run
S	Candidate mixed-precision settings of i-th operator
UE	User defined error threshold
NID	ID of next operator to tune
lv	Level number of mixed-precision settings

and propagates to it through the value C. As the indirectly related one of it, operator 1 first propagates error to operator 2 through the value B. Then after the calculation of operator 2, the error is brought to operator 3 via the value C again. This helps *moTuner* to determine that operator 1 doesn't rely on operator 3 even if data dependency exists between them in static code analysis. But when the body of loop is executed twice, Figure 5(c) shows the corresponding dumped identifier list. Identifier of operator 3's first run occurs before the one of operator 1's second run. It means that error produced by operator 3 in the first run will be propagated to operator 1 in the second run through C, which makes operator 3 related to operator 1.

3.2.2 Optimized Adjustment. We devise the optimized adjustment strategy to track accumulation of error following its propagation path which is the execution path of operators, and adjust the mixed-precision settings as soon as an error larger than the threshold occurs. This greatly reduces the search space of tuning mixed-precision settings for operators. Table 1 introduces the notations we will use later and Algorithm 1 details the optimized tuning approach.

To achieve this approach, we need to classify mixed-precision settings into lv levels at first. The higher level a setting gets, the higher accuracy and performance penalty it produces. For example, mixed-precision settings consisting of INT8/FP32, FP16/FP32 and FP32/FP32 can be levelized into three levels, where INT8/FP32 is the lowest level and the FP32/FP32 is the highest one. Then line 5-9 in the Algorithm 1 find a mixed-precision setting for an operator with the highest performance and acceptable error. If there doesn't exist such a setting, it means that error produced by related operators is too much and this operator cannot produce a qualified result even in FP32 precision. As a result, line 12 upgrades settings of related operators to next higher level. To verify whether its new error is lower than given threshold, we need extra runs to dump the error of these operators. Number of extra run equals to the level number minus one, which guarantees that we are able to generate qualified settings for related operators.

Here we provide an example. We assume that the body of loop in Figure 5(a) only runs once and operators 1, 2 have been tuned to be INT8/FP32 after first three adjusting procedures like Figure 6. Also, the performance and error of operator 3 with different settings are dumped. Then in the fourth one, dumped information of operator 3 is taken as input. If *moTuner* detects that operator 3 cannot satisfy accuracy requirement under all settings, *moTuner* will turn operators 1, 2 and 3 into FP16/FP32 mode. Then the new program will be executed to validate whether the upgraded setting is qualified. If not, their settings will be upgraded to FP32/FP32. Then the latest program is executed again to verify. Because the original program is in FP32/FP32, the latest program always produces a qualified error, which guarantees the correctness of the program optimized by *moTuner* . With this approach, `Adjuster` has the time complexity of $O(N + lv - 1)$, where N is the number of operators.

Algorithm 1: Optimized tuning algorithm.

```
   Data: $O, lv, S, UE$.
   Result: Optimized mixed-precision settings for operators.
1  $E \leftarrow \{\}$;  $T \leftarrow \{\}$;  $NID \leftarrow O_0$;
2  for $i = 0 \rightarrow len(O) + lv - 1$ do
3      if $i \leq len(O) - 1$ then
4          $R_i \leftarrow DDA(O_i)$ ;
             /* Get related operators of operator $O_i$ */
5          $t_{max} \leftarrow float_{max}$;  $k_{best} \leftarrow -1$;
6          for $k = 0 \rightarrow len(S) - 1$ do
7              $err \leftarrow E_i^k$;  $t \leftarrow T_i^k$;
8              if $err \leq UE \wedge t \leq t_{max}$ then
9                  $t_{max} \leftarrow t$;  $k_{best} \leftarrow k$;
10         if $k_{best} \neq -1$ then
11             $set\_operator(S_{k_{best}}, i)$ ;
                 /* Set $O_i$ with $k_{best}$ th setting */
12         else
13             $repair\_setting(R_i, O)$ ;
                 /* Tune related operators to lower error */
14         if $O_i == NID$ then
15             $wrap\_operator(O_i)$ ;      /* Try all mixed −
                 precision settings for operator $O_i$ */
16     for $j = 0 \rightarrow len(ET) - 1$ do
17         $R_j \leftarrow DDA(O_j)$;
18         if $ET_j \geq UE$ then
19             $repair\_setting(R_j, O)$;
20     $NID \leftarrow O_{i+1}$;
21     $ET, E_i, T_i, \leftarrow execute()$;      /* rerun the program */
```

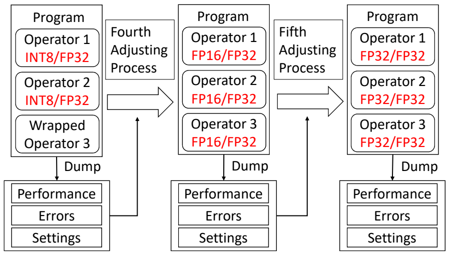


Figure 6: Optimized adjustment of three operators in the worst case.

3.3 Implementation

Our approach is built in HIPCC, an open-source LLVM compiler with full support of HIP. HIP is a heterogeneous programming framework compatible with both AMD and Nvidia GPUs, thus all the code we operate and analyze is implemented in HIP. Now we discuss main components of our implementation, (1) GEMM and precision refinement framework, (2) Optimization passes.

- **GEMM and precision refinement framework.** We target on FP16/FP32 GEMM in HIPBLAS as the mixed-precision operator and build precision refinement framework based on QUANTENSOR [28]. (τ_1, τ_2, τ_3) is the parameter in it to gain tradeoff between performance and accuracy. When $(\tau_1, \tau_2, \tau_3) = (0, 0, 0)$, it runs a GEMM under FP16/FP32 without precision refinements. To take advantage of the optimized adjustment, refinement parameters are levelized into 3 levels. The lowest one is with $\tau_1 + \tau_2 < 2$, the higher one is with $(\tau_1, \tau_2, \tau_3) = (1, 1, 0)$ and the highest one is in FP32 precision. For extra memory space needed by the precision refinement framework to store low-precision data, we collect the size of extra memory space needed by each GEMM in `Marker` and create the biggest one for usage in `Adjuster` and `Finalizer`.

- **Optimization passes.** The `Marker`, `Adjuster` and `Finalizer` are implemented as separated optimization passes in LLVM. In order to record result, ID and dimensions of each GEMM in `Marker`, we use a wrapper function to replace the original GEMM function. This wrapper function not only performs the GEMM under FP32 precision with the same input, but also dumps ID, dimensions and result of original GEMM to file system. To find all directly and indirectly related operators for one, we maintain a directly related operators set for each operators and apply DFS algorithm on detecting indirectly related operators for it.

4 EXPERIMENTAL METHODOLOGY

4.1 Platform and Workloads

4.1.1 Environments.

We conduct experiments on the computing platform with configurations being summarized in Table 2. All programs are compiled using with HIPCC@4.3.1 with option `-O3`, which is commonly switched on to achieve high performance.

Table 2: Specifications for the computing platform.

Hardware		Software	
CPU	EPYC 7302 (Freq.: 3.0-3.3 GHz)	Operating System	CentOS 7.9
GPU	MI100 (FP32 Perf.: 46.1 TFLOPS, FP16 Perf.: 184.6 TFLOPS)	Operator Library	hipblas@4.3.1 rocsolver@4.3.1
Memory	32 GB	Compiler	HIPCC@4.3.1

4.1.2 Benchmarks.

We test three benchmarks to demonstrate that *moTuner* can automatically and efficiently tune mixed-precision setting of each GEMM to improve performance while guaranteeing desired accuracy. These workloads all use FP32/FP32 GEMM by default and set FP16/FP32 as the target mixed-precision setting. The first one is the micro-benchmarks (Micro), contains varying number of GEMMs (e.g., 3, 6 and 9 [1]). Figure 7 shows the dependency in the tested micro-benchmarks. Also to validate the *moTuner*'s robustness for data, we generate matrices with following data distributions: $Normal(0, 0.5)$, $Uniform(-0.5, 0.5)$ and $Random(-1, 1)$.

The other two are cholesky factorization (CF) and HPL-AI [17] and we implement the tiled version of them. The input setting of these two applications are denoted using (N, ts), which decides execution counts of GEMM operator. N and ts refer to input matrix dimension and tile size, respectively.

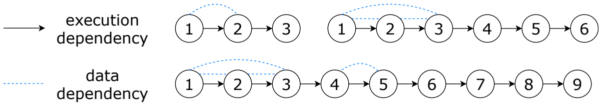


Figure 7: Dependency in micro-benchmarks containing 3, 6 and 9 GEMMs.

4.2 Metrics

For performance measurement, we consider the execution time speedup of the whole application when running Micro and CF. As for the HPL-AI, we consider the speedup of its GEMM part, owing to the high number of non-operator calls. The execution time is the average of five repeated runs.

To measure accuracy of an operator, we select the maximum absolute error (E_δ) and mean relative error (E_γ) of its result. Let X be the matrix produced by FP32/FP32 GEMM, $X^{'}$ be the matrix computed in FP16/FP32 mixed-precision. We define the maximum absolute error E_δ as the maximum difference between matrix X and $X^{'}$:

$$E_\delta(X, X^{'}) = \left\| X_{flatten} - X^{'}_{flatten} \right\|_\infty \tag{1}$$

And the relative error is in the Frobenius norm E_γ:

$$E_\gamma(X, X^{'}) = \frac{\left\| X - X^{'} \right\|_F}{\left\| X^{'} \right\|_F} \tag{2}$$

Settings of error threshold are categorized into eight kinds, which are used to tune operators in the following experiments and listed in Table 3. These thresholds cover a wide range for usual applications.

Table 3: Categories of different error thresholds.

Error Kind	Value	Error Threshold Category
E_γ	0.05	E1
E_γ	0.005	E2
E_γ	0.0005	E3
E_γ	0.00005	E4
E_δ	100	E5
E_δ	10	E6
E_δ	1	E7
E_δ	0.1	E8

As for the accuracy of Micro, we consider the $1 - E_\gamma$ of operators with the most data dependencies. In CF, $1 - E_\gamma$ of the result matrix is used to measure accuracy. For HPL-AI, because it only validates whether the scaled residual of final result is less than 16, we measure its accuracy by $1 - E_\gamma$ of the scaled residual. For a measurement of tuning effectiveness, we denote the execution count instead of search time as the tuning effort because time spent in hand optimization is not objective. And we select the $\log_{10}(execution\ count)$ as the metric.

5 RESULTS AND ANALYSIS

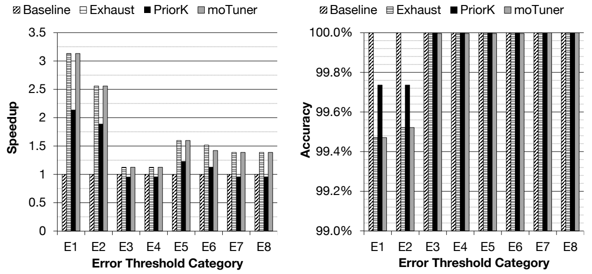
In this section, we present and analyze the respective experiment results of the aforementioned workloads. We evaluate *moTuner* from three different perspectives: performance and accuracy of optimized programs, automation effectiveness and sensitivity of input setting. We studied and compared the following schemes:

[1] The GPU memory capacity allows at most 9 GEMMs, which are occupying approximately 85% memory.

- *Baseline*. We run the program in FP32/FP32 for once.
- *Exhaust*. This method exhaustively searches all setting combinations for executed operators in the program and select the one bringing the highest performance and acceptable error.
- *PriorK*. With expertise or prior knowledge, the search space in *Exhaust* can be effectively narrowed. As such, *PriorK* is aware of error and performance of each operator with every setting in advance. In tests for the Micro, we randomly select the top 1% settings which produce acceptable error for each operator. In tests for the real world application, we randomly choose $M/2$ from fastest 50% settings with acceptable error for each operator, where M indicates the total setting combinations for all operators. For both, we use the average performance and accuracy of all chosen settings as the result.
- *moTuner* . We use *moTuner* to optimize a program for once.

5.1 Performance and Accuracy

Figure 8 compares the speedup and accuracy of the optimized Micro GEMM count is 9 (i.e., GEMM-1, ..., GEMM-9) and the data distribution of input matrix is $Normal$. Figure 8(a) demonstrates that *moTuner* gains up to 3.13x speedup and 1.72x speedup on average, which are mostly the same as the ones of *Exhaust* and higher than *PriorK*. But for E6, performance gained by *moTuner* is less than *Exhaust* because *moTuner* sets the (τ_1, τ_2, τ_3) of GEMM-5 as $(1, 1, 0)$ while it can still satisfy the error requirement with $(\tau_1, \tau_2, \tau_3) = (0, 0, 0)$. *Exhaust* detects such a situation and provides the setting with higher performance to GEMM-5. Figure 8(b) shows the accuracy of GEMM-3 in Baseline and the tuned programs. When error category is E1 and E2, *moTuner* achieves lower accuracy than *PriorK*, which is still higher than 99.4%. For the remaining error categories, *moTuner* can achieve almost 100% accuracy.

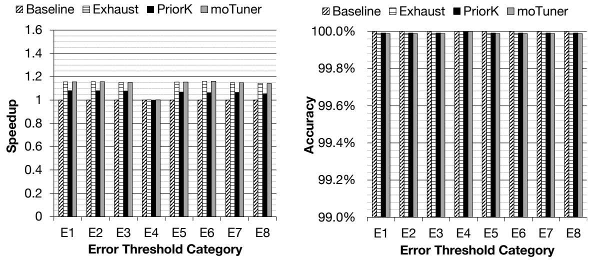


(a) Speedup.

(b) Accuracy.

Figure 8: Speedup and accuracy of Micro gained by different schemes under varying error thresholds.

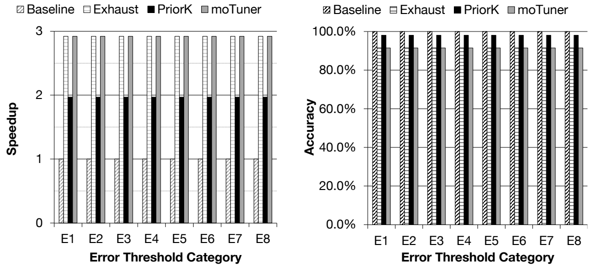
Figure 9 covers the speedup and accuracy of CF optimized by different schemes. The input setting of CF is $(40960, 8192)$. Figure 9(a) shows that *moTuner* can obtain a mean speedup of 1.153x speedup in all error categories except E4. When the parameter level of mixed-precision GEMM with $ts = 8192$ is lower than the one of $(\tau_1, \tau_2, \tau_3) = (1, 1, 0)$, E_γ of its result is above 0.00005. So optimizing with E4 requires it to be upgraded to $(1, 1, 0)$ theoretically. But because mixed-precision GEMM with $(\tau_1, \tau_2, \tau_3) = (1, 1, 0)$ is slower than FP32/FP32 GEMM due to type-casting cost, *moTuner* decides to keep it in FP32 precision for E4 and brings no performance degradation. For the accuracy of CF shown in Figure 9(b), *moTuner* is capable to achieve over 99.99% accuracy under all given thresholds, whereas *PriorK* brings some little advantages under some error thresholds compared to *moTuner* .



(a) Speedup.

(b) Accuracy.

Figure 9: Speedup and accuracy of CF tuned by different schemes under varying error thresholds.



(a) Speedup.

(b) Accuracy.

Figure 10: Speedup of GEMM part in HPL-AI and accuracy of HPL-AI tuned by different schemes under varying error thresholds.

Figure 10 presents the speedup and accuracy of baseline of HPL-AI's GEMM part with the ones optimized by different schemes. The input setting is $(24576, 8192)$. Figure 10(a) shows that while *PriorK* can only gain 1.97x speedup on average, *moTuner* is able to acquire a 2.92x mean speedup, which is the same as the *Exhaust*. Figure 10(b) demonstrates that *moTuner* can achieve over 92% accuracy in all situations while *PriorK* achieves 98%. But because HPL-AI only validates whether the scaled residual is smaller than 16 and the one from programs tuned by *moTuner* is 0.003551, it means that *moTuner* is able to generate qualified HPL-AI programs with faster GEMMs.

5.2 Automation Efficiency

In this section, we demonstrate how efficient *moTuner* can be when tuning the benchmarks, compared to two manual assisting methods. The input of these benchmarks keeps the same as the ones used in Section 5.1. As discussed before, the efforts are estimated using the execution counts. Figure 11 illustrates the tuning efforts needed by different schemes. The *Exhaust* demands the most tuning efforts and *moTuner* requires the least. Also, *Exhaust* and *PriorK* both require manual code modification while *moTuner* provides an end-to-end automatic optimization.

5.3 Sensitivity Studies

In this section, we present how *moTuner* performs on programs with varied inputs. First, we test on Micro with different data distributions of input matrices and GEMM count. Figure 12 shows

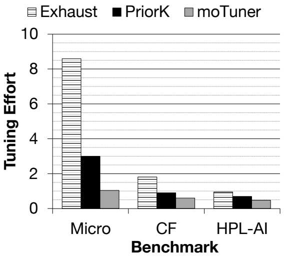
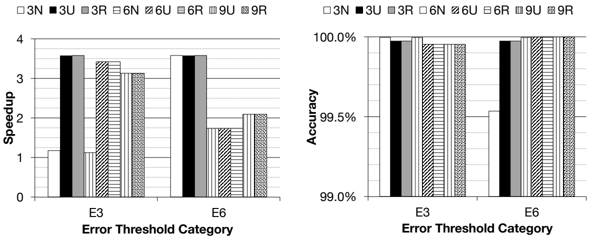


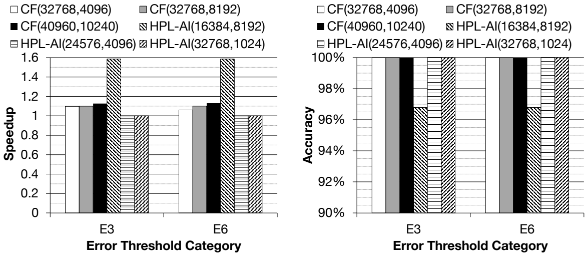
Figure 11: Average tuning effort needed by three schemes under all error thresholds.

that *moTuner* can obtain 2.67x speedup with over 99.9% accuracy in average for Micro with differing inputs. Figure 13 shows how *moTuner* performs on CF and HPL-AI with changing inputs. *moTuner* can gain a mean speedup of 1.10x and 1.19x on CF and the GEMM part of HPL-AI respectively while maintaining over 99% accuracy on average under E3 and E6.



(a) Speedup. (b) Accuracy.

Figure 12: Speedup and accuracy of different micro-benchmarks gained by *moTuner* under E3 and E6, the input setting is denoted by the number of GEMM and data distribution ($N : Normal, U : Uniform, R : Random$).



(a) Speedup. (b) Accuracy.

Figure 13: Speedup and accuracy of real applications gained by *moTuner* with different inputs under E3 and E6.

6 RELATED WORK

Error analysis of floating-point: to evaluate whether a program can take advantage of mixed-precision of floating-point to gain performance, a variety of works have been proposed to predict the error of floating-point arithmetic. They can be divided into two categories: dynamic analysis and static analysis. Dynamic analysis requires for running programs to gather necessary information. A dynamic analysis approach [5] is presented to find potential risk of degrading accuracy. Then Lam et al. [26] apply dynamic analysis to detect cancellation error in floating-point calculations. These dynamic analysis methods only aims at instructions and brings up significant tuning time. *moTuner* belongs to this category and only targets on error of operators instead of error of instruction. As for static analysis methods, they only analyze the characteristics of code to gather information. Darulova proposes Xfp [10] to gain sub-optimal solution without performing an exhaustive exploration. Rosa [8, 9] is a source-to-source compiler which provides a precision mix for a given program on real values. It introduces a contract-based programming paradigm based on the Scala functional programming language, which is not suitable for most existing scientific computing applications. AMPT-GA [23] implements a static analysis aiming at identifying strongly connected variables in the dependency graph. These works are lack of the sensitivity for different input in varying degrees while *moTuner* can generate the appropriate mixed-precision setting for different input.

Mixed-precision tuning: to lessen programming burden, several efforts focused on automatic generation of mixed-precision programs on CPU and GPU. For CPU programs, Precimonious [36] proposes the delta debugging to narrow the search space for single, double and long precision. Then a follow-work devises blame analysis [37] to reduce the search space further. While these two focus on each variable, HiFP-Tuner [14] targets on groups of variables constructed by using the community structure detection, which also reduces the search space. Similarly, FPTuner [7] applies SMT-solver to tune groups of operations and predict an error upper bound. For GPU programs, GPUMixer [25] tunes mixed-precision settings of floating-point operations. Although GPUMixer is performance-driven, it cannot utilize mixed-pecision operators and helps precision refinement frameworks to be applied like *moTuner* . GPU-FPtuner [13] takes into account code patterns prone to error propagation, but it only supports 32- and 64-bits floating-point arithmetic while *moTuner* supports FP16 and INT8. ADAPT [30] uses algorithmic differentiation to estimate error with reduced search space. *moTuner* targets on GPU programs and is orthogonal to these works.

7 CONCLUSION

The paper proposes *moTuner* , a compiler-based auto-tuning approach for mixed-precision operators. *moTuner* automatically handles both the configuration knobs and the quantization/refinement operations, thus eases the programming burden. Further, an efficient tuning strategy is applied to strike a balance on the performance improvement and output quality. We test *moTuner* on micro-benchmark with multiple GEMMs, cholesky factorization and HPL-AI and the preliminary results demonstrate that *moTuner* can efficiently obtain up to 3.13x, 1.15x and 2.92x respectively. In the future, we plan to extend *moTuner* to support more complicated operators.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by the National Natural Science Foundation of China-#62102465/-#U1811461, the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant NO. 2016ZT06D211, the Major Program of Guangdong Basic and Applied Research-#2019B030302002, the Guangdong Natural Science Foundation-#2018B030312002, and CCF-Baidu Open Fund (CCF-BAIDU OF2021032).

REFERENCES

[1] AMD. 2021. AMD Instinct™ MI100 Accelerator. Retrieved 2022-01 from https://www.amd.com/en/products/server-accelerators/instinct-mi100.

[2] AMD. 2021. AMD rocBLAS Library. Retrieved 2022-01 from https://github.com/ROCmSoftwarePlatform/rocBLAS.

[3] Barbara Barabasz, Andrew Anderson, et al. 2020. Error Analysis and Improving the Accuracy of Winograd Convolution for Deep Neural Networks. *ACM Trans. Math. Softw.* 46, 4, 37:1–37:33.

[4] Chaim Baskin, Natan Liss, et al. 2021. UNIQ: Uniform Noise Injection for Non-Uniform Quantization of Neural Networks. *ACM Trans. Comput. Syst.* 37, 1-4, 4:1–4:15.

[5] Florian Benz, Andreas Hildebrandt, et al. 2012. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 453–462.

[6] Cambricon. 2021. Cambricon MLU Accelerator. Retrieved 2022-01 from https://www.cambricon.com/.

[7] Wei-Fan Chiang, Mark Baranowski, et al. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 300–315.

[8] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 235–248.

[9] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, 8:1–8:28.

[10] Eva Darulova, Viktor Kuncak, et al. 2013. Synthesis of fixed-point programs. In *Proceedings of the International Conference on Embedded Software*. IEEE, 22:1–22:10.

[11] Veselin A. Dobrev, Tzanio V. Kolev, et al. 2012. High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5, B606–B641.

[12] Amir Gholami, Sehoon Kim, et al. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR* abs/2103.13630.

[13] Ruidong Gu and Michela Becchi. 2020. GPU-FPtuner: Mixed-precision Auto-tuning for Floating-point Applications on GPU. In *27th IEEE International Conference on High Performance Computing*. IEEE, 294–304.

[14] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 333–343.

[15] Kaiming He, Xiangyu Zhang, et al. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 770–778.

[16] Tsuyoshi Ichimura, Kohei Fujita, et al. 2018. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE / ACM, 49:1–49:11.

[17] ICL. 2021. The High Performance LINPACK for Accelerator Introspection (HPL-AI) benchmark. Retrieved 2022-01 from https://bitbucket.org/icl/hpl-ai/src/main/.

[18] Intel. 2019. Introduction to Intel deep learning boost on second generation Intel Xeon scalable processors. Retrieved 2022-01 from https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-deep-learning-boost-on-second-generation-intel-xeon-scalable.html.

[19] Intel. 2021. Intel MKL. Retrieved 2022-01 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[20] Emmanuel Jeannot. 2012. Performance Analysis and Optimization of the Tiled Cholesky Factorization on NUMA Machines. *Proceedings - International Symposium on Parallel Architectures, Algorithms and Programming*, 210–217.

[21] Weile Jia, Han Wang, et al. 2020. Pushing the limit of molecular dynamics with *ab initio* accuracy to 100 million atoms with machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE/ACM, 1–14.

[22] Norman P. Jouppi, Cliff Young, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, 1–12.

[23] Pradeep V Kotipalli, Ranvijay Singh, et al. 2019. AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications. In *Proceedings of the ACM International Conference on Supercomputing*. Association for Computing Machinery, 160–170.

[24] Alex Krizhevsky, Ilya Sutskever, et al. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6, 84–90.

[25] Ignacio Laguna, Paul C. Wood, et al. 2019. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. In *High Performance Computing - 34th International Conference Proceedings*, Vol. 11501. Springer, 227–246.

[26] Michael O. Lam, Jeffrey K. Hollingsworth, et al. 2013. Dynamic Floating-Point Cancellation Detection. *Parallel Comput.* 39, 3, 146–155.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 75–88.

[28] Guangli Li, Jingling Xue, et al. 2021. Unleashing the Low-Precision Computation Potential of Tensor Cores on GPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 90–102.

[29] Stefano Markidis, Steven Wei Der Chien, et al. 2018. NVIDIA Tensor Core Programmability, Performance, Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE Computer Society, 522–531.

[30] Harshitha Menon, Michael O. Lam, et al. 2018. ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 48:1–48:13.

[31] netlib. 2021. HPL benchmark. Retrieved 2022-01 from https://www.netlib.org/benchmark/hpl/.

[32] Tomás Norton and Da-Wen Sun. 2006. Computational fluid dynamics (CFD) – an effective and efficient design and analysis tool for the food industry: A review. *Trends in Food Science & Technology* 17, 11, 600–620.

[33] NVIDIA. 2008. cuBLAS Library. Retrieved 2022-01 from https://docs.nvidia.com/cuda/cublas/.

[34] NVIDIA. 2021. NVIDIA A100 Tensor Core GPU. Retrieved 2022-01 from https://www.nvidia.com/en-us/data-center/a100.html.

[35] Riken. 2021. Comprehensive software for ab initio quantum chemistry calculations of large and complicated molecular systems. Retrieved 2022-01 from https://www.r-ccs.riken.jp/software_center/software/ntchem/overview/.

[36] Cindy Rubio-González, Cuong Nguyen, et al. 2013. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 27:1–27:12.

[37] Cindy Rubio-González, Cuong Nguyen, et al. 2016. Floating-Point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering*. Association for Computing Machinery, 1074–1085.

[38] Youcef Saad and Martin H. Schultz. 1986. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput.* 7, 3, 856–869.

[39] Zhuoran Song, Bangqi Fu, et al. 2020. DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 1010–1021.

[40] Huanrui Yang, Lin Duan, et al. 2021. BSQ: Exploring Bit-Level Sparsity for Mixed-Precision Neural Network Quantization. *CoRR* abs/2102.10462.

[41] Zhaoyang Zhang, Wenqi Shao, et al. 2021. Differentiable Dynamic Quantization with Mixed Precision and Adaptive Resolution. In *Proceedings of the 38th International Conference on Machine Learning*, Vol. 139. PMLR, 12546–12556.