NOCAP: **N**earby-**o**perand **C**ontinuous **Ap**proximation

ZHIXIANG TEOH, PETER LY, OWEN GOEBEL, and NEEL SHAH*

A framework for generation of lookup tables based on profile data is presented[1]. Lookup tables allow for quicker evaluation of functions by approximating the value of the function by an input which is near the original input. This can alternatively be thought of as hard-coding the function to return certain values on input ranges. We implement these functionalities using LLVM. We find that, for general use, there is a minor speedup when applying this transformation and that, for crafted benchmarks, there is significant speedup.

1 INTRODUCTION

Approximate computation is the idea of optimizing the utility of an application by sacrificing the quality of application output for an increase in computation speed. In some cases, computation of an exact result is not necessary for an application to be used in practice and it is desirable to accept an effectively negligible error in the output of the application for a significant speedup. In these cases, it is beneficial to have a programmatic way to generate versions of the application which trade between the error and speed of the application.

In this project, we study the usefulness of *table lookup* enhanced with profiled data by implementing a table lookup relaxation in LLVM. The benefit here is that application speed can be improved by increasing compilation time and decreasing the quality of the output. Similarly to the ideal setting of having a programmatic way to generate versions of the application which trade between the error and speed of the application, we provide a way for the users of our framework to trade between the amount of error and speed by configuring the amount of memory used to generate the table.

The rest of the paper is organized as follows: section 2 identifies related work, section 3 describes our implementation, section 4 describes how the implementation is evaluated, section 5 provides the results of evaluation and our analysis of said results, and section 6 provides our findings.

2 RELATED WORK

Much of the recent work in approximate computation has been directed in two directions:

- Generating techniques which give trade-offs between application quality and speed.
- Generating frameworks which programmatically apply relaxations to attain target levels of quality.

We survey results of both types.

*All authors contributed equally to the project.

[1]https://github.com/neel-one/nocap

Authors' address: Zhixiang Teoh, zhteoh@umich.edu; Peter Ly, pmly@umich.edu; Owen Goebel, oagoebel@umich.edu; Neel Shah, neelsh@umich.edu.

2.1 Techniques of Approximate Computation

Relaxations may occur at the software level or at the hardware level. Several techniques of both abstraction levels are classified in [1] and [2]. In this section we describe techniques relevant to the approximation of numerical functions.

2.1.1 Linear/Polynomial Approximation. It is well known that a function can be locally approximated by a polynomial fit to the function. Linear/polynomial appproximation [3] is a technique where calls to functions (especially mathematical functions like sin and cos) are replaced by calls to linear/polynomial approximations of the original function (often these approximations are related to the Taylor series approximation of the function). The C `math.h` library implements many mathematical functions using polynomial approximations.

2.1.2 Function Memoization. Function memoization [4] is a technique where the input values of functions are stored in a lookup table as they are computed so subsequent calls to the function with the same operand can be resolved with a table lookup instead of an evaluation of the function which may be expensive to compute. In the context of approximate computing, function memoization refers to the same practice of storing the results of calls as they are computed, but instead of resolving with a table lookup only if the operands exactly match the operands of a previous call, the function resolves with a table lookup if the operands approximately match the operands of a previous call.

2.1.3 Table Lookup. Table lookup is technique rooted in the history where tables of function values (such as the trigonometric functions) would be printed in books so that they might be referred to instead of having to directly compute the value of the function. This differs from function memoization in that here, the table is generated prior to execution of the application, whereas function memoization generates the table "on-the-fly".

In [5], these tables are constructed via what is essentially a brute-force enumeration of the function space and the results are aggregated via a clustering algorithm. More recently, [6] has introduced an FPGA-based approach to implementing table lookup approximation which uses gradient information to identify the intervals for which the function should be approximated.

2.2 The ACCEPT framework

The ACCEPT framework introduced in [7] is a pragmatic framework for generating several versions of a C++ program

with varying levels of quality and speed. The framework extends LLVM 3.2 and EnerJ [8], an annotation system for approximate computing, with several passes to implement a type system for approximate variables to identify regions of code amenable to approximation, estimates the effect of relaxations, and finally greedily generates a set of configurations to attain target levels of error in the output application.

3 METHODOLOGY

At a high-level, we apply profiled input data to generate lookup tables for continuous functions and then redirect calls to the functions to instead use values from the lookup table. We target continuous functions because continuous functions have the property that for inputs sufficiently close to an input x, the values will similarly be similarly close.

3.1 Profiling of inputs

We profile the code using LLVM to identify the input values for functions to which we can apply table lookup. Assuming the profile data is reflective of typical inputs, we identify which intervals on the domain can be implemented instead as a table lookup.

3.2 Construction of table input intervals

Once the intervals which should be resolved to table lookups are identified, we construct the actual intervals which will appear in the table. The exact number of intervals is specified by the user, and the exact intervals are simply determined by splitting the input range evenly by the number of intervals. Then, for each interval, we evaluate the function at the median of the interval and use the value from that evaluation as the table entry for the interval (as depicted in Figure 1).

3.3 Redirection of function calls

After the table is constructed, what remains is to integrate the table into the application. To integrate the table into the application, we modify each function so that if the operand to the function is in the table generated, the function should return the value in the table. Otherwise, the function should execute as normal.

4 EVALUATION

Table lookup techniques trade both memory and precision for speed, so we will measure memory usage, precision, and speed for both the original and table lookup modified version of the binary. Specifically, runtime is measured using the linux `time` utility. We quantify memory usage by the number of values to be stored in the table and compute a normalized error statistic for each number of values to be stored in the table.

5 RESULTS AND ANALYSIS

In this section, we present the results of our experiments and analyze the results.

5.1 Benchmarks

We constructed a small custom C program in which the execution time is dominated by the computation of the function `exp`. For this, we only measure the time it takes to run the program after compilation.

We use the Black-Scholes algorithm (a pricing model used to determine the fair price or theoretical value for a call or a put option) implementation from the ACCEPT framework test suite[2] as a benchmark because it uses functions such as `exp`, `log`, and `sqrt` which can be expensive to compute exactly. We removed the ACCEPT framework type system from the Black-Scholes algorithm implementation as our framework is not implemented on top of the ACCEPT framework.

5.2 Results

For our custom C programs, we find that there is speedup when running NOCAP to approximate `exp` and `log` – this is likely because the implementation of the two functions involve evaluations of Taylor polynomial like functions which can be expensive to evaluate. On the other hand, running NOCAP to approximate `sqrt` seemed to slow down the program possibly because `sqrt` is evaluated by a hardware instruction. These are summarized in 2

We did not measure how precise the approximated custom C programs were because there was no output on which to evaluate the custom programs.

5.2.1 Custom C program - `exp`*.* The C program we implement is a deterministic program which runs the function `exp` repeatedly. We observe an average speedup of about 60% when we replace the original calls of `exp` with table lookups. Specifically, we improve the average running time of the program from 2.4911 seconds to 0.9946 seconds.

5.2.2 Custom C program - `log`*.* We implement a similar program which runs `log` repeatedly (but fewer times than the `exp` program) and observe a speedup of about 42% (from 0.0726 seconds to 0.0421 seconds) when approximating `log`.

5.2.3 Custom C program - `sqrt`*.* We implement a similar program which runs `sqrt` repeatedly (again, but fewer times than the `exp` program) and observe a *slow down* of about 60% (from 0.0262 seconds to 0.042 seconds) when approximating `sqrt`.

5.2.4 Black-Scholes. The Black-Scholes algorithm is a deterministic program which computes the Black-Scholes function.

[2]https://github.com/uwsampa/accept-apps

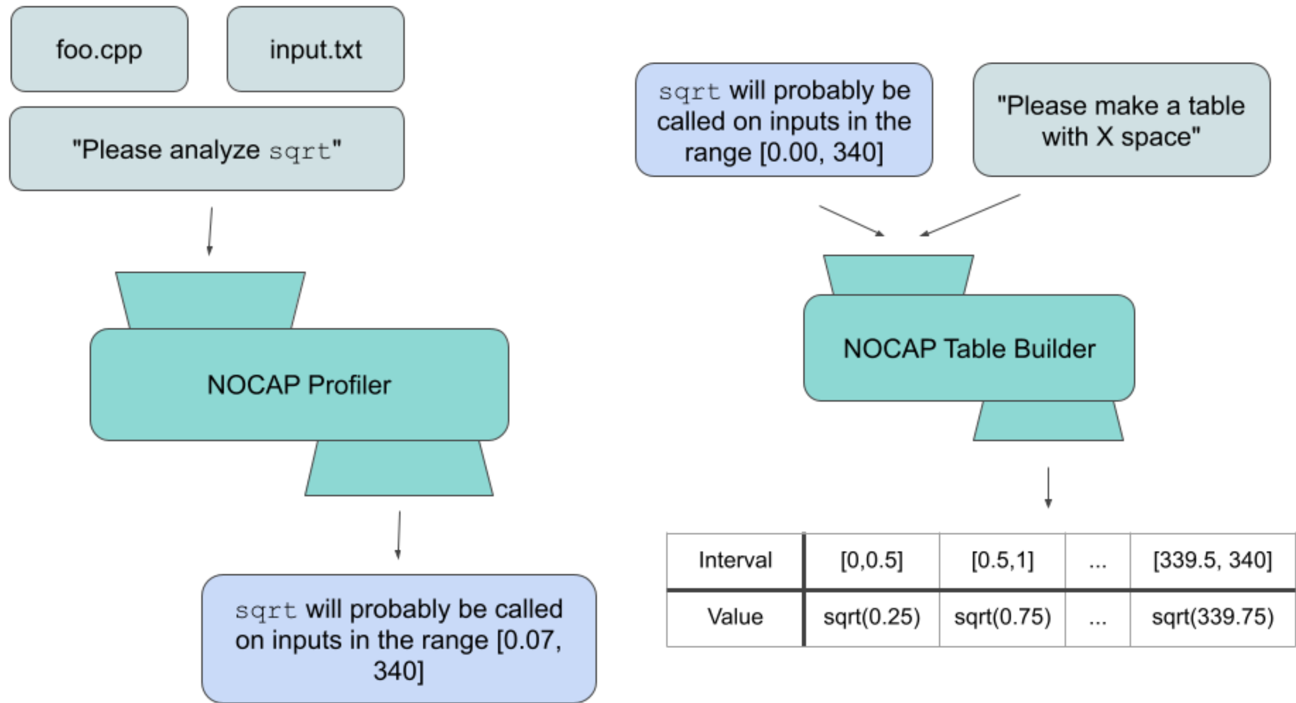

Fig. 1. High-level overview of NOCAP's workflow on `sqrt`. Afterwards, the new source code with the lookup table implementation of `sqrt` is emitted.

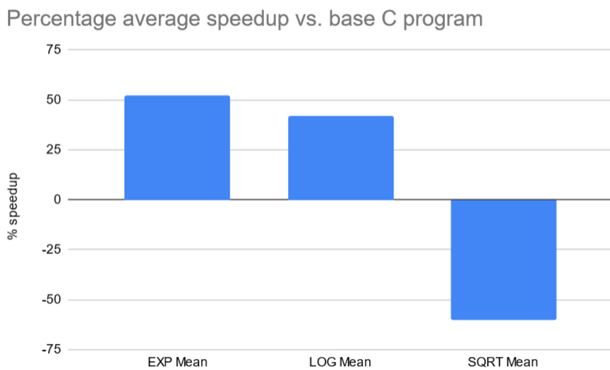

Fig. 2. We find that `log` and `exp` are greatly sped-up while `sqrt` is slowed down.

We profiled the program on an input computing the Black-Scholes price model for 16 options and then ran both the original Black-Scholes program and our approximated version on an input of 10 million options to obtain the following observations.

Following [7], we compute the average normalized difference between the two options as follows to evaluate the accuracy of the approximated program. Specifically, we compute

$$\sum_{\text{option}} \frac{|\text{Original option price} - \text{Approximated option price}|}{\text{Max original option price}}$$

as our accuracy metric. The maximum and minimum option prices computed by the original Black-Scholes program are \$28.6436 and $\$-1.4007 \times e^{-6}$.

When we replace uses of `log` by table lookups, we find that the average normalized difference converges to about 0.036 after using only 10 buckets. For `sqrt`, the average normalized difference tends to 0.039 after using about 50 buckets. Measurements for `log` and `sqrt` are summarized in Figure 4. For `exp`, the normalized difference tends to 0.01680 after using about 100 buckets, though it is of note that this normalized difference is attained before using 100 buckets. Measurements for `exp` are summarized in Figure ??. Note that average normalized error for `log` approximation drops suddenly when the number of buckets is increased from 1 bucket to 2 bucket, and then increases again after the number of buckets is increased to 3. This seems to happen because `log` is computed on values which are near the first and third quartile of the domain which is where the value of the buckets are found when there are two buckets in the table.

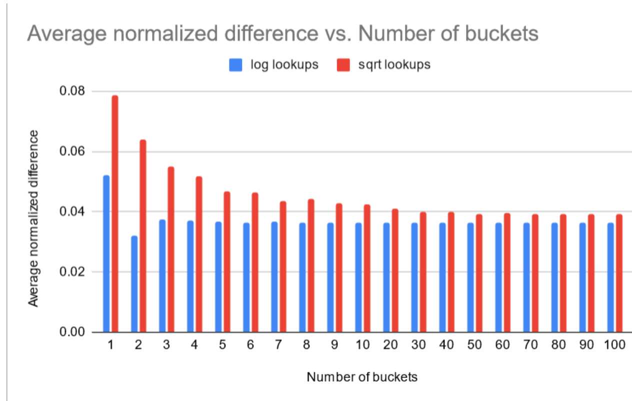

Fig. 3. The average normalized differences tend to their asymptotic values quite quickly for `log` and `sqrt`.

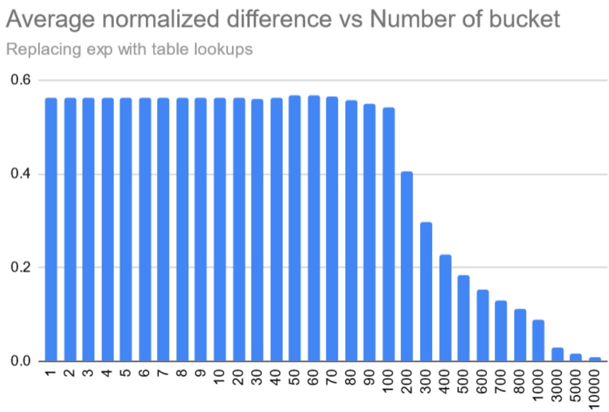

Fig. 4. The average normalized differences tend to its asymptotic values quite slowly when approximating `exp`.

The exact cause of difference in convergence rates is unclear, but it seems that the rate of convergence is related to the gradient of the function being approximated – `log` has a quite small gradient, while `exp` has a quite large gradient, and `sqrt` has a gradient between the two.

The percent speedup we see after replacing each of the functions with a table lookup is summarized in 1. Exact running times are given in 5. The speedup differs from the amount observed in the custom C programs (which had computations which were mainly composed of the functions being approximated) because the Black-Scholes program does many computations other than computations of `log`, `sqrt`, and `exp`.

It is unclear why the Black-Scholes benchmark exhibited speedup when approximating `sqrt` while the custom C program for `sqrt` exhibited slowdown. One possible reason is that the values that `sqrt` was evaluated on were not able to be computed easily for some reason, so the compiler was forced to rely on a software implementation of `sqrt` which runs slower than a table lookup.

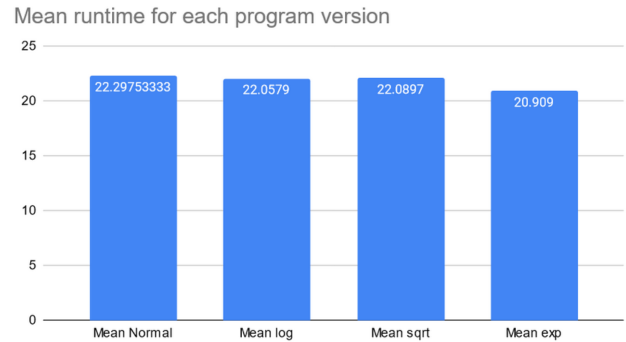

Fig. 5. All versions of the Black-Scholes benchmark exhibited speedup.

5.3 Challenges

One major challenge we faced was building the ACCEPT framework. While the framework is open-source, it was built on top of an older version of LLVM–specifically LLVM 3.2–while the current major release of LLVM is version 15. The difference in version caused several compilation and linking issues which could not be resolved in a reasonable amount of time. As such, we were unable to integrate NOCAP into the ACCEPT framework.

6 CONCLUSIONS

We implement a compiler-guided optimization framework for generating lookup tables for functions based on profile data. Our results indicate that there are considerable improvements in code speed to be had when applying our framework, even if the use cases may be somewhat specific.

We categorize future work into two categories: usability and technical implementation.

6.1 Contributions

We provide an open-source MIT-licensed compiler-guided optimization framework for optimizing C/C++ functions via lookup table approximation based on program profile data. We also demonstrate speedup of continuous mathematical functions defined in `math.h` and evaluate the memory-efficiency of this lookup table scheme with respect to accuracy.

6.2 Future usability improvements

Currently, the input program needs to be profiled (involving a run of the program) for each function-to-approximate. It would be good to only profile the input program once, and be able to generate executables for different combinations

	`log` approx	`sqrt` approx	`exp` approx
Speedup	0.5%	3.9%	6.5%
Normalized Average Difference	0.036	0.039	0.0168

Table 1. Summary statistics for the Black-Scholes benchmark. We approximate the functions `log`, `sqrt`, `exp` individually. The normalized average difference statistics are the difference the program (when profiled on 16 inputs) tends to as the number of buckets increases.

of functions-to-approximate. This would vastly decrease the time needed to build and run NOCAP.

Additionally, it would be good to better customize and measure NOCAP's memory usage via allowing the user to provide a memory limit in byte units.

Finally, it would be good to allow the user to provide custom header file(s) with user-defined functions with the `double` $\rightarrow$ `double` signature, which will augment NOCAP with the ability to approximate user-defined functions.

6.3 Future technical improvements

There are other ways in which the table lookup can be implemented. One is to map each interval to a linear or higher order approximation of the function rather than a constant (so that the function is essentially approximated by a spline).

Additionally, it would be good to determine if setting the bucket width in the table dynamically based on profile data would be useful in increasing the accuracy of the approximated applications. We hypothesize that setting the interval width so that more buckets are constructed when inputs are close together will improve accuracy when approximating functions which have sudden changes in output.

It would also be good to augment NOCAP with dynamic function memoization, which could be applicable in many cases, including long-running online programs like servers.

Finally, it would be good to determine if the gradient-based bucket computation employed by [6] could be applied in software instead of in hardware and what speed-up and memory/accuracy trade-off can be observed in this case.

REFERENCES

[1] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie Enright Jerger, and Adrian Sampson. 2018. A taxonomy of general purpose approximate computing techniques. *IEEE Embedded Systems Letters*, 10, 1, 2–5. DOI: 10.1109/LES.2017.2758679.

[2] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48, 4, Article 62, (Mar. 2016), 33 pages. DOI: 10.1145/2893356.

[3] Jean-Michel Muller. 2020. Elementary functions and approximate computing. *Proceedings of the IEEE*, 108, 12, 2136–2149. DOI: 10.1109/JPROC.2020.2991885.

[4] Priya Arundhati, Sisir Kumar Jena, and Santosh Kumar Pani. 2022. Approximate function memoization. *Concurrency and Computation: Practice and Experience*, 34, 23, e7204. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.7204. DOI: https://doi.org/10.1002/cpe.7204.

[5] Ye Tian, Qian Zhang, Ting Wang, and Qiang Xu. 2018. Lookup table allocation for approximate computing with memory under quality constraints. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 153–158. DOI: 10.23919/DATE.2018.8341995.

[6] Chetana Pradhan, Martin Letras, and Jürgen Teich. 2023. Efficient table-based function approximation on fpgas using interval splitting and bram instantiation. *ACM Trans. Embed. Comput. Syst.*, (Jan. 2023). Just Accepted. DOI: 10.1145/3580737.

[7] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: a programmer-guided compiler framework for practical approximate computing. University of Washington, (2015). https://dada.cs.washington.edu/research/tr/2015/01/UW-CSE-15-01-01.pdf.

[8] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011.* Mary W. Hall and David A. Padua, (Eds.) ACM, 164–174. DOI: 10.1145/1993498.1993518.