GRAM: A Framework for Dynamically Mixing Precisions in GPU Applications

NHUT-MINH HO, HIMESHI DE SILVA, and WENG-FAI WONG,

National University of Singapore

This article presents GRAM (GPU-based Runtime Adaption for Mixed-precision) a framework for the effective use of mixed precision arithmetic for CUDA programs. Our method provides a fine-grain tradeoff between output error and performance. It can create many variants that satisfy different accuracy requirements by assigning different groups of threads to different precision levels *adaptively at runtime*. To widen the range of applications that can benefit from its approximation, GRAM comes with an optional half-precision approximate math library. Using GRAM, we can trade off precision for any performance improvement of up to 540%, depending on the application and accuracy requirement.

CCS Concepts: • **Software and its engineering** → *Source code generation*; • **Computing methodologies** → *Graphics processors*; • **Mathematics of computing** → *Approximation*;

Additional Key Words and Phrases: GPU, approximate computing, half precision, mixed precision

ACM Reference format:

Nhut-Minh Ho, Himeshi De Silva, and Weng-Fai Wong. 2021. GRAM: A Framework for Dynamically Mixing Precisions in GPU Applications. *ACM Trans. Archit. Code Optim.* 18, 2, Article 19 (February 2021), 24 pages. <https://doi.org/10.1145/3441830>

1 INTRODUCTION

In recent years, along with the emergence of deep learning and data analytics, *approximate computing* has regained prominence. A common characteristic among these applications is the use of floating point—which itself is an approximation of real number arithmetic. One of the most widely adopted techniques for approximation in commodity hardware is to use lower precision formats. Furthermore, many hardware vendors have started to provide low precision format as alternatives to the widely used IEEE *float* (FP32) and *double* (FP64) types in traditional hardware. Most notably, the introduction of *half precision* (FP16) in GPUs by both Nvidia and AMD have sparked a general interest in using lower precision data types. Mixed precision for deep learning and tensor-core-based applications [11, 28] can be enabled by re-engineering the BLAS libraries and frameworks to support tensor operations in FP16. However, the same is not true for more general applications, especially those that do not use BLAS libraries. In this article, we introduce GRAM (GPU-based

This work was funded in part by Singapore Ministry of Education research grants T1-251RES1818 and MOE2016-T2-2-150. Authors' addresses: N.-M. Ho, H. De Silva, and W.-F. Wong, National University of Singapore; emails: {minhnh, himeshi, wongwf}@comp.nus.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1544-3566/2021/02-ART19

<https://doi.org/10.1145/3441830>

Runtime Adaption for Mixed-precision), a mixed precision framework that solves this problem. Traditionally, mixed precision tuning frameworks on CPU applications try to search for a subset of variables that can be switched to lower precision for performance gains. This strategy is adopted to some recent works to support GPU applications. However, GPU applications are usually data-parallel and inherently different from CPU applications. Adopting the same searching strategy and precision mixing method may not be beneficial.

The main difference between GPU and CPU programs is that the data dimension (number of data elements) is changing and usually larger than the number of variables for the former. As a result, other approximation techniques on GPUs [24, 40, 47] have exploited the data dimension to provide tradeoff between performance and error. However, these approximations usually result in higher error than what mixed precision tuning can yield.

Based on the above observation, the key innovation in GRAM is to treat the mixed precision tuning problem on GPU applications in a way that is similar to other data-wise approximations. Instead of allocating variables using different precisions, we allocate *data elements* to precision levels. This technique is analogous to a milder version of loop perforation on CPU programs. The “perforated” loops are replaced with low precision computation instead of removing the computation completely.

Given an original CUDA program and a user-defined accuracy objective, GRAM will automatically generate a mixed-precision version that achieves the accuracy using a performant combination of FP16 and FP32. The same strategy works for mixing FP32 and FP64. In summary, our contributions are as follows:

- We introduce a data-wise precision tuning strategy that generates a mixed-precision version for CUDA programs with hundreds of tradeoff points that can be changed at runtime.
- We are the first to introduce an effective method to mix half2 type of FP16 with FP32. The effect of using half precision on error of various applications is also reported.
- Our method can work well either independently or as a complement to traditional mixed-precision tuning techniques to better exploit the performance-accuracy tradeoff in large programs.
- As far as we know, GRAM is the only framework that allows for varying the mix of precision *at runtime*.

The rest of this article is organized as follows: Section 2 briefly reviews mixed precision tuning methods and GPU programming. Section 3 presents the challenges faced and how we solved them. Sections 4 and 5 give the full details of GRAM. Section 6 shows the experimental results, followed by the related works survey and conclusion.

2 BACKGROUND

2.1 Mixed Precision Tuning

There has been extensive research on using mixed precision floating point in programs. Given a program that performs a significant amount of floating point arithmetic operations, the safe choice for most programmers will be to have all variables represented in the higher precision (FP64). However, that choice is usually too conservative. Previous works in mixed-precision tuning methods [4, 27, 39] have shown that most of these variables are actually approximable. By reducing the precision of these variables (usually to FP32), the performance of these programs will most likely improve on many platforms. However, the choice of which variables to have in low precision affects the output error. Therefore, it is important to find a suitable mix of precision in the program that meets some user given accuracy threshold while maximizing performance. The most common

approach to this problem is some form of smart searching and analysis [27, 39]. We will give a thorough review of such techniques in Section 7.

Interestingly, while the CUDA platform was introduced some 10 years ago, only recently have techniques for mixed precision tuning specifically for CUDA programs emerged. The current state-of-the-art techniques, namely, ADAPT [27], GPUMixer [22], and AMPT-GA [21] can target CUDA programs. However, all of the works are variable-wise tuning and can only work well on mixing FP32 and FP64. The support for newly introduced FP16 type is limited due to challenges involving the use of the half2 datatype. Also, because GPU programs are highly data-parallel, switching variables to lower precision yields a very coarse-grained tradeoff. Furthermore, analyzing them is very time-consuming. We will compare our method with all of the related works in Section 6.

The method used in our GRAM framework is more general than previous works. Besides the traditional mixing between FP32 and FP64, we provide solution for mixing FP32-FP16 of highly approximable CUDA applications. To the best of our knowledge, GRAM is the first framework that can mix half2 data type and float efficiently. We will show that besides deep learning that motivated the introduction of FP16, other general purpose floating point CUDA applications can also benefit from its use.

2.2 CUDA Programming and FP16

In this section, we shall briefly introduce the GPU programming model that our method is based on, mainly for completeness. A program written in CUDA will have multiple *kernels*. Each kernel invocation will launch N *threads* running in parallel. To map the threads to actual hardware at runtime, they are grouped into *warps*. Warp typically consists of 32 threads that will execute the same instruction in hardware.¹ This is the smallest unit that the instruction scheduler will issue instruction for. Multiple warps then form a *block*. All threads in a block are executed in a *streaming multiprocessor* (SM), allowing them to share the fast shared memory. All the blocks in turn form a *grid*, which contains all the threads used to compute the problem in that particular CUDA kernel.

An additional level of parallelism inside each thread is required to fully exploit the advantage of FP16 arithmetic. Each half2 instruction inside each thread now can perform a 2-way SIMD operation on the two halves of the half2 datatype. FP16 operands are in half2 type, which is simply two FP16 data located next to each other. To simplify the notation, we shall use FP16 and half2 interchangeably, which means the fastest SIMD half precision version of the program. The same FPU can now either execute a single FP32 instruction or a single SIMD half2 instruction (with the same operation applied to each FP16 data). The throughput of half precision arithmetic in Nvidia GPUs is reported to be twice the throughput of FP32 [34].

3 CHALLENGES IN CUDA MIXED PRECISION

3.1 Usage of Half Precision on GPUs

In their latest architectures, both Nvidia and AMD provide support for FP16 arithmetic. This support is primarily driven by deep learning applications that use enhanced BLAS libraries and, more recently, tensor cores via vendor-supplied libraries. Support for general applications is still limited. Currently, the following options are available for the use of FP16 [12, 16]:

- In linear algebra and deep learning applications, one can use the tensor cores and FP16 CUBLAS libraries as well as automatic tool provided by Nvidia.
- Memory bound applications can double their data throughput with the use of FP16.

¹Nvidia's Volta architecture allows sub-warp synchronization, but the change does not affect the correctness or the efficacy of our work.

- Compute-bound applications must use the half2 data type to enjoy any of the up to $2\times$ theoretical speedup.
- Applications with a mix of compute and memory operations will still need to use the half2 data type for higher performance.

In this article, we shall consider compute-bound general applications as our main targets for tuning.

3.2 Conversion Overhead and Coarse Granularity

3.2.1 An Example of Mixing FP32 and FP64. To highlight the problem with applying the traditional approach for mixed precision tuning to CUDA programs, we ported a classic example for CPU to its CUDA equivalent with the same computations so we can compare their results. Listing 1 shows one such example for a mixed precision program, arclength. It has been used in various papers [8, 39] to showcase the mixed-precision tuning opportunity on FP64, FP80, and FP128. Because CUDA only supports FP64 as maximum precision, we will consider a mix of FP32 and FP64 types in this example.

We introduce TYPE4 as an additional tuning opportunity, as it is the result of the math function $\sin()$ for FP64 or $__\sin f()$ for FP32 in CUDA program.

```
void arclength(double x[], double y[], int nthreads){
    //y = fun(x) for 1E6 elements
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int k, n = 5;
    if (tid < nthreads) {
        TYPE1 t1; TYPE2 d1 = 1.0;
        TYPE3 x_temp = x[tid];
        t1 = x[tid];
        for (k = 1; k <= n; k++){
            d1 = 2.0 * d1;
            TYPE4 sin_res = sin(d1 * x_temp);
            t1 = t1 + sin_res/d1;
        }
        y[tid] = t1;
    }
}
```

Listing 1. Mixed precision example of arclength in CUDA.

There are 4 variables in the sample program; therefore, we have $2^4 = 16$ possible configurations of mixed precision between FP64 and FP32 for all variables. We tested all the combinations using Listing 1 and measured the performance and floating point error of the kernel. The version where all variables are in FP32 and FP64 shall be labeled P_FP32 and P_FP64 , respectively. After obtaining $y[tid]$ from Listing 1, the outputs will be used to calculate the final summation similar to the example in Reference [39]. The error before normalization is the absolute error of the sum compared to the reference output in FP64 (i.e., 5.79577632...). The execution time and absolute error then will be normalized so the runtime and error of P_FP32 are 1.0 with the absolute error of P_FP64 being 0.0. In other words, the normalized error is the current error divided by the maximum possible error (when all variables are in FP32). Because there are 4 variables, there are 4 different levels of approximation (each level may consists of multiple different combinations). These are 0% where all variables are in FP64 precision, 25%, 50%, 75%, and 100% where 1, 2, 3, and all variables are in FP32, respectively. We collected all the runtime (average of 5 runs) and error using the same system as

Fig. 1. Performance and error for all possible mixed-precision configurations of the arclength example.

in Section 6.1 and plotted them in Figure 1. The green columns are the runtime with normalized values on the left y-axis. The series shows the normalized error induced with values on the right y-axis. The blue horizontal line is the normalized runtime of P_{FP64} , which is $1.26\times$ slower than P_{FP32} . Due to the conversion overhead, some mixed versions are actually slower than P_{FP64} .

Figure 1 shows that the variables-wise precision allocation has two main drawbacks:

- The granularity is too coarse. For example, there is no mixed-precision version gives the error between 0.0 and 0.28. Also, the error incurred by mixing precisions in this way fluctuates wildly. Complex analysis tools have to be created to find good configurations.
- The overhead of type conversion can be significant. This varies depending on the architecture. In this example, 9 out of the 16 mixed-precision configurations had *worse* performance than the all-FP64 versions.

When two operands of an operation (e.g., addition) have two different data types, an expensive conversion is unavoidable. This limits the number of useful configurations that actually results in any speedup. Also the coarse granularity makes it hard for any precision tuning tool to find the configuration that matches the desired error threshold given by the user. Taking the arclength example of Figure 1, if we use a normalized error that is 0.2 as the target error tolerance, any existing mixed-precision tuning tool will have to suggest the all-FP64 version with 0.0 error as recommendation. To solve this problem, we introduce a new method that does fair for every error threshold, and, in particular, will work in the region between 0.0 and 0.28 for arclength where no combination of mixed precision variables will yield. This is validated in our experiments in Section 6.3.3.

3.2.2 Mixing FP16 and FP32. Besides the limitations described in Section 3.2.1, the traditional variable-wise method cannot be easily applied for mixing FP16 and FP32. The naïve way of mixing half type and FP32 via conversions and the FPU without SIMD will likely slow down the program, unless the program is memory-intensive as pointed out in Reference [16]. To fully utilize the FPU, the half2 instruction has to be used. Users end up with the option to vectorize the program to operate in the float2 vector type to be compatible to with half2. This vectorization task is not trivial. At worst, when the shared memory and registers are already fully utilized in the FP32 version, the float2 version will result in a non-compilable (shared memory limit) or very slow (registers spilling) program.

4 DATA-AWARE PRECISION TUNING

This section describes the intuition behind our method and details for generating mixed precision CUDA code.

Fig. 2. Two mixed precision approaches on a simplified GPU programming model.

4.1 Data-wise and Variable-wise Mixed Precision in CUDA Applications

Unlike traditional CPU programs, CUDA programs are highly parallel and process on large arrays of data. While the number of variables is static and determined at compile time, the size of data is usually determined at runtime. Let us not consider the overhead of type conversion for the time being so, using a traditional technique, we can create a mixed precision version, as shown in Figure 2. Consider a single instruction performed by each thread in Figure 2; the traditional method points out which variable is approximable (B) or non-approximable (A and C). The decision is fixed for the entire array of data named by a single variable. In case of local variable (not an array), that decision also affects all the local versions of the same variable in all the threads executed. This method is illustrated in Figure 2(a).

In contrast, Figure 2(b) shows a different approach to approximation where we consider each element of the array separately. We decide at runtime which part of the data in an array needs to be in lower precision for faster execution. The example in Figure 2 only demonstrates this for a small number of threads. For larger programs with thousands to millions of threads, we can get a very fine-grained degree of approximation by varying the amount of threads being executed in low precision. As stated, the hardware constraints of GPU do not allow thread level of granularity. Therefore, we have to assign group of threads to be either executed in high precision or lower, but not both. A solution for this will be introduced in Section 4.2.

If the application is compute-bound as described in Section 3, we can keep the data in higher precision and use conversion once before intensive arithmetic operations. This way, the mixed precision version can be constructed with ease. The proportion of threads that use low precision can be adjusted as a runtime argument, removing the need for any recompilation.

To showcase the advantage of GRAM in terms of the conversion overhead, we conduct a set of microbenchmarks to compile the kernel in Listing 1 using different approaches shown in Figure 2. The applications are compiled with the “--ptx” flag and the number of the implicit floating point conversion instructions (e.g., `cvt.fp32.`, `cvt.fp64.`) of each thread is counted and reported in Table 1. The table explains the poor performance of mixed precision program version in Figure 1. When all of the variables are in FP32 (the 100% column in Table 1), we can clearly see that the compiler will implicitly use 3 floating point conversions, shown in Listing 1. When all the variables are in FP64 (the 0% column in Table 1), no floating point conversion is needed. Besides these two configurations, other mixed precision versions suffer from a significant conversion overhead (up to

Table 1. The Number of Floating Point Conversion Instructions of the *Arclength* Kernel in Listing 1 When Using the Conventional Mixed Precision Method Compared with Our Method in Figure 2

Percentage of variables in FP32	0%	25%	50%	75%	100%
Conventional method (Figure 2(a))	0	0–12	7–22	7–22	3
GRAM (Figure 2(b))	0	0–3 25% of threads: 3 75% of threads: 0	0–3 50% of threads: 0 50% of threads: 3	0–3 25% of threads: 0 75% of threads: 3	3

The percentage of variables in FP32 was defined and used in Section 3.2.1.

22 instructions used in the worst case, where each loop of Listing 1 requires several conversions). The conversions are added by the compiler to make sure each two-operand instruction will have both operands in the same type. It is a low-level hardware constraint. In contrast, the number of conversion instructions of the compiled program generated by GRAM is between the two extremes (0 conversions or 3 conversions). Each thread using GRAM will execute either a copy of the “no-conversion” version (the 0% column in Table 1) or a “3-conversion” version of the program (the 100% column in Table 1).

4.2 Warp-level Approximation

In CUDA, a warp is the unit of instruction issuance. Any difference in execution path between threads in a warp will cause divergence. Therefore, the smallest granularity possible for approximation is a warp. This technique was introduced in Reference [24] where the authors consider the effect of the whole warp doing an approximate version of math function or not. In a similar manner, we must make the whole warp run the same instruction, be it FP16 add or FP32 add, say. Hence, our mixed precision code will run each warp either in higher or lower precision, but not both. The technique we use is similar to Reference [24]. We will generalize it to the block level as in the next section. The overhead from this approach is reported to be insignificant, because no divergence occurs.

4.3 Block-level Approximation

The usage of shared memory to improve speed is common in real-world applications. To avoid type conversion, maintaining two versions of shared memory data, one using FP32 and another using FP16, can be used. However, when the original program is probably optimized to fully utilize the limited shared memory capacity, allocating extra shared memory space for half precision may not be possible. Therefore, we have to set the granularity at block-wise approximation; in other words, the whole block of threads either doing all FP32 computation or half2 computation, and not mix. Inside each block doing half precision, we can fuse two adjacent threads to form 2-way SIMD version that makes use of half precision as was done in Reference [16]. Given a program running with hundreds of blocks, this blocks level of granularity can give enough degree of freedom to create any mixed precision version at the resolution of 1% of threads.

Listing 2 demonstrates the fusing technique for creating a half2 version of the *arclength* example in Listing 1. Fusion is applied within each FP16 block, which will see half of the threads disabled. Other blocks in FP32 are not affected. For this to work, the occupancy after fusion must remain acceptable. As will be demonstrated in our experiments, this is the case for a variety of benchmarks.

We enhanced the techniques from References [12, 16] by adding the block localization control to isolate the usage of FP16 to a specific block instead of the entire program. Note that, for the example in Listing 2, we can also vectorize the **for** loop to use half2 type and achieve similar

```

void arclength(float x[], float y[], int nthreads, int  $\alpha$ ){
  if(isApproximable(block_idx,  $\alpha$ )){
    //doing FP16 computation
    if (threadIdx.x < blockDim.x/2){ //fused threads
      int tid1 = blockDim.x*blockIdx.x + 2*threadIdx.x;
      int tid2 = blockDim.x*blockIdx.x + 2*threadIdx.x+1;
      int k, n = 5;
      if (tid2 < nthreads) {
        half2 t1; half2 d1 = __float2half2_rn(1.0);
        half2 x_temp = __floats2half2_rn(x[tid1], x[tid2]);
        t1 = x_temp;
        for ( k = 1; k <= n; k++ ){
          d1 = 2.0 * d1;
          half2 sin_res = h2sin(d1 * x_temp);
          t1 = t1 + sin_res * h2rcp(d1);
        }
        //convert result back to float
        float2 y_temp = __half2float2(t1);
        y[tid1] = y_temp.x;
        y[tid2] = y_temp.y;
      }
    } else return; //50% the warps are inactive.
  } else {
    //doing FP32 computation in Listing 1
  }
}

```

Listing 2. Block-localized thread fusing of arclength.

results. We consider both options when processing benchmarks, especially when the number of threads per block is too small for effective threads fusing.

4.4 Constructing a Mixed Precision Program

Based on the block-level of approximation, we can construct the mixed precision version of the given program. In our experiments, block-level approximation is good enough to provide a 1% step in our tuning process, since most benchmarks use more than 100 blocks. The next task is to realize the `isApproximable` condition in Listing 2. Finally, we simply copy the code of the two versions (e.g., FP16 and FP32) and enclose them in the `if-then-else` condition statement as shown in Listing 2. We choose to keep all the input and output data in high precision to maintain compatibility with the original version. For mixed precision between FP32 and FP16, we use the thread fusion technique as described in Section 4.3. To mix FP32 and FP64, no thread fusing is needed to create the two versions.

`isApproximable` can be realized using some simple policies based on the *degree of approximation*, α , (expressed as a percentage) as a runtime parameter. There are two simple ways to realize the `isApproximable` condition:

- *Scattered* distribution policy: $[\text{block_idx} \bmod 100 < \alpha]$ or $[\text{block_idx} \bmod 10 < \alpha/10]$. In this policy, $\alpha\%$ of blocks will be executed in lower precision. All the lower precision blocks will be scattered evenly every 100 or 10 blocks. This is the default policy, because it scatters the low precision elements more evenly on the output arrays.

- *Clustered* policy: $\text{block_idx} < \alpha\%$ of max_block_idx . This policy differs from the first in that the first $\alpha\%$ of blocks, arranged by the block_idxs , will be assigned low precision first.

In the end, we have a single mixed precision program that can be configured at runtime via the α parameter without the need for recompilation. α is an intuitive knob for adjusting how much of the operations will be run in lower precision. For all the benchmarks we tested on, the two policies are enough for generating mixed precision programs from many application domains. When the original programs generate input randomly and the error metrics are average relative error or root mean squared error, we observed no significant error variation between the two schemes. In rare cases, for some benchmarks with custom error metrics and fixed test inputs, the variations between two policies can be significant. If no error threshold is specified, GRAM chooses the policy that results in almost linear error tradeoff in the entire tuning range. For a particular error threshold supplied by user, we simply try the two policies and select the one that gives the most performance. Both policies are included in the mixed precision code. Selecting the appropriate policy can be done by simply using the corresponding `isApproximable` implementation.

4.5 Finding α for a Given Accuracy

After constructing the mixed-precision version, we now need to find the degree of approximation, α , described above that will maintain a given accuracy threshold, ϵ . From our experiment in Section 6, the error monotonically increases as α is increased. We can use a binary search for the search. If we choose to configure the program with the smallest tunable resolution of 1%, say, the search will terminate at $\lceil \log_2(100) \rceil = 7$ steps. As the minimum tunable resolution is set to a constant, the complexity for searching is constant $O(1)$. The binary search procedure is presented in Algorithm 1. `RunProgram(mid)` will return the error with mid percent of threads running in lower precision (FP16 in general).

ALGORITHM 1: Finding α satisfies an error threshold ϵ

```

lower_bound = 0 ; upper_bound = 100 ; stop_resolution = 1;
while (upper_bound - lower_bound) > stop_resolution do
    mid = (upper_bound + lower_bound)/2;
    if RunProgram(mid) <  $\epsilon$  then
        | lower_bound = mid;
    else
        | upper_bound = mid;
    end
end
return lower_bound as  $\alpha$ ;

```

5 IMPLEMENTATION OF GRAM

5.1 Overview of the Framework

Figure 3 shows the overview of GRAM. In our framework, given a single version of the program and an error threshold, we first create three extra program versions that will be used for mixing. Depending on the error supplied, the controller will decide which pair of program to be mixed together. In all of the benchmarks, the errors of the FP16 version and FP32 version are vastly different (error of FP16 version is usually more than $100\times$ FP32). Given an error threshold, we can clearly pick the two versions so the error threshold is between the high precision “high_prec” version and low precision “low_prec” version. Then, the automatic script in Figure 4 will construct the mixed precision version given the two versions of each kernel (“low_prec” and “high_prec”),

Fig. 3. Overview of GRAM.

Fig. 4. Mixing script to combine two versions of the same kernel.

using the policies in Section 4.4. Then, with a representative input and an error threshold, the user will run our searching script to determine α . Finally, the α value found is supplied as a program argument at runtime to achieve the desired output quality. The workflow is developed in Python. The construction of the “low_prec” kernel using FP16 uses our extension of the framework from Reference [16]. This module was developed from Clang LibTooling to rewrite CUDA kernels to a new half2 version that is using CUDA math library by default. The main modification is to include block-localized indexing as described in Section 4.3. The **fast_math** version is simply the output program with a new math header file inserted to extend the error/performance tradeoff region. The **fast_math** library will be introduced in Section 5.4. For the FP32-FP64 pair, we simply switch between type float and double in all variable declarations in the target kernels, which is straightforward. The input and output arrays are kept in their original precision. Hence, no additional global memory storage is required. However, more registers are needed to handle the conversions inside each low precision thread. This register usage, along with register reuse strategy and other optimizations from nvcc compiler, resulted in the speedups reported in Section 6.1.

For the extension of Reference [16], we reused the original code rewriting strategy. Reference [16] automatically finds all the floating point declarations in the kernel and puts them in a configuration file before rewriting. By default, all the variables will be marked to be converted to lower precision. In this extension, because we keep the original input and output in their higher precision, the tool needs to read such information from the same configuration file that specifies the variable names that belong to input and output of a kernel. From users’ perspective, the user can either remove the input/output variables from the “to_be_converted” list or add the list of variable to the exclusion list. *floats2half2_rn* and *half22float2* will be used when a variable in the exclusion list is read or written, respectively. We targeted a single kernel in one pass of rewriting,

Fig. 5. Runtime calibration for Blackscholes.

the kernel name is also supplied by user in the configuration file. The main rewriting method does not change from the original work in Reference [16]. More specifically, *RecursiveASTVisitor* is used to traverse the program and rewrite each type of statement to the predefined patterns that use half2 (e.g., when binary operator is detected, *VisitBinaryOperator* will be called to replace the current operation with the appropriate half2 equivalence).

After obtaining the two versions of the same kernel for mixing the precisions, we can use the mixing script to combine the two versions. An example is shown in Figure 4. Entire function bodies are copied, and the predefined if/else condition is added along with the new argument for the kernel ($int\alpha$). We implemented this functionality using straightforward string processing functions in the aforementioned Python script without the need of compiler tools.

5.2 Adapting with New Inputs and Error Thresholds

Previous works use expensive searching on a subset of variables, and therefore the result is dependent on having a representative input. To adapt to new input, the search has to be performed again, which makes runtime quality calibration or adapting to new input nearly impossible. In GRAM, to choose how many percent of operations should be done in lower precision requires at most seven re-executions of the program. Therefore, it can be quickly reused for new inputs set or error threshold with a few additional re-executions needed in Algorithm 1. We use a calibration method when running the program with new inputs similar to Sage [40]. The approximation degree is represented as α with maximum calibration overhead $\lceil \log_2(100) \rceil = 7$ steps for the resolution of 1% change in error. Furthermore, the resolution can be reduced to 10% change in α if marginal tradeoff is not necessary. This effectively reduced the number of steps to $\lceil \log_2(10) \rceil = 4$. As with Sage, the calibration should only happen after a significant number of executions or after the user decides to change the error threshold. This variable is called calibration interval, which was thoroughly studied in Sage [40]; we adopt a similar controller to GRAM. In Sage, it has been proven to effectively provide output quality under a certain confidence ($>90\%$) with minimal overhead depends on the calibration interval used. We adopt Sage's default scheme in GRAM: The calibration interval (the number of normal executions between two consecutive calibrations) increases by 10 after each calibration.

5.3 Runtime Precision Adaptation: An Example

Figure 5 shows runtime precision adaptation using the *Blackcholes* benchmark as an example. The L1norm error of the all-FP32 version on a random set of 20M inputs is $\approx 1.6 \times 10^{-7}$. We assume that the user wants *Blackcholes* to run with a L1norm that does not exceed $\approx 1.26 \times 10^{-7}$ (the error when $\alpha \approx 75\%$). GRAM automatically finds the approximation degree α in seven runs. During

Fig. 6. Relative speedup and average relative error of math functions in FP16 provided by CUDA and ours, compared to the fastest CUDA FP32 math functions.

tuning, apart from running with *mid* value of α in Algorithm 1, we always use the correct output. The subsequent calibration intervals happen according to Section 5.2. At each calibration, the error drops to 0, because we run the exact version to compare with the current tuned version. If the output error exceeds the original threshold, Algorithm 1 is applied again. Otherwise, the calibration ends, because changing α is not necessary. To change the error threshold, we reset the calibration by using Algorithm 1 similar to what happens at the first execution of the program. Figure 5 shows the error and the cumulative performance of Blackscholes during the first 100 executions of the above use case. Each of the 100 runs used a different inputs set generated by the original benchmark with a different random seed. All of the error values induced by these inputs do not exceed our defined error threshold. We observed a very small variation that is almost unnoticeable in Figure 5 in the error between different inputs. This example shows how users (or some autotuner) can freely switch between different error thresholds at *runtime* when using GRAM. It will adapt the subsequent executions of the same program accordingly.

5.4 Approximate FP16 Math Library

Because our target will be a broad approximable program domain, we also consider the speed gained in FP16 math functions, which is executed by the Special Functional Unit (SFU) in GPUs. Reference [16] pointed out that the SFU unit does not support half precision natively in Nvidia Pascal GPUs. To use FP16 math functions, the nvcc compiler will implicitly convert the input to FP32 precision, perform a FP32 SFU call, and convert the result back to FP16 precision. Therefore, this will slow down the half-precision kernels compared to FP32 if they use the slow math functions more often. We also observe the similar behavior in Volta and Turing by inspecting the generated SASS code using `nvdiasm` in CUDA 10. The upper half of Figure 6 shows the speedup of the supported half2 math function relative to FP32 (with `-use_fast_math=true`, which provides the fastest and most approximate math functions available) in CUDA 10. Using the blue dashed horizontal line at speedup = 1.0× as reference, we can clearly see that the CUDA's half2 math functions are slower than float in most of the cases. Note that *asin* and *acos* are still not available in CUDA half2 math library version 10.1 [33]; therefore, we did not measure their speed and error.

To overcome the issues with FP16 SFU, we develop an entire half-precision math library using all SIMD instructions, which is much faster than the math functions provided in CUDA 10. By replacing all conversion instructions and SFU calls with full SIMD half2 arithmetic instructions, our implementation comes with an acceptable error compared to naïve SFU calls, especially when we consider the fact that half precision programs are highly approximable. We converted all the approximate math functions from FP32 to the equivalent half2 bit patterns using the instruction provided in the respective source. The `rsqrt` and `sqrt` functions use the method in Reference [37], while `div` and `rcp` functions use the Newton-Raphson method [7]. The `exp` and `log` functions are the faster versions from References [29, 43]. For other functions, namely, `sin`, `cos`, `asin`, and `acos`, we used the standard Taylor expansions [1] and range reduction from Reference [29]. We chose these approximation schemes because they cause little divergence between the two halves of half2 (due to conditional statements). They are widely used in different domains from hardware design to machine learning. Figure 6 shows the relative speedup and error of our math functions compared to calling CUDA 10's standard math library using the half2 data type. The lower half of Figure 6 shows the average relative error compared to double precision math functions over 10^7 randomly generated values in the working range of each function, which does not cause arithmetic overflow in CUDA FP16 math library. The execution time is measured by microbenchmarking a Titan V GPU. The execution times of each function is averaged over five runs and are compared with the same number of math function calls with the “-use_fast_math” compiler flag. Note that this relative speedup is for reference only. Each math function consists of multiple instructions, and in real applications, there may be effects from latency hiding and resource contention in the GPUs. For a better insight, we will provide the actual speedup gained in real applications from several benchmark suites in Section 6.1. This library gives some benchmarks that are slower than FP32 a competitive chance. This fast_math library uses an API compatible to the half2 math functions in CUDA. Therefore, switching between fast math and CUDA math is simply done by replacing a header file. For quality control, our framework automatically detects whether the use of CUDA math or our fast math is appropriate according to user's supplied error threshold. The approximate math library and microbenchmarks are available online.²

6 EXPERIMENTS

6.1 Methodology

To validate our method, we conduct experiments on the Nvidia Titan V GPU, which supports half precision arithmetic natively [34]. The system and software used for performance measurement in this section, if not explicitly described, has installed CUDA version 10 with the host running Ubuntu 18.04. All performance values reported are the averages of five runs each. The variation between the measurements within the five runs of the same experiment is insignificant and is not reported. Unlike other mixed-precision tuning tools, we have a single version of program with runtime tunable approximation degree. We plotted the entire tunable region from all threads executing high precision to all threads executing low precision at the resolution of 10% of threads. Table 2 lists all benchmarks involved in this section. The main error metric is *Average_Rel.*, which is the average relative error. This metric is used widely in data-parallel applications on GPUs and in various approximation works [24, 40, 50]. Some benchmarks (e.g., *Lulesh*) have customized error measurement; therefore, we keep their original error metrics without alteration. For the first nine benchmarks, we are able to achieve a reasonable accuracy with half precision (<15% error); thus,

²<https://github.com/minhnh2910/fp16-fast-math>.

Table 2. List of Benchmarks Used in Our Experiments

Name	Domain	Input Size	Error metric	Ref
<i>billateral filter</i>	Image processing	512x512 image	Average Rel.	[35]
<i>nbody</i>	Simulation	65,536 bodies	Average Rel.	[35]
<i>matrix mult</i>	Linear algebra	3,200x6,400	Average Rel.	[35]
<i>lavaMD</i>	Molecular Dynamics	-boxes1d 20	Average Rel.	[2]
<i>kmeans</i>	Data mining	640,000 points	Average Rel.	[2]
<i>mri-q</i>	Medical imaging	64x64x64 dataset	Average Rel.	[44]
<i>cp</i>	Molecular Dynamics	100,000 atoms	Average Rel.	[44]
<i>inversek2j</i>	Robotics	2 [^] 20 coordinates	Average Rel.	[49]
<i>newtonraph</i>	Numerical analysis	2 [^] 20 equations	Average Rel.	[49]
<i>lulesh</i>	Hydrodynamics modelling	50x50x50	Symmetry	[20]
<i>blackscholes</i>	Financial	200M options	L1 Norm	[35]
<i>arclength</i>	Numerical analysis	k=5, n=1M	Absolute Error	[39]

Table 3. Speedup (on Nvidia Titan V) and Error Characteristics of Each Benchmark When $\alpha = 50\%$

	<i>billateral filter</i>	<i>nbody</i>	<i>matrix mult</i>	<i>lavaMD</i>	<i>kmeans</i>	<i>mri-q</i>	<i>cp</i>	<i>inversek2j</i>	<i>newtonraph</i>
Speedup	1.14	1.36	1.34	1.35	1.35	1.36	1.56	1.63	1.49
Average Rel. Error	0.0011	0.094	0.0074	0.0031	0	0.028	0.06	0.0056	0.0035
Error_std	0.009	0.1	0.007	0.0068	0	0.083	0.12	0.03	0.16

they are included in the mixing FP16 and FP32 experiments. For the remaining, we have to use FP32 as the lowest possible precision due to FP16's numerical limitations.

For clarity, we shall now give the definition of speedup and average relative error used in this section. Let us consider the programs in the experiments in Section 6.2 where we mixed the two versions of FP32 and FP16. Let T_{FP32} and T_{FP16} be the execution time of the program where all blocks are in FP32 and FP16, respectively. For any measurement, the execution time of the mixed program is $T_{measured}$. First, we define the *normalized runtime* ($T_{normalized}$), which will be used in Section 6.2 and Figure 8. The *normalized runtime* is calculated as in Equation (1). It simply normalizes the execution time by computing the ratio between the mixed precision program and the fastest version of the program (T_{FP16}). We use the normalized runtime in Figure 8 and all of our tradeoff curves, because it will produce the increasing trend when the degree of approximation increases. It avoids the overlap in plotting two similar increasing series of error and speedup when we increased the approximation degree.

$$T_{normalized} = \frac{T_{measured}}{T_{FP16}} \quad (1)$$

As was defined by previous works, *speedup* refers to the ratio between the reference and the mixed precision program. When the reference is FP32, we use the execution time T_{FP32} to compute the speedup as in Equation (2). This definition of speedup is used in Figure 9 and Table 3. Furthermore, Equation (3) shows the relationship between speedup and $T_{normalized}$ used the tradeoff chart in Figure 8. Note that the ratio T_{FP32}/T_{FP16} is actually the largest possible value of $T_{normalized}$ in each benchmark. To compute the speedup based on normalized runtime, we simply take the maximum value of the left y axis of each benchmark in Figure 8 and divided it by the corresponding

normalized runtime value.

$$speedup = \frac{T_{FP32}}{T_{measured}} \quad (2)$$

$$speedup = \frac{T_{FP32}}{T_{FP16}} \times \frac{1}{T_{normalized}} \quad (3)$$

For the definition of error, we used the error metric defined by the benchmarks (e.g., Lulesh, arclength) when available. For other applications that do not have any predefined error metric, we used the average relative error to report on the output quality, as shown in Equation (4). Suppose the reference program's output is an N -dimensional array A , while the approximated program yields an array A' that also has N elements. Let R_i be the relative error of the i th element, and the average relative error is the average of all the relative error values (all R_i). In rare cases where A_i is 0 and $|A_i - A'_i| > 0$, we simply set the relative error of that element (R_i) to 1.0. To normalize the error, we simply divide the measured *Average_Rel* with the maximum possible *Average_Rel* when all blocks are in FP16.

$$R_i = \frac{|A_i - A'_i|}{|A_i|}$$

$$Average_Rel = \frac{\sum_{i=0}^{N-1} R_i}{N} \quad (4)$$

To generalize to other pairs of precision (e.g., FP64:FP32), we simply replaced FP16 in the above equations with the lower precision in the pair. Likewise, FP32 is replaced by the higher precision in the pair.

6.2 Mixing FP16 and FP32 for Highly Approximable Programs

This section shows the trading-off curves for nine approximable benchmarks in Table 2. The benchmarks selected are compute-intensive floating-point applications from different domains. We measured the speed and error of FP16 version of each benchmark using CUDA standard library and using our fast math library described in Section 5.4.

In summary, our approximate math library opens up tuning opportunity in some benchmarks (*mri-q*, *inversek2j*, and *bilateral-filter*) where the standard CUDA math library actually slows down the half precision program. The cost is an extra amount of error. When half precision gives high error as in *nbody* with 9.9% and *cp* with 14%, the error caused by the inherent issues with half precision numbers (low precision, limited range) overshadow the extra error incurred by our approximate math library. Therefore, for benchmarks resulting in high output error, our math library is faster while maintaining similar accuracy. Note that the use of our math library is entirely optional. Users can easily change to any other, possibly more precise, math library in the future simply by replacing a header file. The error is controlled by checking the user-supplied error threshold; if our math library gives higher error in a particular error metric than requested by user, then we simply use the original CUDA library as the low precision version.

cp and *kmeans* have approximability beyond half precision. If we use the original error measurement in *cp*, which only considers one array of the output set, the error of that array is ≈ 0 when using half precision. In *cp*, we measured the error on the full set of output instead. For *kmeans*, the outputs are the centroids of clusters and the membership of data points in the clusters. We observe no error when using full FP16 for this benchmark. This is reasonable, because the output of *kmeans* is only affected by the comparison operations between the distances.

Figure 8 presents the tradeoff curve with the axes normalized for readability. For each program, the x-axis shows the α value mentioned in Section 4.4, i.e., the percentage of blocks in lower precision (from 0% to 100%). The **runtime** series connects all the measurements of normalized

Fig. 7. The effect of using our approximate math library and standard CUDA math library for FP16.

runtime (the minimum runtime equals 1.0). The **error** series connects all measurements of all normalized error values. Hence, for each α value, there are two measurements—the normalized runtime with the scale on the left y-axis and the normalized error induced scaled by the right y-axis. As expected, when we move from left to right, i.e., increase the percentage of blocks running in FP16, the normalized runtime will decrease while the error will increase. The maximum error values (1.0) on the right axis are equivalent to the numbers on top of Figure 7. When the reference output is in FP64, the error of FP32 is diminishingly small compared to FP16 (see Section 6.3.1). Therefore, for simplicity, we use minimum error ≈ 0.0 on all benchmarks. We use the fast math library for all applications where possible. In Figure 8, most applications show a fairly smooth tradeoff curve when the percentage of threads running in FP16 is changed. The error profile is also mostly linear, except for *nbody*, where the error increases much faster than the reduction in execution time. This is because *nbody* repeatedly computes the force from all bodies to each body. Therefore, the error induced by a small percentage of threads will propagate quickly to all threads.

To better understand the error characteristic and the relationship between speedup and Normalized Runtime, we show the error and its standard deviation (Error_std) in Table 3. The actual speedup derived from Figure 8 is also provided when $\alpha = 50\%$ with the formulae described in Section 6.1.

6.2.1 Efficacy across GPU Architectures. We ran the experiments on the Titan V (Volta architecture), Tesla P100 (Pascal architecture), and Tesla T4 (Turing architecture). Their FP32 throughput is half of the FP16 throughput [34]. The speedup values we measured in this section use the reference runtime of FP32 versions of the respective architecture. The P100 and Titan V systems used the same setup as they ran on our servers. For T4, we used a Google Cloud instance [17] with 2vCPUs, 7.5 GB RAM, and default installation of CUDA 10 and Ubuntu 16.04. To simplify the chart, we shall only report the speedup of the mixed precision program where $\alpha = 50\%$ of operation is in FP16 for each benchmark. The speedups are reported in Figure 9 while the normalized errors can be retrieved from Figure 8.

Fig. 8. Mixed precision tuning result for various benchmarks.

Fig. 9. Speedup of mixed-precision across different architectures with $\alpha = 50\%$.

6.3 FP32-FP64 Mixed Precision

6.3.1 Error Performance Tradeoff. This section presents the error performance tradeoff of FP32-FP64 pair. Because most of the tradeoff curves are similar to the FP16-FP32 pair, we present the results for the three benchmarks that have the highest errors in Figure 7. In addition, we also include some large benchmarks from related works to present in the next section. Most of these benchmarks could be considered to have unacceptably high error when running on half precision and therefore FP32-FP64 mixing is more viable for approximation. For complex application, we included LULESH [20], which is an application used by the US Department of Energy with more than 5,000 lines of CUDA code. It is considered a standard benchmark for mixed precision tuning by established works [21, 22, 27].

The results for the first three programs are presented in Figures 10(a), (b), and (c). We can clearly see that the error induced by FP32 is much (hundreds to thousands times) smaller than FP16. Therefore, there is no need for mixing between FP64 and FP16, as a single block of threads in FP16 can cause catastrophic error with insignificant performance improvements compared to an FP32 version.

Fig. 10. Mixed precision tuning between FP32 and FP64 in various benchmarks. In figures (d) and (e), the blue filled boxes represent the normalized runtime of other approaches while achieving the same output error as green boxes (GRAM) at the same x coordinate. When we consider a single vertical blue line connecting a blue box and a green box, the higher position of the blue box means that the normalized runtime of their mixed precision program is longer than GRAM while achieving the same output error.

6.3.2 Comparison with Other Precision Tuning Schemes. The most recent works on variable-wise or operation-wise precision tuning specifically for GPU are GPUMixer [22] and AMPT-GA [21]. ADAPT [27] and fpTuning [13] can also be used in the GPU setting via proxy applications in CPU code. All of these works showed results that are comparable and sometimes better than that of Precimonious [39], which was the state-of-the-art for precision tuning on CPU applications. With whatever is available in the public domain, we were able to reproduce the results of GPUMixer, AMPT-GA, and ADAPT for LULESH [20] and fpTuning for Blackscholes [35] on our system, with the input sizes of “-s 50” for LULESH and 20M options for Blackscholes, respectively. The results are presented in Figures 10(d) and (e).

We compared in total seven error thresholds for the three frameworks (three from GPUMixer, three from AMPT-GA and one from ADAPT) on LULESH. For readability in LULESH’s chart, we plotted the error threshold that achieved the highest performance compared to GRAM in each framework and plotted in Figure 10(d). The blue filled squares ■ are the normalized execution times (with values on the left y-axis) of the tuned version produced by the other works. We use “(1),” “(2),” “(3),” and “(4)” to denote the results for ADAPT, AMPT-GA, GPUMixer, and fpTuning, respectively. For LULESH, they are mainly in the region less accurate than three digits. For other higher accuracies, the variable-wise approaches cannot find good configurations with the given input size of “-s 50.” For BlackScholes, we choose the threshold to be 25%, 50%, and 75% of the error induced by the all-FP32 version given by Nvidia (with an L1 norm error of 1.6×10^{-7}).

For our method to work on large programs, in this section, we create a simple kernels profile estimating runtime reduction and error increase when converting each kernel to FP32 separately. This approach is used in AMPT-GA and GPUMixer to narrow down the list of kernels that can benefit from precision tuning. In case of LULESH, the top three kernels contribute to $\approx 80\%$ of the application’s runtime. The target kernel for mixing precision is *CalcKinematicsAndMonotonicQ-Gradient*. Tuning other kernels results in minimal performance gain while inducing significant additional error.

In summary, for these error thresholds, GRAM can create a version of the code that is up to 15% faster than that produced by these state-of-the-art precision tuning approaches. This improvement can be seen in Figure 10(d), where the blue square denotes the normalized runtime of ≈ 1.2 while GRAM’s normalized runtime is at ≈ 1.05 ; both versions give the same output error on the right

y-axis. Furthermore, GRAM only needed to execute the program seven times to do this, while other searching methods mostly require hundreds of executions, the lowest being 61 executions when using GPUMixer for LULESH. Although we do not exclude the possibility that GRAM can be slower than other approaches in certain cases, GRAM stands out, as its complexity is deterministic and is much faster than other approaches most of the time. The only error threshold where the variable-wise approaches achieve slightly higher performance (2% improvement) than GRAM is in LULESH at $\alpha = 97\%$ when the error equals 3.6, which is near the maximum error achieved by having all variables in FP32.

6.3.3 Synergy with Variable-wise Precision Tuning. GRAM can be combined with variable-wise analysis. Consider the *arclength* example in Listing 1. By default, the method can generate a tradeoff region that is $[1.0, 1.26]$ normalized runtime and $[0, 1]$ normalized error between FP32 and FP64. Since this is a small program, by exhaustive search, we can get an “Optimal” version with a normalized error and runtime of 0.28 and 1.02, respectively, with two of the variables in FP32, the other two in FP64. In the region of error of $[0, 0.28]$, any variable-wise method has to pick the 0.0 error version, because there are no configurations with an error threshold between 0.0 and 0.28. In this case, GRAM can create a mixed version between “Optimal” and FP64 with a normalized runtime in $[1.02, 1.26]$ (see Figure 10(f)). Interestingly, this version is faster with the same error amount than either our method or variable-wise approximation alone in the $[0, 0.28]$ error region.

7 RELATED WORK

The problem of mixed-precision tuning and approximate computing have been studied for the past few decades [30, 46, 48]. In this section, we only review a relevant subset.

For CPU applications, one of the early attempts was EnerJ [41]. It was followed by the introduction of dynamic techniques such as Precimonious [39] and more recent works [3, 9, 38]. Precimonious can analyze fairly mid-size programs having 50 variables. Recently, ADAPT [27] was introduced to scale to HPC applications. There are also rigorous analyses [4, 5, 18] that can give exact error bounds or calculate the sensitivity of variables to approximation error. However, scalability remains an issue in such approaches. Moreover, these methods do not work on CUDA’s paradigm.

FPGAs and customized ASIC provides the means to create customized hardware for any precision. Hence, there are many techniques to find an optimized representation for each variable or operation in FPGA applications. Some notable works on the static and hybrid analysis are References [6, 23]. More recently, more dynamic approaches solve the scalability of fine-grained precision allocation on larger programs, with some supporting CPU programs [13, 45]. Notably, bit-level reduced precision was recently exploited for the deep neural network inference where even FP16 is too precise [10, 14, 15, 19, 25, 36].

For GPU applications, there were different approximation techniques for memory reads [51], memory refresh reduction [31, 32], task skipping including threads fusion [47] and loop/thread perforation [26], SFU [24], neural accelerators [50], or combined different approaches [40]. Compared to these, GRAM has a lower error tradeoff than other more aggressive approximation approaches. More recently, some works target GPU floating point programs and half precision [11, 28, 42]. Unfortunately, they only support tensor core operations, which are not applicable to normal programs. Recently, ADAPT [27], AMPT-GA [21], and GPUMixer [22] are able to analyze GPU programs. Some of these preliminary experiments can work on half type, which mostly improves speedup in memory-bound applications. However, they cannot fully exploit FP16 (in half2 type) without using thread fusing as in our work. The performance of half precision

in most general applications is much slower when using half compared to half2 as reported in Reference [16]. Variable-wise or operation-wise approaches rewrite the program to be used identically on all threads. To the best of our knowledge, fully utilizing half2 interchangeably with float or double in a single thread is a hard problem, because we need to gather pairs of the same operations to use the SIMD half2 instructions.

8 LIMITATIONS AND FUTURE WORK

8.1 Program Characterization

Although our method is fast and deterministic, it works more effectively with certain types of programs. GRAM works well with applications that are compute-intensive and use average error as the standard error metric. For application-specific error, we showed that the tradeoff may not be in the ideal form of near-linear relationship as when we use average error.

Besides, some applications may be sensitive on local error. One example happens when half precision blocks produce too high local error that is beyond user's defined tolerance. Thus, we recommend using GRAM on application that has either specific error metric that can be relied on. Otherwise, the user must be aware of the worst-case error caused by half precision. Such error is application-dependent and can be estimated by running the whole program in half precision using our method.

Although GRAM may not work in all programs, we believe its simplicity can make it a good candidate to apply before trying much more complex analyses, which could take hours or days to complete. In the future, we plan to explore more complex methods to solve the above issues by possibly integrating variable-wise analysis into the framework as described in Section 6.3.3, where the variable-wise solution ensures the worst-case error requirement if requested by the user.

8.2 Block Allocation Strategy

As described in Section 4.4, there is a variance in the output error between different block allocation schemes. The variance can be observed when supplying a fixed input array and changing the *if-then-else* condition statement in Listing 2. Between the two default allocation schemes, namely, *scattered* and *clustered*, we observed that the variance vastly depends on each application. More specifically, given a fixed input, switching allocation scheme may change the normalized error to up to 10% in *nbody* and *cp*. The variance is much smaller in other benchmarks. Thus, the variance in error when using different allocation schemes is a known issue of our approach. It is conceivable that, given a fixed input, there will be an optimal blocks mapping scheme for each α value. When we change the input (reading new input files or generating new inputs set), such allocation is also changed. Thus, the overhead to analyze and derive the optimal allocation scheme for every unique input overshadows the performance gained by the mixed precision program. Searching for the best block allocation scheme becomes the problem of considering the tradeoff between gaining accuracy ($<10\% * 10\%$ maximum error = 1% actual error) and the possible overhead the search may incur.

In a more realistic setting, we observed that the inputs are always changing. Although the variance exists, it becomes less noticeable when we consider the average error over hundreds or thousands of runs with different inputs. It is the case in our experiment in Section 5.3 and Figure 5. In summary, the current scheme supported by GRAM is only suboptimal when we consider each input alone. The search for each optimal block allocation scheme for each unique input becomes less relevant and less beneficial when considering the runtime adaptation scenario in Section 5.3. Nonetheless, GRAM will benefit from future proposal on a runtime method to quickly identify better allocation scheme than our default schemes.

8.3 Multi-kernel and Multi-precision Combination

8.3.1 Multi-kernel Mixed Precision. The multi-kernel scenario has been studied in previous works such as GPUMixer. The current idea is that we only need to focus on a subset of kernels that contributes the most to runtime and accuracy. The default profiler tool (*nvprof*) from Nvidia can identify such kernels. Thus, GRAM can leverage the profiling technique on top of our framework to select the subset of kernels for processing. After profiling the list of top kernels, we simply apply the same approximation degree α to all of the kernels. As our benchmarks mostly reduce to 1–2 top kernels, we currently support setting the same α for all kernels as default.

If we consider the output of each kernel independently and let the user decide on how much error of each kernel can be tolerated, the problem in multi-kernel mixed precision allocation is reduced to multiple single-kernel mixed precision. Such problem is well solved by GRAM and only takes around seven runs to analyze each kernel.

When we consider a combined output after passing through multiple kernels, the problem is more sophisticated. We can either set the same α for all kernels and achieve smooth overall tradeoff like in Section 6.3.1. If we assume the final output error is positively correlated with α as in the single kernel method, we can simply use Algorithm 1 to find the suitable α for each kernel. This approach costs approximately $\log_2(10^N)$ runs, assuming N is the number of kernels and 10 levels of α is supported for each kernel. However, this approach may not give the optimal α for each value when the impact of each kernel to the final output error is not trivial. When the positive correlation cannot be assumed, this problem could have the difficulty of searching in the search space of 10^N possibilities.

In summary, we currently support setting the same α for the top kernels profiled by “nvprof” as in GPUMixer. Although this approach gives a smooth tradeoff curve when changing α , it cannot guarantee that the mixed precision program constructed is always the optimal choice. We plan to study sophisticated method of analyzing multiple kernels independently in our future work.

8.3.2 More Than two Precision Levels. For mixing more than two precision levels. GRAM can support adding extra if-else condition to Listing 2 to achieve the required effect. For example, if we consider three types, there will be two different α_1 and α_2 values to identify which block runs which data type. α_1 controls which blocks run in FP16, α_2 controls which blocks run in FP32, and the rest will run in FP64. Assume that we allow M levels of approximation degree on each α_1 and α_2 , the tuning complexity is approximately $\log_2(M^2)$.

In the real benchmarks, the output quality is vastly different between FP32, FP64, and FP16 and such three types mixing does not give any clear advantage to GRAM. More specifically, our matrix multiplication benchmark shows that the relative error of FP32 is 0.000035, while the relative error of FP16 is around 0.014768 compared to the FP64 version. If we consider the normalized error of FP16 is 1.0 as in Figure 8, the normalized error of FP32 is ≈ 0.00237 . In performance, FP32 is 1.72 $\times$ faster than FP64. Thus, replacing blocks in Figure 8 from FP32 to FP64 only gives < 0.00237 changes in error while reducing the performance significantly.

The effect can be seen in Figure 11 when we measure the speedup of mixed precision program compared to FP64 version. The *speedup_FP32* columns show the speedup of the mixed precision program when the corresponding $\alpha\%$ of blocks in FP16 and the remaining blocks are in FP32. The *speedup_FP64* columns show the speedup when the higher precision blocks use FP64 instead of FP32. This clearly shows that there is no visible advantage in terms of speedup and error when using FP64 blocks in the mixed precision constructed between FP16 and FP32.

When any block of FP16 presents, switching any of the remaining blocks from FP32 to FP64 does not give significant changes to the final output error while inducing extra overhead due to

Fig. 11. Speedup and normalized error of *matmul* benchmark when the higher precision blocks in FP32 are replaced by FP64. Because the errors of the two versions completely overlap each other in the plot, we only plot one error series for readability.

computation in FP64. This explains our design of GRAM to select two appropriate precision levels before running the mixing script instead of including every datatype choice to the final program.

9 CONCLUSION

We presented GRAM, a framework that mixes precision for approximable programs on GPUs. We first introduce our data-wise precision tuning techniques that allocate different precision to different data elements instead of the entire arrays. We then build the GRAM framework, which supports the wide spectrum of error-performance tradeoff spread from half precision to double precision. To accelerate some approximable benchmarks, we include the fast math library as an option for GRAM to further extend the performance improvement. We have also shown that GRAM is effective on its own or as a complement to other techniques for the purpose of mixed precision tuning of CUDA programs. Furthermore, it works on all architectures that support FP16 natively. What is even more novel is that GRAM can adaptively change the mix of precision at runtime, all with no additional hardware. We believe that GRAM opens up new possibilities in adaptive, system-wide approximation. The framework is available under an open source license.

REFERENCES

- [1] Claudio Brunelli, Heikki Berg, and David Guevorkian. 2009. Approximating sine functions using variable-precision Taylor polynomials. In *Proceedings of the IEEE Workshop on Signal Processing Systems*. IEEE, 057–062.
- [2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, 44–54.
- [3] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. 2020. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.* 17, 2 (2020), 1–26.
- [4] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. *ACM SIGPLAN Not.* 52, 1 (2017), 300–315.
- [5] Himeshi De Silva, Andrew E. Santosa, Nhut-Minh Ho, and Weng-Fai Wong. 2019. ApproxSymate: Path sensitive program approximation using symbolic execution. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 148–162.
- [6] Claire Fang Fang, Rob A. Rutenbar, Markus Püschel, and Tsuhan Chen. 2003. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In *Proceedings of the 40th Design Automation Conference*. ACM, 496–501.
- [7] Michael J. Flynn. 1970. On division by functional iteration. *IEEE Trans. Comput.* 100, 8 (1970), 702–706.
- [8] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. 2016. Auto-tuning for floating-point precision with discrete stochastic arithmetic. <https://hal.archives-ouvertes.fr/hal-01331917>.

- [9] 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.
- [10] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning*. 1737–1746.
- [11] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 47.
- [12] Mark Harris. 2016. Mixed-Precision Programming with CUDA 8. Retrieved from <https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/>.
- [13] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Ansoosheh. 2017. Efficient floating point precision tuning for approximate computing. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17)*. IEEE, 63–68.
- [14] Nhut-Minh Ho, Duy-Thanh Nguyen, Himeshi De Silva, John L. Gustafson, Weng-Fai Wong, and Ik-Joon Chang. 2021. Posit arithmetic for the training and deployment of generative adversarial networks. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'21)*. IEEE.
- [15] Nhut-Minh Ho, Ramesh Vaddi, and Weng-Fai Wong. 2019. Multi-objective precision optimization of deep neural networks for edge devices. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. IEEE, 1100–1105.
- [16] Nhut-Minh Ho and Weng-Fai Wong. 2017. Exploiting half precision arithmetic in Nvidia GPUs. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'17)*. IEEE, 1–7.
- [17] Google Inc. 2020. Google Cloud Platform. Retrieved from <https://cloud.google.com/nvidia/>.
- [18] Anastasiia Izycheva and Eva Darulova. 2017. On sound relative error bounds for floating-point arithmetic. In *Proceedings of the 17th Conference on Formal Methods in Computer-aided Design*. FMCAD Inc., 15–22.
- [19] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.
- [20] Ian Karlin, Jeff Keasler, and J. R. Neely. 2013. *Lulesh 2.0 Updates and Changes*. Technical Report. Lawrence Livermore National Lab (LLNL), Livermore, CA.
- [21] Pradeep V. Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. 2019. AMPT-GA: Automatic mixed precision floating point tuning for GPU applications. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, 160–170.
- [22] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, and Saurabh Bagchi. 2019. GPUMixer: Performance-driven floating-point tuning for GPU scientific applications. In *Proceedings of the International Conference on High Performance Computing*. Springer, 227–246.
- [23] D.-U. Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. 2006. Accuracy-guaranteed bit-width optimization. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 25, 10 (2006), 1990–2000.
- [24] Ang Li, Shuaiwen Leon Song, Mark Wijtvlit, Akash Kumar, and Henk Corporaal. 2016. SFU-driven transparent approximation acceleration on GPUs. In *Proceedings of the International Conference on Supercomputing*. ACM, 15.
- [25] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *Proceedings of the International Conference on Machine Learning*. 2849–2858.
- [26] Daniel Maier, Biagio Cosenza, and Ben Juurlink. 2018. Local memory-aware kernel perforation. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 278–287.
- [27] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 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.
- [28] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).
- [29] Paul Mineiro. 2017. fastapprox. Retrieved from <https://github.com/romeric/fastapprox>.
- [30] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Comput. Surv.* 48, 4 (2016), 62.
- [31] Duy-Thanh Nguyen, Nhut-Minh Ho, and Ik-Joon Chang. 2019. St-DRC: Stretchable DRAM refresh controller with no parity-overhead error correction scheme for energy-efficient DNNs. In *Proceedings of the 56th Design Automation Conference*. 1–6.
- [32] Duy-Thanh Nguyen, Chang-Hong Min, Nhut-Minh Ho, and Ik-Joon Chang. 2020. DRAMA: An approximate DRAM architecture for high-performance and energy-efficient deep training system. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD'20)*. IEEE, 1–8.
- [33] Nvidia. 2020. CUDA half2 math functions. Retrieved from <https://docs.nvidia.com/cuda/cuda-math-api/>.

- [34] Nvidia. 2020. CUDA instruction throughput. Retrieved from <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions>.
- [35] Nvidia. 2020. CUDA Samples Reference Manual. Retrieved from https://docs.nvidia.com/pdf/CUDA_Samples.pdf.
- [36] Salonik Resch, S. Karen Khatamifard, Zamshed Iqbal Chowdhury, Masoud Zabihi, Zhengyang Zhao, Jian-Ping Wang, Sachin S. Sapatnekar, and Ulya R. Karpuzcu. 2019. PIMBALL: Binary neural networks in spintronic memory. *ACM Trans. Archit. Code Optim.* 16, 4 (2019), 1–26.
- [37] Matthew Robertson. 2012. *A Brief History of invsqrt*. Department of Computer Science & Applied Statistics.
- [38] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1074–1085.
- [39] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*. IEEE, 1–12.
- [40] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture*. ACM, 13–24.
- [41] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Not.* 46 (2011). ACM, 164–174.
- [42] Anumeena Sorna, Xiaohu Cheng, Eduardo D'Azevedo, Kwai Won, and Stanimire Tomov. 2018. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In *Proceedings of the IEEE 25th International Conference on High Performance Computing Workshops (HiPCW'18)*. IEEE, 3–7.
- [43] Ian Stephenson. 2020. A Fast Power Function. Retrieved from <http://www.dctsystems.co.uk/Software/power.html>.
- [44] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Cent. Reliab. High-perf. Comput.* 127 (2012).
- [45] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benin. 2018. A transprecision floating-point platform for ultra-low power computing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. IEEE, 1051–1056.
- [46] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2015. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd Design Automation Conference*. ACM, 120.
- [47] Daniel Wong, Nam Sung Kim, and Murali Annavaram. 2016. Approximating warps with intra-warp operand value similarity. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, 176–187.
- [48] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2016. Approximate computing: A survey. *IEEE Des. Test* 33, 1 (2016), 8–22.
- [49] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Des. Test* 34, 2 (2017), 60–68.
- [50] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural acceleration for GPU throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 482–493.
- [51] Amir Yazdanbakhsh, Bradley Thwaites, Hadi Esmaeilzadeh, Gennady Pekhimenko, Onur Mutlu, and Todd C. Mowry. 2016. Mitigating the memory bottleneck with approximate load value prediction. *IEEE Des. Test* 33, 1 (2016), 32–42.

Received March 2020; revised October 2020; accepted November 2020