Mesa: Automatic Generation of Lookup Table Optimizations

Chris Wilcox
Computer Science Dept.
Colorado State University
Fort Collins, Colorado 80523
wilcox@cs.colostate.edu

Michelle Mills Strout
Computer Science Dept.
Colorado State University
Fort Collins, Colorado 80523
mstrout@cs.colostate.edu

James M. Bieman
Computer Science Dept.
Colorado State University
Fort Collins, Colorado 80523
bieman@cs.colostate.edu

ABSTRACT

Scientific programmers strive constantly to meet performance demands. Tuning is often done manually, despite the significant development time and effort required. One example is lookup table (LUT) optimization, a technique that is generally applied by hand due to a lack of methodology and tools. LUT methods reduce execution time by replacing computations with memory accesses to precomputed tables of results. LUT optimizations improve performance when the memory access is faster than the original computation, and the level of reuse is sufficient to amortize LUT initialization. Current practice requires programmers to inspect program source to identify candidate expressions, then develop specific LUT code for each optimization. Measurement of LUT accuracy is usually *ad hoc*, and the interaction with multicore parallelization has not been explored.

In this paper we present Mesa, a standalone tool that implements error analysis and code generation to improve the process of LUT optimization. We evaluate Mesa on a multicore system using a molecular biology application and other scientific expressions. Our LUT optimizations realize a performance improvement of 5X for the application and up to 45X for the expressions, while tightly controlling error. We also show that the serial optimization is just as effective on a parallel version of the application. Our research provides a methodology and tool for incorporating LUT optimizations into existing scientific code.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Patterns; D.3.4 [**Processors**]: Code Generation, Compilers, Optimization; G.1.2 [**Approximation**]: Special function approximations

General Terms

Performance, Languages

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Keywords

Lookup table, error analysis, performance optimization

1. INTRODUCTION

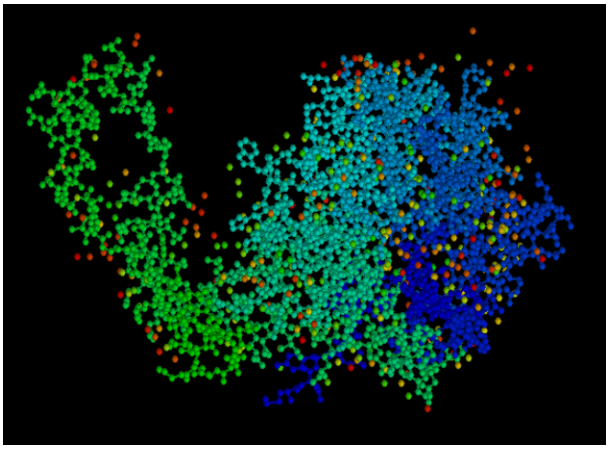
The computational needs of scientific applications are always growing, driven by increasingly complex models in the physical and biological sciences. Scientific programs often require extensive tuning to perform well on multicore systems. Performance optimization can consume a major share of development time and effort [14], and software engineering practices are sometimes ignored in the rush to attain performance [12]. Manual tuning, including parallelization, is inefficient and can obfuscate application code, making it harder to maintain and adapt [10]. One solution is automated performance tuning, which improves on manual methods by reducing the programming effort and simplifying the code [7].

This paper examines a serial optimization that uses precomputed lookup tables (LUTs) to avoid computation. We study LUT optimizations with Mesa, a standalone tool that we have developed. Mesa supports the application of LUT optimizations to existing code, without the loss of abstraction caused by manual tuning. The primary goal of Mesa is to decrease the cost of LUT optimization while giving the scientific programmer control of the tradeoff between accuracy and performance. Our secondary goal is to show that LUT methods can improve the performance of scientific applications on multicore systems.

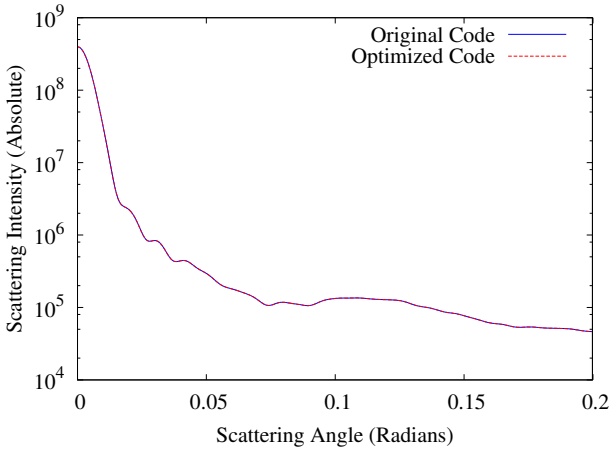
This work was motivated by a research collaboration with the small angle X-ray scattering (SAXS) research project at Colorado State University [1]. The SAXS software simulates X-ray scattering of proteins using Debye's formula, shown in Equation (1).

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

For performance evaluation of the SAXS code, we use enzymes from the Protein Data Bank (PDB), including the 1xib molecule, which has 3052 atoms. We defer a detailed discussion of computational complexity to Section 4, but we have measured $\approx 4.7 \times 10^9$ evaluations of the formula to scatter the 1xib molecule. The SAXS scattering code therefore has a significant amount of computation in which we have observed a large amount fuzzy reuse [3]. The initial version of scattering code required more than 1.5 hours to scatter the 1xib molecule. Removal of redundant code and precomputation of scattering constants resulted in a 6X improvement on a 32-bit system. Porting the code to a 64-bit system yielded another 3X improvement.



(a) 1xib molecule, which contains 3052 atoms.



(b) Original and optimized scattering curves.

Figure 1: 1xib molecule and scattering curves.

The project goal was to scatter thousands of rotations of various molecules. This raised the performance requirement, so we decided to investigate LUT methods. Table optimizations date back to the early days of computing [4], and are commonly used to optimize function evaluation in hardware. Programmers often use LUT methods, but the technical literature is very limited with respect to a software methodology. Our initial LUT optimization was manual, so we had to empirically determine the LUT size to meet the required level of accuracy and performance. After extensive experimentation we were able to gain an additional 7X improvement. We achieved these numbers despite a modest table size of 3.2MB and average error of < 0.0014 percent.

Figure 1 shows the 1xib molecule and intensity curve. Both the original and optimized curves are plotted, but the difference between them is negligible and impossible to see visually. To improve the process we developed the Mesa tool to perform error analysis and automatic generation of LUT optimization code for a broad range of scientific expressions. Mesa, described in Section 3, allows us to characterize the performance and accuracy of LUT optimizations.

In addition, we parallelized the SAXS scattering loop with OpenMP, showing a further 1.8X improvement on a dual-core, and 3.6X on a quad-core system. We conclude that the benefit of LUT optimization applies equally to the parallel version, that is the single core and multicore optimizations are independent and complementary for this application. Table 1 shows the progression of performance improvements for the scattering code.

Table 1: SAXS scattering code improvements.

Run Time	*Cumulative Factor*	*Delta Factor*	*Version Description*
5365s	1X	1X	Original code
872s	6X	6X	Removed redundancy
279s	19X	3X	64-bit system and compilers
41s	131X	7X	Manual table optimization
23s	233X	1.8X	Parallel version, dual-core
13s	413X	1.8X	Parallel version, quad-core

Current trends in computing platforms include the broad availability of multicore hardware and a decrease in memory access performance relative to processor performance. As a result, the focus in scientific computing has shifted to parallel execution and transformations that reduce the number of memory accesses on multicore systems. However, single core performance still remains important enough to justify continued study of optimizations that improve performance by eliminating redundant operations. Such optimizations must be evaluated to ensure that they remain effective when the code is parallelized, as we have done in this paper.

Mesa makes the process of applying LUT optimizations to existing code less costly and more repeatable. Our contributions are as follows: (1) we show that that LUT optimizations can benefit scientific codes, (2) we present a tool for the partial automation of LUT optimization, including error analysis, and (3) we demonstrate that LUT optimization is applicable in a multicore environment.

The remainder of this paper is organized as follows: Section 2 provides background information on LUT optimization, Section 3 introduces the Mesa tool, Section 4 describes case studies performed with Mesa, Section 5 gives details on error analysis, Section 6 shows related work, and Section 7 presents our conclusions.

2. LUT APPROXIMATION

LUT optimizations represent continuous functions as sets of discrete values, with each LUT access returning a stored approximation of the original function. Increasing the number of LUT entries improves accuracy, but reduces the level of reuse that occurs when different inputs share the same LUT entry. Decreasing reuse causes an increase in the penalties associated with memory usage, notably cache misses. Thus LUT optimizations have an inherent tradeoff between accuracy and performance. In this section we define how to characterize the error introduced by LUT optimization.

LUT approximations introduce error because memory limitations make it impractical to match the floating-point accuracy of the processor. IEEE floating-point standards provide accuracy of $\pm 1.19 \times 10^{-07}$ for single-precision (SP) and $\pm 2.22 \times 10^{-16}$ for double-precision (DP). To match this precision for an input variable with a domain $[0.0, 1.0]$ would require a 32MB table for a float, and a 16PB table for a double. The latter is clearly out of reach for modern computers, even for limited domains.

We define a LUT approximation as a function $l(x)$ with identical parameters to the original function $f(x)$, but less accuracy. The $l(x)$ function performs a table lookup by di-

viding the input value by LUT granularity and rounding it to an integer LUT index. Each LUT entry defines an input interval that maps to a single output value. The output value for each LUT entry is computed by evaluating the expression for an input value in the interval. The ideal output value is the average of the function over the input interval. This is costly to compute so many LUT implementations simply use the value at the interval center.

To evaluate accuracy, we need statistics that characterize the magnitude of the introduced error. We refer to the computation of error statistics for a LUT optimization as error analysis, described in Section 5. The most basic statistic is the error for an individual LUT access, computed as the absolute value of the difference between $l(x)$ and $f(x)$, as shown in Equation (2):

$$e(x) = |l(x) - f(x)| \tag{2}$$

We can calculate maximum and average error statistics based on the individual errors. The maximum error is important because it bounds the error imposed on the application for a single LUT access. In the worst case, each LUT access can produce the maximum error. We define the maximum error e_{MAX} as the largest absolute error within a LUT entry. We can combine the error terms for entries into a maximum error E_{MAX} for the entire table by iterating over all e_{MAX} to find the largest value. Error terms can be absolute or relative. Absolute error is the magnitude of the error term, and relative error is the error term divided by the expected value. Both representations are used in this paper, but relative errors are shown as a percentage.

In the best case, a LUT access exactly matches the original function, so the minimum error is zero. The average error is therefore bounded by zero and the maximum error. If we assume a random distribution of LUT accesses within an entry, then we can compute e_{AVG} as the arithmetic mean value of all outputs. If we assume an equal number of accesses to each LUT entry, then we can compute the average error E_{AVG} for the table as the mean of e_{AVG} values. The distribution of input data can vary widely, so the average error is only an approximation.

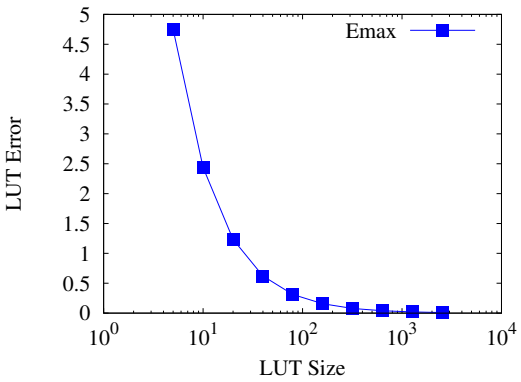
The key parameter for a LUT optimization is the number of LUT entries, since accuracy increases along with LUT size. Figure 2 plots the error statistics on a log scale for the function $f(x) = x^2$, with a progression of LUT sizes. Graphs like the one shown can help a programmer to select a LUT size that meets the error requirements of the application. Building the graphs requires extensive computation, so fast error analysis is needed to automate LUT size selection.

3. MESA SOFTWARE

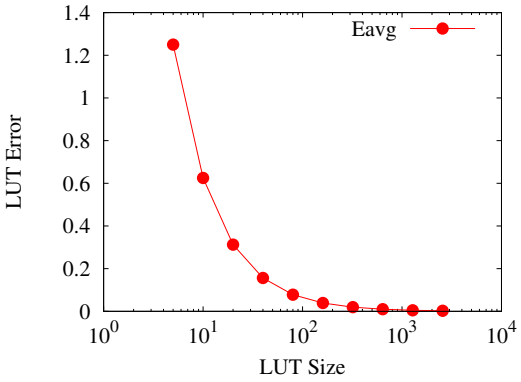
To make the process of LUT experimentation less cumbersome, we have developed a software tool called Mesa that automatically generates LUT optimization code for expressions commonly found in scientific code. Mesa code is available for download and use under the BSD license [1].

3.1 Mesa Methodology

This section describes the workflow for applying a LUT optimization with Mesa. As previously stated, the predominant practice is to perform the steps shown below by hand. The goal of Mesa is to automate the process as much as possible, although some of steps remain manual in the current version.



(a) Maximum error



(b) Average error

Figure 2: LUT error by LUT size for $f(x) = x^2$.

1. The first step is *expression identification*, which is done most effectively by manually running profiling tools. The ideal candidate expression is a computationally expensive function with high levels of fuzzy reuse, meaning that the expression is evaluated with the same *approximate* inputs repeatedly. Once the programmer has identified an expression, a specification file must be written for Mesa.

2. The second step is to *determine domains and distributions* of the input variables for the candidate expression. In some cases the input domain is easy to infer, otherwise this step require instrumentation of the target expression. The domain size is critical because it affects LUT size, and the distribution determines the level of reuse. The domain of each input variable must be included in the Mesa specification file.

3. The third step is *size specification*, which currently makes the user specify the LUT size on the command line. This may require multiple Mesa runs to characterize the error statistics before finding a favorable LUT size. We also have a prototype of the system that allows the user to specify a threshold for the maximum error on the command line, letting Mesa search for the smallest LUT size that keeps the maximum error within the specified value. The future direction of Mesa is automatic selection of LUT size based on error constraints and system resources.

```
$ cat square.dat
variable x 0.0 5.0 center;
expression x * x;
$ Mesa square.dat optimized.cpp 5 exhaustive
Mesa, version 1.0
LUT optimization started
Input Parameter: x [0.00, 5.00]
Lut size: 5
Analysis method: exhaustive
Number samples (per interval): 8388608
Interval[0.00,1.00] emax: 0.75, eavg: 0.25
Interval[1.00,2.00] emax: 1.75, eavg: 0.75
Interval[2.00,3.00] emax: 2.75, eavg: 1.25
Interval[3.00,4.00] emax: 3.75, eavg: 1.75
Interval[4.00,5.00] emax: 4.75, eavg: 2.25
Table: Emax: 4.75, Eavg: 1.25
LUT optimization completed
```

Figure 3: Mesa specification file and output.

4. The fourth step is *error analysis*, which is fully automated in Mesa. The user invokes Mesa with command-line arguments for the specification file (input), code file (output), table size, and error analysis method.

5. The fifth step is *code generation*, which is fully automated in Mesa. Mesa generates LUT data or initialization code for the LUT, along with the LUT approximation function that replaces the original expression.

6. The sixth step is *code integration*, which requires the user to include the generated code, call the initialization and deallocation methods on entry and exit, and replace the original expression with a call to the approximation method. This introduces an extra method call that can impact performance, but the user can substitute code from the approximation function directly to avoid the call overhead.

7. The seventh step is to *compare performance and accuracy*. This is done by switching back and forth between the optimized and original versions of the application, comparing accuracy and performance. Accuracy can be measured as a percentage difference between the output of the original application, which is assumed to be exact, and the output of the optimized application.

3.2 Mesa Operation

The input to Mesa is a specification file containing the expression and declarations for each constant and variable used by the expression. Input variables consist of a name and a lower and upper value that defines the domain. Our expression parser handles constants, variables, math operators $(+, -, *, /)$ and library functions (sin, cos, tan, sqrt, exp, log, fabs, fmod, pow). When Mesa is invoked from the command line, it parses the specification file, optionally performs error analysis, then generates LUT code. Figure 3 shows the specification file and a sample Mesa run for a LUT to optimize the function $f(x) = x^2$. The output shows that exhaustive error analysis was requested, resulting in display of maximum and average error terms for each entry and the entire table.

Figure 4 shows the code generated by Mesa. Mesa reads expressions from the specification file and builds a 1D LUT for one independent variable or a 2D LUT for two independent variables. SP values are used for LUT data, because

```
0  // Code generated by Mesa, version 1.0
1
2  // Expression variables
3  float xLower = 0.00e+00;
4  float xUpper = 5.00e+00;
5  float xGran = 1.00e+00;
6
7  // Original Expression
8  float Original(float x)
9  {
10    return(x*x);
11 };
12
13 // LUT Create
14 vector<float> lut;
15 void Create()
16 {
17   for (double d=xLower; d<xUpper; d+=xGran)
18   {
19     lut.push_back(Original(d+(xGran/2.0)));
20   }
21 }
22
23 // LUT Destroy
24 void Destroy()
25 {
26     lut.clear();
27 }
28
29 // LUT Approximation
30 float Lookup(float x)
31 {
32   // Compute index
33   int uIndex = (int) (x * 1.00e+00f);
34
35   // Table access
36   return(lut[uIndex]);
37 }
```

Figure 4: Optimized code generated by Mesa.

they provide sufficient accuracy for most LUT approximations, however the code could easily be changed to use a DP representation. LUT optimization may be unreliable when functions exhibit high frequencies, discontinuities, and other abrupt changes in slope. Mesa detects these cases by computing unacceptable error terms.

The generated code is C++ that uses an STL vector container, however a simple modification to use dynamic arrays would allow C compatibility. Support for other languages such as Fortran can be added by rewriting a single method within the Mesa implementation. Examining the generated LUT code, we see the lower bound, upper bound, and granularity of each input variable declared as constants on lines 3-5. The original function on lines 8-11 is simply a reconstruction of the expression from the specification file. The LUT table is declared on line 14, initialized on lines 15-21, and deallocated on lines 24-27. The LUT approximation function is declared on lines 30-37.

4. CASE STUDIES

In this section we evaluate Mesa by optimizing the SAXS molecular biology application and four scientific expressions. For the application we compare solutions using 1D and 2D LUT optimizations, which exhibit very different accuracy and performance characteristics.

4.1 SAXS Project

For the SAXS project, we wrote software to evaluate the Debye's scattering formula, shown again in Equation (3).

$$I(\theta) = 2\Sigma_{i=1}^{N-1}\Sigma_{j=i+1}^{N} F_i(\theta)F_j(\theta)sin(4\pi r\theta)/(4\pi r\theta) \qquad (3)$$

Table 2: SAXS performance and error comparison.

Protein	*Version*	*Time*	*Speedup*	E_{MAX}	E_{AVG}
1xib	Original	243.1s	-	-	-
	Optimized	49.8s	4.88x	0.0045%	0.0009%
1hbb	Original	503.1s	-	-	-
	Optimized	101.7s	4.95x	0.0083%	0.0014%
4gcr	Original	60.1s	-	-	-
	Optimized	12.4s	4.85x	0.0039%	0.0008%

The summations in the formula are implemented with a doubly nested loop over the N atoms in a molecule. The formula shown computes the intensity $I(\theta)$ for a single value of the angle θ. To construct a scattering curve we must evaluate the intensity for each of M different values of the scattering angle, thus the computational complexity of the entire operation is approximately $O(N^2M)$. By inspection, each evaluation of Debye involves multiple floating-point operations, including an expensive sine calculation. The number of steps for the scattering angle defaults to 10^3, which we have found provides a sufficiently detailed intensity curve for the biochemists. To scatter the 1xib molecule, the formula must be evaluated $(3052)(3052)(10^3)/2 \approx 4.7 \times 10^9$ times.

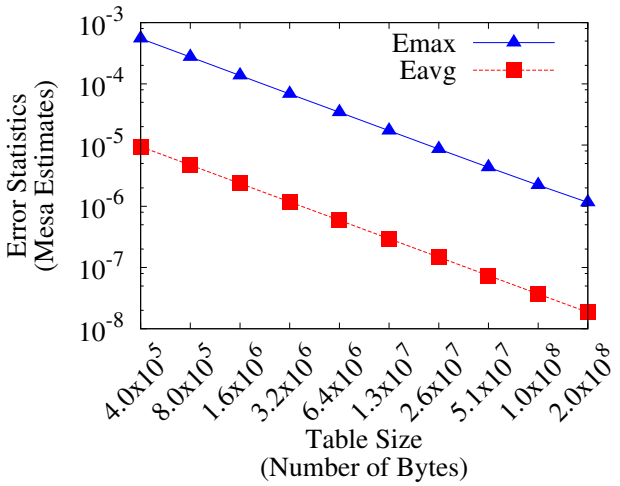
In the code, $F_i(\theta)$ and $F_j(\theta)$ are precomputed constants. This means the only independent variables are the scattering angle θ and the distance between atoms r. We can limit the domain of both variables in order to maintain a reasonable LUT size. The scattering energy falls off quickly, so θ is restricted to less than 0.2 radians. The proteins we study have distances r of less than 100.0 angstroms. The product of the variables is duplicated in the expression, so we can optimize further by multiplying the variables before we evaluate of Debye's formula. This allows the substitution of a 1D LUT based on the product of r and *theta*, with a domain of 0.0 to 20.0, instead of a much larger 2D LUT.

Table 2 shows performance and error rates for three PDB molecules: 1xib, 1hbb, and 4gcr. The replacement of the original formula with a LUT optimization increases the performance of the entire application by ∼5X. The resulting intensity values incur a reasonable error despite a modest table size of 8×10^5(3.2MB) entries. Two more molecules show a similar speedup due to the LUT optimization. The performance numbers shown in Table 2 and throughout the paper are measured on an Intel dual core Xeon E5405 processor (family 6, model 23, 2.0GHZ) with a 256KB L1 cache, 6MB L2 cache, and 2GB memory, running Fedora Core 10.

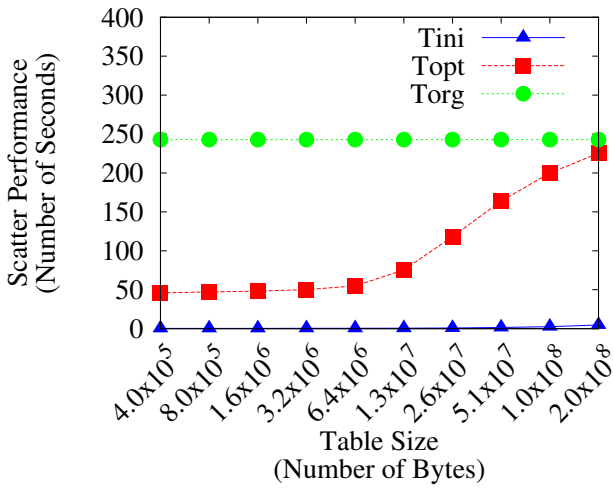
4.1.1 SAXS Results (1D)

This section shows the result of applying Mesa LUT optimization to SAXS code, with a 1D LUT. The specification file contains the Debye formula as the expression, along with a constant *pi* and the domain-restricted variable *rtheta*, which has a domain of $[0.0, 20.0]$. This variable represents the product of r and *theta* from the original Debye's formula, which can be used to index a 1D LUT. The value of automation is shown by the graphs in this section, which were produced by running Mesa from a shell script that varied LUT size and granularity.

Figure 5 shows the error statistics and performance numbers misses for the 1D optimization with different LUT sizes. LUT size starts at 1×10^5 entries (400kb) and doubles in size with each entry to 5.1×10^7 entries (200mb) in both graphs. The LUT error for the expression decreases with



(a) Error statistics versus table size.



(b) Execution time versus table size.

Figure 5: SAXS error and performance (1D LUT).

LUT size, and both the maximum error E_{MAX} and average error E_{AVG} are cut in half each time the LUT size doubles. The difference between the maximum error and average error is approximately two orders of magnitude. Our results confirm the expected relationship in which increasing LUT size improves accuracy but reduces performance.

The performance graph shows the number of seconds to execute the original code T_{ORG} along the top. The optimized performance T_{OPT} starts out around $5X$ faster than the original, then decreases until it almost degrades to the original performance on the last entry. We attribute the dropoff in performance primarily to L2 cache misses, which are shown in Figure 6. Statistics on L2 cache misses are gathered with PAPI hardware performance counters [6]. The PAPI library allows us to measure L1 and L2 cache misses, resource stalls, and other relevant performance numbers.

The evidence for L2 cache misses as the primary cause of the performance dropoff is as follows: (1) the performance and L2 cache misses curve are close to the same shape, (2) we observe that the dropoff in performance starts very close

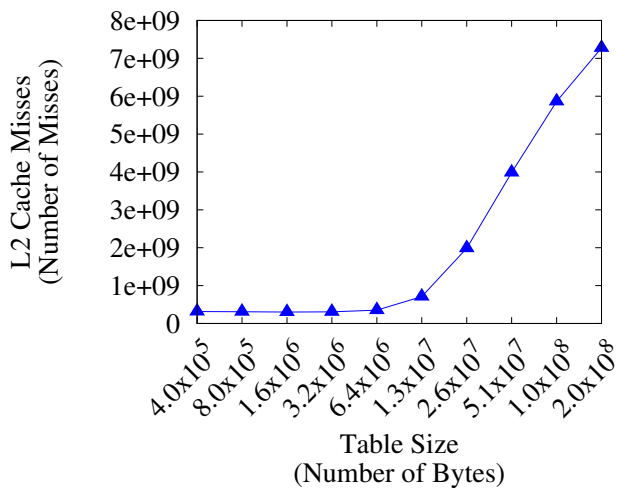


Figure 6: SAXS L2 cache misses vs. table size.

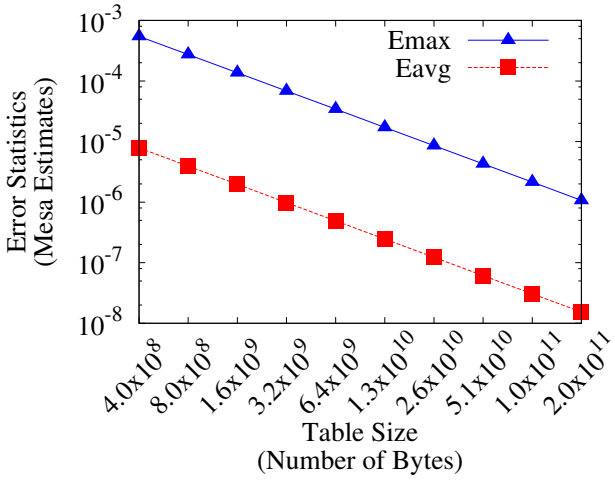
to the point where the LUT data fills the L2 cache, and (3) we have analyzed SAXS 1D performance with PAPI without finding other resource stalls that would explain the performance decrease. In addition, the performance dropoff happens at the point where the LUT size exceeds the L2 cache size on several different systems.

The performance graph also plots the LUT initialization time T_{INI} along the bottom, showing that LUT initialization is not a significant factor in the 1D LUT optimization. The combination of error and performance graphs allow us to select the optimal LUT size for the application. We also parallelized the 1D LUT version of the code using OpenMP, achieving an additional 1.8X on a dual-core or 3.6X on a quad-core processor. This demonstrates that the LUT optimization is relevant in the context of parallel execution on a multicore system.

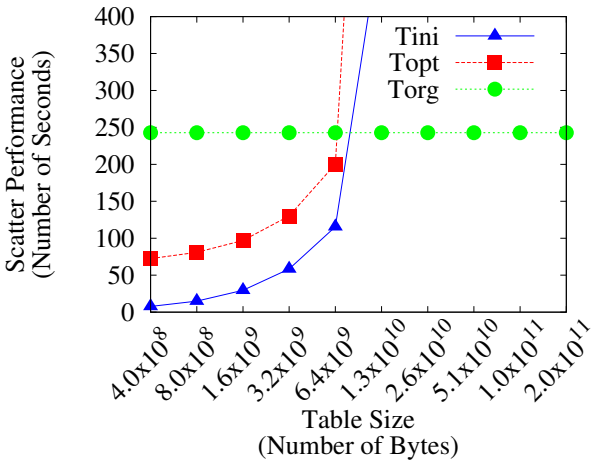
4.1.2 SAXS Results (2D)

A more straightforward LUT optimization of the Debye formula would use the variables r and *theta* as independent variables to create a 2D LUT. Figure 7 shows the error statistics and performance numbers for the 2D optimization. The magnitude of the error statistics match the 1D LUT, but the memory footprint is 10^3 larger. Thus the combination of these variables has apparently yielded an enormous advantage in memory footprint. This happens because both r and *theta* are only used in the Debye expression within the subexpression r times *theta*. As can be seen in Figure 8, the expression r times *theta* has the same value along multiple lines in the 2D LUT input space. The 2D LUT computes outputs for all points along this line, but the 1D LUT optimization computes values for only one point in each line.

The 2D optimization performance chart confirms the effects of the increased memory usage. Disregarding the error terms, the performance for the smallest 2D LUT (56.2s) is at a disadvantage as compared to the smallest 1D LUT (54.9s). In contrast with the 1D case, the 2D LUT initialization time causes the performance to degrade quickly, as shown in the graph. In addition, PAPI shows a large increase in resource stalls for the 2D optimization, even when we exclude LUT initialization. The cause may be virtual memory penalties



(a) Error statistics versus table size.



(b) Execution time versus table size.

Figure 7: SAXS error and performance (2D LUT).

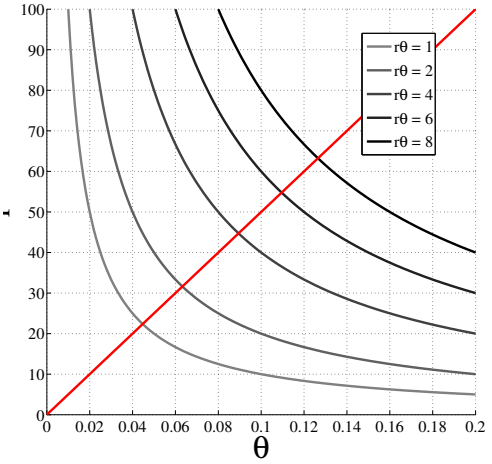


Figure 8: Explanation of 1D vs. 2D size difference.

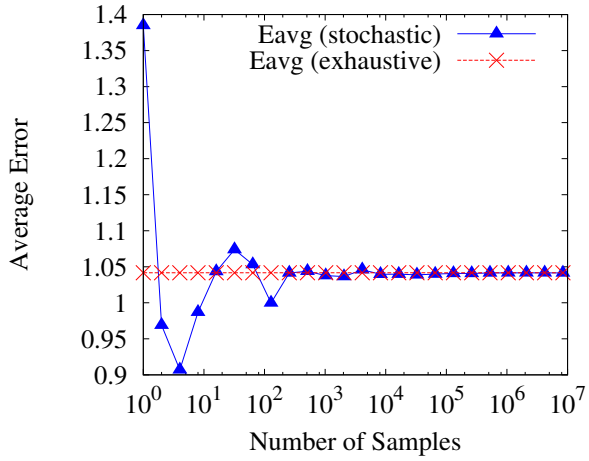


Figure 9: Stochastic versus exhaustive sampling.

that occur when the LUT exceeds the size of system memory. PAPI does not measure virtual memory penalties directly, so all we can confirm is that some type of resource stall is responsible, and that L1 and L2 cache misses are not the direct cause. The 2D LUT appears to be impractical for the SAXS implementation because of excessive memory usage.

4.2 Other Scientific Expressions

To generalize the result from SAXS, we gathered expressions from several different scientific algorithms and created LUT optimizations for them with Mesa. The algorithms include Gamma Correction, a color intensity adjustment common in computer graphics, Cauchy's Equation, a refractive index calculation from optics, the Normal Distribution, used by many statistics programs, and the Logistic Curve, which estimates population growth. Table 3 shows error statistics and performance results for each expression.The E_{MAX} and E_{AVG} statistics are computed by Mesa. An empirical measurement of performance from Mesa is shown, including the execution time of the original function t_{ORG} and optimized function t_{OPT}. The factor is the ratio of the original time divided by the optimized time. This factor indicates how much the expression evaluation performance will improve, but does not predict the performance gain of the full application, which will improve significantly only if one of these expressions accounts for a major share of the execution time.

5. ERROR ANALYSIS

Error analysis provides the information needed to control accuracy versus performance, a key tradeoff for LUT optimizations. Mesa does error analysis by computing the LUT error statistics defined in Section 2. Computation of error statistics is implemented by comparing the LUT approximation to the original function across the entire domain. We have prototyped three methods in Mesa for error analysis: *exhaustive*, *stochastic*, and *analytical*.

The *exhaustive* method implements a numerical traversal of LUT entries at a resolution of the minimum epsilon for SP, but error terms are maintained in DP for accuracy. Error values for all entries are accumulated to compute statistics for the entire LUT. The *stochastic* method samples the input domain randomly, the accuracy is dependent on the number of samples. Stochastic sampling appears to converge towards the same answer as the exhaustive method with many fewer samples, as shown in Figure 9. This suggests that stochastic sampling is more efficient. The *analytical* method computes error statistics by evaluating the area between the original and approximation functions. Mesa currently supports the analytical method but requires the user to include the integral of the expression in the specification file. One limitation of the analytical method is its failure to detect local minima and maxima for the function, which can cause inaccurate results. The analytical method is the fastest technique, so future work is planned to resolve accuracy issues.

The literature shows numerous examples of error analysis for hardware and software LUT optimizations [15, 8]. The assumption in this prior work is that table values are precise, with interpolation or polynomial reconstruction for the computation of accurate values in between table entries. Our error analysis is unique because we omit interpolation in favor of higher resolution tables. We do this to avoid interpolation, which is costly in software. The ramifications are that our technique uses more memory but less computation, and error terms are entirely dependent on LUT size. As a consequence simpler expressions can benefit from optimization because of lower overhead, and performance is limited more by memory usage than computation. Our experimental results show performance degradation when the LUT data falls out of L2 cache, so our strategy is to make the LUT into L2 cache if we can do so while ensuring the required accuracy.

6. RELATED WORK

LUT optimization in hardware is commonly used to accelerate the evaluation of elementary functions [15, 5], including exponentials [5], logarithms [16], and trigonometric functions [13]. Dedicated hardware memory is expensive, so hardware LUT optimizations usually depend on interpolation between table entries to limit LUT size.

LUT optimization in software has been used for many years to reduce instruction counts. Hyde [9] discusses the idea of optimizing performance by replacing computations with table lookups. A recent paper by Zhang et al. [17] provides a starting point for exploring software LUT methods. The work explores code generation for software LUT optimizations on multicore systems. The compiler in the Zhang et al. paper reads expressions from a specification file, in a manner similar to Mesa. An important difference from our work is that their algorithm depends on interpolation, thus the authors conclude that LUT size is not a significant performance limiter. Our experiments show that interpolation is not always necessary, and LUT size may in fact be important. For this reason, we believe that more research is required to quantify the tradeoff between memory usage and interpolation overhead.

Other techniques for reusing function results include fuzzy memoization [2, 3] and function memoization [11]. Fuzzy memoization resembles LUT optimization in that an approximation is used to increase reuse. Function memoization [11] does not allow approximation, and is therefore applicable only when identical inputs are repeatedly reused. Memoization algorithms require a conditional to check whether a result has been stored, and hashing is generally used for table access. This differs from LUT optimization, which precomputes and stores results for the entire input domain in advance of the calculation.

Table 3: Mesa performance on other scientific expressions.

Expression Name	Expression Description	Parameter Domain	Table Size	Estimated Error E_{MAX}	Estimated Error E_{AVG}	Measured Performance t_{ORG}	Measured Performance t_{OPT}	Measured Performance Factor
Gamma Correction	$I_\Gamma = I_{ORIGINAL}^{(1/\gamma)}$	intensity = [0, 1]	1.0 $\times 10^5$	1.3 $\times 10^{-3}$	2.2 $\times 10^{-6}$	14.6 ns	0.34 ns	43x
Cauchy's Equation	$i = 1.458 + (.00354/\lambda^2)$	λ = [0.0, 1.8]	1.0 $\times 10^3$	7.0 $\times 10^{-4}$	2.2 $\times 10^{-5}$	0.66 ns	0.17 ns	4x
Normal Distribution	$Z = 1/\sqrt{2\pi}e^{(-x^2/2)}$	x = [0, 10]	1.0 $\times 10^5$	1.2 $\times 10^{-5}$	9.8 $\times 10^{-7}$	15.3 ns	0.34 ns	45x
Logistic Curve	$P(t) = 1/(1 + e^{-t})$	t = [0, 10]	1.0 $\times 10^3$	1.3 $\times 10^{-3}$	1.2 $\times 10^{-4}$	4.7 ns	0.17 ns	27x

7. CONCLUSIONS

We have presented a methodology for LUT optimization and an associated software tool called Mesa that provides error analysis and code generation. By replacing manual tuning, Mesa makes the application of LUT optimizations less expensive and more reliable. We conclude that LUT optimizations can improve the performance of scientific programs without requiring extensive manual tuning. LUT optimizations also appear to be effective in conjunction with parallel execution on multicore systems. We have two future directions for our research. The first is a comparison of the direct table access method described in this paper against linear interpolation, which provides more accuracy at the cost of some extra computation. The second is to evaluate the interaction between LUT optimization and parallel performance on systems with different cache architectures. Mesa is freely available for download and use [1].

8. ACKNOWLEDGMENTS

The authors thank Stephanie Dinkins for assistance she provided in doing the performance analysis of the LUT optimization. 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.

9. REFERENCES

[1] SAXS software project at Colorado State University. http://www.cs.colostate.edu/hpc/SAXS, 2010.

[2] C. Alvarez, J. Corbal, E. Salami, and M. Valero. Initial results on fuzzy floating point computation for multimedia processors. *Computer Architecture Letters*, 1(1), January-December 2002.

[3] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.

[4] G. M. Amdahl. Computer architecture and Amdahl's law. *Solid-State Circuits Newsletter*, 12(3):4–9, Summer 2007.

[5] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23(2):242–251, 1976.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[7] 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&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[8] M. Grand. *Patterns in Java, Volume 2*. John Wiley & Sons, Inc., 1999.

[9] R. Hyde. *The Art of Assembly Language*. No Starch Press, 2003.

[10] 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.

[11] P. McNamee and M. Hall. Developing a tool for memoizing functions in C++. *SIGPLAN Notices*, 33(8):17–22, 1998.

[12] T. Panas, D. Quinlan, and R. Vuduc. Tool support for inspecting the code quality of HPC applications. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*, SE-HPC '07, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society.

[13] M. Schulte and E. Swartzlander. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8):964–973, 1994.

[14] S. Squires, M. Van De Vanter, and L. Votta. Software productivity research in high performance computing. *CTWatch Quarterly*, 2(4A), nov 2006.

[15] P.-T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In *The Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991.

[16] M. Zhang, J. Delgado-Frias, and S. Vassiliadis. Table driven Newton scheme for high precision logarithm generation. In *Proceedings of the IEEE Computers and Digital Techniques*, volume 141, 1994.

[17] 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. pages 1130 –1135, Mar. 2010.