Preprint of C. Wilcox, M.M. Strout, J. Bieman. Tool Support for Software Lookup Table Optimization. Scientific Programming, 19(4):213-229, 2011

Tool Support for Software Lookup Table Optimization

Chris Wilcox*, Michelle Mills Strout, and James M. Bieman
Computer Science Department, Colorado State University

Abstract

A number of scientific applications are performance-limited by expressions that repeatedly call costly elementary functions. Lookup table (LUT) optimization accelerate the evaluation of such functions by reusing previously computed results. LUT methods can speed up applications that tolerate an approximation of function results, thereby achieving a high level of *fuzzy reuse*. One problem with LUT optimization is the difficulty of controlling the tradeoff between performance and accuracy. The current practice of manual LUT optimization adds programming effort by requiring extensive experimentation to make this tradeoff, and such hand tuning can obfuscate algorithms.

In this paper we describe a methodology and tool implementation to improve the application of software LUT optimization. Our Mesa tool implements source-to-source transformations for C or C++ code to automate the tedious and error-prone aspects of LUT generation such as domain profiling, error analysis, and code generation. We evaluate Mesa with five scientific applications. Our results show a performance improvement of 3.0× and 6.9× for two molecular biology algorithms, 1.4× for a molecular dynamics program, 2.1× to 2.8× for a neural network application, and 4.6× for a hydrology calculation. We find that Mesa enables LUT optimization with more control over accuracy and less effort than manual approaches.

Keywords: lookup table, performance optimization, error analysis, code generation, scientific computing, memoization, fuzzy reuse

*Corresponding Author: *1873 Campus Delivery, Fort Collins, CO 80523*
phone 970.491.5792, fax 970.491.2466, wilcox@cs.colostate.edu

1 Introduction

Applications in scientific computing are often performance-limited by elementary function calls [28, 33]. Such functions are common in scientific code, so designers have long studied how to accelerate them with lookup table (LUT) hardware [9, 29]. Despite hardware support, software libraries often equal or exceed hardware performance, as shown in Table 1, possibly because software evolves more quickly than hardware [9]. Software LUTs can improve elementary function performance [33], but determining if a software LUT is applicable and optimizing its parameters by hand is cumbersome.

LUT optimization partitions the input domain of expressions or functions into intervals. Each interval is represented by a LUT entry that stores a single output value that is shared across all inputs in the interval. The original expression is replaced by an indexing function that locates the interval and returns its corresponding LUT entry. A LUT optimization is beneficial when (1) enough *fuzzy reuse*[1]occurs to amortize the LUT initialization and overhead, (2) the LUT access is significantly faster than evaluation of the original function, and (3) the LUT can provide the needed accuracy without excessive memory use, such as is the case when the input domain is restricted.

Performance tuning methods such as LUT optimization require a substantial development effort for scientific programmers [26], because such domain-specific optimizations are usually applied manually [17]. Manual LUT optimization is time-consuming because of the need to explore parameters such as the table size and sampling method that determine the tradeoff between performance and accuracy. This paper presents a methodology and source-to-source transformation tool that automates the most time-consuming and

[1]Fuzzy reuse is a concept introduced by Alvarez et al. [2] in which function results are approximated in order to increase reuse.

Table 1: Performance of elementary function instructions.
(Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

x86 Instruction	**Execution Time**	**Math Library**	**Execution Time**	**Relative Performance**
FSIN	35.2ns	sin	36.5ns	+3.6%
FCOS	33.9ns	cos	36.9ns	+8.8%
FPTAN	72.9ns	tan	51.8ns	−28.9%
FSQRT	8.1ns	sqrt	1.8ns	−77.7%

error-prone steps required for LUT optimization, including error analysis, domain profiling, and code generation.

Our LUT research was motivated by a collaboration with the Small Angle X-ray Scattering (SAXS) project [24] at Colorado State University (CSU). For the project we received R code that simulates the discrete X-ray scattering of proteins. We ported the SAXS code from R to C++, but its performance still failed to meet requirements. To reduce execution time we manually incorporated a LUT optimization for the dominant calculation. The result was a significant speed up, but our *ad hoc* approach for determining optimal LUT parameters was costly and had to be repeated for each architecture of interest. To simplify future LUT tuning we developed the Mesa tool. We have since used Mesa to tune a continuous version of SAXS scattering, a molecular dynamics program, a neural network, and a hydrology calculation.

Table 2 shows the performance improvement and error statistics achieved using Mesa. The top two rows show variants of the SAXS application [24], the third row shows the Stillinger-Weber molecular dynamics program [14], the fourth row shows a CSU neural network application [19], and the fifth row evaluates a calculation from a Precipitation-Runoff Modeling System (PRMS) [21]. We attribute the effectiveness of LUT optimization to the high level of fuzzy reuse inherent in these applications, which are described in more detail in Section 3. Relative error is listed in all cases except PRMS, which reports absolute error. The relative error is misleading in this case because the computation of relative error divides the error by the original output. If this value is close to zero then the resulting relative error is huge, despite the small magnitude of the absolute error.

Table 2: Mesa performance improvements and error statistics.
(Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)
Relative error is shown for all applications except hydrologic analysis, which reports absolute error because its results are close to zero.

Application Name	**Orig. Time**	**Optd. Time**	**Speedup**	**Maximum Error**	**Memory Usage**
SAXS Discrete Algorithm	283s	41s	6.9×	$5.4X10^{-3}\%$	4MB
SAXS Continuous Algorithm	.726s	.239s	3.0×	$1.8X10^{-3}\%$	3MB
Stillinger-Weber Program	14.7s	10.3s	1.4×	$2.9X10^{-2}\%$	400KB
CSU Neural Network	11.0s	3.9s	2.8×	$6.2X10^{-2}\%$	4MB
PRMS Hydrologic Analysis	234ns	53ns	4.6×	$3.2X10^{-5}$	800KB

In a previous workshop paper [31], we described version 1.0 of the Mesa tool, which lacked support for domain profiling and linear interpolation and required a separate specification of candidate expressions outside the source code. We used this preliminary version of Mesa to optimize SAXS discrete scattering with 1D and 2D LUT optimization, and we showed that performance improvements in the scope of the SAXS application required the LUT data to fit into mid-level cache. A programmer using this version had to (1) manually identify candidate expressions and their domains, (2) specify the expressions and constituent variables in a file, (3) run Mesa to generate code, and (4) manually integrate the resulting code back into the application.

In this paper, we present a pragma-based approach to apply LUT optimization to expressions in C or C++ source code, thereby extending the reach of Mesa to full applications. We show that version 1.1 of Mesa reduces programming effort by automating the generation and integration of LUT optimization code. We have added boundary error analysis to improve tool performance, and linear interpolation sampling to increase accuracy. The major benefit of this work is to allow the programmer to more easily find an effective set of optimizations, while receiving feedback about the introduced error. We have also evaluated version 1.1 of Mesa in the context of four additional scientific applications, as shown in Table 2 and described in Section 3.

The primary contributions of this paper are:

- A methodology for software LUT optimization, so that programmers will no longer need to depend on *ad hoc* methods.
- A demonstration of domain profiling, error analysis, and code generation in the context of full applications using our Mesa tool.
- Additional experimental results that suggest the LUT data must reside primarily in mid-level cache to be effective.
- An investigation of the error and performance differences between direct access and linear interpolation.
- A case study showing that LUT optimization maintains its effectiveness in the context of parallel execution on a multicore architecture.

Section 2 introduces our LUT optimization methodology and the Mesa tool. Section 3 presents case studies on the use of Mesa to optimize applications, and evaluates the effectiveness of the tool in terms of performance and programming effort. Section 4 explores related work and Section 5 describes limitations. Section 6 lists threats to validity and Section 7 concludes.

2 LUT Optimization Methodology and Mesa

The goal of our research is to help scientific programmers use LUT optimization in a more effective and efficient manner. Because the existing literature lacks a systematic methodology for LUT optimization, we define our own. We have broken down the process into the following steps:

1. Identify elementary functions and expressions for LUT optimization.
2. Profile the domain and distribution of the LUT input values.
3. Select the LUT size based on the desired input granularity.
4. Analyze the error characteristics and memory usage of the LUT data.
5. Generate data structures and code to initialize and access LUT data.
6. Insert the generated LUT code into the application.
7. Compare performance and accuracy of the original and optimized code.

The current practice for LUT optimization is to write code manually, often without careful analysis of the performance or error implications. Our methodology is independent from its implementation in a tool, for example we can apply the steps shown above manually. However, a methodology is especially important to when considering automation. We developed the Mesa tool to automate the most time-consuming part of the methodology shown above, including steps 2, 4, 5, and 6. Mesa can optimize elementary function calls including sin, cos, tan, exp, log, and sqrt. Mesa also optimizes expressions identified by pragmas, including combinations of elementary functions and arithmetic operators: +, -, *, and /. A description of how Mesa supports the methodology follows.

Mesa does not support identification of candidate functions and expressions (Step 1), since many suitable profiling tools such as gprof [13] already exist. Mesa automates domain profiling (Step 2) by generating an instrumented version of the application to capture the domain boundaries and distribution. LUT size selection is specified on the command line (Step 3), however Mesa provides detailed error analysis (Step 4) so that the programmer can iterate size selection and error analysis until the LUT optimization meets the application requirements. Mesa reduces development time by completely automating code generation and integration (Steps 5 and 6). The comparison of the original and optimized code (Step 7) remains manual.

2.1 Elementary Function Optimization

To demonstrate the methodology with a simple example, we show the optimization of a single elementary function with Mesa. Figure 1 shows command line and program output from a Mesa run that optimizes sine calls. The command line specifies the original and optimized source files, elementary functions(s) to be optimized, table size, and error analysis method. Mesa uses the Rose compiler infrastructure [23, 16] to parse the file into an abstract syntax tree (AST) on which it can operate.

Figure 2 shows the generated code, which is defined and instantiated as a C++ class with everything needed for the optimization, including a public method for the LUT approximation function. Mesa inserts the C++ class at the beginning of the module, and replaces instances of the original function with a call to this method. Mesa calls Rose to unparse the modified AST into an output file with the LUT code. The optimized file is substituted for the original file and the application is rebuilt.

The sine tables in Figure 3 are generated by running Mesa with a direct access sampling method, meaning without any form of interpolation between entries. We show small LUT sizes of 16 and 32 entries to illustrate the error terms. The original function is shown by $f(\theta)$ and the approximation is shown by $l(\theta)$. The area between the original function and approximation represents the magnitude of the error introduced by the LUT, which is plotted as $e(\theta)$. For direct access, the error term is error at the center of each LUT interval, because Mesa evaluates the original function at the center to compute the output value. The maximum error occurs at the boundaries of the LUT interval. The graphs in Figure 3 show the tradeoff between LUT size and accuracy. Tables with more entries exhibit smaller errors and vice verse.

```
./Mesa original.cpp optimized.cpp -exhaustive -lutsin -lutsize 2048
Mesa LUT optimization started
Lower Bound: 0.000000e+00
Upper Bound: 6.383185e+00
Granularity: 3.116790e-03
Lut size: 2048
Error analysis: exhaustive
Emax: 1.53e-03,  Eavg: 4.88e-04
Replaced sin with clut.sin
Mesa LUT optimization completed
```

Figure 1: Optimizing elementary functions with Mesa.

```
// Start of code generated by Mesa, version 1.1
const float fsinLower = 0.000000e+00;
const float fsinUpper = 6.383185e+00;
const float fsinGran = 1.595796e-03;
class CLut {
  public:
    // LUT Constructor
    CLut() {
      for (double dIn=fsinLower; dIn<=fsinUpper; dIn+=fsinGran)
        lutsin.push_back(sin(dIn + (fsinGran / 2.0)));
    }
    // LUT Destructor
    ~CLut() {
      lutsin.clear();
    }
    // LUT Approximation
    float sin(float fsin) {
      while (fsin < 0.0f) fsin += (2.0f * M_PI);
      while (fsin > (2.0f * M_PI)) fsin -= (2.0f * M_PI);
      int uIndex = (int) (fsin * (1.0f / fsinGran));
      return(lutsin[uIndex]);
    }
  private:
    // LUT Data
    std::vector<float> lutsin;
};
// Object instantiation
CLut clut;
// End of code generated by Mesa, version 1.1
```

Figure 2: Listing of code generated by Mesa.

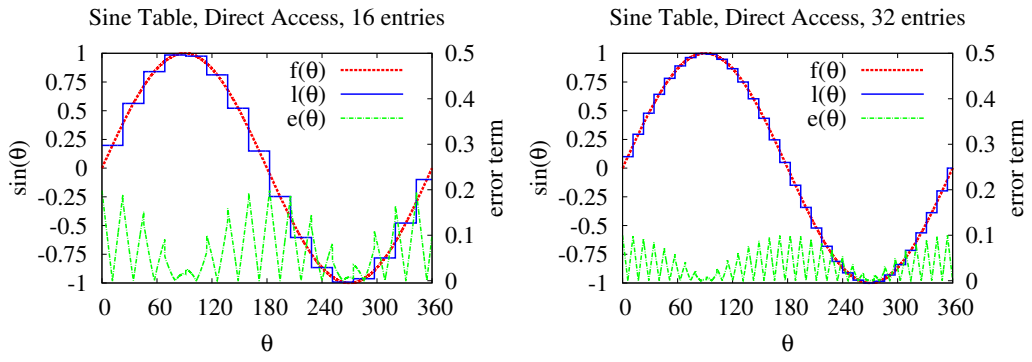

Figure 3: Lookup tables for sine function: direct access.

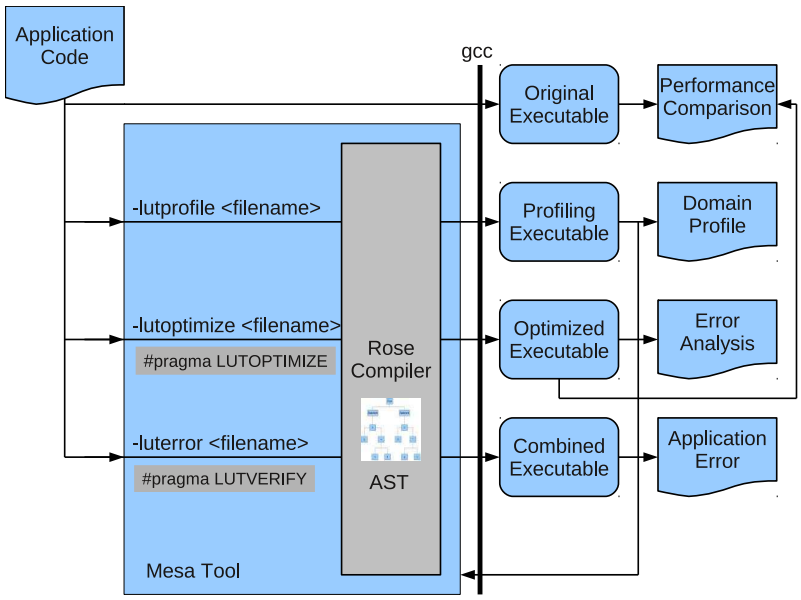

Figure 4: Workflow diagram for Mesa.

2.2 Pragma-Based Expression Optimization

Figure 4 shows the workflow for expression optimization, which is a superset of elementary function optimization. The programmer starts by running the original code to establish a baseline for performance and accuracy. Domain profiling and optimization require the programmer to insert a pragma into the C or C++ source code to identify the target expression. Elementary function optimization does not require pragma insertion because elementary functions are easily identified by Mesa.

Figure 5 shows a code fragment after addition of the pragma. The code shown is the dominant calculation of SAXS discrete scattering. Domain profiling is initiated with a command line option that creates an instrumented version of the unoptimized program. When the program runs, profiling information is gathered and written to a data file. The workflow continues by running Mesa with the same pragma flag, this time requesting expression optimization. During optimization Mesa reads the profiling data, performs an error analysis, and generates LUT code and data structures. As with elementary function optimization, the LUT size must be specified.

```
// Iterate steps (outer loop)
for (step = 0; step < 1000; ++step) {
  // Iterate atoms (middle loop)
  for (atom1 = 0; atom1 < vecAtoms.size(); ++atom1) {
    // Iterate atoms (inner loop)
    for (atom2 = atom1; atom2 < vecAtoms.size(); ++atom2) {
      ...
      // Compute distance between atoms
      float fDistance = distance(atom1, atom2);
      // Compute scattering angle
      float fTheta = m_fStep * (float)(step + 1);
      // Combine parameters to scatter
      float rTheta = fDistance * fTheta;
      ...
      // Optimize subexpression shown below
      #pragma LUTOPTIMIZE
      fIntermediate = sinf(FOURPI * rTheta) / (FOURPI * rTheta);
      ...
    }
  }
}
```

Figure 5: Insertion of a pragma for expression optimization.

```
./Mesa original.cpp optimized.cpp -pragma -lutsize 200000 -exhaustive
Mesa LUT optimization started
Enter parameters for rTheta
Lower bound: 0.0
Upper bound: 0.2
Variable:    rTheta
Lower Bound: 0.000000e+00
Upper Bound: 2.000000e-01
Granularity: 1.000000e-06
Lut size: 200000
Error analysis: exhaustive
Emax: 2.79e-06,  Eavg: 1.00e-06
Mesa LUT optimization completed
```

Figure 6: Optimizing an expression with Mesa.

Figure 6 shows the Mesa command line and output for expression optimization. The *-pragma* flag causes Mesa to insert code for the specified LUT optimization into the application, and Mesa optionally performs error analysis. The programmer may need to run Mesa several times to determine whether the LUT optimization can meet accuracy requirements while fitting into mid-level cache, but repeated runs can be scripted. When this is complete the programmer compiles and runs the generated code and compares it performance and accuracy against the original version. The next sections provide detail on domain profiling, error analysis, and code generation.

2.3 Domain Profiling

To determine the extent of LUT data, we must capture the domain of each input variable. Mesa does this through a profiling option that generates a transformation to add instrumentation to the program. The programmer can characterize the input domain by running the instrumented program with representative data sets. Mesa stores domain information in a data file that can be edited, so that a programmer can use domain expertise to adjust the domain boundaries if necessary. If the data file is missing, Mesa prompts for domain values. When the domain is known in advance, the programmer can optimize the code without profiling by creating the data file manually. Mesa additionally supports the generation of assert statements that halt execution and report an error condition if the domain is exceeded during program execution.

Some elementary functions are cyclical, so the input domain is intrinsically known. For example, sine and cosine tables need only store the interval from 0 to 2π radians. Input values outside of that interval can be folded back by a modulo operation or iterative addition and subtraction. This operation is called *domain conditioning* or *range reduction*. Mesa has a command line option to generate domain conditioning code for cyclical functions such as sine and cosine, as shown in Figure 2.

Mesa captures the domain of input variables and the number of executions of the expression. The former is necessary to build LUT data, and the latter is intended for future support of performance modeling. We have experimented with a modified version of Mesa that captures the complete distribution of

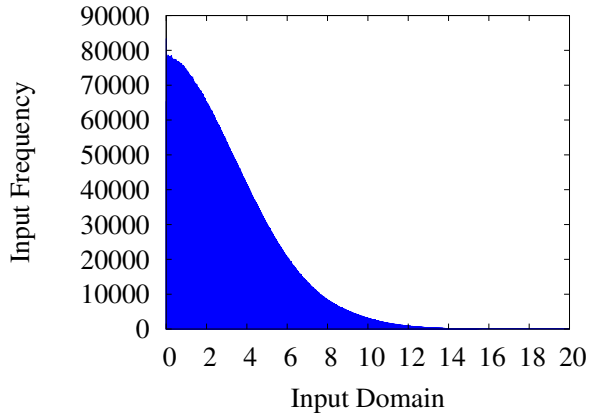

Figure 7: Domain distribution for SAXS lookup table.

input data. Figure 7 shows the distribution of the input variable $rTheta$ from the SAXS discrete scattering application. Note that the left half of the LUT is used much more frequently than the right side.

Figure 7 suggests that some LUT optimizations may benefit from storing only a partial domain. This requires the generation of conditional code that accesses LUT data only inside the partial domain. Outside of the partial domain the original function must be called. For the applications we have evaluated, the cost of the conditional negates the benefit of the reduced LUT size, but our preliminary data is too limited to generalize this result.

2.4 Error Analysis

Error analysis computes LUT error statistics by combining the individual error terms as shown in Figure 3. We are primarily interested in the maximum error $E_{maximum}$ and the average error $E_{average}$ over the entire LUT domain. These statistics allow us to characterize the error introduced by the LUT approximation. Computing error statistics is essentially a problem of sampling the LUT domain. Within each interval the error analysis code computes error terms for a set of samples by evaluating the original and approximation functions. Mesa implements three different algorithms for sampling: exhaustive, stochastic, and boundary. Each computes the error statistics for all LUT intervals, then combines the results to get values for the entire LUT.

The exhaustive method performs a brute force numerical traversal of the input domain, at a resolution of FLT_EPSILON or $1.2X10^{-7}$. This yields an exact answer for single-precision LUT data, but the method is very expensive. On our benchmark system (Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core), exhaustive sampling requires approximately 40 seconds to analyze the domain $[0.0, 20.0]$ for the SAXS discrete scattering. This fails to meet our goal of performing error analysis on multiple expressions in real time.

The stochastic method samples the domain randomly. The key drawback for this method is the lack of a general rule for determining the number of samples required to converge on an accurate answer. Our experiments show stochastic sampling computes a value of $E_{average}$ that approaches the exhaustive method with an order of magnitude fewer samples, however its computation of $E_{maximum}$ can vary widely. The performance of stochastic sampling for the domain listed above is less than 5 seconds on the benchmark system. Because of the inaccuracy of the $E_{maximum}$ computation, we have deprecated the stochastic method in Mesa.

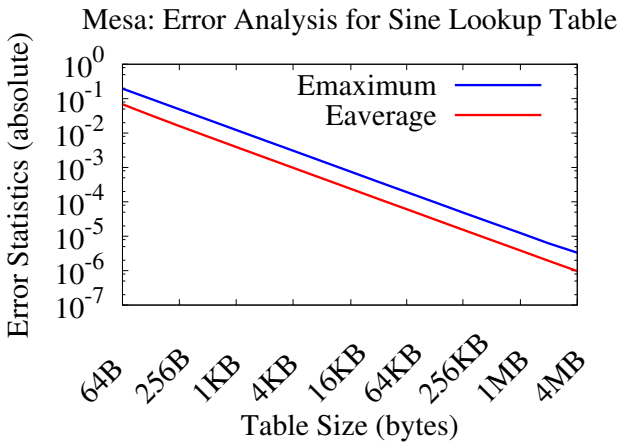

Figure 8: Error for different sizes of sine tables.

The boundary method makes the simplifying assumption that the maximum error can be found at the LUT interval boundaries for direct access, and at the LUT interval center for linear interpolation, as proposed by Zhang et al. [33]. This computation takes less than 0.2 seconds on the benchmark system, and the results are identical within $1.2X10^{-5}$ of exhaustive sampling for the applications we evaluate in this paper. The boundary method computes only $E_{maximum}$, since $E_{average}$ requires more extensive sampling. We use the boundary method for fast determination of error statistics.

Figure 8 shows error statistics generated by Mesa. We collected the data by calling Mesa from a script for the series of LUT sizes shown. Exhaustive error analysis was used to compute both $E_{maximum}$ and $E_{average}$. The graph shows one of the main benefits of Mesa, which is that the tool allows the programmer to evaluate the accuracy of different LUT sizes. Note the linear relationship between accuracy and table size. This is common for smooth functions such as trigonometric operations. For the sine table we see a $2\times$ decrease in error for each $2\times$ increase in table size.

We conclude from the graph that LUT sizes that fit easily within mid-level cache on a modern processor can produce usable error values. For example, the 256KB sine table shown in Figure 8 has a maximum error of $4.82X10^{-5}$ and an average error of $1.53X10^{-5}$. In Section 3 we show that such tables can significantly increase performance.

So far we have only discussed the error that occurs at the time that a function result is approximated. Returning an imprecise function result causes a computational error, which is propagated through the application by further calculations that use the function result. The numerical analysis of this error propagation is beyond the scope of the current version of Mesa, but measuring the application error is critical to the success of the LUT optimization. For this reason, our methodology includes a manual comparison of the accuracy of the original and optimized application. For each application in this paper, we compute the maximum error and average error based on the final output of the application. These statistics, along with the measured performance improvement, form the basis for deciding whether a particular LUT optimization is beneficial.

2.5 Code Generation

We have shown the code generated by Mesa in Figure 2. Mesa uses the Rose compiler [23] to implement code transformations, thereby leveraging the ability of Rose to parse and unparse arbitrary C and C++ syntax. Code generation and code integration is done by the Mesa tool itself. The current implementation inserts a C++ object at the beginning of the source module being processed. A global declaration ensures that the object is constructed when the application starts and destroyed when it ends. The initialization of LUT data is performed by the constructor, based on a reconstructed function that mirrors the original computation, and LUT data is freed by the destructor on application exit. The LUT approximation function is public and can therefore be called from anywhere in the application.

The current implementation operates on C or C++ code, but the resulting program must be compiled with C++ because Mesa generates code that contains C++ objects and containers. Mesa generates single-precision LUT data, which is sufficient for the applications and LUT sizes we have studied. Single-precision uses less cache memory, and we find linear interpolation is more effective for increasing accuracy than double-precision. However, double-precision LUT data could potentially benefit applications with high accuracy requirements and very restricted domains. Further investigation is require to characterize how such a change of precision would affect accuracy and performance. The topic of using single-precision versus double-precision math in scientific computations is explored by Buttari et al. [4].

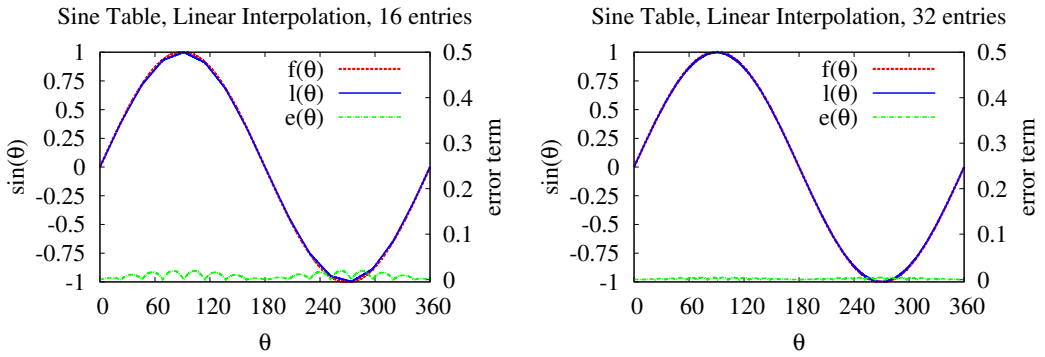

Figure 9: Lookup table for sine function: linear interpolation.

2.6 Sampling Methods

The sampling method determines how the output is computed from LUT entries. The simplest form of LUT optimization uses *direct access*, which returns the closest individual LUT entry to an input value. We can reduce approximation error without increasing memory usage by employing *linear interpolation* between adjacent LUT entries. Interpolation improves the LUT accuracy but adds computation and an extra LUT access for each evaluation, since the LUT value on both sides of the input value must be fetched.

The sine tables in Figure 3 are made using direct access, while Figure 9 graphs the same 16 and 32 entry sine tables using linear interpolation. Linear interpolation combines the two closest LUT entries based on the relative distance from the input value. In contrast to direct access, linear interpolation has zero error at the boundaries instead of the center. Conversely the maximum error is close to the center for linear interpolation. Accuracy can be further improved by techniques such as polynomial reconstruction, which is commonly used in hardware solutions where the increased computational load can be handled by additional circuitry. We plan to add polynomial reconstruction as a future enhancement to Mesa. Mesa supports direct access by default, and linear interpolation is enabled through a command line option. We find that linear interpolation improves the error by approximately an order of magnitude for the elementary functions. In Section 3 we compare the accuracy and performance of direct access and linear interpolation on the SAXS discrete scattering code.

In summary, we have defined a methodology for LUT optimization and its implementation in the Mesa tool. Mesa optimizes elementary functions and

expressions that combine those functions with basic math. A programmer uses Mesa by inserting one or more pragmas to identify candidate expressions. The programmer then runs Mesa to create a profiling version of the application that captures and stores domain information. Mesa reads the domain information and automatically generates a LUT optimized version of the application. The process completes when the user evaluates the performance and accuracy of the optimized code versus the original. In the next section we evaluate Mesa on a set of scientific applications.

3 Case Studies

We evaluate our methodology in terms of ease of use, accuracy, and performance by using Mesa to optimize five scientific applications. The first two applications are part of the SAXS project [24], a multi-disciplinary project at CSU between the Molecular Biology, Mathematics, Statistics, and Computer Science departments. SAXS is an experimental technique that explores the structure of molecules [11]. SAXS can be simulated on a computer via discrete or continuous algorithms. A partial discrete scattering simulation was written in the R language by members of the Statistics department, then ported to C++ and completed by one of the authors of this paper. A complete simulation of continuous scattering was written in $MATLAB$ by members of the Math department, then ported to C++ by various people in the Computer Science department. The SAXS code base for both algorithms currently consists of around 5000 lines of C++, not including documentation, data, and test code. The third application is Stillinger-Weber, a molecular dynamics program developed and used for research at Cornell University [14]. It consists of slightly more than 3000 lines of C. The fourth application is neural network code [19] developed by a faculty member in our Computer Science department and used in [3], which contains around 1100 lines of C. We have modified the C applications minimally to support C++ compilation and Mesa optimization. The fifth case study evaluates a computation from the Precipitation-Runoff Modeling System (PRMS) developed by the United States Geologic Survey [21] for hydrologic modeling. We were given Java code containing an expensive function that computes a *slope aspect*, which we have converted into approximately 1000 lines of C++. We refer again to Table 2 for a summary of the performance improvement and error statistics for each application.

3.1 Molecular Biology

Our first case study is the SAXS application that simulates the discrete scattering of a molecular model using Debye's formula [11] shown in Equation (1).

$$I(\theta) = 2\Sigma_{i=1}^{N-1}\Sigma_{j=i+1}^{N} F_i(\theta)F_j(\theta)sin(4\pi r\theta)/(4\pi r\theta) \quad (1)$$

The formula computes the intensity based on the interaction of pairs of atoms in the molecule. The code with the dominant calculation was shown in Figure 5. The $fDistance$ variable represents the distance between atoms, calculated in the middle loop. The $fTheta$ variable is the scattering angle, which varies for each iteration of the outer loop. The only elementary function called in the loop is the sine function.

Performance profiles of the SAXS scattering code show that the expression with the sine call dominates the computation, so we identify this expression with a pragma. We invoke Mesa multiple times with the pragma option, varying the LUT size. Mesa constructs a LUT optimization of the expression on the right-hand side of the assignment statement following the pragma. A previous run using the Mesa profiling option captured the input domain as approximately 0.0 to 20.0, so Mesa builds a LUT over this domain.

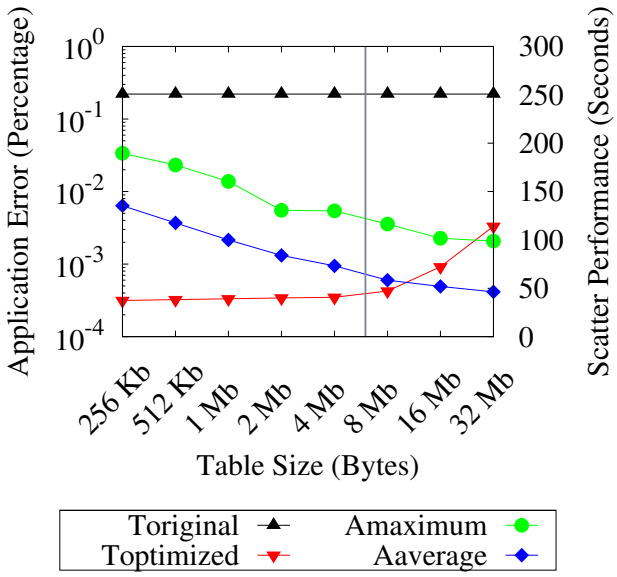

Figure 10: SAXS discrete scattering simulation results
(Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

```
for (int j = 0; j < m_vecGeometry.size(); j++) {
    ...
    // Scattering equation
    dSum0 += m_vecGeometry[j].fDensity * exp(dSigma * (fProduct/2.0))
                                       * sin(vecProduct[j]);
    dSum1 += m_vecGeometry[j].fDensity * exp(dSigma * (fProduct/2.0))
                                       * cos(vecProduct[j]);
}
```

Figure 11: SAXS continuous scattering code.

For each table size, we run the application and compare the performance and accuracy with that of the original code. Figure 10 shows the results from this experiment. The original time $T_{original}$ and optimized time $T_{optimized}$ are measured against the right axis. The vertical line in the graph indicates the amount of L2 cache in the system. The graph shows that the original time is constant and the optimized time increases with the memory usage. $A_{maximum}$ and $A_{average}$ are measured against the left axis.

Based on the graph in Figure 10, we select 4MB as the optimal LUT size to stay within the biochemists' requirements for accuracy without overflowing the L2 cache. The performance improvement is 6.9×, with a maximum error $4.6X10^{-3}$%. For the SAXS discrete scattering code we initialize the table by evaluating the expression approximately 1 million times. The resulting table is accessed 4.6 billion times by the application, so each table entry is reused more than 4,600 times on the average.

Our second case study uses code that implements another set of equations that model X-ray scattering, using a continuous instead of a discrete algorithm. The C++ code for the inner loop of continuous scattering is shown in Figure 11. Note the use of three elementary functions: exponential, sine, and cosine. The equations that define continuous scattering are shown in Equation (2).

$$I(q,\psi) = \left(\sum_{j=1}^{N} d_j e^{-\sigma_j^2 q \cdot q/2} cos(q \cdot \mu_j(\psi)) \right)^2 + \left(\sum_{j=1}^{N} d_j e^{-\sigma_j^2 q \cdot q/2} sin(q \cdot \mu_j(\psi)) \right)^2 \quad (2)$$

Performance profiles of the SAXS continuous code show that elementary function calls dominate the computation. We invoke Mesa to optimize the sine, cosine, and exponential function calls, varying the table size. Figure 12 shows the results of optimizing the continuous scattering code, with the same statistics as for the discrete case. Optimized performance varies from 4.5× to

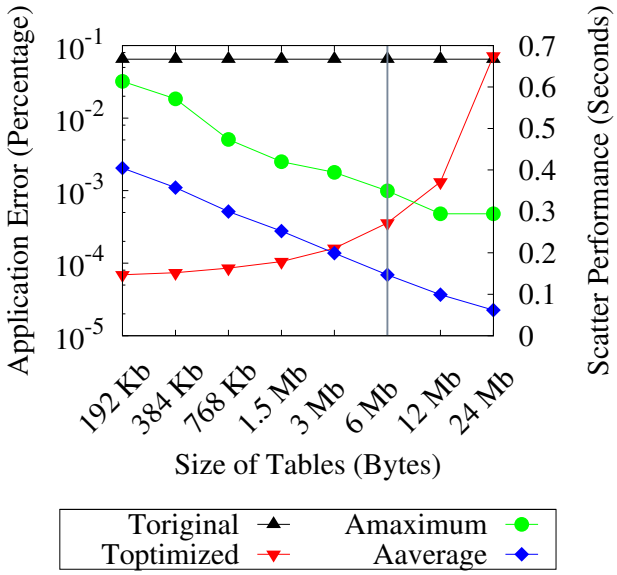

Figure 12: SAXS continuous scattering simulation results.
(Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

2.5× of the original when the LUT fits into L2 cache, then quickly degrades to slower than the original code when L2 cache is exhausted. The graph in Figure 12 shows 3MB as the optimal LUT size. The performance improvement at this point is 3.2×, with a maximum error $1.8X10^{-3}$%. The LUT size on the graph represents total memory usage of all three elementary function tables. During initialization, the continuous scattering computes 100,000 LUT entries for each of the three elementary functions, but the application calls each function at least 7.9 billion times, for an average of over 70,000 reuses per LUT entry. LUT optimization of discrete and continuous scattering improves for larger molecules, as shown in Table 3. The main reason for this is that LUT initialization is amortized over more computation.

3.2 Molecular Dynamics

Our third case study is Stillinger-Weber [14], a molecular dynamics program that models the physical movement of atoms and molecules by computing the potential energy and interaction forces of particles. The simulation is performed over a series of time steps to predict particle trajectories. Many molecular dynamics applications exist, but we have chosen Stillinger-Weber

Table 3: Saxs optimization performance and error on different molecules (Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

Algorithm Type	Molecular Model	Number Atoms	Speedup	Maximum Error	Memory Usage
Discrete Scattering	4gcr.pdb	1556	6.8×	$3.7X10^{-3}\%$	4MB
Discrete Scattering	1xib.pdb	3052	6.9×	$4.6X10^{-3}\%$	4MB
Discrete Scattering	3eqx.pdb	5897	7.7×	$9.4X10^{-3}\%$	4MB
Continuous Scattering	4gcr.pdb	1556	2.9×	$7.1X10^{-4}\%$	3MB
Continuous Scattering	1xib.pdb	3052	3.0×	$1.8X10^{-3}\%$	3MB
Continuous Scattering	3eqx.pdb	5897	3.8×	$1.9X10^{-2}\%$	3MB

because the code contains a manual LUT optimization done by the original authors. The dominant calculations in Stillinger-Weber are based on the potential energy equations [27] of the same name, which take into account 2-body (ϕ_2) and 3-body (ϕ_3) interactions that call the exponential function, as shown in Equations (3) and (4):

$$E = \sum_i \sum_{j>i} \phi_2(r_{ij}) + \sum_i \sum_{j\neq i} \sum_{k<i} \phi_3(r_{ij} r_{ik} \theta_{ijk}) \tag{3}$$

$$\phi_2(r_{ij}) = A_{ij}\epsilon_{ij}[B_{ij}(\frac{\sigma_{ij}}{r_{ij}})^{p_{ij}}]exp(\frac{\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}}) \tag{4}$$

$$\phi_3(r_{ij}, r_{ik}\theta_{ijk}) = \lambda_{ijk}\epsilon_{ijk}[cos\theta_{ijk} - cos\theta_{0ijk}]^2 exp(\frac{\gamma_{ij}\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}})exp(\frac{\gamma_{ik}\sigma_{ik}}{r_{ik} - a_{ik}\sigma_{ik}})$$

The original version of Stillinger-Weber optimized the 2-body and 3-body calculation by precomputing multiple lookup tables for series of expressions. To evaluate Stillinger-Weber we removed LUT optimization code from the original version and inserted straightforward implementations of the Stillinger-Weber equations into the 2-body and 3-body loops. We used the resulting unoptimized version of Stillinger-Weber as a baseline for performance and accuracy. The process required several minor modifications before we could run Mesa, including type casts to allow C++ compilation and the coalescing of the 3-body computation into a single expression. Profiling showed that the 3-body computation was dominant, with more than 55% of execution time as compared to 8% for the 2-body, so we focused on the 3-body code.

Table 4: Stillinger-Weber molecular dynamics simulation results. (Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

Program Version	Execution Time	Performance Speedup	Application Error	Memory Usage
Original	8.82s	1.67×	0.135%	5.6MB
Mesa (manual)	8.94s	1.62×	0.038%	2.0MB
Mesa (automated)	10.29s	1.42×	0.030%	400KB
Unoptimized	14.76	1.00×	0.000%	0.0MB

We applied the Mesa tool by identifying the 3-body expression with a pragma and entering the domain limits, which are constant in this application. The performance and error results are shown in Table 4. The Mesa version achieves 4.5× better accuracy with 14× less memory usage than the original code, but the performance is 16% slower. We modified the Mesa-generated code by hand to include all of the expressions optimized by the original version. By doing so we were able to closely match the performance of the original version of Stillinger-Weber, with 3.5× better accuracy and 2.8× less memory usage. Mesa currently cannot handle this combination of expressions, but we are working on a new version that can.

We also investigated the difference in accuracy between the original and Mesa versions. We found that the *ad hoc* optimization propagates and magnifies error terms by combining LUT values in successive expressions. Mesa avoids this problem by optimizing only the critical expression. The disparity in memory usage between these versions is because (1) Mesa stores single-precision values and the original tables were double-precision, and (2) many fewer expressions were optimized in the Mesa version. Mesa allowed us to experiment with different LUT sizes, and we discovered that we could improve accuracy significantly with a relatively small table. Another benefit from using Mesa is that the entire optimization required only the addition of a a single pragma to the code.

3.3 Neural Networks

Our fourth case study evaluates neural network code [19] developed by Chuck Anderson at CSU. Profiling showed that the evaluation of transfer functions in the neural network was a performance bottleneck that consumed approximately 47% of the execution time. Two commonly used transfer functions

Table 5: Mesa results on neural network code
(Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

Transfer Function	Orig. Time	Optd. Time	Speedup	Orig. Result	Optd. Result	Maximum Error
Logistics	7.9s	3.8s	2.1×	0.12407	0.12408	$8.8X10^{-2}\%$
Hyperbolic Tangent	11.0s	3.9s	2.8×	0.000295	0.000277	$6.1X10^{0}\%$

are logistics $f = 1.0/(1.0 + e^x)$, and hyperbolic tangent $f = tanh(x)$, both of which call elementary functions. We optimized both functions, and the results are shown in Table 5. The error terms may be less significant than shown, since we conservatively compute them as the difference of small numbers, instead of scaling them based on the range of possible solutions. The memory usage in both cases was 4MB, but we were also able to increase the size of the LUT to exceed L2 cache without significant performance degradation. It appears that the number of LUT accesses is much smaller than the entire table, thus the LUT data does not actually overflow L2 cache.

3.4 Hydrology Modeling

Our final case study is a slope aspect computation from the PRMS application [21] developed by the United States Geological Survey (USGS). The function we optimized is used by PRMS to compute the slope aspect for a single point on a terrain grid based a variety of parameters including the latitude and declination. The computation is shown in Figure 13, and the results of the optimization are shown in Table 2. The prevalence of sine and cosine calls make this code a good candidate for elementary function optimization, allowing Mesa to achieve a 4.6× speed up with a small error. The number of variables in the slope aspect computation preclude expression optimization with version 1.1 of Mesa.

3.5 Interpolation versus Direct Access

The results shown so far were generated with the direct access sampling method. Mesa supports linear interpolation, which samples the adjacent LUT entries and combines them in a linear fashion according to their distance from the input value. Figure 14 compares direct access and linear

```
float CGeospatial::Calculation(int julDay,
    double aspect, double slope, double latitude)
{
  // Declination calculation
  double decline  = 0.4095*sin(0.01720*(julDay-79.35));

  double sin_decline  = sin(decline);
  double cos_decline  = cos(decline);
  double sin_latitude = sin(latitude);
  double cos_latitude = cos(latitude);
  double sin_slope    = sin(slope);
  double cos_slope    = cos(slope);
  double cos_aspect   = cos(aspect);

  double sloped =
    (sin_decline*sin_latitude*cos_slope) -
    (sin_decline*cos_latitude*sin_slope*cos_aspect) +
    (cos_decline*cos_latitude*cos_slope) +
    (cos_decline*sin_latitude*sin_slope*cos_aspect);

  double horizontal =
    sin_decline*sin_latitude+cos_decline*cos_latitude;

  double slopeAspect = sloped / horizontal;
  return slopeAspect;
}
```

Figure 13: PRMS slope aspect computation.

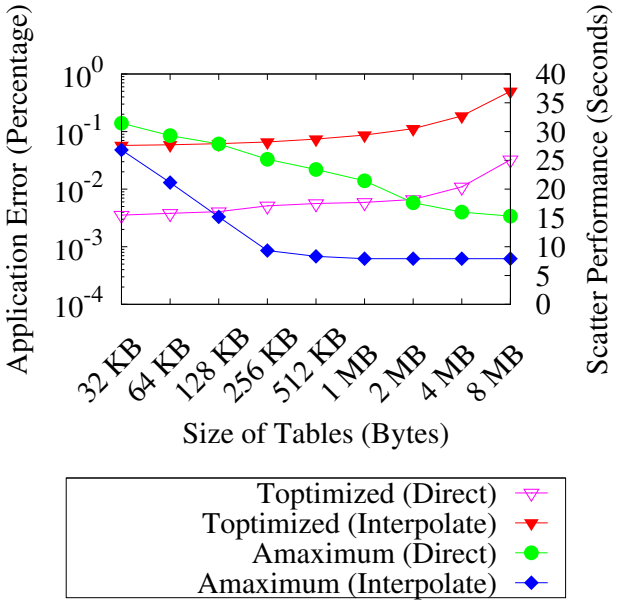

Figure 14: Comparison of interpolation versus direct access.
(Intel Xeon E5450, 3.00GHz, 6MB L2 cache, single core)

interpolation using the SAXS discrete scattering code. Execution time and maximum application error are shown in the graph for both methods. The extra computation for interpolation makes it slower than direct access. For the LUT sizes shown, linear interpolation yields a speed up from 4.5× to 6.1×, as compared to a 6.1× to 10.7× speed up for direct access. The shape of the performance curves is almost identical, implying that cache penalties affect both cases similarly. The maximum error for linear interpolation starts at almost an order of magnitude better for the smallest LUT size and improves quickly until an advantage reaches more than two orders of magnitude in the center of the graph. Thus linear interpolation may allow the LUT optimization to meet accuracy requirement in cases where direct access would overflow the mid-level cache.

3.6 Case Study Summary and Evaluation

We define two criteria for evaluating our methodology. The first criterion is effectiveness, which we measure quantitatively by comparing the performance and accuracy of the Mesa code against the original version. The second criterion is programming effort, which we define qualitatively as the effort needed to apply a LUT optimization with Mesa as compared with the manual process. We believe that programming effort is reduced in proportion to ease of use of the tool. The LUT optimizations shown in this section are effective because they achieve a significant speed up while meeting application accuracy requirements, as shown by results of the case studies in Section 3.

Programming effort is greatly reduced over the original *ad hoc* implementation for the SAXS discrete code, which required several weeks of development time and experimentation, even after completion of the original algorithm. Characterization of error was especially time-consuming, because it required multiple runs of the entire SAXS application. Mesa error analysis allowed us to quickly identify efficient set of LUT parameters without overflowing mid-level cache. The SAXS continuous code was never optimized manually because Mesa was available during its development period. Optimization of both the SAXS discrete and continuous code was done in a matter of hours with Mesa, including the error analyses shown in Figures 10 and 12. We additionally find that the Mesa code very closely matches the performance of the manually developed SAXS code. The other applications were just as easy to optimize with Mesa, requiring at most a single pragma. The development time for the manual optimization of the Stillinger-Weber

code is unknown, but it is obvious that many complex code changes were required, along with experimentation with table size versus performance and accuracy. The other applications did not contain a prior LUT optimization, so no direct comparison of development time is impossible.

3.7 Parallel Efficiency

Much of the current research in scientific computing focuses on multi-core performance and programming effort. An important trend in multi-core systems is the decrease in memory access performance relative to processor throughput. This motivates research on program transformations that minimize memory accesses. Despite this, we see no reason to neglect single-core performance, as long as the resulting optimizations remain equally beneficial in the multi-core environment. We verify that our optimizations meet this criteria by comparing multi-core scaling on programs optimized with Mesa. We have parallelized the SAXS discrete and continuous scattering loops with OpenMP directives. Figure 15 shows that our optimizations scale well on a Cray XT6m computer, and we have replicated this on several multi-core systems include the benchmark system shown throughout the paper (Intel Xeon E5450, 3.00GHz, 6MB L2 cache, eight cores). We conclude that our single-core optimizations are independent from and complementary to parallelization.

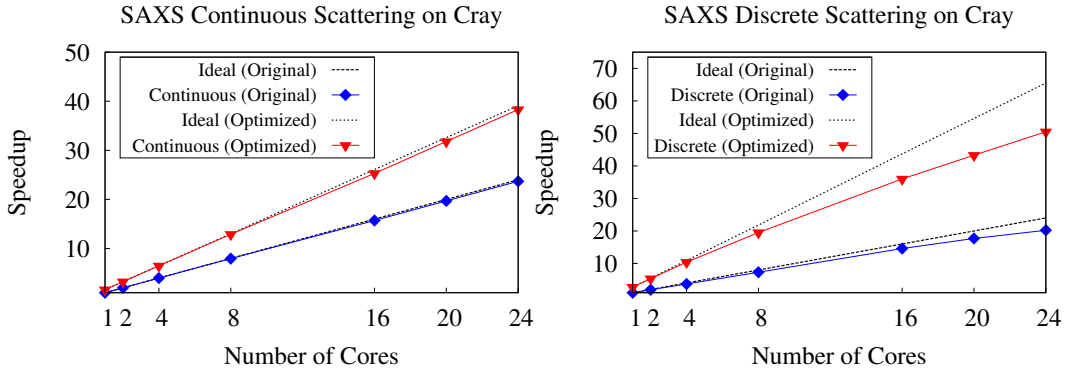

Figure 15: Parallel efficiency of SAXS application.
(Cray XT6m, AMD Opteron 6100, 2.5Ghz, 512KB L2, 6MB L3, 24 cores)

4 Related Work

Optimizing compilers are efficient at improving the *serial* performance of applications. However, the scope of optimizations provided by compilers is somewhat limited. Current compilers do not generate algorithmic improvements or more efficient numerical techniques, nor can they completely automate parallelization. As a result, manual tuning consumes a significant amount of the development effort for scientific applications [26]. Besides increasing programming effort, manual tuning has the disadvantage of obscuring algorithms [17].

A current research topic in scientific computing is how to decrease the programming effort [17] required to parallelize existing codes. Previous models such as POSIX threads required an intensive programming effort and special expertise. Current models such as OpenMP [6] raise the level of abstraction for the programmer, reducing the amount of code that needs to be written. Inserting pragmas is well accepted by scientific programmers using OpenMP, so we have adopted the same model for Mesa. The end goal is automation, which reduces effort by freeing programmers from low level details [5].

The LUT optimization approach described in this paper is motivated by the observation that some applications evaluate elementary functions repeatedly with inputs within a restricted domain. This repeated computation can be avoided by caching the results of function evaluation for later reuse. Memoization [1] is a similar approach that reduces computation by caching function results. Typically memoization algorithms guarantee *precise reuse*, meaning that function results are always exact. Alvarez et al. [2] propose *fuzzy memoization* in which *fuzzy reuse* allows results to be reused when the input closely matches a previous evaluation. This achieves a much higher level of reuse, but introduces error into the computation. The main difference between LUT optimization and memoization is that LUT methods compute results for the entire domain in advance, eliminating the overhead of identifying whether or not a result has been previously cached.

Considerable research has focused on optimizing the performance of elementary functions. The idea of approximating elementary functions in hardware is long established, for example Gal [10] proposed combining a LUT with a minimax polynomial. Frequently cited papers by Tang [29, 30] apply similar methods to implement elementary functions under the IEEE 754 floating-point specification [12]. LUT methods remain popular because they provide good performance at a reasonable hardware cost. Much of the recent

literature focuses on variants of *polynomial reconstruction* to improve the accuracy of function evaluation between reference points [20].

There are few academic references on software LUTs, but some books [22] and articles encourage the use of *ad hoc* techniques. A series of papers on LUT hardware for function evaluation in FPGAs [25, 8] has culminated in a paper by Zhang et al. [33] that explores both hardware and software LUT methods. Zhang et al. present a compiler that transforms functions written in a "MATLAB-like" language into C or C++ code suitable for multi-core execution. Mesa performs a similar transformation in the context of C or C++ source code, allowing optimization of entire applications. Similar to our research, Zhang et al. show that LUT optimizations outperform standard C and C++ code that calls elementary functions in the math library. Zhang et al. conclude that (1) linear interpolation is a minimal requirement, (2) LUT size is not a problem because of modern L2 cache sizes, and (3) the extra expense of polynomial reconstruction may not be worthwhile in software.

The only other related work that we are aware of that addresses the impact of cache usage on LUT performance is Defour [7]. Defour concludes that LUT sizes must fit within L1 cache, however this observation is based on very small tables in conjunction with highly accurate polynomial reconstruction. We show that LUTs up to the size of the L2 cache can improve the performance of function evaluation. We also extend the literature by comparing the performance of direct access and linear interpolation across a range of LUT sizes, and we find that lower overhead makes direct access worthwhile in some cases.

5 Limitations

The primary limitation of the Mesa tool is that it parses only C and C++ code, and the resulting optimized code must be compiled with a C++ compiler. Mesa is currently limited to the elementary functions sin, cos, tan, exp, log, and sqrt, but new functions are easily added. Mesa does not support all possible C++ syntax, thus minor modifications may be required before using the tool. For example, variables declared as *const* are not detected as constants, and must be replaced with preprocessor defines, and type casts are not allowed in candidate expressions. Mesa is also limited to expressions that depend on a single free variable, so the tool will not detect and generate multi-dimensional LUT data.

The main limitation of LUT optimization is that the technique is suitable only for programs that are performance-limited by computations with elementary functions. Such computations are common in scientific computing, but many applications are performance-limited by memory accesses. These applications may not benefit from LUT techniques, and the increase in memory usage can actually decrease performance. However, having access to a lightweight performance tool such as Mesa greatly reduces the effort required to see whether an application can benefit from LUT optimization. A second limitation is that LUT data must share mid-level cache memory with the application to avoid cache penalties. In practice this is often done successfully, as shown by the case studies in this paper. The final limitation of LUT optimization is that some applications may be unable to tolerate the decreased accuracy inherent to LUT optimization. In contrast, many scientific simulations are known to be based on very imprecise data, yet they often make pervasive use of expensive high-precision floating-point operations. These applications may be able to achieve a significant benefit without comprising results. Several precedents for reducing the precision of elementary functions exist, including the *Enhanced Performance* mode of the Intel Vector Math Library (VML) [15].

6 Threats to Validity

There are four general types of validity relevant to empirical research: conclusion validity, internal validity, construct validity, and external validity [32]. A study has *conclusion validity* if we can conclude that there is a relationship between study variables of interest. Our study exhibits conclusion validity since we can safely conclude that, among the programs in our study, programs with LUT optimization have improved performance than those without LUT optimization. We can also safely conclude that it takes more effort to perform the optimizations by hand than by using Mesa.

Internal validity focuses on whether or not there is a causal relationship between the treatment and external variables. Threats to internal validity include having no rationale for the relationship between treatment and outcome. In addition, other unmeasured factors that might be the real cause of the outcome are threats to internal validity. In our study, there is a clear rationale for a cause and effect relationship between LUT optimization and the dependent variables: performance, accuracy, and programming effort. Other

unmeasured factors should not affect the outcome, since the only change to the program is the replacement of function calls with a LUT access.

A study has *construct validity* when the treatment used in the study accurately represents the concepts that we mean to study, and the measurements of variable attributes are valid. Mesa performs LUT optimization in a manner consistent with the descriptions of manually performed LUT optimization in the literature. The measures of performance and accuracy are clearly valid. The measurement of programming effort in terms of programming time is consistent with the general notion of effort; a reduction in the time required to implement a program represents a reduction in effort.

Results of a study with *external validity* will generalize beyond the study data itself. External validity is often an issue with empirical research, because of the limited number and scope of the applications that can be evaluated. We would need to study a random sample of all scientific applications to eliminate all threats to external validity, but such a sampling is not practical. Our empirical evaluation consists of case studies of five such applications in four scientific areas. Further research is required to demonstrate applicability to other scientific domains, and to better understand where our methodology is most appropriate. However, we believe that the results in this paper will generalize to applications that share the characteristic of being limited primarily by the evaluation of elementary functions.

7 Conclusions

This paper presents the Mesa source-to-source transformation tool and an associated methodology for LUT optimization. We evaluate Mesa on five scientific applications from four domains, and find that our tool reduces the effort associated with LUT optimization. Our approach improves the current practice of manual tuning by allowing programmers to create and analyze LUT optimizations with very little effort and without writing code. Mesa error analysis lets programmers improve performance with clear knowledge of the effect on accuracy, without costly experimentation. The case studies in this paper provide additional evidence that software LUT optimization can exploit the fuzzy reuse inherent in many scientific programs to produce significant performance gains of 1.4× to 6.9×, while maintaining reasonable error bounds. The code for version 1.1 of Mesa as described in this paper is available from our web site [18].

8 Acknowledgements

This project is supported by Award Number 1R01GM096192 from the National Institute Of General Medical Sciences. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute Of General Medical Sciences or the National Institutes of Health. This project is also supported by grant number DE-SC0003956 from the Department of Energy. Additional support comes from seed funding from the Vice President of Research and the Office of the Dean of the College of Natural Sciences at Colorado State University and from a Department of Energy Early Career grant. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386. We gratefully acknowledge the many partners who have shared code with us, including Stefan Sillau, Ryan Croke, and Mark van der Woerd for the SAXS code, Paulette Clancy for the Stillinger-Weber program, Chuck Anderson for the neural network application, and Olaf David and George Leavesley for the PRMS code, and the anonymous reviewers from the Scientific Programming journal whose comments helped improve this paper.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Selective Memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 14–25, New York, NY, USA, 2003. ACM.

[2] C. Alvarez, J. Corbal, and M. Valero. Fuzzy Memoization for Floating-Point Multimedia Applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.

[3] C. W. Anderson, S. V. Devulapalli, and E. A. Stolz. Determining mental state from EEG signals using parallel implementations of neural networks. *Sci. Program.*, 4:171–183, September 1995.

[4] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. *ACM Trans. Math. Softw.*, 34(4):1–22, 2008.

[5] G. Cong, S. Seelam, I. Chung, H. Wen, and D. Klepacki. Towards a Framework for Automated Performance Tuning. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[6] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-memory Programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, Jan. 1998.

[7] D. Defour. Cache-optimised methods for the evaluation of elementary functions. Technical Report 2002-38, Ecole normale superieure de Lyon, 2002.

[8] L. Deng, C. Chakrabarti, N. Pitsianis, and X. Sun. Automated Optimization of Look-up table Implementation for Function Evaluation on FPGAs. In *Proceedings of SPIE*, volume 7444, 2009.

[9] J. Detrey, F. de Dinechin, and X. Pujol. Return of the Hardware Floating-point Elementary Function. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 161–168, Washington, DC, USA, 2007. IEEE Computer Society.

[10] S. Gal. Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance. In *Proceedings of the Symposium on Accurate Scientific Computations*, pages 1–16, London, UK, 1986. Springer-Verlag.

[11] O. Glatter and O. Kratky, editors. *Small angle x-ray scattering*. Academic Press, London, UK, 1982.

[12] D. Goldberg. What every Computer Scientist should know about Floating-point Arithmetic. *ACM Comput. Surv.*, 23:5–48, March 1991.

[13] B. Gough. *An introduction to GCC for the GNU compilers gcc and g++*. Network Theory Ltd., 2004.

[14] M. Haran, J. A. Catherwood, and P. Clancy. Diffusion of group v dopants in silicon-germanium alloys. *Applied Physics Letters*, 88(17):173502, Apr 2006.

[15] Intel Vector Math Library, 2011. http://software.intel.com/sites/products/documentation/hpc/mkl/vml/vmld%ata.htm.

[16] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In *IWOMP*, pages 15–28, 2010.

[17] E. Loh, M. L. Van De Vanter, and L. G. Votta. Can Software Engineering Solve the HPCS Problem? In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 27–31, New York, NY, USA, 2005. ACM.

[18] MESA Project, 2010. http://www.cs.colostate.edu/hpc/MESA.

[19] Neural Network Software, 2011. http://www.cs.colostate.edu/~anderson/meOther.html.

[20] J. A. Piñeiro, J. D. Bruguera, and J. M. Muller. Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree. In *ARITH '01: Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, page 40, Washington, DC, USA, 2001. IEEE Computer Society.

[21] PRMS Project, 2010. http://water.usgs.gov/software/PRMS.

[22] J. Riley. *Writing Fast Programs: A Practical Guide for Scientists and Engineers*. Cambridge International Science Publishing, 2006.

[23] ROSE Project, 2011. http://www.rosecompiler.org/.

[24] SAXS Project, 2010. http://www.cs.colostate.edu/hpc/SAXS.

[25] K. Sobti, L. Deng, C. Chakrabarti, N. Pitsianis, X. Sun, J. Kim, P. Mangalagiri, K. Irick, M. Kandemir, and V. Narayanan. Efficient Function Evaluations with Lookup Tables for Structured Matrix Operations. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 463 –468, Oct. 2007.

[26] S. Squires, M. Van De Vanter, and L. Votta. Software Productivity Research In High Performance Computing. *CTWatch Quarterly*, 2(4A):52–61, Nov. 2006.

[27] F. Stillinger and T. Weber. Computer simulation of local order in condensed phases of silicon. *Physical Review B*, 31(8):5262–5271, 1985.

[28] J. E. Stine and M. J. Schulte. The Symmetric Table Addition Method for Accurate Function Approximation. *J. VLSI Signal Process. Syst.*, 21:167–177, June 1999.

[29] P.-T. P. Tang. Table-driven Implementation of the Exponential Function in IEEE Floating-point Arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, 1989.

[30] P.-T. P. Tang. Table-lookup Algorithms for Elementary Functions and their Error Analysis. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991.

[31] C. Wilcox, M. Strout, and J. Bieman. Mesa: Automatic generation of lookup table optimizations. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, New York, NY, USA, 2011. ACM.

[32] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[33] Y. Zhang, L. Deng, P. Yedlapalli, S. Muralidhara, H. Zhao, M. Kandemir, C. Chakrabarti, N. Pitsianis, and X. Sun. A Special-Purpose Compiler for Look-up Table and Code Generation for Function Evaluation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1130 –1135, Mar. 2010.