AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications

P. Kotipalli (1), R. Singh (1), P. Wood (1), I. Laguna (2), S. Bagchi (1)
(1) Purdue University; (2) Lawrence Livermore National Lab

ABSTRACT

Mixed precision computations improve high performance computing throughput for applications that can tolerate decreased mathematical precision in their computations. Native mixed precision computation is commonplace in today's GPGPU accelerators where it is applied to applications with well-known tolerances for reduced mathematical precision. Applications with stricter accuracy needs lack support for selecting precisions that both improve performance *and* satisfy these accuracy requirements. Prior works have focused primarily on accuracy, leaving performance concerns such as the overhead of casting unanswered in GPGPU contexts. In this paper, we present a system called AMPT-GA that selects application-level data precisions to maximize performance while satisfying accuracy constraints. We combine static analysis for casting-aware performance modeling with dynamic analysis for modeling and enforcing precision constraints. We further improve our optimizations with application-aware mutations in our genetic algorithm-based search function. AMPT-GA improves the performance efficiency of our target applications more than the prior state-of-the-art approach called Precimonious.

ACM Reference Format:
. 2019. AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications. In *ICS '19: 33rd ACM International Conference on Supercomputing, June 26–28, 2019, Phoenix, AZ.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

High-performance computing (HPC) scientific applications rely heavily on floating point arithmetic to compute the high-precision outputs needed by end users. Often in these programs, it is possible to reduce the floating-point precision of certain variables in order to speedup execution or to reduce computation energy. In recent NVIDIA GPUs, for example, the throughput of single-precision floating-point operations is twice that of double-precision operations. Being able to leverage this trade-off becomes increasingly important as we seek to execute applications at large scales driven by increases in problem size. One crucial dimension in this problem space is the accuracy of application results, which must satisfy user-driven accuracy or error thresholds.

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Combining different precisions for different floating point variables is known as *mixed precision computing*. Mixed precision computing has been known for many years [14], but several recent changes in the GPU space have created new opportunities for improving efficiency. GPU manufacturers have begun to include native FP64 (double precision) and FP16 (half precision) arithmetic units inside of their processing units, in addition to FP32 (single precision). As a result, FP64/FP32/FP16 instructions can coexist providing different performance levels, e.g., the ratio of FP64:FP32 throughput is 1:3 in the K40 series and 1:2 in the P100 series of Tesla GPUs, while the ratio is 1:2 for FP32:FP16 on the same P100 series. Prior to the introduction of these processors, mixing FP64 and FP32 instructions had limited performance impact because mixed precision math units were rare. Selecting mixed precisions resulted in data types conversions, or *casting*. As examples of the acceptability of results of lower precision arithmetic, consider LULESH [11], an important proxy application for the exascale co-design efforts of the US Department of Energy (DOE). While computing everything on FP32 precision significantly perturbs the desired symmetry of the solution, computing everything on FP64 may be unnecessary—making some variables FP32 precision, while the rest stay at FP64 precision has been shown to yield acceptable output [1]. In the area of LSTM (a form of neural network) used for speech recognition and machine translation, Baidu researchers have recently found significant gains using a mix of FP32 and FP16 variables on a GPU with no difference in accuracy [18]. However, no prior work has shown how to navigate this space in a rigorous algorithmic manner balancing the improvement in execution speed with a hard bound on the tolerable error.

Background on Prior Work: Prior work on mixed precision optimization falls short in three crucial ways. First, prior works do not support parallel codes found in GPU programming models as they rely on serial instrumentation or profiling support that does not span the GPU programming and execution model, such as CPU-to-GPU calls. Second, accuracy-centric approaches [3, 14] lack a performance model and simply assume that minimal precision results in the fastest running time. For example in [3], the objective function is to maximize the number of FP32 variables. In practice, mixing precision requires casting to satisfy the mathematical operation with the most precise operand. However, casting is an expensive operation (e.g., twice as expensive as FP64 operations in our target GPU architectures) and therefore reducing precisions may actually *increase* the execution time. Further, when there are parallel resource pools for FP64 and FP32 (as in several GPU architectures), mixing precision allows for an additional opportunity for parallelism, which prior work ignores. In some previous work [3, 4, 7], precision levels are selected that meet overall accuracy requirements, but they often fail to improve the execution time of the application. Third, approaches that solely use online runtime information [20] suffer from very large search space problems. Consider that there are n floating point variables that can be tuned and there are 3 precision levels supported by the architecture; then the search space is 3^n. In large production-level scientific applications, n can be very large (hundreds), making this method impractical.

Our Solution: We present AMPT-GA, pronounced "Amp-ed GA", the first mixed precision optimization system that solves an accuracy-constrained performance maximization problem for GPU programs. AMPT-GA seeks to select the set of precision levels for floating point variables at an application level that maximizes the performance while keeping the introduced error below a tolerable threshold, as defined by the application user. The complete set of assignments of each floating point variable to a precision level is called the *Precision Vector (PV)* and the optimal PV is the final output of AMPT-GA. *The key insight behind AMPT-GA is that the dynamic search technique through the large space of possible precision vectors is aided by static analysis.* Our static analysis identifies groups of variables whose precisions should preferentially be changed together to reduce the performance impact of the precision change of any variable through casting. Such information speeds up the online search through the large search space. Further, considering the irregular nature of the search space, we use a Genetic Algorithm for the search, which helps avoid local minima that prior approaches such as [10, 20] have a tendency to fall into.

To make our rationale concrete, consider the operation $x + y$ which translates to either FP32 addition of two FP32 variables or FP64 addition of two FP64 variables as an architecture requirement. At a compiler level, if the precision of x and y do not match, then the operation FP32 x + FP64 y yields a FP64 add with x being converted from FP32 to FP64 following the C/C++ standard, our target language. Such conversion takes time of 4 units in our target GPU architecture, while the FP32 add takes time of 1 unit and the FP64 add takes time of 2 units. Consequently, many mixed precision configurations run *slower* than the all-double case—41.8% of the configurations sampled by our naïve algorithm were slower than all-double. Our solution takes this cost explicitly into account while searching, aided by its static analysis phase. Thus it avoids many poorly performing PV configurations where the cost of casting will be high.

Another novelty is that AMPT-GA takes a global view of the error and avoids the complex issue of mapping of per-operation error to final application-level error. We utilize global error models in our analysis, a departure from the Blame Analysis work [19], where pruning is done based on local, per-operation error models that are difficult to come up with and that do not easily map to global application errors for complex programs. As an end result, the selected PV from AMPT-GA results in a performance improvement that takes into account both the cost of casting and the speedup of operations due to reduced precision arithmetic, at a cost of reduced but tolerable accuracy.

We find that AMPT-GA is able to outperform the state-of-the-art approach Precimonious [20] by finding 77.1% additional speedup in LULESH's mixed precision computations over its baseline of all double variables, while using a similar number of program executions as Precimonious. Our casting-aware static performance model allows AMPT-GA to find precision combinations that Precimonious was unable to identify due to local minima issues. With three representative Rodinia benchmarks, LavaMD, Backprop, and CFD, we achieve additional speedups relative to Precimonious of 11.8%–32.9% when the tolerable error threshold is loose, and -5.9%–39.8% when the error threshold is the tightest. We choose these three as they span the range of number and sizes of kernels as well as the spectrum of precisions available on latest generation GPU machines. We also evaluate the efficiency of the three solution approaches, which is the performance gain over the number of executions needed to search for the optimal PV, and AMPT-GA outperforms in this metric.

Our claims to novelty in this paper are as follows:

(1) We improve the efficiency of finding viable mixed precision programs by integrating GPU-compatible static analysis into a genetic searching process. A statically built performance model provides an execution filter that eliminates unprofitable precision vectors, reducing the number of executions that AMPT-GA has to perform during its search.
(2) AMPT-GA provides application-level guidance on precision level for entire GPU applications rather than localized kernels, functions, or instructions seen in prior works.
(3) Experimentally we find that our search strategy can avoid local minima compared to the state-of-the-art approaches, thus leading to execution time gains.

2 RELATED WORK

Prior work on mixed precision performance optimization can broadly be divided into two categories, works that tune the precision to control the error and those that also take the performance of the tuned system into consideration.

2.1 Error Analysis

Prior work in static error analysis provides a foundation for rigorously determining what precisions are required to meet error constraints for particular closed form equations. Work in FPTuner [3] provides *expression-level* precision guarantees by using Taylor series expansions and formulates an error-constrained mixed integer optimization problem that attempts to find the lowest precision possible, subject to casting penalties, for a given error bound. The mathematical model obtains a rigorous error bound but is unable to deal with loop and conditional statements, making it unsuitable for LULESH and most other real programs. Our work provides precision guidance for application-level parameters rather than at the individual expression level. Related to FPTuner, work in [7] utilizes an SMT solver to determine precisions from program annotations to meet tight error bounds, but the runtime of the resulting application is not considered.

There are also models such as [15] and [13] that perform *Shadow Value Analysis* by inserting low level instructions at runtime to simulate floating point instructions at another precision. However, this is only a tool for analysis with respect to error, not for finding faster configurations.

A few tools exist that rewrite code segments, basic blocks, or instructions so that they perform faster for floating point operations [17, 22], but these improvements are complementary to AMPT-GA.

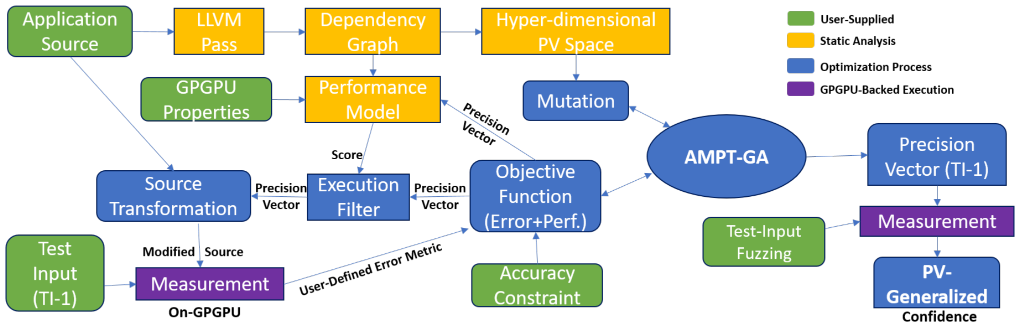


Figure 1: Overview of AMPT-GA. The inputs to AMPT-GA, shaded in green, are the application, the GPGPU properties (such as FP32 vs. FP64 instruction performance, casting cost), the error metric of interest and the corresponding error threshold, a test input, and the fuzzing process to generate similar inputs. The static path (in yellow) creates a performance and dependency model and the dynamic path (in blue) finds the Precision Vector (PV) to minimize the execution time of the application while staying under the error threshold. The blue optimization boxes are part of an iterative optimization process that run multiple generations of a Genetic Algorithm to identify the output PV. The purple boxes are measurements executed on the target GPGPU platform. During the search, each PV is evaluated through actual execution on the GPGPU unless filtered by our execution filter. The final PV for the test input is then measured against a region of similar inputs to determine its generality.

2.2 Application-Level Precision Tuning

Our solution focuses on application-level precision tuning where all variables in a program (or in a chosen set of GPU kernels) are under consideration for precision updates. Precimonious [20] is the most closely related work that provides precision tuning guidance. It aims to find the *1-minimal* configuration, i.e., a configuration where changing even a single variable from higher to lower precision would cause the configuration to cease to be valid. A valid configuration is defined as one in which the relative error in program output is within a given threshold and there is a performance improvement compared to the baseline version of the program. Precimonious uses the *delta debugging* algorithm [23] to speed up the search process. However, it does not use any knowledge of the structure of the program to identify potential variables of interest. Furthermore, the impact of casting plays a significant role in precision selection, and we incorporate such analysis directly in our optimization approach unlike Precimonious. Other approaches have used stochastic searching, similar to the genetic algorithm used in our work, such as [21], but they focused on loop-free code segments while AMPT-GA works across loops and functions.

Work in [14] uses a search technique which however does not guarantee a reduction in execution time.

A recent solution called HiFPTuner [10] alleviates this problem to some extent by taking a white box approach. It performs static analysis of the program to generate read-after-write dependencies among variables and uses these dependencies to limit the search space of Precimonious. It has the insight that we share, i.e., avoiding frequent shifts in precision at runtime contributes to better overall performance.

Blame Analysis [19] on the other hand performs *dynamic* analysis to achieve floating point precision tuning, considering the application to be white box. It creates a blame set for each instruction, which comprises the variables whose precisions can be reduced to reach a given error threshold of the instruction under consideration. Building the blame sets and the repeated executions with different possible precision values makes this technique expensive and it demonstrates overheads of about 50X, which is typical of dynamic instrumentation programs. However, it does reduce the search space for Precimonious and can be used as a pre-processing step for Precimonious as also for us.

Our work will be complementary in another aspect—Blame Analysis does not provide any guarantees about ultimately speeding up the program through mixed precision arithmetic; it only reduces precision in an intelligent manner, but the cost of casting can overwhelm such improvements (as the authors themselves acknowledge). When coupled with our solution, we can provide the guarantee that the transformed program will execute faster, while Blame Analysis will narrow down the GA search we will have to do. We do not perform this experimental comparison because the code base of Blame Analysis does not run on GPUs and our attempts to port it were unsuccessful.

3 OVERVIEW OF AMPT-GA

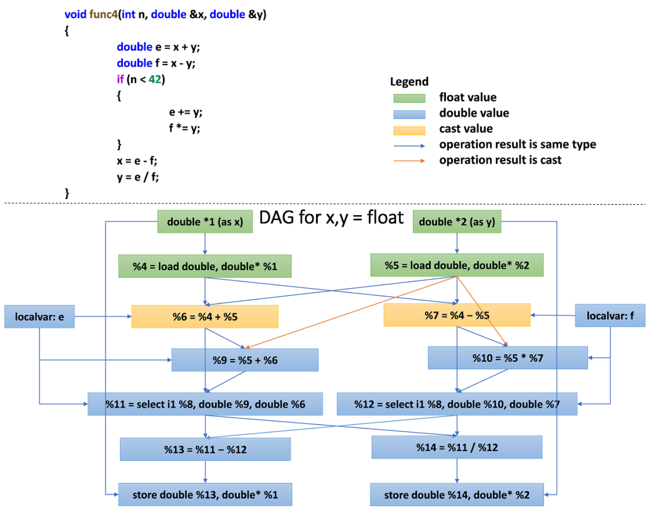
AMPT-GA is a multi-step process that solves the mixed precision optimization problem. The target application has N tunable floating point variables, and our goal is to calculate a precision vector (PV) of size N where each entry can be any of the supported precision levels, such as (for our target GPU) FP64 (double), FP32 (float), or FP16 (half). AMPT-GA's end goal is to pick PV such that the performance is maximized and the error is less than a user-defined threshold T. In this context, we define performance as operations per second.

AMPT-GA operates with an input of the target application or individual kernel with a test harness, a list of variables that can be changed, and a specific error metric with its target error threshold (T) for a given test input. If the list of variables is not specified, then we perform the analysis using all the floating point variables in the application or a target GPU kernel. For this paper, we assume the architecture processes only identical-precision inputs for an instruction, e.g., add with inputs FP64, FP64 or FP32, FP32 but not FP32, FP64. This assumption holds in CUDA language definitions. The compiler uses casting to enforce this constraint and we model its performance implications.

We construct a performance model to predict the impact of lowering the precisions of the considered floating point variables. The model is constructed using static analysis through an LLVM pass that constructs a dependency graph between the variables in the program. The graph has instructions and variable definitions as nodes, and edges connect variable definitions with their uses in an instruction. This graph, when provided with the precision level for each variable, measures the number of operations of each precision and the number of castings that occur. This information is translated into a performance score that captures the relative performance due to faster lower precision instructions across candidate PVs. This static pass does not capture the dynamic execution parameters such as the number of times a loop will be executed. While there is no guarantee that running time of equally-scored functions will be the same, the score provides a relative ordering of the potential for speedup of different PVs.

We use a modified genetic algorithm (GA) [5, 8, 9] to search through the space of PVs to determine the one that maximizes performance, while keeping the error below the specified threshold. We use a GA-based solution because the PV space is binary constrained, the gradients between the PVs are not smooth, and the objectives, both error and performance, are non-linear. The GA at each step of its search queries the performance model and only executes a point in the search space if the estimated performance is better than the best obtained so far. To search any given point, the GA actually executes the program with the PV being considered and then measures the performance and the error. If the performance is lower than the best performance obtained so far (which will happen due to inaccuracy in our performance model) or the error is higher than the threshold, then this search point is rejected. Otherwise, it stays in contention to be picked again for a future generation of the GA. Our solution is aware of the casting-induced penalties associated with dependent parameters (groups of related variables), and the mutation function is modified to change some variables in the hyper-dimensional group space, i.e., changing some set of variables together to avoid casting penalties. The final result from the GA is the optimal PV that the real application can be executed with. Fig. 1 shows how these different steps are interconnected.

To repeat a significant point from above, the GA needs to execute an application with the currently considered PV and since execution of a program is expensive (relative to querying our performance model), we choose to execute only if the performance model predicts that this will be a potentially worthwhile configuration. To account for the inaccuracy in the performance model, a certain relative range, say $X\%$ can be used, such that if the predicted performance is within $X\%$ of the best achieved so far, then this search point will be exercised. We validate empirically that filtering out likely unprofitable PVs and not executing the program with them is highly beneficial (Figure 9). In testing, we implement the PV as type definitions in the original source code for each variable of interest as a source-to-source transformation. For example,



```
void func4(int n, double &x, double &y)
{
        double e = x + y;
        double f = x - y;
        if (n < 42)
        {
                e += y;
                f *= y;
        }
        x = e - f;
        y = e / f;
}
```

Legend
🟩 float value
🟦 double value
🟨 cast value
→ operation result is same type
→ operation result is cast
DAG for x,y = float
double *1 (as x)
double *2 (as y)
%4 = load double, double* %1
%5 = load double, double* %2
localvar: e
%6 = %4 + %5
%7 = %4 – %5
localvar: f
%9 = %5 + %6
%10 = %5 * %7
%11 = select i1 %8, double %9, double %6
%12 = select i1 %8, double %10, double %7
%13 = %11 – %12
%14 = %11 / %12
store double %13, double* %1
store double %14, double* %2

Figure 2: Example DAG for AMPT-GA.

the variable "double t" in C++ is replaced with "typeOft t", and a header file is defined with "typedef typeOft PV(t)" where PV(t) maps to either "double", "float", or "half". Thus AMPT-GA can utilize proprietary compilers, such as NVIDIA's nvcc, to compile and test final versions of the mixed precision application. AMPT-GA also allows improvements available in nvcc to persist through our precision modifications, an advantage over more tightly integrated approaches that require intermediate representation manipulations, such as those found in [20].

End user workflow. The end user deploys AMPT-GA by providing the application source to AMPT-GA, defining a test input, and defining how the error will be measured and its target threshold. From this, AMPT-GA will find the precision vector for all the floating point variables that maximizes performance while meeting the threshold. Optionally the end user can indicate which floating point variables should be modified, e.g., she has the intuition that these are heavily used and therefore likely to be performance critical.

4 OPERATION OF AMPT-GA

In this section, we describe the design details of AMPT-GA.

4.1 Optimization Objective

AMPT-GA is fundamentally an optimization process designed to select the best precision vector (PV) such that the performance is maximized while the error stays below the user-specified threshold. We implement the constraint as an objective function penalty so that invalid solutions are rejected while information about the magnitude of the error is still incorporated into the score to aid in guiding the search.

$$\min_{PV} \text{perf}(PV) + (\text{error}(PV) - T) \cdot K \tag{1}$$

where perf(PV) is the performance score, with smaller values indicating higher performance. The performance model is described in Section 4.3. The term error(PV) is the value of the error metric due to the chosen PV while the threshold for this error metric is T. K is a multiplicative factor which is defined as follows: 1, if error$(PV) \leq T$ and P, a large value greater than the maximum perf value, if error$(PV) > T$.

User-Defined Error Metric. By default we use an application-generic error metric, which is the number of digits of precision compared to the most precise result. For example, if the output of an application is $y = 3.14159$, the developer may specify that the error of any output must match for the first 2 digits after the decimal point, e.g., y=3.142 and y=3.140 are both acceptable outputs. We use this metric in a global sense, for the entire application's output or outputs. In some applications, the developer may define an application-specific error metric and the corresponding threshold. For our test application LULESH, we utilize such an application-specific metric that captures the expected symmetry of particles at the end of the hydrodynamics simulation (Section 5.2).

Variables of Interest. Users may include all floating point variables in an application including arrays, structures, and individual variables, but parameter growth will increase the searching time. In AMPT-GA, we select the variables that are utilized in those GPU kernels that have high running times as measured b y standard profiling tools. We use a criteria that we start with the longest running kernel, include that, and move down the list sorted according to running time. When we have included enough kernels to capture a given percentage of the total program running time we stop—in our experiments, we include kernels which together account for at least 75% of the program's running time. For the chosen kernels, we use floating point variables that are inputs to the kernel, outputs from the kernel, or are local variables in these kernels. In prior work [19], *Blame Analysis* is used to identify variables that cannot be reduced in precision because they violate local precision requirements. This approach can also be used as a pre-processing step for us to limit the number of variables that need to be included in the search.

4.2 "Black Box" Search

As a baseline for our optimization approach, we utilize a "black box" optimization process which uses a genetic algorithm (GA) without any program analysis to direct the search. The approach by prior techniques, Precimonious [20] and its precursors [14, 15], can be considered to be black box as well. The GA randomly samples an initial population of PVs and then builds subsequent PVs by evolving them based on the performance of the prior generations. The intuition is that the algorithm will continually build a population that contains some entries close to the error threshold where the opportunity for lower precision is maximized. Each objective function evaluation is executed on the test application, by compiling and running it for the given PV. This approach can find solutions but suffers from general problems that all genetic algorithms have, such as large numbers of objective function evaluations. We compare the results of the naïve GA to the full AMPT-GA.

4.3 Dependency Graph and Static Performance Model

The static performance model and intermediate dependency graph estimate the performance gain from a particular PV. In general, the process takes an input of a PV and application source code, and it estimates the number of floating point and double operations along with the number of casts that appear statically in the code. The dependency graph is based on the program structure for the all-double case and applied to the mixed precision cases for calculating the number of casts and operation types that would change in a compiled mixed precision version of the program. We apply relative performance metrics for the target hardware to these operations and cast counts to generate a performance score for the particular PV. For example, our target GPU computes 1 double (FP64) or 2 floating point (FP32) operations per normalized cycle and takes 4 cycles to perform a casting operation. The specific PV is then translated into number of casts, number of FP32, and number of FP64 operations which are mapped to the normalized cycles and summed as a score. Since this analysis is static, loop counts are not captured, but loops are rare in of GPU codes since the code itself is executed in parallel.

We obtain the LLVM intermediate representation (IR) for each kernel and then use the *def-use* and *use-def* chains to build a DAG corresponding to all the floating point variables of interest in a kernel, as shown in Fig. 2. The DAG has nodes which correspond to different LLVM IR level instructions, with parents being the definitions of the variables being used in that instruction and children being the uses of the result of the instruction at a virtual register access level. Each node also contains information about the original data type of the instruction and the basic block of the program to which the instruction belongs. To simulate the changes in performance due to a different configuration, the root level nodes, corresponding to parameter and local variable definitions, are changed to the new types as per the configuration. The children of these nodes are then changed from higher to lower precision if *all* of its parent nodes are of lower precision. If even a single parent is in a higher precision, nodes corresponding to casting the lower precision parent to higher precision are introduced. In this manner, the change in type configuration is propagated down the DAG and a new version of the DAG for each PV, with different data types corresponding to the nodes, is obtained.

We then go through each basic block and add the expected number of cycles for each floating point instruction from the architecture specification (such as [6]) to calculate perf(PV) for that configuration. It should be noted that this score only considers floating point instructions and so, the score will not be strictly be proportional to the expected running time of the kernel. But it serves to rank different configurations, with a configuration with a lower score expected to perform better.

By analyzing the def-use chains in the DAG, we get guidance for which combinations of variables should be changed together, called *groups* in AMPT-GA. If a particular variable is a target for change, then it is possible to list the successors and predecessors that must also be changed to avoid inserting a cast, i.e., the set of changes required to maintain a uniform math operation precision if that particular variable's precision is changed. For example, if the series of instructions exists: $a = x + y$, $b = a + z$, then the precision of x, y, z should match to avoid casting the $x + y$ and $a + z$ operation, since the representation for a will follow from x, y. The DAG will have variables x, y, z in the same chain, so selecting z for a precision change will introduce casting unless the other operand (a), and its predecessors (x, y) are all the same type. Therefore, during optimization, the variables (x, y, z) are considered as a group. These variable groups can also be associated with the number of times the variables were used together, with variables being used together more often being more likely to be changed together in the mutation function.

4.4 Precision Vector Optimization

Our optimization approach in AMPT-GA selects the optimal PV for the given error threshold. Its goal is to optimize the function shown in Eq. 1. This problem is particularly challenging for several reasons. First, the space is very large even for fairly simple applications. In one of our test applications, LULESH, there are 76 floating point variables in the 3 longest GPU kernels. This makes the search space size 3^{76} when choosing between FP64, FP32, and FP16. Second, the space is not smooth, and nearby points have many local minima. For example, consider the PV for a simple operation $a = x + y$. In this case, the PV is {1,1} corresponding to the precision of {a,b} as both FP64 (FP64=1, FP32=0). If we consider {1,0}, then we have introduced a cast without any performance benefit, i.e., the performance differential is -4. If we consider {0,1}, then the differential is also -4. This means that a 1-variable change makes the performance worse, and a gradient-based optimization may converge to a local minimum without discovering {0,0} which provides a +1 performance benefit. The delta debugging approach of Precimonious runs into such local minima because it makes its decisions to further explore or not based on a single step exploration. We find empirical proof for this phenomenon as explained in Section 5. For these reasons, we rely on a genetic algorithm to optimize the PV.

The genetic algorithm by default does not learn any relationships among the variables in the PV. In fact, while some stochastic elements help it escape local minima, it is only by chance that the {0,0} PV would be considered from the earlier example. This is especially true with score-weighted crossover and mutation functions that bias the genetic population toward the current optimal values. The probability that a group of size N will be selected at random for casting to the same type is $\frac{1}{P^N}$, where P is the number of precision levels (= 3 in our target GPU architecture). This means that for a group of size 5 say, there is only a 0.41% chance that stochastic exploration will cause all 5 variables to be all float and the same low probability for all double. This probability becomes much lower when other independent parameters in the PV are also explored simultaneously. Thus, relying on a "blind search" (including a blind GA search) will not be able to find desirable configurations.

We overcome this problem in AMPT-GA by introducing a hyper-dimensional space with groups of variables for stochastic exploration. The mutation function, with a certain bias, will perform the mutation by changing an entire group of variables to a single precision level. We take the original random mutation PV that exists in space $\{v_1, v_2, ..., v_n\}$ and map this to $\{g_1, g_2, ..., g_m\}$ where $n \geq m$ (and typically $n \gg m$), g are the groups, and v are the variables. So the union of the groups covers all the variables and some groups can have a single element. If the mutation in the v space changes any group g, then all of the variables in that group g are changed as a unit for these G% of the mutations ($G = 25\%$ in our experiments). LULESH, for example, has 48 such groups. The remaining 100-G% of the population is mutated using the default *Adaptive Feasible* mutation function that chooses a direction and step length that meets feasibility requirements for the current PV, adapted from the last successful generations of PVs. We also seed the initial population with the two cases consisting of all variables being double and all variables being float (or half, if half precision is supported in the architecture) to define the upper and lower bounds of the search space. This guarantees that AMPT-GA will find the solution in the degenerate cases of the accuracy threshold being zero or ∞.

4.5 Execution Filtering

AMPT-GA relies on actual program executions to measure the error metric and verify if it is below the threshold. This means that *every* objective function evaluation for a yet unseen PV requires *executing the application*. Since this consumes execution time, AMPT-GA minimizes the number of actual program executions that occur using an execution filtering module. This module evaluates the PV against the performance score and compares the best seen performance score to that of the test PV. If the score for the test PV is better, then the execution is performed, else, the PV is not executed nor considered for further search. This forces the GA to find performance-improving PVs first before evaluating their accuracy.

4.6 Program Transformation

In AMPT-GA, we ultimately must apply the PV to the application so that the actual operations in the program occur at the modified precision. Instead of modifying the compiler, we utilize source code-level program transformations to replace the variable definitions and then allow recompilation with any supported compiler. This allows AMPT-GA to benefit from continuing improvements in the vendor-specified compiler.

AMPT-GA achieves this by modifying the declaration lines in the program and giving each variable of interest its own type name. For instance, "`Real_t x`" is replaced by "`Real_t_x x`". Then we can arbitrarily control the data type of each variable by choosing the appropriate one of "`typedef double/float/half Real_t_x`". By static analysis, we catch the dependent variables which should be declared as the same data type and use typedef to control this condition as well. In some cases, a given PV may not compile because of pointer-based variable accesses where data types have to be of a certain type. For example, if an array `x` has type double and a variable `y` is being used to access its elements in a loop, such as `y = x[i]` in a loop, then the data type of `y` and `x` must be the same. Infinite penalty is set for such infeasible PVs.

4.7 Input Generalization

One challenge that all mixed precision programs face is generalizing PVs across a variety of inputs. The relationship between input values and error is complex enough that the optimal PV is highly input-dependent. Instead of developing a guarantee, in AMPT-GA we develop a confidence that inputs similar enough to the test input will also satisfy the error constraint. We define the local input space as:

$$x' = x + \mathcal{N}(\mu = 0, \sigma = V) \tag{2}$$

where $\mathcal{N}(\mu = 0, \sigma = V)$ is the zero mean normal distribution with variance V. The variance V captures how far from the user-provided input we want to generalize. During the training phase of AMPT-GA, we generate a set of points x'. We measure for the optimal PV found for the given input x, what fraction of the points obeys the error threshold, which gives a confidence metric C. The higher the value of C, the higher is the reliance that the user can have on the generalizability of the output of AMPT-GA. Using this approach, we can amortize the time it takes to find a PV across runs that have similar but not identical inputs to the original x.

5 EXPERIMENTAL RESULTS

In this section, we evaluate AMPT-GA on a complex hydrodynamics code from the US Department of Energy (DOE) called LULESH [11] followed by evaluation on 3 diverse benchmarks from the Rodinia suite [2], LavaMD, Backprop, and CFD.

5.1 LULESH

Lulesh is used as a proxy app by the US DOE for benchmarking large-scale clusters because it is representative of many large codes. It models hydrodynamics equations, which describe the motion of materials relative to each other when subject to forces. We use the CUDA version of LULESH, with input size `-s 50`. We run all experiments on an NVIDIA Tesla P100 GPU machine.

5.2 Target Application and Limitations

We conduct our experiments by changing the precision at a variable level for LULESH in the FP32 or FP64 space, controlled by the PV. For performance, we measure "Figure of Merit" (FOM) as the rate at which elements in the simulation are processed, synonymous to throughput. FOM is the standard metric used for reporting performance on LULESH [12] and is output by default at the end of the code. There are somewhat esoteric reasons for relying on FOM instead of simple execution time that are related to scaling, but the improvements in wallclock and FOM are directly proportional. The actual metric we evaluate is the FOM as a percentage of the FOM with the lowest possible precision level (all floats for most of the applications, and all half precision for Backprop). Compared to evaluations in prior works, LULESH is a relatively complex code spanning 7,000 lines of C++ code on the GPU side and 600 LOC on the CPU side. We consider a total of 76 floating point variables in our tests. For comparison, in Precimonious [20], the largest program for which any mixed precision solution was found had 32 variables.

For the error constraint in LULESH, we rely on the the application-specific metric `TotalAbsDiff`, which measures the difference between energy values at symmetric points in the different dimensions. The error thresholds were selected by sampling 20,000 uniformly random PVs and collecting the 25%, 50%, and 75% quartiles from the data as 0.65 (tightest), 2.5, 3.6 (loosest) respectively.

5.3 Experiment 1: Optimized Precision

In this experiment, we seek to evaluate the relative execution benefit of AMPT-GA vis-á-vis the state-of-the-art Precimonious and a naïve GA. For Precimonious to work on GPU applications, we replace its variable interaction mechanism with our source code transformation process using the same variables as in AMPT-GA. In this experiment, we use Precimonious with its default terminating conditions while AMPT-GA and Naïve use fifty stall generations so that if after fifty rounds of mutation and crossover no better result is found, then the process stops with the best result.

The performance in terms of FOM is shown in Fig. 3 for the three protocols. The foremost goal of AMPT-GA is to improve the performance of the target application and we accomplish that for all but the tightest error constraint. For calibration, the FOM with all double variables is 267.6 while with all floats it is 360.0. AMPT-GA outperforms Precimonious for two error thresholds (2.5, 3.6) while under-performing for the tightest threshold by 4.2%, due to the inaccuracies in its performance model (quantified in Experiment 3). For the 2.5 case, Precimonious is not designed with group casting in mind, and it is unable to achieve the same level of performance gain as AMPT-GA. For the 3.6 case, pre-population helps AMPT-GA identify a solution with higher performance (near the all-float case) because mutations and crossovers are driven toward this solution point. This allows AMPT-GA to outperform even the naïve solution significantly for the highest threshold case. The slight underperformance compared to the naïve solution in the other cases is traded off with the reduced number of executions in the filtering operation. In Fig. 4, we see the number of program executions for the three protocols. Expectedly, naïve has by far the highest number of executions, while AMPT-GA*without* the execution filter comes close. With the execution filter, AMPT-GA is either smaller than or comparable to the number of executions of Precimonious. We see a tradeoff between speedup achieved and solution generation.

The efficiency of the various solutions, which is the FOM by the number of executions, is shown in Fig. 5. The number of program executions is directly proportional to the wallclock time spent by a technique to find the optimal PV. Our example search space is very large (2^{76}), so an efficient algorithm in terms of program executions is very important for finding precision vectors, and AMPT-GA is the most efficient of the three approaches. With only 10% of the executions used by the naïve solution, AMPT-GA is able to achieve 96.3% of the speedup in the 2.5 case. Even in the 0.65 case, AMPT-GA has higher efficiency than Precimonious despite finding a lower performing solution. The efficiency benefit over Precimonious is 14-63% and over naïve is 563-577%. The benefit of AMPT-GA becomes more pronounced as the error threshold increases because AMPT-GA has more room for aggressive reduced precision.

Fig. 6 shows the efficiency gain of each algorithm per program execution. The results shows that AMPT-GA is able to converge faster to a higher FOM solution when compared with the other two approaches. This happens because the GA can make big jumps unlike the more local search of delta debugging.

5.4 Experiment 2: Error Threshold Adherence

In this experiment, we validate that the PVs selected in the previous experiment satisfy the user-defined error constraint. Fig. 7 shows the performance of each approach and its ability to leverage available error. These results show that all approaches satisfy the error constraints, whether tight or loose. However, Precimonious is unable to leverage available error headroom for the 2.5 threshold.

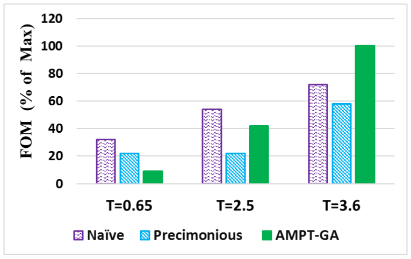


Figure 3: We compare the application's performance after precision optimization using the "Figure of Merit" (FOM), synonymous with throughput, normalized to the FOM with all floats. AMPT-GA is able to outperform Precimonious for two of the three thresholds, and it also outperforms the naïve solution for the top threshold.

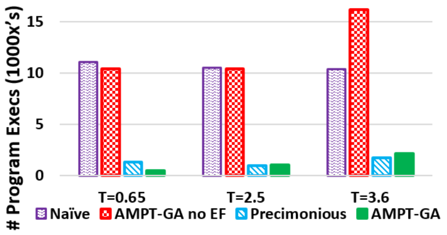


Figure 4: The number of function evaluations are shown for the three approaches. The naïve approach uses the most executions as it has no program knowledge. Precimonious uses a similar number of executions in its delta-debugging process as AMPT-GA. Also shown is AMPT-GA without the execution filter, showing the filter is highly effective.

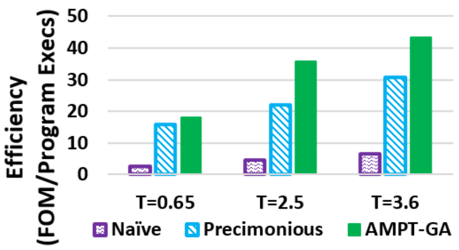


Figure 5: The efficiency of the three approaches. Efficiency is calculated as the FOM by the number of program executions. AMPT-GA is the most efficient approach for all three error thresholds.

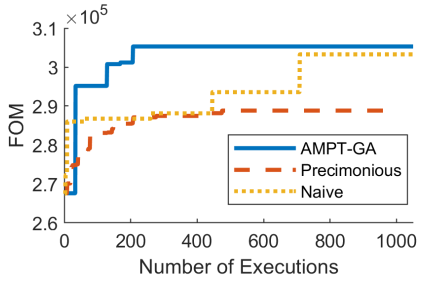


Figure 6: The FOM increase over number of program executions is shown for all three approaches. AMPT-GA is able to identify better configurations more quickly compared to the others.

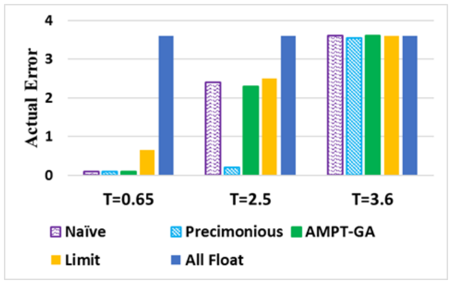


Figure 7: The error thresholds and actual errors are shown for the three methods. For the 2.5 threshold, Precimonious does not take advantage of its available error headroom, due to being stuck in a local minima. AMPT-GA follows the limit fairly closely for all points, leveraging the available error headroom.

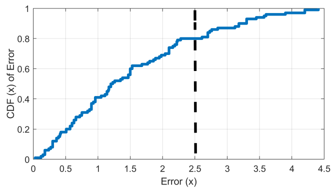


Figure 8: The error CDF is shown for the PV with a target error of 2.5 for inputs that are near the tested input. These results show that most of the neighbors (80%) fall within the error bound, so the result can be generalized with 80% confidence in this space.

5.5 Experiment 3: Component Testing

This experiment examines the impact of AMPT-GA's various components on the FOM metric. Figure 9 shows the resulting FOM for AMPT-GA with different features enabled and disabled for the T=2.5 case. We run AMPT-GA with all of its features, mirroring the result in Experiment 1 (labeled AMPT-GA). The first component comparison we make is against the naïve case where the performance, but not efficiency, is higher than in AMPT-GA. This is because the naïve case uses actual program performance rather than the static performance model, which allows it to measure the impacts of data movement and other non-arithmetic or dynamic operations that AMPT-GA does not model. Additionally some scores from our performance model are inaccurate because of un-modeled dynamic behaviors, resulting in execution filtering of potentially better candidate PVs. We then compare the impacts of including the all-float and all-double case in our initial population, labeled as population seeding in the figure. This has only a minor impact on the search because it steers the search toward the most aggressive (all float) case during the mutation/crossover operations, provided it meets the error constraints.

The second feature enables the group mutation without the pre-population, as the hyper-dimensional space with $G = 25\%$. This has the largest impact as it helps AMPT-GA overcome local mimima issues. The final feature is the execution filter and this has no impact on the performance, though it does significantly reduce the number of program executions as seen in Experiment 1. There is no impact because the GA to select the PV is driven by the performance model score, which is the same in the full AMPT-GA and AMPT-GA minus execution filter. When execution filter is tuned to randomly pass 90% of the rejected PVs, we achieve only minute FOM improvement over AMPT-GA.

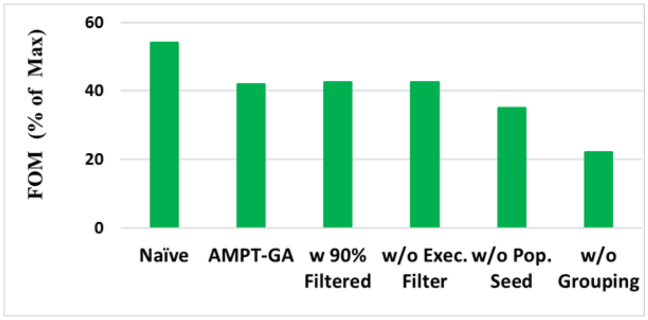


Figure 9: The FOM for the T=2.5 case is shown with different modules of AMPT-GA activated. The far left case replaces the performance model with actual program executions where it performs best, but at the cost of 10x additional executions. The grouping and hyper-dimensional space mutations has the largest impact, while the execution filter and the prepopulation have marginal impact, for this specific error threshold.

5.6 Experiment 4: Local vs. Global Models

In this experiment, we quantify the benefit of using application-level tuning over kernel or function level tuning. We observe that prior works focus on per-line (e.g., [4, 19]) or per-kernel (e.g., [10, 20]) optimizations. We validate this claim by running a simple linearity test between the individual kernel contributions and the application-level contribution to error and performance.

Table 1 shows the performance gain and error from the three kernels in LULESH, simplified as K1, K2, and K3, for a particular locally optimized precision vector. For example, in K1, the PV terms that impact K1 are set to all-float and the terms for K2 and K3 are set to all-double and the resulting application-level change in FOM and error are shown. The same is done for K2 and K3. If the kernel-level tuning is sufficient, then the total application FOM should be a summation of the individual kernel-level gains, but the application-level metric is actually lower by 15.1%. Similarly, for the error, the overall system error is much lower (52.6%) than the individual kernel errors combined.

5.7 Experiment 5: Precision Vector Generalization

In this experiment, we test the ability to generalize to inputs within a small range of the user-provided input value. For LULESH, we modify the template source code that defines the symmetric input evaluation points. These points lie in [0, 1.125], and we apply a distance (variance) of 0.1 to these points as $x' = x + 0.1 \cdot \mathcal{N}(0, 1)$. From this we collect 100 samples, and the CDF for these sample errors is shown in Fig. 8. The results have a confidence of 0.80 for the 100 test cases. This indicates that our searching time can be amortized across inputs that are nearby to these test inputs if the confidence level is acceptable to the end user.

Table 1: Independent Kernel Vs. Application Level (All Float Case)

	FOM Gain	Error
K1	32,286	3.596
K2	54,561	2.994
K3	8,701	0.000
K1+K2+K3	95,549	6.590
Application Level	81,088 (-15.1%)	3.124 (-52.6%)

5.8 Evaluation with Rodinia Benchmark programs

We evaluate our approach with three programs from the Rodinia GPU benchmark suite, namely, LavaMD, Backprop and CFD. LavaMD, a Molecular Dynamics program, is a single kernel program with 15 floating point variables. Backprop implements an algorithm in the domain of Pattern Recognition and is a two kernel benchmark with 21 floating point variables. CFD, a standard benchmark in the field of Computational Fluid Dynamics, is a four kernel program with 26 floating point variables. In the Backprop kernels, the choice of the precision of the floating point variables is between FP16 and FP32. All the other benchmarks are evaluated for choice between FP32 and FP64. These programs return single or multiple arrays of floating point data as the end result. Application-generic error is obtained by the average of the mean square root of the error associated with each element of each array.The error thresholds are chosen as with LULESH—analyzing the error behavior of the program with 1,000 unique PVs and selecting the 25%, 50%, and 75% quartile error values. Fig. 10, Fig. 11, and Fig. 12 show the results for the three different applications. We plot the same metrics as before—FOM, number of program executions, and efficiency and show for the naïve case, Precimonious, and AMPT-GA. We believe that the drop in FOM gain by AMPT-GA with respect to Precimonious in LavaMD is because of the smaller length of the PV (smaller by a factor of 5 in comparison to LULESH), which hugely reduces the search space and hence, both algorithms are able to sufficiently explore the search space resulting in close FOM values. With Backprop and CFD, AMPT-GA achieves the highest efficiency in all the cases, excepting the 2e-6 error threshold case in CFD. The general trend with these two benchmarks is that AMPT-GA achieves the FOM values as high as the Naïve GA and significantly higher than that of Precimonious but consumes more executions than Precimonious. The increase in FOM compensates for this additional number of executions making AMPT-GA the most efficient approach.

6 DISCUSSION

GPU-specific benefit. One benefit of reducing precision, specific to GPUs, is that it reduces the communication across the CPU-GPU interface, say through a PCIe interface. This is a bandwidth-constrained interface [24], with the throughput being far lower than the memory bandwidth of the GPU. Reducing the precision has the effect of reducing the memory traffic on this interface and speeds up the total execution time. For our test applications, the traffic volume is not significant enough for this effect to be substantial,

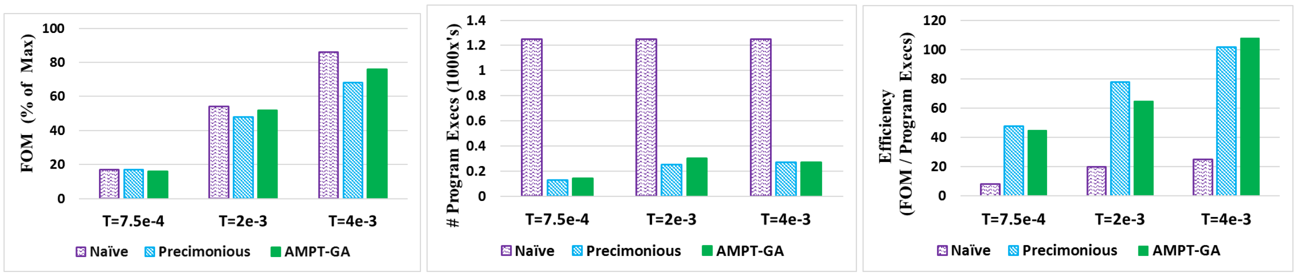


Figure 10: LavaMD results: Naïve GA achieves the highest FOM values in all the cases but uses significantly larger number of program executions compared to the other approaches. FOM achieved by AMPT-GA is higher than that by Precimonious.

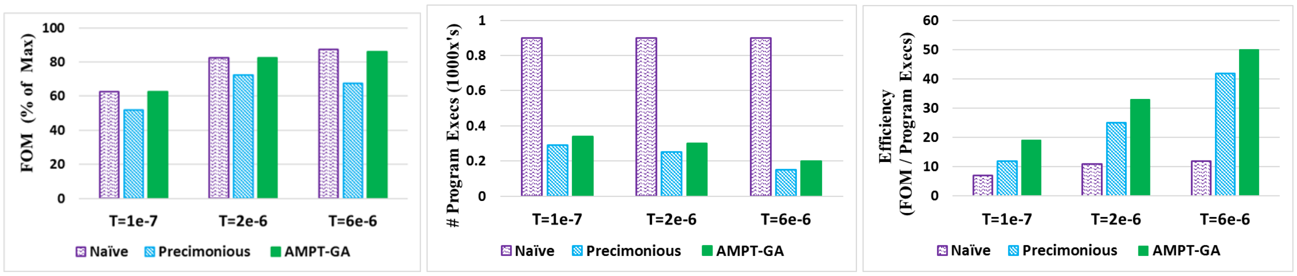


Figure 11: Backprop results: AMPT-GA achieves the same FOM values as Naïve GA while using less than a third of the number of executions. AMPT-GA emerges as the most efficient approach for all three error thresholds.

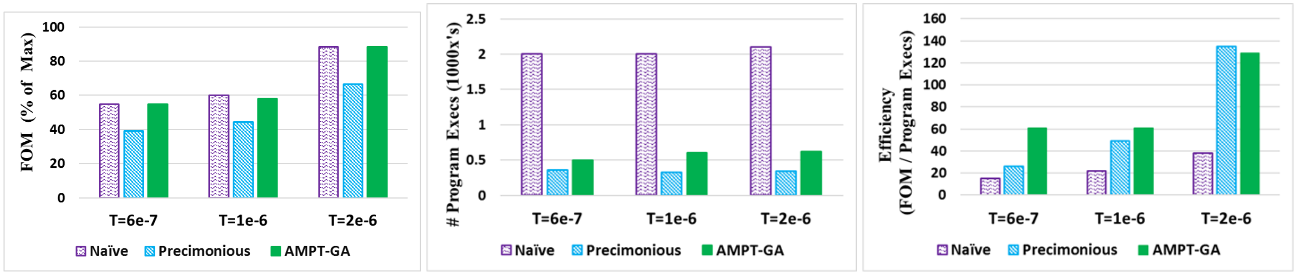


Figure 12: CFD results: Naïve GA and AMPT-GA achieve the highest FOM values in all the cases but Naïve uses significantly larger number of program executions compared to AMPT-GA.

but for applications with higher data transfer volumes [16] this will be important.

Error Accuracy and Overall Speedup. Many mixed precision papers focus on precise error calculations and guarantees for satisfying $E \leq T$. These apply at the operation level while guarantees at the application level require executing the actual program which creates a paradox: if the program must be executed in double precision to determine an error guarantee, then executing it in mixed precision provides no runtime benefit since the exact result is already known. Our results on generalizing the PV about an input region are useful in understanding risk. In scientific computing applications, error checking and uncertainty quantification are common and serve as a means to measure the error in an application-specific manner.

7 CONCLUSION

In this paper, we present AMPT-GA, a solution for optimizing the precision of variables in applications that have demanding GPU kernels. It brings two innovations: a static analysis to identify groups of variables whose precisions should be changed together and a search technique that is aware of the relevant penalties such as type casting and can eliminate unprofitable PVs promptly. AMPT-GA is able to improve the performance of a real-world application LULESH with 77.1% higher speedup than Precimonious for a mid-range error threshold. AMPT-GA achieves 96.3% of the speedup compared to a naïve search using only 10% of the program executions. AMPT-GA outperforms Precimonious in efficiency by 14-63% and naïve “black box” search by 563-577%. In future, we will create an error model that can act as another execution filter and further reduce the number of program executions.

REFERENCES

[1] [n. d.]. CORAL Benchmarks. *https://asc.llnl.gov/CORAL-benchmarks/* ([n. d.]).

[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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* Ieee, 44–54.

[3] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846

[4] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM, 300–315.

[5] A Conn, Nick Gould, and Ph Toint. 1997. A globally convergent Lagrangian barrier algorithm for optimization with general inequality constraints and simple bounds. *Mathematics of Computation of the American Mathematical Society* 66, 217 (1997), 261–288.

[6] NVidia Corporation. [n. d.]. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html# arithmetic-instructions__throughput-native-arithmetic-instructions. ([n. d.]).

[7] Eva Darulova and Viktor Kuncak. 2017. Towards a compiler for reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 2 (2017), 8.

[8] Kusum Deep, Krishna Pratap Singh, Mitthan Lal Kansal, and C Mohan. 2009. A real coded genetic algorithm for solving integer and mixed integer optimization problems. *Appl. Math. Comput.* 212, 2 (2009), 505–518.

[9] David E Goldberg and John H Holland. 1988. Genetic algorithms and machine learning. *Machine learning* 3, 2 (1988), 95–99.

[10] 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 (ISSTA).* ACM, 333–343.

[11] Ian Karlin, Abhinav Bhatele, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. 2012. *LULESH Programming Model and Performance Ports Overview.* Technical Report LLNL-TR-608824. Lawrence Livermore National Lab. 1–17 pages.

[12] Lawrence Livermore National Lab. 2017. LULESH 2.0 Benchmark Summary. https://asc.llnl.gov/CORAL-benchmarks/Summaries/LULESH_Summary_v1.pdf. (2017).

[13] Michael O Lam and Jeffrey K Hollingsworth. 2016. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications* (2016), 1094342016652462.

[14] Michael O Lam, Jeffrey K Hollingsworth, Bronis R de Supinski, and Matthew P LeGendre. 2013. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing.* ACM, 369–378.

[15] Michael O Lam and Barry L Rountree. 2016. Floating-point shadow value analysis. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools.* IEEE Press, 18–25.

[16] Reza Mokhtari and Michael Stumm. 2014. BigKernel–High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, 819–828.

[17] Andres Nötzli and Fraser Brown. 2016. LifeJacket: verifying precise floating-point optimizations in LLVM. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis.* ACM, 24–29.

[18] Mostofa Patwary, Sharan Narang, Eric Undersander, Joel Hestness, and Gregory Diamos. 2018. Experimental Evaluation of Mixed Precision Training for End to End Applications. http://research.baidu.com/Blog/index-view?id=103. (May 2018).

[19] 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.

[20] 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.* ACM, 27.

[21] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices* 49, 6 (2014), 53–64.

[22] Laurent Thévenoux, Philippe Langlois, and Matthieu Martel. 2017. Automatic source-to-source error compensation of floating-point programs: code synthesis to optimize accuracy and time. *Concurrency and Computation: Practice and Experience* 29, 7 (2017).

[23] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[24] Jia Zhan, Onur Kayıran, Gabriel H Loh, Chita R Das, and Yuan Xie. 2016. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 1–13.