Undergraduate Research Opportunity Program

(UROP) Project Report

Approximate Computing using Variable Floating-Point Precision

By

Asha Anoosheh

Department of Computer Science

School of Computing

National University of Singapore

2014

Undergraduate Research Opportunity Program

(UROP) Project Report

Approximate Computing using Variable Floating-Point Precision

By

Asha Anoosheh

Department of Computer Science

School of Computing

National University of Singapore

2014

Advisor: Prof Weng Fai Wong

Abstract

The choice of precision in computation requiring floating-point values is a significant one. High precision produces accurate results at the cost of performance and power. Lower precisions come with much less computational expense, though their use can lead to undesirably inaccurate results. The paradigm of approximate computing has emerged with the goal of finding compromises between accuracy and efficiency. It relies on the tendency that reducing the quality of a result during computation leads to significant improvements in energy and/or performance efficiency. It is hypothesized that lowering the precisions of values in a given program to variable, non-standardized lengths will result in nontrivial gains in power and performance aspects. This report explores the topic of floating-point numbers, past works on approximate computing related to floating-point numbers, and methods to quantitatively test the assumption.

Subject Descriptors:

G.1.2 Approximation
I.2.8 Problem Solving, Control Methods, and Search

Keywords:

Floating-point, precision, approximate computing, field-programmable gate arrays

Implementation Software:

Altera Quartus II, Altera AOCL SDK and compiler, Python 3.4, PyCParser 2.10

Acknowledgements:

I would like to thank Pooja Roy and Elavarasi Manogaran for aiding with the project and mentoring me along the way, as well as Professor Wong for giving me this opportunity to partake in undergraduate research at NUS.

Table of Contents

1. INTRODUCTION:

The notion of relaxing floating point precisions in programs has been around for a while; it provides faster computation and lower power consumption in most cases, though in exchange for decreased accuracy of results due to the loss of information. One common application of this is lossy compression via MPEG formats: the precision of uncompressed data is reduced to levels where approximation produces a comparable result, while significantly reducing the data size and increasing media playback speed. Previous works on finding optimal precision configurations for programs are all done using the fixed-length, standardized formats for floating-point numbers. In this study the tuning of floating-point values using variable, non-standard precisions is explored, and the outcome is to be tested for effectiveness for possible use with FPGAs. The crux of the process is the search process for the best precisions per program, and because of the non-standard nature of the problem, it requires a non-standard search procedure.

2. BACKGROUND:

Here the topics of floating point numbers, search algorithms, FPGAs, and similar previous studies on precision tuning used in the research are described.

2.1. FLOATING POINT VALUES AND ERROR:

Floating-point numbers, first standardized by the IEEE in 1985, are real numbers represented by a sign, exponent, and significand, with bit representation in that order. As digital storage of floating point numbers is in binary, the significand is a binary encoding of the digits following a floating point, with an implied 1 on the left hand side. Thus the final value is $(-1)^b \cdot 1.s \cdot 2^e$, where *b, s,* and *e* are the sign bit, significand, and exponent, respectively, notwithstanding special values and denormalized numbers, which are not necessary for this discussion.

One can grasp the fact that the infinity of all real numbers are not representable by a finite segment of space, thus only a certain range and selection of numbers can be stored properly in floating-point format - everything else is approximated. On top of this, it is not hard to see why performing arithmetic operations involving an approximated value and another value will almost inescapably result in further approximation. We call the former

truncation error, and the latter roundoff error. Because of the need for base-2 representation, any value whose denominator is not a power of two cannot be represented fully in binary. These unrepresentable values are irrational in base-2, and therefore require infinite bits. There are also some other fractions can be represented in full, but require a large number of digits to be stored in binary, often more than most registers have room for, as well as numbers that are too small to be stored in full and require a negative exponent much larger in magnitude than can be stored.

The other type of error involves floating-point arithmetic. Operations involving these values often requires truncation or rounding of the final computed value to fit into its register. For example, addition between two numbers with different exponents is done most easily by shifting one of the values to give it the same exponent as the other, then they are summed. The error is very clear in the following example where we assume 8-digit length:

1.2345678 x 10^7 *1.2345678 x 10^7*
\+ *1.9876543 x 10^3* becomes \+ *0.000019876543 x 10^7*

1.234587676543 → Rounded to 8 digits = ***1.2345877***

We can see the shifting causes the last digits (76543) to be dropped down, not added to anything, and the restriction on value length causes us to cut the digits in those places out of the final value, resulting in an imprecise value. This is also easily seen in other operations such as multiplication where the product, by nature, usually has more digits than the operands.

The length of the significand is what determines the precision of the value: the longer the length, the more bits are able to be stored, and thus the effects of precision-related error are further to the right - further from the most significant digits. The current IEEE 754 standard provides multiple precision levels, most commonly 32-bit (single-precision) and 64-bit (double-precision) values. In many languages, the former corresponds to the “float” type and the latter to the “double” type, which have the following number of bits for {sign, exponent, significand}, respectively: {1,8,23} and {1,11,52}. Other standard formats are available as well, including 16-bit (half precision,) 80-bit (extended precision,) and 128-bit (quad precision.)

We can see, for example, the difference in using single vs. double-precision values in a scientific computation. We can observe a sample program that calculates the arc length of a function given by $g(x) = x + \sum_{n=0}^{5} 2^{-k} sin(2^k x)$ over the interval $(0, \pi)$, which sums $\sqrt{h^2 + (f(x_k + h) - f(h))^2}$ for $x_k \in (0, \pi)$, divided into n subintervals with $n = 1000000$ (therefore $h = \pi/1000000$ and $x_k = kh$). This program run on an Intel-based machine produces an output of 5.795776322413031 when using single-precision values, and the more precise 5.795776322412856 when using double-precision (Bailey, 2012). In general, the maximum error, or machine epsilon, possible when using 32-bit variables is exactly 2^{-24}, while for 64-bit, the max epsilon is 2^{-53}.

Using larger-precision variables also comes with significant cost. Obviously it requires at least twice the amount of space to store, and more to perform operations on inside memory, than single precision values. This means operations where four single-precision values could be held are reduced to two doubles; more bits are carried through the hardware, and more bits have to be operated on. In practice, due to additional overhead of operations on these larger values - sometimes larger than the registers of the machine - the performance of single-precision operations is often observed to be at least 2.5 times as fast as their double-precision counterparts (Lam, Hollingsworth, Supinski, and Legendre, 2013). On the Intel Xeon 5100, the double precision peak is fourteen times as slow as the single precision. And on NVIDIA's GPUs running CUDA code, double-precision operations take anywhere from 3 to 24 times longer than single-precision ones (Whitehead and Fit-Florea). The increase in time corresponds to more clock cycles, and therefore more power spent.

The topic of interest, though, is not merely for finding ways to approximate computations using standard, fixed-size floating-point units, but rather using precisions of any size. Certain calculations that would, for example, fit acceptable error had there been a 33-bit format instead of just a choice between 64 and 32 would now be able to be found in a configuration. Unfortunately, this also results in a vastly increased search space, for instead of two or three precisions, there are now infinite - though we would not want to go above any precision that a program already utilizes as it only makes sense to decrease precision. A GNU library known as MPFR (Multiple Precision Floating-Point Reliably) is the means to

actualize variable-precision in the proposed project, and specifics pertaining to it will be covered later on.

2.2. POTENTIAL SEARCH METHODS:

The novel approach in this study is to observe the effect of tuning variables to any precision. The problem such an approach presents is that there aren't two or three precisions per variable, but, assuming finite memory, exponentially more options to search through. In fact, even with restricting MPFR precision to no larger than 128-bit variables, we have 128^n total search configurations per piece of source code; for a sample program with, say, 25 floating-point variables, we'd already have nearly $4.8 \cdot 10^{52}$ config options. Exhaustive search is clearly not an option, and binary search or BFS won't work well here, but there are other ways to find approximate solutions.

The simplest method is by randomly searching: either random walks or random sampling. Random sampling involves picking randomly from a search space in the purest sense. If we have a specified target T in the sample space then (assuming uniform probability) p=**U**(T) will be small due to T being small; the probability of hitting the target within m trials would be $(1-(1-p)^m)$, which stays close to zero unless m approaches 1/p, in which case it approaches $1-e^{-1} \approx 0.63$. A random walk is a search starting at a randomly selected point and randomly choosing neighboring options to search next, rather than scattering the search all over the space. Ensuring the random walk has no biases for which neighbor is chosen next, the expected time to reach the target T will be on the same order of magnitude as $\frac{1}{U(T)}$, which is no better than a random sampling in the number of steps required to reach a certain target (Dembski, 2005). The advantages random search has is that it can quickly provide a desired number of samples to search within an overall space too large to search. The sampled points may or may not be adequately representative of the entire space, since both methods presuppose a uniform probability on the search space.

The solution to the shortcomings of fully random searches come in the form of assisted searches, where each candidate provides information about what candidate would be better to choose next. They usually involve heuristic functions or probabilities for determining the strength of selected points. For example, once a point is selected, it may be analyzed, and the search finds out that it is a bad choice and avoids future candidates with

similar characteristics. Their effectiveness is completely dependent on the problem type and implementation at hand.

2.3. FPGAs:

The variable-precision programs need somewhere to be run and tested on. Even more importantly, they need a platform where they will likely make a very significant difference. This can be done using field-programmable gate arrays (FPGAs,) integrated circuits that are configurable after they are built, possible due to their large amounts of logic and RAM. The prediction is that in the near future, FPGAs will have variable-precision registers, since they are aimed at providing configurable hardware. Altera's latest FPGAs, the Stratix V, Cyclone V, and Arria V, all contain variable precision digital signal processing (DSP) blocks. These can hold 9-bit to single-precision floating point values and perform operations with them (along the DSP datapath only for now.) Additionally, a paper written in 2006 from researchers at Northeastern University describes a library they've created named VFloat that allows for variable precision on FPGAs only. This may prove useful to look further into for reference and implementation details.

3. RELATED WORKS:

A number of previous studies done on tuning floating-point precision in programs provide groundwork for this research. Though they were done using fixed precisions rather than variable.

3.1. "PRECIMONIOUS":

The most critical inspiration for our experimentation in this field is a graduate-student project conducted at Lawrence Berkeley National Laboratory by the name Precimonious. Precimonious aimed and succeeded in analyzing user-given programs written in C in order to find a reconfiguration of numeric variable types, reducing the precision of certain variables in the program such that the final result still stays within a user-provided error-range. The authors claim it has resulted in performance improvements of up to 41% (Rubio-Gonzalez et al, 2013). Precimonious operates on doubles and long doubles (usually 80 or 128-bit depending on the architecture and compiler) in C, changing select double variables to floats and long doubles to either doubles or floats. Then it runs each new configuration of code

compiled using LLVM. Precimonious is also integrated into the frontend of the LLVM C compiler, allowing it to extract variables and their types during the compilation process, intercepting it and creating bitcode with the variables modified each time before running. Though this eliminates the need to parse code for variables and their types, it introduces an extra layer in the process, as well as dependency on external software (LLVM,) which is generally not desired. Precimonious utilizes a modified form of an algorithm known as delta-debugging, where the search space, defined by the floating-point types and the variable they belong to, starts at the default precision given and is broken down similar to binary search. The search space is first broken into two equally-sized spaces, where the variables in each half are reduced in precision, separately, and the program run twice (once with each new half) and if neither result in an acceptable result (determined by a user-provided epsilon) then the search space is instead broken into four chunks and the process continues until a viable configuration is found. This process will often not result in the optimal solution, or global minimum, but rather finds a local minimum, as the search on the entire space of every configuration would have to evaluate P^n combinations, where P and n are the number of precision types and number of tunable variables, respectively. The delta-debugging algorithm instead averages $O(n \log n)$ with a worst case of $O(n^2)$. The search is done based on the user inputting sample values for the variables, which should ideally represent the most common case or range expected. So even a configuration that has a significant performance improvement with the sample values may have highly different results in worse cases.

3.2. "AUTOMATICALLY ADAPTING PROGRAMS…":

Another experiment relating to the problem at hand was done by researchers in both the University of Maryland and the Center for Applied Scientific Computing in California. The project is titled "Automatically Adapting Programs for Mixed-Precision Floating-Point Computation." The premise is similar to that described in Precimonious, though here it deals only with two precision types, double and single. The search is done via breadth-first search (BFS), with a binary search optimization that breaks up large functions in half to reduce the number of configurations that need to be searched when there are large blocks with unmodifiable sections spread throughout sparsely. Another optimization is made to speed up the search, where a profiling is done on the code initially to determine which lines are run the most, and to tune precision of variables in those lines first to give the most promising

outcome. After the BFS, the program then attempts to union all the previously found successful configurations, and tests the new configuration out to see if it still meets requirements, since tuning different sections of code is not independent of each other. In one case this variant of precision tuning has led to a speedup of 2X.

3.3. "FLOATWATCH":

A project known as FloatWatch, integrated into the popular code analysis framework ValGrind, was released in 2007 as a way of reducing precision and/or range of variables in floating-point computations. For each assembly instruction produced, FloatWatch provides bucketized sub-ranges of values, overall range of values, and the maximum difference between 64 and 32-bit executions. It will optimize in many different ways, first of which is removing excess zeros: quite often, zeros are received as inputs during computation and performing the operations with a number and zero is a waste of resource. Another way is by identifying denormal values (numbers smaller than the standard floating point format can store) and, if possible, use specially-optimized hardware to be produced for such numbers, as often times these are implemented in software by typical processor designs (Brown, Kelly, and Luk). In addition to converting some variables from double to single precision, FloatWatch also determines when to convert some floating-point variables into fixed-point (integer) formats when narrow range is required. Another type of conversion FloatWatch recommends is to convert values to Dual Fixed-Point representation, which has the efficiency of fixed-point with the flexibility of floating-point. The format contains a bit at the front to switch between fixed-point representations by giving two options for the location of the floating point; the most significant bits of the remaining 63 bits of the value act as the exponent, and as the range (exponent) is increased, the precision decreases. FloatWatch identifies said cases and returns the areas of code where variables can be altered to improve performance, rather than running it, itself.

3.4. "ACCELERATING SCIENTIFIC COMPUTATIONS…":

A 2008 publication in Elsevier Journal by a group of researchers from various locations titled "Accelerating scientific computations with mixed precision algorithms" aimed to enhance the performance of many algorithms while maintaining 64-bit accuracy of the solution. The paper states, "in many cases, a single precision solution of a problem can be

refined to the point where double precision accuracy is achieved" (Baboulin, Buttari, Dongarra, and Kurzak, 2008). The topic focuses on linear systems, either sparse or dense, whose solution can be found via LU-factorization of the coefficient matrix using Gaussian elimination. At each step in the elimination, round-off errors carry onto the final solution of the problem; the proposed algorithm iteratively creates a correction to the computed solution at each step, which yields the iterative refinement algorithm. Provided the refinement procedure converges in a small number of steps, its cost will be relatively little compared to the factorization. The iterative refinement itself can be achieved using Newton's method, an algorithm for computing the zero of a function. The refinement process is done in double-precision, unlike the factorization (most expensive operation) which is single-precision. The article points out that because the round-off errors are nonlinear, the iterative refinement process is equivalent to solving Newton's method for $f(x) = b - Ax$, where our system is of the form $Ax=b$. The results of this mixed-precision method turned out to be relatively close to claims: on a Cell Broadband Engine running at 3.2 GHz, the LU Solve program resulted in performance near that of full single-precision configuration (mixed at 100 Gflop/s and single at 130 Gflop/s for problem size of 3500) while the peak performance for double precision was around 17 Gflop/s.

4. IMPLEMENTATION:

Multiple tools and steps are used to physically implement, run, test, and analyze the entire proposed process as a whole. Precimonious, FloatWatch, and all the other papers gave various useful examples of searching floating-point configurations as well as evidence that significant performance gains are achievable. MPFR provides the actual way to represent and use variable-precision. Conversion to MPFR-compatible source code is a necessary subprocess and will be performed as part of the experiment. The information on large search spaces, MCMC, and genetic algorithms lend insight into ways to search the enormous search space unique to this problem. All this is to be tested for results via FPGA using software specialized in simulation and performance analysis of the code.

4.1. VARIABLE PRECISION WITH MPFR:

The MPFR C library provides floating-point software-based implementations to store and perform common operations on variables of precision that can be between the smallest

and largest integer values (depending on the size of an integer for a specific machine.) Initialization of variables using MPFR can be done using the *mpfr_init2(var, precision)* function, where *var* is a proprietary MPFR type (*mpfr_t)* and precision is the integer length of the significand. Assignment is done via *mpfr_set(rop, op, rnd)*, assigning op to rop and using the given rounding method, either rounding to nearest, toward zero, toward infinity, toward minus-infinity, or away from zero. Arithmetic and other operations (only binary ops) are done via special functions that take a two variables and assign the result of the operation into a third variable, much like a triple in assembly instruction. Various other functions exist as well.

Rarely are user programs written using MPFR, so to tune third-party source code using the variable precision MPFR can provide, code would have to either be manually or automatically re-written to fit MPFR notation. And as there are no tools currently out there supporting this operation, creating a standard-C to C-with-MPFR converter, as well as a way to recursively run programs with different precision configurations would be a necessary tool to create to facilitate the process of testing large existing benchmarks using variable precision.

4.2. C TO MPFR TRANSLATION:

The process of translating standard C source code to MPFR compliant code involves parsing the source for all floating point variables and changing all operations involving them into MPFR library function calls.

The basis for this conversion is done using a C parser written in Python by the name of PyCParser. The parser reads through a given C source file and breaks apart each instruction, storing them into a specialized parse tree. This tree can then be traversed by another module of the parser package called the generator, which reads off the instructions step by step recursively and prints them back out. The returned code is identical in function to the source code (comments are stripped away and some extra parenthesis may be inserted.) The objective here was intercepting the regeneration of the pre-parsed code and replacing as many instances of floating point instructions with their MPFR equivalents.

The syntax structure required for MPFR operations brings about some complications. Firstly, MPFR variables must be initialized in addition to assignment. Secondly, arithmetic operations such as addition, subtraction, multiplication, and division must be in the form of

an assembly triple, with a destination and two operands only. Thirdly, comparison operators must be replaced with special MPFR versions. There are many scenarios much more complex to translate, but the goal remains to automate as much of the conversion process as possible.

Sample C floating-point operations and their MPFR equivalents:

float a; or *long a;*	*mpfr_init2(a, custom_precision);*
float a = 4.2000;	*mpfr_init2(a, custom_precision);* *mpfr_set_d(a, 4.2000, rounding_mode);*
a = b + c;	*mpfr_add(a, b, c, rounding_mode);*
long x = a;	*mpfr_init(x, custom_precision);* *mpfr_set(x, a, rounding_mode);*
if (a > b)	*if mpfr_greater_p(a, b)*
a++;	*mpfr_add_si(a, a, 1);*
*a = b * c / d + e - f*	*mpfr_init(dummy_1, custom_precision);* *mpfr_init(dummy_2, custom_precision);* *mpfr_mul(dummy_1, b, c, rounding_mode);* *mpfr_div(dummy_2, dummy_1, d, rounding_mode);* *mpfr_add(dummy_1, dummy_2, e, rounding_mode);* *mpfr_sub(a, dummy_1, f, rounding_mode);*

The current stage of the translation tool can perform the above operations, with rounding modes set by default to a single chosen type. For example, an incoming line being visited by the generator script "*double c = a - b;*" would be in the form of a declaration node. A special function to handle declarations would then be called automatically, which would notice the declaration as a double, and initializes as "*mpfr_init2(c, config_array[n]);*" where *n* is the number of mpfr-variables initialized prior to this. Then, due to the equals sign, it realizes it is an initialization in addition to just a declaration, so instead of ending here, it calls the assignment function which splits the right hand side by arithmetic operators, sees there are only two operands, and chooses the appropriate mpfr arithmetic function based on the

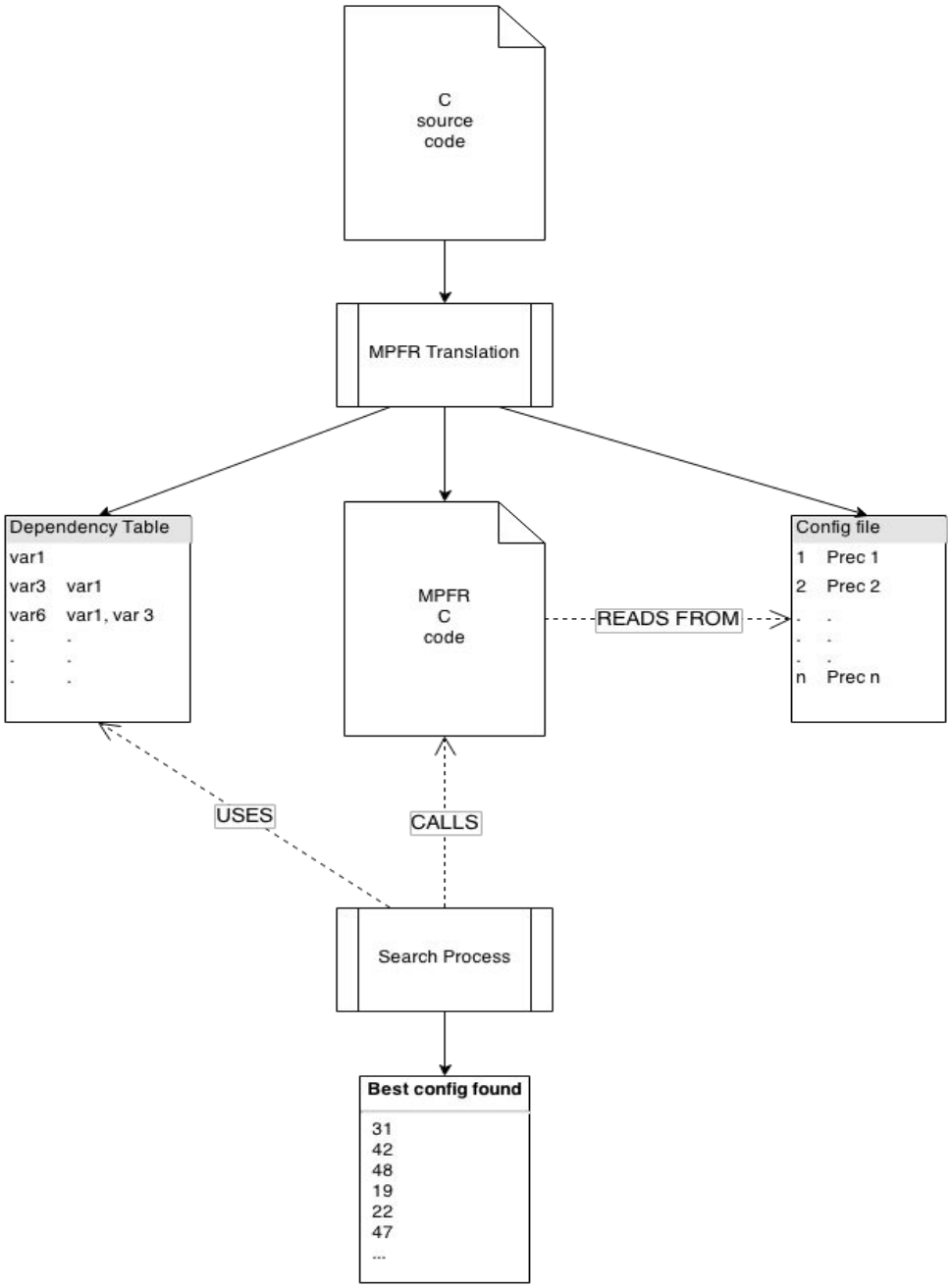
minus sign found - in this case "*mpfr_sub(c, a, b, MPFR_RNDD)*" where *MPFR-RNDD* is a predefined rounding-direction flag - not important to us.

The parser also ignores operations involving integers and no floating-point variables, and even has the ability to automatically generate dummy variables for arithmetic involving over two operands. Though there are still things that need human proofreading to ensure correctness. For example, setting an mpfr value to a function requires knowing the function's return type to use the correct setter method. Complex arithmetic operations with mixed operators must be checked for correct order-of-operations. And there are many more cases in which the limitations of MPFR require manual intervention. The tool takes care of the most common cases that would be too tedious to replace by hand.

4.3. FRAMEWORK:

The process of searching different configurations involves running the program with such values at each step to ensure its result remains within acceptable error bounds. Doing this in an efficient manner requires the framework to be structured such that a search algorithm can alter precision numbers and run the program many times without re-compiling in between. To achieve this, the MPFR-translation tool logs the precisions of floating-point variables in an external file as comma-separated values. This configuration file must be read by the target program at first, with the values being read into an array, using a snippet of generic code that can be placed manually anywhere near the start of the MPFR-converted program prior to compiling. The MPFR conversion tool also automatically sets the precision of each MPFR variable to an index in the array, where a variable's index is its order of initialization in the source code. (For example, variables *a* and *b* declared in that order in the original C code will have precisions *config_array[0]* and *config_array[1]*, respectively.) The dynamic reading of precision values from an external file means the program does not need to be recompiled at each step. The entire setup itself allows a search algorithm to tweak values in the configuration file, call the compiled program executable to run, check the results, and repeat with the next search step. In addition, the converter has been outfitted to also keep track of variables that (most likely) are dependent on previous variables in the code, where any declaration or assignment statement with an MPFR variable on the left and right hand side means the left depends in some way on the right hand side; these are stored in a

dictionary where keys are the MPFR variable names and their values are arrays of variables they possibly depend on, according to the definition of dependency above.



Overview of workflow

4.4. TRAVERSING THE SEARCH SPACE:

As introduced before, assisted searches are prime candidates for search algorithms in the experiment. Here we introduce possible search algorithms for our experiment and their effectiveness.

4.4.1. MCMC:

A certain class of algorithms, known as Markov-Chain Monte Carlo (MCMC) methods, are well-known for searching large sample spaces. These algorithms sample from a probability distribution based on creating a Markov chain with the same desired distribution as its equilibrium distribution. The chain itself is then used as the final sample of the distribution, with each step improving in desired quality as the search progresses. The most common type of MCMC are random-walk Monte Carlo methods. These are a type of Monte Carlo method, which is a general method for obtaining numerical results using random sampling of statistically independent samples. The random walk Monte Carlo methods operate on samples that are not independent, but correlated. At each point a random "walker" moves around, adding the integrand value of the point to an overall multi-dimensional integral; it then moves around the neighborhood of that sample looking for a spot that will contribute greatly to the integral. The most famous of these random walk MCMCs is the Metropolis-Hastings algorithm: given a probability distribution *P(x)* this algorithm draws a sequence of samples such that they eventually approximate the desired distribution, *P*. The distribution of the next-to-be-chosen sample value depends solely on the current sample value (since it is an iterative process,) making the sequence of samples a Markov Chain (a markov chain is a system that changes states according to probabilities of where it goes next; the next state is memoryless - only dependent on the current state.) For the Metropolis-Hastings algorithm a random point x_0 is chosen, and an arbitrary yet symmetric proposal density *D(x|y)* (commonly a Gaussian distribution centered at y) is used to choose the next value *x* given *y*. Then iteratively, a new candidate *x'* is picked each time after the previous value *x* using *D(x|y)*, and the acceptance ratio $\alpha = f(x')/f(x)$ is calculated to decide whether to keep the candidate or not (since *f* is proportional to *P*, $\alpha = P(x')/P(x)$ as well.) If $\alpha \geq 1$ then it is a better candidate and we add *x'* to the sequence and it becomes the next *x*; if not, *x'* is kept with probability α. This way, the points more probable than the last, according to *P(x)*, are

always chosen, while the rest are sometimes chosen, returning samples that follow *P*'s distribution. Overall, it is a semi-random walk with probabilistic decision-making, that is useful for returning a desirable set of samples from a very large search space.

Adapting this method to the problem presented would involve providing an appropriate proposal density and customizing the search to work with the specifics of the problem. The proposal densities can be given as input interchangeably, and though none have been tested yet, the initial hunch is one centered at a midway value between single and double precision - say 48 bits of precision.

The optimization that can be used is reducing the number of dimensions in our search space. Knowing that some variables in the program depend to a degree on other variables means changing the precision of one would likely require the precisions of those dependent on it to also change to the same precision anyway. As mentioned before, the MPFR converter also keeps track of variables during generation. This allows for searching along search spaces of specific variables rather than all of them, reducing the search space significantly. Another very simple yet effective way to increase productivity of the search is to focus on lines in the code that are run most often. GCC compiler can provide this information, and searches can be directed toward those instructions containing floating-point values. Details of implementation and results of MCMC-based searches are still to come.

4.4.2. GENETIC SEARCH:

Another form of algorithm that may prove useful in the search is falls under the umbrella of genetic algorithms. This type of search uses biological natural selection as its basis; it requires a heuristic function that evaluates the strength of each sampled value. First, a randomly selected set of values are taken from a large space and evaluated using the heuristic function. The strongest of these values is then chosen (some percentage) for the next round of evaluation. Then pairs (or other tuples) of these values are mixed together, with the idea that they probably possess certain beneficial aspects, and combining them may lead to a stronger value. Additionally, some samples can be mutated (random changes made within them) for increased variety. Then the next round is run and is repeated until a best value or best values are returned. Such algorithms depend on the power of their heuristic function and are well suited for problems where parameters are to be tweaked, as ours is.

A basic genetic search algorithm has been implemented in Python to read values from the configuration file, create a user-specified number of initial random configuration lists, runs each one (this can be done in parallel,) throws out those that don't fit error requirements, as well as cutting off the bottom user-specified percentage of configurations that scored most poorly. Then the remaining population is merged, where a user-specified percentage of lists are merged by pairs, and then a percentage of them are mutated randomly. The initial population is completely randomized, with each precision element of the list chosen randomly between a user-provided minimum (recommended 16 bits) and the original precision of the variable. The scoring is currently not based on running time of the program, but rather the sum of the precisions in the program. This is due to the assumptions that MPFR variables will all take approximately the same time to run (on a PC) since they're emulated in software and the assumption that lower precision overall will correlate to faster performance and lower power consumption when run on a device with hardware support for variable-precision. The merging is done by taking a pair of configuration lists, and creating a new list by choosing an element either from one of the two lists with a 50-50 chance per index. The mutation is a separate percentage supplied to the algorithm performed after merging, so it may only operate on some of the merged lists and some non-merged lists. Mutation creates a random step size and goes through the list, giving those indices landed upon a new random value drawn from a 32-mean gaussian distribution. The final configuration set at each step can contain a many types of lists to promote variety: ones that were merged only, mutated only, merged and mutated, or untouched and carried over. Additionally, the algorithm keeps track of the maximum scoring configuration of the entire search, in case it is lost along the way through non-beneficial merging or mutation. The algorithm works on a sample set but has yet to be put into practice with an actual program.

4.5. FUTURE ANALYSIS:

Results of the search are meant to be utilized on FPGAs, the only devices capable of variable-width registers for holding our variable-precision floating point numbers. To test the effectiveness, an FPGA logic design software from Altera Corporation called Quartus II is used to perform performance analysis on the proposed hardware design for an FPGA board. The current state of the analysis step is blocked for the completion of the search portion of the research.

For FPGAs, a hardware description language (HDL) is used to determine the configuration of the board's circuit, with each configuration optimized to perform the specific application/program provided. HDL can be easily generated using high-level synthesis (HLS,) whereby source code written in a native language (C/C++) is compiled through numerous processes down to clock-level timing and other hardware specifications for the FPGA board, then automatically produces readable HDL. This removes the massive complexity of handwriting HDL for large applications and allows for the HDL to be put through an analysis tool - in this case, the Quartus II software - to return power-usage, timing, and performance reports. The current HLS program used is the AOCL (Altera OpenCL) compiler which can take OpenCL (written in a host language, either C or C++) and produce design files compatible with Quartus II for analysis.

5. CONCLUSION:

To conclude, the work so far has resulted in a framework for searching various precision configurations in a program, a conversion tool to translate C to MPFR-syntax code, and multiple embryonic search programs to be developed further. The current research is a work in progress to be carried on after the writing of this paper and by graduate students in the upcoming months. Once the setup of the experiment is completed, the searches and analyses will be up and running. Then the results of the analyses can tell whether or not tuning the precision of variables to nonconventional lengths produces significant power reductions and performance speedup for applications fit to run on FPGAs.

References

Baboulin, M., Buttari, A., Dongarra, J., & Kurzak, J. (2008, November 13). ScienceDirect. *Accelerating scientific computations with mixed precision algorithms*. Retrieved October 19, 2014, from http://www.sciencedirect.com/science/article/pii/S0010465508003846

Bailey, D. (2012, April 5). Resolving Numerical Anomalies in Scientific Computation. DavidHBailey. Retrieved October 16, 2014, from http://www.davidhbailey.com/dhbpapers/numerical-bugs.pdf

Brown, A. W., Kelly, P. H., & Luk, W. (n.d.). Profiling floating point value ranges for reconfigurable implementation. FloatWatch. Retrieved October 17, 2014, from http://valgrind.org/docs/floatwatch2007.pdf

Dembski, W. A. (2005, March 4). Searching Large Spaces: Displacement and the No Free Lunch Regress. Detecting Design. Retrieved October 18, 2014, from http://www.detectingdesign.com/PDF%20Files/Dembski%20-%20Searching_Large_Spaces.pdf

Goldberg, D. (2000, April 5). Appendix D. What Every Computer Scientist Should Know About Floating-Point Arithmetic. Retrieved October 17, 2014, from http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Kroese, D., Taimre, T., & Botev, Z. (n.d.). Handbook of Monte Carlo Methods - Homepage. University of Queensland - Maths. Retrieved October 19, 2014, from http://www.maths.uq.edu.au/~kroese/montecarlohandbook/

Lam, M. O., Hollingsworth, J., Supinski, B. d., & Legendre, M. P. (2013, June 10). Automatically adapting programs for mixed-precision floating-point computation. Retrieved October 19, 2014, from http://dl.acm.org/citation.cfm?id=2465018

Melquiond, G. (n.d.). Gappa Documentation. Gappa. Retrieved October 19, 2014, from http://gappa.gforge.inria.fr/

Rubio-Gonzalez, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., et al. (2013, November 17). PRECIMONIOUS. Lawrence Berkeley Laboratory Library. Retrieved October 19, 2014, from http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/corvette/precimonious/

Sentieys, O. (n.d.). Approximate Computing: a New Paradigm for Energy-Efficient Computing Architectures. Institut de Recherche en Informatique et Systèmes Aléatoires. Retrieved October 19, 2014, from http://www.irisa.fr/en/offres-theses/approximate-computing-new-paradigm-energy-efficient-computing-architectures

Stratix V FPGA Variable-Precision DSP Block Architecture. (n.d.). Altera News. Retrieved October 19, 2014, from http://www.altera.com/technology/dsp/variable-precision/architecture/stratix-v-dsp-block.html

Wang, X., Braganza, S., & Leeser, M. (2006, April 26). Advanced Components in the Variable Precision Floating-Point Library. IEEE Xplore. Retrieved October 19, 2014, from http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4020913&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D4020913

Whitehead, N., & Fit-Florea, A. (n.d.). Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. Nvidia Developers. Retrieved October 16, 2014, from https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf