GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications *

Ignacio Laguna[1], Paul C. Wood[2], Ranvijay Singh[3], and Saurabh Bagchi[4]

[1] Lawrence Livermore National Laboratory, Livermore CA 94550, USA
ilaguna@llnl.gov
[2] Johns Hopkins Applied Physics Lab, Laurel MD 20723, USA
[3] NVIDIA Corporation, Santa Clara CA 95051, USA
[4] Purdue University, West Lafayette IN 47907, USA

Abstract. We present `GPUMixer`, a tool to perform mixed-precision floating-point tuning on scientific GPU applications. While precision tuning techniques are available, they are designed for serial programs and are accuracy-driven, i.e., they consider configurations that satisfy accuracy constraints, but these configurations may degrade performance. `GPUMixer`, in contrast, presents a *performance-driven* approach for tuning. We introduce a novel static analysis that finds *Fast Imprecise Sets (FISets)*, sets of operations on low precision that minimize type conversions, which often yield performance speedups. To estimate the relative error introduced by GPU mixed-precision, we propose shadow computations analysis for GPUs, the first of this class for multi-threaded applications. `GPUMixer` obtains performance improvements of up to 46.4% of the ideal speedup in comparison to only 20.7% found by state-of-the-art methods.

1 Introduction

GPU accelerated computing has reached a tipping point in the high-performance computing (HPC) market. As HPC scientific applications increasingly rely on GPU accelerators to perform floating-point arithmetic, tools to extract the maximum performance out of floating-point intensive computations are also becoming increasingly important.

This paper presents `GPUMixer`, the first tool to tune floating-point mixed-precision scientific applications on GPUs. While most mission-critical scientific applications use double-precision floating-point arithmetic (FP64) because of accuracy requirements, current generations of GPU architectures have higher peak computation rates in single-precision floating-point arithmetic (FP32) or lower precision [19]. To take advantage of the performance that lower precision offers, programmers can use mixed-precision computing to perform some computations in high precision (e.g., FP64) and some in low precision (e.g., FP32, or lower). `GPUMixer` provides a practical method to select the computations to be performed on FP32 or FP64 precision so that (a) user-defined accuracy constraints are maintained and (b) performance is significantly improved.

* This work was performed when P. C. Wood and R. Singh were at Purdue University.

Tuning mixed-precision programs is challenging. Programmers are interested in finding mixed-precision *configurations*, i.e., sets of operations on more than one precision, that satisfy both accuracy and performance demands. However, because the number of possible configurations is very large, manually exploring all configurations is impractical, even in small programs. For FP64/FP32 mixed-precision programs, for example, the number of possible configurations is 2^N, where N is the number of floating-point arithmetic operations.

In the domain of serial applications, a number of techniques for automatic tuning have been proposed to address this problem [3,4,6,11,23,22,13,7,16], however, practical and efficient tuning tools for multi-threaded applications are scarce, making mixed-precision programming for GPUs hard.

Irrespective of their architectural focus, a limitation of the majority of these methods is that they focus on mixed-precision tuning with accuracy as a target. That is, the configuration space search is driven by accuracy constraints in the program solution. We call this methods *accuracy-driven* approaches. Because performance is not explicitly modeled, these approaches have the disadvantage of suggesting configurations that provide *no performance guarantees*, and in many cases configurations that degrade performance.

`GPUMixer`, on the other hand, is designed as a *performance-driven* approach. We introduce the concept of *Fast Imprecise Sets (FISets)*, a set of arithmetic operations in a GPU kernel on which the data that enters and that leaves the set is in high precision, but on which the operations of the set are in low precision (hence an *imprecise* set). A *FISet* has the property that the ratio of arithmetic operations to cast operations is high; thus, an *FISet* is a configuration that, almost always, yields performance speedups (hence a *fast* set).

We demonstrate that *FISets* can be found via static analysis, which eliminates the need for running configurations to determine whether they provide performance speedup or not, as existing techniques do (e.g., [7,16]). Our algorithm for finding *FISets* locally maximizes the number of low-precision arithmetic operations while it minimizes the number of type cast operations in input/output boundaries of operation sets.

To find the *FISets* that also satisfy accuracy requirements, we perform *shadow computations*, a dynamic analysis that calculates an approximation of the relative error introduced when the precision is decreased from FP64 to FP32 in GPU kernels. While previous shadow computations techniques exist to tune serial programs [22,13], to the best of our knowledge, we present the first shadow computations framework for multi-threaded/GPU programs.

The contributions of this paper are:

1. We introduce the concept of *FISets*, floating-point configurations that provide performance speedups, and present an algorithm to find them statically.
2. We describe an implementation of our algorithm in the LLVM compiler [14] for the NVIDIA CUDA programing model. We show that our method can be applied efficiently to realistic multi-kernel GPU programs.

3. We implement the first GPU shadow computations framework for mixed-precision tuning, a dynamic analysis to compute the relative error introduced when the precision of FP64 operations is decreased to FP32.
4. We evaluate our implementation in three computationally intensive CUDA programs. We show that our approach finds configurations that are always faster than the default (all in FP64) for a given error threshold and input.

We compare our approach to the Precimonious approach [23,22], a state-of-the-art method for mixed-precision tuning on serial programs. On our evaluation, our approach finds performance speedups that can vary between 9.8%–46.4% of the maximum speedup, whereas the comparison approach (Precimonious) finds speedups of only 1.4%–20.7%.

2 Related Work

Formal Methods. FPTuner [3] is a rigorous approach for precision tuning based on Symbolic Taylor Expansions and interval functions; FPTuner meets error thresholds across all program inputs, however it has been demonstrated only on small programs and it has limitations handling conditional expressions. Rosa [6] is a source-to-source compiler that uses an SMT solver to annotate a program with mixed-precision types; the compiler operates on the Scala programming language. Paganelli and Ahrendt [21] present an approach that formally proves that an increased precision in a variable causes only a limited change of the result; it uses SMT solvers and is demonstrated on FPhile, a toy sequential imperative language. Other formal methods include Salsa [5] and S3FP [4].

Although these methods perform rigorous analysis and can verify properties for all inputs, they scale poorly and/or do not support common HPC programing languages (C/C++) and coding patterns (branches and loops), thus their applicability to realistic HPC applications is limited.

Heuristics for Automated Search. These methods cannot prove properties but they are able to scale to real-world programs, and as a result have broader practical utility. Our approach falls in this category.

CRAFT [11,12] performs an automated search of a program's instruction space, determining the level of precision necessary in the result of each instruction to pass a user-provided verification routine assuming all other operations are done in high precision, i.e., FP64. While it uses heuristics to sample a fraction of the search space, it can be very time consuming even for very small programs (worst case complexity is $O(2^N)$). Precimonious [23] uses the delta-debugging algorithm to search for configurations. While this algorithm helps in speeding up the search, this can still lead to a high number of builds and runs of the program. Blame Analysis [22] finds configurations that satisfy user-given error constraints, using a *single execution of the program*. The analysis finds a set of variables that can be in single precision, while the rest of the variables are in double precision; however, the output configurations may or may not improve performance, so to use the analysis in practice one must perform runs of the program to determine which configurations actually improve performance. The

experiments in [22] use `Precimonious` to perform the program runs in a guided manner. `ADAPT` [16] uses algorithmic differentiation to provide estimates about the final output error, which can be used for mixed-precision tuning.

The above techniques are *accuracy-driven* approaches, i.e., the configuration space search is driven by accuracy constraints in the program solution. Because performance is not explicitly modeled—the cost of operations is seen as a black box—these approaches may suggest configurations that provide no performance guarantees. `GPUMixer`, in contrast, is driven by performance gains.

A recent approach, HiFPTuner [7], considers performance by avoiding frequent precision casts on program variables. This approach, however, focuses on serial programs and is not available on GPUs and/or CUDA. Another difference is that [7] requires dynamic profiling to build a weighted dependence graph of the program, which is non-trivial to build efficiently on CUDA. One of the challenges to gather the per-instruction error introduced on multi-threaded code is to do it with reasonable overhead (one of the problems that we solve partially in our shadow computation framework). Because of the above limitations, we compare our method to the Precimonious method [23,22] instead of comparing it to [7]. The Precimonious approach (via delta debugging) is a more generic approach that can be easily adapted to GPUs (see Sec. 5.1 for more details).

3 Background and Overview

3.1 Example of Mixed-Precision Tuning

To illustrate the problem of mixed-precision tuning, we present an example using a CUDA kernel from an N-body simulation [18]. Listing 1.1 shows an implementation of the force calculation in an n-body simulation obtained from [8]. After the kernel calculates the forces and velocities of particles, the positions of the particles, `x`, `y`, and `z`, are updated in the main function.

Table 1: Error and speedup for different configurations of Listing 1.1 on a NVIDIA Tesla P100 GPU

Case	x	y	z	Error	Speedup (%)
1	-0.599775587166981	-0.906326702752302	-0.217694232807352		
2	-0.508669376373291	-0.906326711177825	-0.217694222927093	15.19	53.70
3	-0.575293909888785	-0.906326702752302	-0.217694232807352	4.08	5.78
4	-0.611327409124778	-0.906326702752302	-0.217694232807352	1.93	-43.35
5	-0.588951610438680	-0.906326702752302	-0.217694232807352	1.80	11.69

We perform mixed-precision tuning on the kernel variable declarations to find a configuration that yield both accurate results and a performance speedups. The *baseline* configuration is where all variables are declared as FP64, i.e., as `double`, the one shown in Listing 1.1. To illustrate the calculation of the error introduced by mixed-precision, we focus on the error introduced to the particle positions (`x`, `y`, and `z`) for a single particle (`i=0`). Programmers of scientific codes may

```
__global__ void bodyForce(double *x, double *y,
 double *z, double *vx, double *vy, double *vz,
 double dt, int n)
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if (i < n) {
  double Fx=0.0; double Fy=0.0; double Fz=0.0;
  for (int j = 0; j < n; j++) {
   double dx = x[j] - x[i];
   double dy = y[j] - y[i];
   double dz = z[j] - z[i];
   double distSqr = dx*dx + dy*dy + dz*dz + 1e-9;
   double invDist = rsqrt(distSqr);
   double invDist3 = invDist * invDist * invDist;
   Fx += dx*invDist3; Fy += dy*invDist3; Fz += dz*invDist3;
  }
  vx[i] += dt*Fx; vy[i] += dt*Fy; vz[i] += dt*Fz;
 }
}
```

Listing 1.1: Force computation in an N-body simulation

define their own metric for error, however, for this illustrative case, we define the relative error introduced by mixed-precision as: $error = (|(x - x_0)/x| + |(y - y_0)/y| + |(z - z_0)/z|) * 100.0$, where x, y, z are the particle positions for the baseline, and x_0, y_0, z_0 are the particle positions for a new configuration.

Table 1 shows the particle values, error, and performance speedup of four configuration with respect to the baseline, case 1. Case 2 shows the configuration where all variables in the kernel are declared as FP32, i.e., as `float`. We observe that while the speedup is significant, 53%, the error is high, 15.19. Case 3 shows the case where only variable `invDist3` is declared as FP32 and the rest as FP64—in this case the error decreases, but the speedup is not too high, only 5%. Case 4 shows an interesting case: when the variable `invDist3` is the only one declared as FP32, the error is very low, but the speedup is negative, i.e., performance degrades. Case 5 shows the best we found when the `distSqr,invDist`, and `invDist3` variables are declared as FP32: the error is lower than as in case 4 while the speedup is about 11%. This example illustrates that some configurations can produce low performance speedup or even performance degradation; the goal of our approach is to find via static analysis configurations such as 3 and 5 that improve performance and discard cases such as 4.

3.2 Configurations

While mixed-precision configurations can be expressed in terms of the precision of variable declarations (as in the previous example), a more precise approach is to express configurations in terms of the precision of floating-point operations. The reason behind this is that a variable can be used in multiple floating-point operations; the precision of each of these operations can be decreased/increased.

More formally, given a program with N floating-point arithmetic operations and two classes of floating-point precision, e.g., FP32 and FP64, a *configuration*

Table 2: Classes of configurations of a program

Satisfy accuracy constraints	Improve performance	Class
Yes	Yes	A
Yes	No	B
No	Yes	C
No	No	D

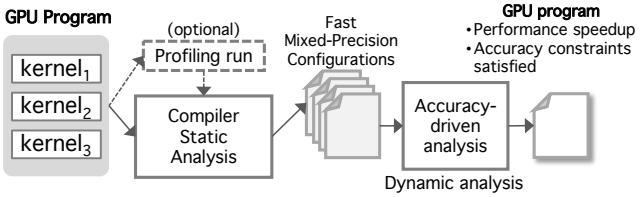


Fig. 1: Workflow of the approach.

is a set of operations on which a subset of n_1 operations are executed in one precision and a subset of n_2 operations are executed in another precision, such that $n_1 + n_2 = N$. For k classes of floating-point precision, subsets of operations can be executed on different precision, such that $n_1 + n_2 + \ldots + n_k = N$. Table 2 shows the four possible classes of mixed-precision configurations. The goal is to automatically find configurations that belong to class A.

3.3 Overview of our Approach

Figure 1 shows the overview of our approach. Given a GPU program, we optionally run a profiling run to determine kernels on which precision reduction can potentially give the highest performance benefits, e.g., by analyzing the kernels where the application spends most of its time or kernels that are computationally intensive. Note that this step is **optional**—if the programmer is not interested in profiling the application, our method analyzes all kernels.

Next, the compiler transforms kernels into an intermediate representation and searches for code regions where precision reduction could speedup the program execution, i.e., *FISets*. For each identified case, the compiler automatically performs code transformations and generate a program configuration. This configuration will likely yield a performance speedup when executed, thus it belongs to $\mathcal{P} = A \cup C$ (see Table 2).

Finally, since some of the configurations in $\mathcal{P}$ may not satisfy the user accuracy constraints, configurations must be analyzed to identify those that satisfy such constraints. Note that the user is free to use any existing accuracy-driven tuning method that is available in conjunction with *FISets*. However, since there is no accuracy-driven analysis available for GPUs, we develop our own method (shadow computations for GPUs), to fill this gap.

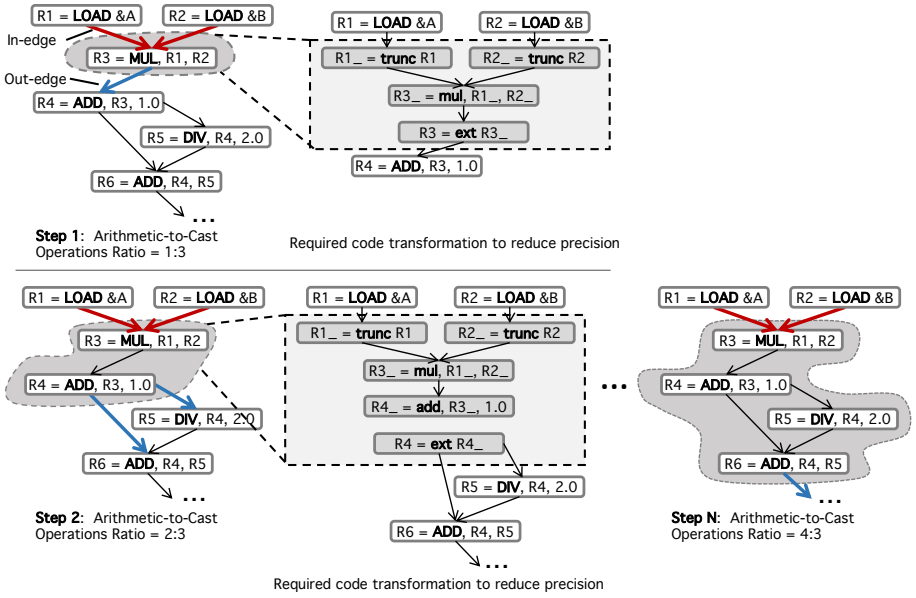


Fig. 2: Illustration of the algorithm to find *FISets*.

4 Approach

We describe our approach to model performance of mixed-precision code regions via static analysis with *FISets*, and our shadow computations approach to compute the error of mixed-precision configurations in GPUs.

4.1 Kernel Intermediate Representation

We use the NVVM IR [17] as the intermediate representation for GPU kernels. This representation is based on the LLVM IR and allows us to use high-level language front-ends, such as Clang to generate NVVM IR. Our approach performs transformation on the NVVM IR, a binary format to represent CUDA kernels.

4.2 *FISet* Design

The base working abstraction of *FISets* is a *data dependence graph* $G = (V, E)$. This is a directed graph whose nodes V represent NVVM IR instructions whose edges E represent dependencies between nodes. We assume that the compiler (in our case, LLVM) generates a data dependence graph for each kernel.

Roughly speaking, a *FISet*, which we denote as Φ, is a group of operations $v \in V$ on which the data that enters and that leaves the group is in high precision, i.e., FP64, and on which the operations that compose the group are in low precision, i.e., FP32. A *FISet* can contain both arithmetic floating-point operations and

non-arithmetic operations, such as comparison or select operations, that join together groups of arithmetic operations.

Type Conversions. Any mixed-precision approach incurs type conversion operations, or *casting*, to transform data from one precision to another. Type conversions are expensive in GPU architectures. Our algorithm to find a Φ in a kernel attempts to minimize the number of conversion operations and to maximize the number of floating-point operations in the set.

A key idea of the algorithm is that if we can perform conversions only *at the beginning and at the end* of a large sequence of floating-point operations that have high degree of dependence among them, we can increase the ratio of arithmetic-to-cast operations, therefore increasing the arithmetic throughput of the code region. Formally, we define the arithmetic-to-cast operations ratio for a code region as

$$r_{ac} = O/C, \tag{1}$$

where O is the number of floating-point operations and C is the number of casting operations.

4.3 *FISet* Illustration

Consider a portion of a graph as shown in the beginning of Fig. 2, where two data values are loaded and stored into registers `R1` and `R2`, which are then used by a multiplication. In step 1, the algorithm considers the code transformations that are required to lower the precision of the multiplication operation. In this example, we use a three-input instruction format with operations in FP64 denoted in upper case (e.g., `MUL`), and operations in FP32 in lower case (e.g., `mul`).

The second column of step 1 in the figure shows the required transformation to reduce the precision. Since data in registers `R1` and `R2` is in FP64, we need to perform two type conversions to truncate their data to FP32. After the multiplication in FP32 is performed, we need to extend the result to FP64, incurring another conversion (from FP32 to FP64). In this step, $r_{ac} = 1/3$. This ratio will likely not improve performance; in fact, it will degrade performance since for the same `MUL` operation we are performing three additional instructions, i.e., type conversion operations. The goal of the algorithm is to find cases where $r_{ac} > 1.0$.

In step 2 (second row of the figure), we consider the *neighbors* of the previous `MUL` operation. Here, neighbors are operations that depend on `MUL` and operations that influence `MUL`. Since the only operations that influence `MUL` are load operations, we do not consider them (they are not arithmetic floating-point operations); however, we consider the `ADD` operation that depends on the result of `MUL`. The second column of step 2 shows the required transformation to reduce the precision, which would produce $r_{ac} = 2/3$; this can be easily seen by noticing that there would be two arithmetic operations and three type conversions, $r_{ac} = O/C = 2/3$ after the corresponding transformation. Since $r_{ac} < 1.0$, the algorithm keeps expanding the neighbors set and performs the same estimations.

Finally in step N we find a set with $r_{ac} > 1.0$, i.e., $r_{ac} = 4/3$ (see last part of Fig. 2). Here we declare this set a Φ.

4.4 *FISet* Properties and Algorithm

Loops. If all nodes of a *FISet* are in the same loop (or loop level), or there is no loop in the kernel, we do not do anything special because all the instructions will be executed the same number of times, which will not affect r_{ac}; this is the common case for most kernels. When this is not the case, we consider the following two cases—we assume kernels can have nesting loops, $L_0 > L_1 > L_2 > ...$, where L_0 encloses L_1, L_1 encloses L_2, and so on:

- *Arithmetic operation nodes are in loop level L_x and conversions are in loop level L_y, where $L_x > L_y$.* We assume that the arithmetic operations will be executed equal or more times than the conversions so we do not do anything special. Note that this applies even for $L_x \geq L_y$, for a given input. In this case, r_{ac} may be higher than expected, which is fine for performance speedups.
- *Arithmetic operation nodes are in loop level L_x and conversions are in loop level L_y, where $L_y > L_x$.* In this case, conversions may be executed more times than arithmetic operations. We use a heuristic to handle this case: if we find the same number of arithmetic operations as the number of conversions in the loop that contains the conversions, we allow this to be a *FISet*; otherwise, we discard this case, and the algorithm proceeds.

Algorithm. The *FISet* search algorithm is shown in Algorithm 1. The algorithm starts by taking a node from the dependence graph and by calculating the number of in/out edges, which is then used to calculate r_{ac}. If $r_{ac} > 1.0$, it adds it to the list of *FISet*. Next, it increases the set to explore by adding the neighbors of the node, which are then used to calculate r_{ac} like in the previous step. The nodes to be explored are added to the *neighborsList*. It does not add neighbors to the list if the node is a *terminating node*, i.e., it is a load/store operation or a function call since these operations do not have lower precision versions. For GPU kernels with very large dependence graphs, the algorithm can find many *FISets*. In those cases, we allow the user to specify the maximum number of *FISets* that the algorithm return, using the parameter ϕ.

Multiple *FISets*. Algorithm 1 can identify multiple disjoint *FISets* in the same kernel. If two *FISets* overlap, i.e., they have instructions in common, the algorithm will return the union of the two. If *FISets* do not overlap, multiple configurations combining these *FISets* are considered. In practice, however, we found that a single *FISet* per kernel typically gives reasonable speedups.

Compilation Process. Once CUDA modules are transformed to NNVM IR (by the clang front-end), the *FISets* search is performed in the NVVM IR representation. After this, the kernel is transformed to PTX, which is then assembled into object files. Finally the NVIDIA nvcc compiler is used to link objects.

4.5 Shadow Computations

FISets per se give no information about the error introduced by lower precision arithmetic. To calculate this error we use dynamic *shadow computations*. Shadow computations analysis for mixed-precision tuning has been used before [22,13];

```
   input  : Dependence graph DG
   output: FISets: list of FISets found
 1 for node n ∈ DG do
 2     if n is not arithmetic op then
 3         continue
 4     else
 5         currentSet = [n]
 6         neighborsList = [n]
 7         while neighborsList is not empty do
 8             tmp = neighborsList.getFirstElement()
 9             for node m ∈ neighbors(tmp) do
10                 if m is not load/store or function call then
11                     numConversions += numInEdgesOfNode(m) +
                        numOutEdgesOfNode(m) - numFloatingPointConstants(m) - 1
12                     numOperations += 1
13                     add neighbors of m to neighborsList
14                     remove m from neighborsList
15                     add m to currentSet
16                     if numOperations/numConversions > 1.0 then
17                         add currentSet to FISets
```

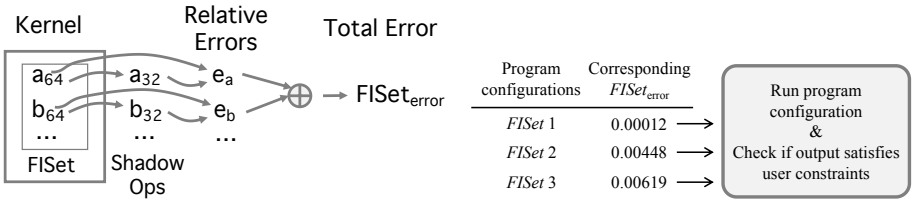
Algorithm 1: *FISet* Search Algorithm. Symbols and operation definitions: *neighborsList* is the list of nodes to visit; *currentSet* is the set of nodes we have visited and may become a *FISet*; *neighbors()* returns the in- and out-edges of a node that have not been visited; *numInEdgesOfNode()* and *numOutEdgesOfNode()* return the number of in- and out-edges of a node respectively; *numFloatingPointConstants()* returns the number of constant input parameters of an operation (they do not require conversion). Note that line 11 subtracts 1 because we need to subtract the edge that connects m to the *currentSet*, otherwise it would be counted twice when we calculate *numInEdgesOfNode()* or *numOutEdgesOfNode()*.

however, none of the previous frameworks handle multi-threaded programs, so, as far as we know, ours is the first.

The idea of shadow analysis is that, for each floating-point arithmetic operation in high precision, e.g., FP64, a similar operation is performed side-by-side on lower precision, e.g., FP32. By comparing the result of the operation in low precision with the result of the operation in high precision, we calculate the relative error that the low-precision operation would introduce.

Calculating the Kernel Total Error We compute an approximation of the total error that is introduced in the kernel when the precision of portion of the kernel (a *FISet*) is downgraded. This allows us to guide the search for *FISet* configurations that introduce low total error.

More formally, let us say that a kernel comprises FP64 operations $\{a_{64}, b_{64}, c_{64}\}$. Operations are of the form $[x_{64} = \text{OP}, y_{64}, z_{64}]$, and $\text{OP} \in \{+, -, *, /, <, >, =, \neq\}$. When an operation is transformed to FP32, its operands y_{64}, z_{64} must be truncated to FP32. Both the truncations and the operation performed to lower precision introduce errors. Shadow computations analysis computes an approx-



(a) Total error calculation (b) Trial runs sorted by their $FISet_{\text{error}}$

Fig. 3: Shadow computations used to calculate the mixed-precision error

imation of the total error introduced by these transformations. The word *total* means that the contribution to the error of **all** the GPU threads is considered.

Kernel Instrumentation We start with a kernel with all its instructions in FP64. Each FP64 operation is instrumented with a callback function. The function takes as input the operands of the FP64 operation (in FP64 precision) and truncates them to FP32 precision. It then computes two values: v_{64} and v_{32}. v_{64} corresponds to the result of the operation as if the operation is performed in FP64 precision; v_{32} corresponds to the result of the operation as if the operation is performed in FP32 precision. The following calculates the relative error:

$$e = abs((v_{64} - v_{32})/v_{64}), \tag{2}$$

where $abs()$ is the absolute value function. The result of e is stored in FP64 precision. Because of the SIMT execution model of GPUs, all threads in a warp in the kernel block execute the same callback function.

GPU In-Memory Structure We keep a structure in the GPU global memory of the form:

$$total\_error[INST][THREADS],$$

where *INST* is the number of static instructions of the kernel, and *THREADS* is the maximum number of threads that the kernel can use. This keeps track of the error values for all static instructions and for all the threads that execute the instruction. Since a thread can execute a static instruction multiple times, each calculated e is aggregated (added) into a single e for the static instruction—this allows us to calculate a total relative error for the instruction (see Figure 3a).

Assigning an Error Value to a FISet. We run the GPU program once with a set of inputs from the user to obtain a total error value for each static instruction. Given a *FISet* with N instructions, we assign an error value to the *FISet* by merging the *total_error* value of each *FISet* static instruction, using this formula: $\sum_i^N total\_error_i$. We call this error $FISet_{\text{error}}$.

Trial Runs. Given several *FISets*, and their corresponding $FISet_{\text{error}}$, to satisfy accuracy constraints, we search for *FISets* configurations in the order of their error, starting with those with the smallest $FISet_{\text{error}}$ (see Fig. 3b). Searching for configurations means that we run the program to determine its output. We call this a *trial run*. Trial runs are independent of the shadow computations run.

Putting it All Together Error and Performance Thresholds. To search for configurations, the user provides two independent parameters: *error threshold* and *performance threshold*. *Error threshold* specifies the number of digits of accuracy that is expected in the program output with respect to the baseline FP64 precision case. For example, if the output of the FP64 case is 3.1415 and the output of the mixed-precision case is 3.1479, we say that the latter is accurate up to 3 digits (i.e., from left to right, digits 3, 1 and 4).

Performance threshold specifies the minimum performance speedup that is expected. Here, performance speedup is defined with respect to the maximum ideal speedup, i.e., the performance of the program when it is compiled using fully FP32. We use the figure of merit (FOM), which represents the metric of performance of the program. Specifically, we define the speedup of the mixed-precision case as:

$$s = ((p_{\text{mixed}} - p_{64})/(p_{32} - p_{64})) * 100, \tag{3}$$

where p_{mixed} is the performance of the mixed-precision case, p_{64} is the performance of the FP64 case, and p_{32} is the performance of the FP32 case. Thus, $s = 100\%$ when the mixed-precision case performs as the FP32 case, i.e., when all instructions are converted from FP64 to FP32.

Modes of Operation. Our approach has three modes of operation to search for configurations:

- **Mode 1:** the user cares only about the output error and does not care about the magnitude of performance speedup (as long as there is some performance speedup). In this case, the user provides only an error threshold. The search is based on the *FISets* total error value—we start running the *FISet* configuration with the smallest total error, then continue with the configuration with the second smallest total error, and so on. The search ends when the output error meets the error threshold.
- **Mode 2:** the user cares about both output error and performance speedup, *but output error has priority*. Here, the search is performed like in Mode 1, but it ends when both the output error meets the error threshold and the performance speedup meets the performance threshold.
- **Mode 3:** the user cares about both output error and performance speedup, *but performance speedup has priority*. Here, the search is based on the ratio r_{ac} of the *FISets* (high r_{ac} implies high chances of performance improvements)—we start by running the *FISet* configuration with the largest r_{ac}, then continue with the configuration with the second largest r_{ac}, and so on. The search ends when both the output error meets the error threshold and the performance speedup meets the performance threshold.

4.6 Limitations

Accuracy of Ratio r_{ac}. A limitation of r_{ac} is that it does not consider the actual cost of operation types. Unfortunately, we are limited by the fact that the NVIDIA CUDA C Programming Guide [20] does not specify the cost of all GPU operations—it specifies the throughput of `add` and `multiply` operations but it lacks throughput specs for other common operations, such as `division` and math operations, e.g., `sqrt`. We believe that per-instruction costs could be empirically estimated for specific GPU architectures, but it requires significant benchmarking that is out of the scope of this paper. Nevertheless, we have found that r_{ac} is practical for most cases.

Register Pressure. Mixed-precision programs can incur register pressure because new type conversions introduce additional instructions, thus more registers may be required. Registers, along with other resources, are scarce in GPU Streaming Multiprocessors (SM). There is a maximum number of available registers in an SM—255 per thread for NVIDIA compute capability 6.0. If a kernel uses more registers than the hardware limit, the excess registers will *spill* over to local memory impacting performance. *FISets* can increase registers usage by a small amount. This may be a problem only on kernels with a register usage that is close to the limit. In such cases, a configuration may not yield any speedup. In our experiments, however, we only saw one kernel in this category.

5 Evaluation

We present our evaluation of `GPUMixer`. We implement `GPUMixer` in the Clang/LLVM compiler [14] 4.0, using the CUDA ToolKit 8.0. Experiments are conducted in a cluster with IBM Power8 CPU Core nodes, 256 GB of memory, and NVIDIA Tesla P100 GPUs (compute capability 6.0), running Linux.

5.1 Comparison Approach: *Precimonious*

While none of the existing mixed-precision tuning methods handle multi-threaded and/or CUDA codes, the Precimonious technique [23,22] uses a generic search algorithm, *delta debugging*, that can be implemented for CUDA programs (the original version in the paper works on CPU-base serial programs). This algorithm is considered the state-of-the-art on automatic mixed-precision tuning and it is also used as a comparison baseline in several works [7,22]. We implement the delta debugging tuning algorithm as described in [23] as a comparison framework for our approach as well, which we call `Precimonious-GPU`.

As described in [23], our implementation finds a 1-minimal configuration, i.e., a configuration for which lowering the precision of any one additional variable would cause the program to compute an insufficiently accurate answer or violate the performance threshold. To generate program variants, we use static changes to the source code to modify the declarations of variables from FP64 to FP32.

Mode of Operation. Since Precimonious does not perform a search separately driven by error or by performance, we only use one mode of operation: once both the error and performance constraints are met, the algorithm stops.

Table 3: Profile of Top Kernels in LULESH

Kernel	Time	r_{ac}	Registers Usage	
			FP64	Mixed
CalcVolumeForceForElems	25.21%	8.13	254	255
ApplyMaterialPropertiesAndUpdateVolume	24.62%	1.01	62	65
CalcKinematicsAndMonotonicQGradient	18.87%	3.45	128	128

5.2 CUDA Programs

We evaluate our approach on three scientific computing CUDA programs: LULESH [10], CoMD [1], and CFD [2]. LULESH is a proxy application that solves a Sedov blast problem. This simulation is useful in a variety of science and engineering problems that require modeling hydrodynamics. CoMD is a reference implementation of typical classical molecular dynamics algorithms and workloads. CFD (from Rodinia benchmarks) is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. We use -O2 optimization in all programs. As inputs we use: `-s 50` for LULESH, `N=20, nx=25,ny=25,nz=25`, for CoMD, and `fvcorr.domn.193K` for CFD.

Output. For LULESH, we consider the TotalAbsDiff as the main output, a symmetry value for the final origin energy of the simulation. For configuration, we also perform other correctness checks, including making sure that the final energy and iterations count is the same as in the FP64 version. For CoMD, we use the simulation final energy as the main output since this is one of the key interesting final results for molecular dynamics simulations. For CFD, we use the total density energy as the output.

Figure of Merit (FOM). For LULESH, we use zones per second as the FOM; for CoMD we use the average atom rate, i.e., processed atoms per time (atoms/usec); for CFD we use execution time in seconds. Note that for LULESH and CoMD, higher FOM is better, while for CFD, lower is better.

5.3 Overhead of Shadow Computations

The overhead of shadow computations analysis is on average 24× (61× for LULESH, 1.5× for CoMD, and 11.12× for CFD), which is comparable to the overhead of static and binary instrumentation tools [9,15]. Note that shadow computations analysis is run only once with a given input and is independent of the trial runs (see Section 4.5).

5.4 Threshold Settings

We present results for three levels of accuracy (3, 6, and 9 digits of accuracy) with respect to the baseline FP64 precision case, and four performance thresholds (5%, 10%, 15%, and 20%). We experimented with higher digits of accuracy and higher performance thresholds, however, none of the approaches found solutions in such cases, so we limit the results in the paper to 9 digits of accuracy and 20% of performance threshold. Note that for CFD, where lower FOM is

Table 4: Results of using *FISets* and shadow computations: performance speedup (% of maximum ideal speedup) for three error thresholds, four performance thresholds and tree modes of operation; number of runs in parenthesis.

	Error Thold. (digits)	Mode 1	Mode 2 Performance Threshold 5%	10%	15%	20%	Mode 3 Performance Threshold 5%	10%	15%	20%
LULESH	3	9.8% (1)	9.8% (1)	30.4% (2)	30.4% (2)	30.4% (2)	46.4% (1)	46.4% (1)	46.4% (1)	46.4% (1)
	6	0.3% (12)	8.4% (79)	–	–	–	–	–	–	–
	9	0.3% (12)	–	–	–	–	–	–	–	–
CoMD	3	24.2% (1)	24.2% (1)	24.2% (1)	24.2% (1)	24.2% (1)	10.9% (1)	10.9% (1)	37.5% (7)	37.5% (7)
	6	24.2% (1)	24.2% (1)	24.2% (1)	24.2% (1)	24.2% (1)	10.9% (1)	10.9% (1)	37.5% (7)	37.5% (7)
	9	2.3% (3)	19.7% (62)	19.7% (62)	19.7% (62)	–	19.3% (8)	19.3% (8)	19.3% (8)	–
CFD	3	8.3% (1)	8.3% (1)	13.3% (3)	15.3% (35)	–	5.1% (9)	12.6% (15)	15.1% (39)	–
	6	8.34% (1)	8.3% (1)	13.3% (3)	15.3% (35)	–	5.1% (9)	12.6% (15)	15.1% (39)	–
	9	–	–	–	–	–	–	–	–	–

Table 5: `Precimonious-GPU` results: performance speedup (% of maximum ideal speedup) for the error thresholds and performance thresholds; number of runs are in parenthesis. See Fig. 4 for the maximum speedup reported for each approach.

	Error Thold. (digits)	Performance Threshold 5%	10%	15%	20%
LULESH	3	11.6% (11)	11.4% (11)	17.4% (32)	20.7% (34)
	6	11.5% (11)	11.4 (11)	–	–
	9	–	–	–	–
CoMD	3	12.6% (2)	12.9% (2)	–	–
	6	13.6% (2)	12.7% (2)	–	–
	9	5.4% (24)	–	–	–
CFD	3	–	–	–	–
	6	–	–	–	–
	9	–	–	–	–

better, speedup is $-s$. We set the maximum number of *FISets*, ϕ, to 200 in all experiments. In practice, the number of trial runs is always less than this value.

5.5 Case 1: LULESH

Table 3 shows the result of LULESH's profile. The first and second columns show the three kernels that consume most of the execution time and the percentage of time, respectively. Since time in the remaining kernels is small (less than 5%), we do not consider them in the rest of the analysis as they are unlikely to yield high speedups when using mixed-precision. The third column shows the average arithmetic-to-cast operations ratio, r_{ac}, for the kernel *FISets*.

As we observe in the table, `CalcVolumeForceForElems` has a high average r_{ac}, which means that the *FISets* of this kernel could potentially give high speedups. As we observe in the fourth and fifth columns of the table, which show the register usage for the baseline (FP64) and mixed precision versions, the register usage of this kernel is very close to the limit, i.e., 254 out of a maximum of 255 registers per thread in this GPU. The average register usage for the mixed-precision version is 255, which indicates that this is kernel is not a good candidate for mixed-precision, therefore, we discard this kernel in the analysis.

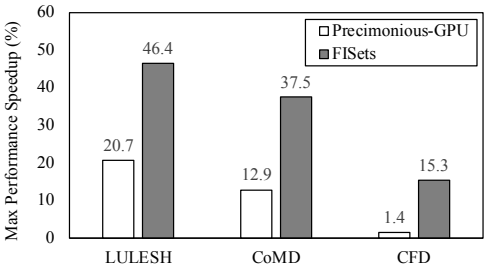


Fig. 4: Maximum performance speedup (% of the ideal speedup) reported by `Precimonious-GPU` and the *FISets* approach.

`ApplyMaterialPropertiesAndUpdateVolume` is the next kernel that we consider (second in the table). While the algorithm found a few *FISets* in it (4), the average r_{ac} of these *FISets* is quite low: only 1.01. This indicates that there is almost no potential for performance improvements in this kernel, thus, we also discard this kernel in the analysis.

`CalcKinematicsAndMonotonicQGradient`, the third kernel is next considered. This kernel has the appropriate characteristics: the average *FISets* r_{ac} is 3.45 and its average register usage is 125, even when *FISets* are used, i.e., for this kernel *FISets do not increase register usage*. Therefore, we focus on this kernel in the rest of the analysis and experiments.

Table 4 (first section) shows the performance results for LULESH, for the error thresholds, performance thresholds and the three modes of operation; the number of trial runs are shown in parenthesis. For Mode 1, we find a configuration with 3 digits of accuracy and 9.8% of speedup with a single trial run; the cases for 6 and 9 accuracy digits do not produce significant performance improvements.

Except for the 6-digit case in Mode 2 (5% of performance threshold), which requires 79 runs, Mode 1 and Mode 2 both generally find configurations with high performance improvement (up to 46%) with *only a few runs* (1–2 runs). We did not find configurations for the 9-digit case in Modes 2–3.

Precimonious. The `Precimonious-GPU` results are shown in Table 5. We observe that the maximum speedup found is about 20.7% for the 3-digit case. Like in our approach, it cannot find good solutions for the 9-digit case.

Input Sensitivity. We measure the performance speedup (using Eq. 3) for multiple LULESH inputs. We use two *FISet* configurations: one with a low r_{ac} of 2.08 (case 1), and another one with a high r_{ac} of 6.90 (case 2). Fig. 5 shows the results; digits of accuracy are shown as labels. We observe that for case 1, the speedup for a small input (20) is small, but it increases for larger inputs. For case 2, the speedup for a small input is large and it decreases for larger input. In both cases the speedup stays almost the same for several large inputs, 50–80. The digits of accuracy for case 1 tend to be higher than for case 2 because case 1 has less FP32 operations than case 2 (its *FISet* is smaller) and as a result it incurs smaller error.

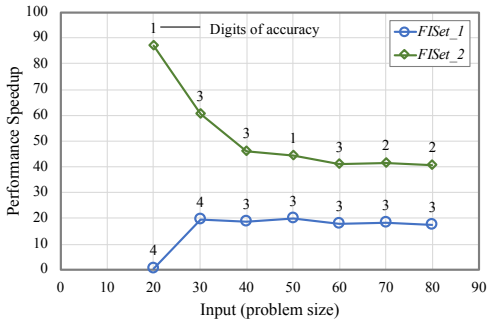


Fig. 5: Performance speedup for multiple LULESH inputs for two *FISet* configurations. Labels are the digits of accuracy.

5.6 Case 2: CoMD

CoMD is a compute-intensive workload, where a large portion of time is spent computing forces between particles—these operations involve several addition and multiplication operations versus a few load/store operations. This code is a good candidate for *FISets* and mixed-precision in general.

We follow a profiling phase that is similar to the one we did for LULESH. Out of the top four time-consuming kernels, `SortAtomsByGlobalId`, `LoadAtomsBufferPacked`, `fill`, and `LJ_Force_thread_atom`, our algorithm only found *FISets* in `LJ_Force_thread_atom`. Thus, this was the only candidate for performance improvements for our technique. The average *FISets* r_{ac} for this kernel was 3.10. By inspecting the code more carefully, we found that `LJ_Force_thread_atom` is where particle force calculations is done, so this finding makes sense. We did not find any kernel with high register pressure in this code.

Table 4 (mid section) shows the performance results for CoMD. As expected, the algorithm finds configurations that meet both error and performance thresholds for all modes of operation, in many cases with a single trial run. The best case in terms of performance was about 37% for 6 digits of accuracy with only 7 runs. As shown in Table 5, while it can find solutions with a few trial runs, `Precimonious-GPU` finds a maximum speedup of about 12.9%.

5.7 Case 3: CFD

CFD presents high potential for performance improvements via mixed-precision since the code core computations, flux computations, involve a number of compute-intensive operations. While this program is smaller than the LULESH and CoMD, it challenges our approach because its main kernel is relatively large, potentially causing *FISets* to put pressure on register usage.

After profiling the code, we find that 67% of the time is spent in `cuda_compute_flux`, while the rest of time is spent mostly on `cuda_time_step` (22%). Our algorithm did not find *FISets* in `cuda_time_step`; so we focus on `cuda_`

`compute_flux` on which the average r_{ac} of *FISets* is 3.56. Note that we did not find any kernel on which *FISets* causes a register pressure in this code.

Table 4 (third section) shows the performance results for CFD. We find configurations of up to 15.1% with up to 6 digits of accuracy running the code 39 times. It can also find a case for 8% of speedup on 6 digits of accuracy with a single trial run. `Precimonious-GPU` is, however, unable to find solutions for the target error and performance thresholds—the maximum performance speedup ever reported during the search was about 1.4% as shown in Figure 4.

6 Conclusions

While floating-point mixed-precision tuning techniques exist, they are accuracy-driven and do not provide significant performance speedups to GPU programs. We introduce and evaluate `GPUMixer` a new tool to tune floating-point precision on GPU programs with a focus on performance improvements. `GPUMixer` is engineered on novel concepts, such as *FISets* to statically identify regions that yield performance, and shadow computations analysis to compute the error introduced by mixed-precision. Our evaluation shows that our approach can be used in realistic GPU applications, and that it can find configurations that produce higher speedups (up to 46% of the maximum ideal speedup) than those of current state-of-the-art techniques.

Acknowledgments. We thank the anonymous reviewers for their suggestions and comments on the paper. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-748618).

References

1. CoMD-CUDA. `https://github.com/NVIDIA/CoMD-CUDA`, 2017.
2. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54. Ieee, 2009.
3. W. F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. Association for Computing Machinery, 2017.
4. W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 43–52, New York, NY, USA, 2014. ACM.
5. N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 31–46. Springer, 2015.
6. E. Darulova and V. Kuncak. Towards a compiler for reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2):8, 2017.

7. H. Guo and C. Rubio-González. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 333–343. ACM, 2018.
8. M. Harris. mini-nbody: A simple N-body Code. `https://github.com/harrism/mini-nbody`, 2014.
9. T. Iskhodzhanov, A. Potapenko, A. Samsonov, K. Serebryany, E. Stepanov, and D. Vyukov. Threadsanitizer, memorysanitizer.
10. I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.
11. M. O. Lam and J. K. Hollingsworth. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications*, page 1094342016652462, 2016.
12. M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378. ACM, 2013.
13. M. O. Lam and B. L. Rountree. Floating-point shadow value analysis. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*, pages 18–25. IEEE Press, 2016.
14. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
15. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
16. H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger. ADAPT: algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 48. IEEE Press, 2018.
17. NDIDIA. CUDA ToolKit Documentation - NVVM IR Specification 1.5. `https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html`, 2018.
18. H. Nguyen. In *GPU Gems 3*, chapter 31, pages 677–694. Addison-Wesley Professional (August 12, 2007), 2007.
19. Nvidia. Nvidia Tesla P100 GPU. *Pascal Architecture White Paper*, 2016.
20. Nvidia. CUDA C Programming Guide, v9.0. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2018.
21. G. Paganelli and W. Ahrendt. Verifying (in-) stability in floating-point programs by increasing precision, using smt solving. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 209–216. IEEE, 2013.
22. C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1074–1085, New York, NY, USA, 2016. ACM.
23. C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 27. ACM, 2013.