

Static Analysis for Efficient Affine Arithmetic on GPUs

by

Bryan Chan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

©Bryan Chan 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Bryan Chan

Abstract

Range arithmetic is a way of calculating with variables that hold ranges of real values. This ability to manage uncertainty during computation has many applications. Examples in graphics include rendering and surface modeling, and there are more general applications like global optimization and solving systems of nonlinear equations.

This thesis focuses on affine arithmetic, one kind of range arithmetic. The main drawbacks of affine arithmetic are that it taxes processors with heavy use of floating point arithmetic and uses expensive sparse vectors to represent noise symbols.

Stream processors like graphics processing units (GPUs) excel at intense computation, since they were originally designed for high throughput media applications. Heavy control flow and irregular data structures pose problems, though, so the conventional implementation of affine arithmetic with dynamically managed sparse vectors runs slowly at best.

The goal of this thesis is to map affine arithmetic efficiently onto GPUs by turning sparse vectors into shorter dense vectors at compile time using static analysis. In addition, we look at how to improve efficiency further during the static analysis using unique symbol condensation. We demonstrate our implementation and performance of the condensation on several graphics applications.

Acknowledgments

This thesis has taken far too long, but hopefully it has improved with age. Many thanks to my supervisor Michael McCool for being a boundless source of advice and interesting research ideas. I appreciate the patience of both Mike and my new advisor Pat over the past two years.

I would also like to thank my readers Gord Cormack and Stephen Mann for their feedback.

None of this work would exist without the Sh system. Kudos to Stefanus Du Toit and everybody involved in the Sh project. Working with all of you as GPGPU took off was an exciting experience.

Finally, I would like to thank the following organizations for helping to fund this research: Ontario Centres of Excellence (OCE), ATI Technologies, Natural Sciences and Engineering Research Council of Canada (NSERC), and Ontario Graduate Scholarship (OGS).

Contents

1	Introduction	1
1.1	Affine arithmetic applications	1
1.2	Harnessing GPUs	2
1.3	Contributions	2
2	Background	4
2.1	Range Arithmetic	4
2.1.1	Properties of a Closed Range Arithmetic	6
2.1.2	Computation Using Range Arithmetic	7
2.1.3	Notation	8
2.1.4	Interval Arithmetic	9
2.1.5	Affine Arithmetic	15
2.2	Applications	22
2.3	Stream Processing	23
2.3.1	Stream Programming Model	24
2.3.2	Stream Architectures	25
2.3.3	Stream Programming Languages	28
2.4	Previous Work	33
2.4.1	Range Arithmetic Libraries and Compilers	33
2.4.2	Affine Arithmetic Optimizations	33
2.4.3	Data Flow Analysis	34
2.5	Contributions	37

3	Static Analysis of Noise Symbols	38
3.1	Data Flow Analysis of Noise Symbols	38
3.1.1	Lattice for AA	38
3.1.2	Static Storage Allocation	39
3.2	Merging Symbols	40
3.2.1	Unique Condensation	42
3.2.2	Hierarchical Condensation	45
3.3	Implementation	46
3.3.1	Additional types and operators	47
3.3.2	From affine forms to float vectors	47
4	Applications	52
4.1	Performance Evaluation	52
4.2	Raytracing Implicit Surfaces	55
4.2.1	Test Cases	57
4.2.2	Results	58
4.3	Global Optimization on Generative Models	62
4.3.1	Global Optimization Algorithm	66
4.3.2	Results	67
5	Conclusions	71
5.1	Future Work	71
5.1.1	More General Applications	72
5.1.2	Smarter Condensation	72
	Bibliography	74
	Index	79

List of Tables

2.1	Summary of range arithmetic notation	8
3.1	Static analysis of f_{circle} showing the noise symbols for the result of each operation	41
3.2	Static analysis of f_{circle} showing the noise symbols after unique condensation, respecting input symbols.	44
4.1	Implicit functions using squared Euclidean distance from a generator curve.	57
4.2	Performance measures for the minimum distance examples and ratios showing the improvement of AAUC over AA. Total render time is over all iterations.	69
4.3	Total times for all parts of the global optimization algorithm	70

List of Figures

2.1	Calculating acceleration due to gravity	4
2.2	Interval arithmetic bounds for $y = \exp(x)$	11
2.3	Interval arithmetic bounds for $y = \sin(x)$	12
2.4	Joint range in affine arithmetic versus interval arithmetic	16
2.5	Chebyshev approximation of $\exp(x)$ over $[0, 2]$	19
2.6	Min-range approximation of $\exp(x)$ over $[0, 2]$	20
2.7	Root-mean-square image difference stream program	25
2.8	Overview of compilation in Sh	31
2.9	Structures extracted during structural analysis in Sh	32
2.10	Lattice diagram of \mathbb{Z}_3	35
3.1	Adding support for affine arithmetic to Sh.	46
3.2	Vectorizing affine forms	51
4.1	Execution time versus number of instructions	54
4.2	GIPS versus number of temporaries for a program with 2048 MAD instructions	54
4.3	Raytracing an implicit function	56
4.4	Sphere functions varying in number of spheres and bump deformations applied	59
4.5	Performance results for sphere functions.	60
4.6	Reduction in number of noise symbols in twists versus bumps	61
4.7	Minimum distance between point and teapot	68
4.8	Test cases for minimum distance between shapes	69

Formatting Notes

If possible, please read this thesis in colour. While the content does make sense in grayscale, some of the preattentive pop-out effects in the text and diagrams are lost without colour.

Where a term is defined, the *font and colour* changes. In the electronic version, terms used elsewhere in the text point back to the definition. For example, here is a link to a paragraph describing [range arithmetic](#). Most software for reading PDFs support jumping back to quickly resume reading after checking a definition (Alt+← in Adobe Acrobat). In the printed version, use the index to find the page for a definition.

Chapter 1

Introduction

1.1 Affine arithmetic applications

Range arithmetic is a powerful extension of real arithmetic that computes using ranges of numbers instead of single values. It provides an efficient way to understand how a function behaves over a set of input values. This has made it useful in numerical computation as it gives verifiable bounds even with many sources of uncertainty. Whether a measured input is inaccurate or floating point math introduces errors, range arithmetic can provide global guarantees on the validity, existence, and uniqueness of solutions. Although it is not universally applicable, it is a simple and efficient means for solving many kinds of problems.

Range arithmetic has proven especially useful in graphics as it can fill in the gaps between discrete samples and ensure correctness despite numerical instability. These benefits have led to numerous applications of range arithmetic in graphics including implicit surface raytracing [39, 12], accurate graphs of parametric functions [55], generative (procedural) surface modeling [50], sampling shader functions [25], and displacement mapping [23, 41]. It is also useful in applications outside graphics, for solving problems like nonlinear global optimization and solving systems of nonlinear equations.

While there are many flavours of range arithmetic with different advantages, the two most common in graphics have been *interval arithmetic* and *affine arithmetic*.

Interval arithmetic [40] operates on closed intervals of real numbers. For example, in standard interval arithmetic, $[0, 1] + [-1, 1] = [-1, 2]$.

Affine arithmetic is a generalized form of interval arithmetic, where instead of a lower and

upper bound, affine forms represent a range using a central value with a sequence of pairs of noise symbols and partial deviations. These pairs track uncertainty from different sources and take the form of a sparse vector. With this extra information, affine arithmetic can provide much tighter bounds than interval arithmetic and often allows range arithmetic computations to converge faster. The trade-off is that individual operations are more expensive in affine arithmetic than in standard interval analysis.

Part of this expense is due to an increase in arithmetic computation, but the expense of managing noise symbols is equally significant. Existing implementations deal with these symbols using extensive data-dependent control flow and dynamic memory management.

1.2 Harnessing GPUs

While modern graphics processing units (GPUs) are primarily designed as powerful image renderers, their raw speed is a good match for more general parallel programming problems. GPUs are a form of stream processor, a kind of parallel architecture designed for large data-parallel computations. Stream processors are optimized for running a program with high arithmetic intensity over sequences of elements [44]. On these kinds of programs, they offer more performance-per-watt than conventional general-purpose processors by taking advantage of different levels of locality and parallelism. As chip manufacturing technology improves, this performance gap continues to grow larger.

Affine arithmetic has a greater arithmetic intensity than real or interval arithmetic, so it is well suited for stream processors in this respect. However, to achieve high throughput, stream processors give up some control flow efficiency offered by the out-of-order pipelines and branch predictors in other architectures. Unfortunately, this makes standard approaches to noise symbol management prohibitively expensive.

While we focus on GPUs here, our approach is also suitable for other high throughput processors such as the Cell BE.

1.3 Contributions

To map affine arithmetic efficiently onto GPUs, we need to reduce the amount of control flow, so we have explored and further developed the solution suggested by Comba [7]: statically allocating noise symbols by resolving noise symbol manipulation at compile time. This removes the control

flow and leaves only the arithmetic computations on partial deviations for run-time.

Such a static assignment limits the ability to dynamically shorten sparse vectors by eliminating noise symbols that represent an insignificant or redundant amount of uncertainty. We explore one shortening method that is still available in a static context: **unique condensation**. This approach detects when multiple noise symbols appear uniquely in a single affine form. In that case, they can be combined into a single noise symbol.

We formulate these optimizations as **data flow** problems and use iterative techniques to solve them. Then we demonstrate the performance of our implementation on two graphics applications using branch-and-bound algorithms: raytracing of implicit surfaces and finding the closest approach between parametric surfaces. Both of these problems depend on finding the solution to a system of nonlinear equations, and the latter also involves solving an optimization problem.

Chapter 2

Background

The story begins with the development of [range arithmetic](#) leading to [affine arithmetic](#), with a sample of the numerous [applications](#) for range arithmetic. While affine arithmetic has advantages over other range arithmetics, it is computationally demanding. This makes throughput maximizing [stream processors](#) a good match for achieving high performance. However, by aiming for one extreme of the [parallel architecture design space](#), stream processors tend to perform poorly on the data structures and algorithms used in a traditional affine arithmetic implementation. This thesis builds on previous work in [range arithmetic implementation](#) and [compiler theory](#) to implement affine arithmetic in a way that avoids stream processor pitfalls.

2.1 Range Arithmetic

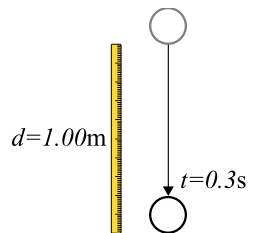


Figure 2.1: Calculating acceleration due to gravity

One of the earliest skills taught in science class is how to deal with measurement inaccuracy.

Suppose an initially stationary mass drops towards the earth, and we measure a time of $t = 0.3\text{s}$ to fall a distance of $d = 1.00\text{m}$. There is some leeway in these numbers since the ruler and stopwatch have limited precision. Assuming the stopwatch gives a reading in tenths of a second, then the true value of t may lie anywhere between 0.25s and 0.35s . Using the convention of *significant figures*, this uncertainty is indicated by the number of digits written down.

As a next step, suppose we want to compute a result from these measured values, like the Earth's acceleration due to gravity, g , for example. There is a convenient formula

$$d = gt^2$$

that gives $g = 11.111 \dots \text{m/s}^2$. Because of the uncertainty in the original values, though, most of these digits are meaningless.

To deal with this uncertainty the rules of significant figures state how many digits to keep after each basic arithmetic operation.

- Addition and subtraction
Keep only as many decimal places as there are in every operand
- Multiplication and division
Keep only as many significant figures as the operand with the least number of significant figures has

After an operation, round the result to remove extra digits.

For the gravity formula above, the result using significant figures is $g = 1 \times 10^1 \text{m/s}^2$, which suggests a more appropriate uncertainty in the computed result of about $\pm 5 \text{m/s}^2$.

For numerical computation, in addition to uncertainty in measuring the input and calculating results, there is also error from representing numbers digitally. Without using expensive infinite precision numerals, problems with rounding and underflow arise. Range arithmetic was developed to deal with all these sources of uncertainty by working with variables that represent not only a single value at a time, but any value within a set.

There are many models of range arithmetic. The first, interval arithmetic, was conceived independently by several mathematicians [14, 58, 54] and developed into a usable tool for automatic error analysis by Moore [40] while he was working for the Lockheed Missile and Space Division. It was inspired by the rules of significant figure arithmetic and an earlier arithmetic on sets of numbers [59].

Intervals by themselves, though have weaknesses, and other improved models of range arithmetic soon emerged. Some move closer to a statistical view of uncertainty by carrying confidence intervals or probability distributions along with ranges. Others like affine arithmetic use tighter higher-order approximations and track correlations between different values. Correlations help detect cancellation of related errors.

The next sections summarize the set-based foundations of all range arithmetics; introduce *interval arithmetic*, the precursor of affine arithmetic; and finally describe the focus of this thesis, affine arithmetic.

2.1.1 Properties of a Closed Range Arithmetic

For now think of a range as a set of real values, and assume all arithmetic is exact. Range representations and the pitfalls of *floating point* will be handled in more detail later.

While Moore [40] built interval arithmetic upon the standard real numbers, the more recent approaches based on the extended real line [45, 56, 26] handle special cases more cleanly. Real functions can be undefined or indeterminate for some values, giving results like $\frac{1}{0}$ and $\frac{\infty}{\infty}$ not on the real line. Using more flexible multivalued set functions and ranges on the extended reals, $\mathbb{R}^* = \mathbb{R} \cup -\infty, +\infty$, removes the need for special cases. Here is an overview of the fundamentals of range arithmetic from a set perspective, using the containment set notation of Walster [57].

Range arithmetic deals with *range functions* that map a range to another range. Its main goal is to take a real function $f : \mathbb{R} \rightarrow \mathbb{R}$ and construct a range function F that predicts the set of outputs of f over a given set of inputs. If computing F remains relatively efficient, then it is possible to understand the behaviour of f over infinite sets, which is unreliable with point sampling alone. No matter how close point samples are, there are always other real numbers in between the samples that can lead to an unexpected result. This is problematic with floating point representations of real numbers, since it is possible that none of the numbers in the gap are representable. Range arithmetic is helpful when such uncertainty arises. To be useful and computationally efficient, a *range extension* F of a real function f should have certain properties.

The first step is to give a sense of what the ideal output set should be. The *containment set* (cset) of real function f for a single value x is

$$\text{cset}(f, x) = \left\{ y \mid y = \lim_{z_n \rightarrow x} f(z_n) \right\},$$

for any sequence z_n converging to $x \in \mathbb{R}$.

Then for a set $X \subseteq \mathbb{R}$,

$$\text{cset}(f, X) = \bigcup_{x \in X} \text{cset}(f, x).$$

For example, when

$$\begin{aligned} f(x) &= \frac{1}{x}, \\ \text{cset}(f, 0) &= \{-\infty, +\infty\}, \\ \text{cset}(f, [0, 1]) &= \{-\infty\} \cup [0, +\infty]. \end{aligned}$$

For simple functions, it may be possible to symbolically derive the exact cset, but this becomes difficult to do for complicated functions. Range arithmetic aims to predict the cset as closely as possible. This suggests two complementary approaches: one method gives a subset of the cset, trying to make it as large as possible. An example of this is inner interval arithmetic. The more commonly used method is to give some superset of the cset as a result. That is F must meet the *containment constraint*, also known as the Fundamental Invariant of range arithmetic:

Given a real function $f(x)$, a corresponding range function F satisfies the containment constraint over an input set X if

$$\text{cset}(f, X) \subseteq F(x).$$

While an F that always returns $[-\infty, +\infty]$ certainly works here, a good range function should also aim to be tight. That is the result set should be as small as the range representation allows. Sometimes though, when tight bounds are inefficient to compute, loose bounds may be more useful.

2.1.2 Computation Using Range Arithmetic

So far in this section functions have been purely a mapping between a domain and range. For numerical computation, though, evaluating a function involves an algorithm. There is a distinction between the meaning of a function f and an expression f . While a function is uniquely defined, there may be several algebraically equivalent expressions for for a function; for example,

$$\frac{1}{x} = 1 + \frac{1-x}{x}.$$

Taking a real expression f and substituting range variables for real variables gives the *natural range extension* of f . As long as each of the range operations satisfies the containment constraint, the whole expression also does.

	Notation
real value (\mathbb{R}^*)	x, y, z
interval arithmetic value (\mathbb{IR}^*)	$\mathbf{x}, \mathbf{y}, \mathbf{z}$
affine arithmetic value (\mathbb{AR}^*)	$\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$
lower bound	$\underline{x}, \underline{y}, \underline{z}$
upper bound	$\overline{x}, \overline{y}, \overline{z}$
function cset	$f(x), f(\mathbf{y}), f(\hat{\mathbf{z}})$
expression	$\mathbf{f}(x), \mathbf{f}(\mathbf{y}), \mathbf{f}(\hat{\mathbf{z}})$

Table 2.1: Summary of range arithmetic notation

Another concern for evaluating range expressions is that manipulating general sets of values is unwieldy. Instead *range arithmetics* normally use only connected sets, or intervals, on the real line. In *interval arithmetic*, these interval variables are independent, while in *affine arithmetic* they have linear correlations. In the rest of this section, we go into more detail on these two kinds of range arithmetic.

2.1.3 Notation

Range arithmetic gives rise to a considerable amount of specialized notation. While many incompatible notations exist, there is a proposed standard [32] for interval arithmetic. This thesis follows that proposal extended with the notation from the original affine arithmetic paper [7] for *affine forms*. Here is a quick description of the notation used in the rest of the thesis (Table 2.1).

Real variables use standard lowercase italic letters, while the corresponding interval values use a bold face for interval arithmetic and a caret for affine arithmetic. The bounds on the values in a range are indicated using an underline for the lower bound and an overline for the upper bound.

There is no difference in notation for functions between the different kinds of arithmetic. The expression f always represents a mapping where the domain and range are implied by the argument. For example $f(x)$ is a real-valued function, and $f(\mathbf{x})$ is an interval-valued function. We also follow Hansen here by making f always mean the *cset* unless otherwise indicated. The notation for an expression \mathbf{f} is similar, and range expressions corresponding to a real expression are always natural range extensions unless specified otherwise.

Along with the specialized notation for values, there are several useful functions for extracting properties from a range variable. The definitions below show only interval arithmetic variables, but the notation is the same for any range arithmetic using connected sets, like affine arithmetic.

The *midpoint* of a range is

$$\text{mid}(\mathbf{x}) = \frac{1}{2}(\underline{x} + \bar{x}).$$

The *width* and *radius* of a range indicate how much uncertainty there is about the midpoint.

$$\begin{aligned}\text{wid}(\mathbf{x}) &= \bar{x} - \underline{x} \\ \text{rad}(\mathbf{x}) &= \frac{1}{2}\text{wid}(\mathbf{x})\end{aligned}$$

The *hull* of a set S is the smallest interval containing S .

$$\square S = \left[\inf_{x \in S} x, \sup_{x \in S} x \right]$$

When treated as sets, ranges can accommodate all the usual set operations. These are not directly useful since often the result is not representable as a range. There are two special functions though corresponding to the smallest representable set union and intersection: the *meet* and *join*.

$$\begin{aligned}\text{join}(\mathbf{x}, \mathbf{y}) &= \square(\mathbf{x} \cup \mathbf{y}) \\ \text{meet}(\mathbf{x}, \mathbf{y}) &= \square(\mathbf{x} \cap \mathbf{y})\end{aligned}$$

2.1.4 Interval Arithmetic

Interval arithmetic (IA) operates on values that are independent real intervals. Each real interval is a set of values

$$\mathbf{x} = \{x \mid \underline{x} \leq x \leq \bar{x}\},$$

where $\underline{x} \in \mathbb{R} \cup \{-\infty\}$ is the lower bound, and $\bar{x} \in \mathbb{R} \cup \{+\infty\}$ is the upper bound.

The set of all real intervals is denoted by \mathbb{IR}^* . This is a generalization of \mathbb{R} , since for every $x \in \mathbb{R}$ there is a *degenerate interval* $[x, x] \in \mathbb{IR}^*$

Basic Arithmetic Operations

All of the basic operators, $+$, $-$, \times , \div are relatively straightforward to define for intervals. Given two interval variables $\mathbf{x} = [\underline{x}, \bar{x}]$ and $\mathbf{y} = [\underline{y}, \bar{y}]$,

$$\begin{aligned}
-\mathbf{x} &= [-\bar{x}, -\underline{x}], \\
\mathbf{x} + \mathbf{y} &= [\underline{x} + \underline{y}, \bar{x} + \bar{y}], \\
\mathbf{x} - \mathbf{y} &= \mathbf{x} + (-\mathbf{y}), \\
\mathbf{x} \times \mathbf{y} &= [\min(\underline{x} \times \underline{y}, \underline{x} \times \bar{y}, \bar{x} \times \underline{y}, \bar{x} \times \bar{y}), \max(\underline{x} \times \underline{y}, \underline{x} \times \bar{y}, \bar{x} \times \underline{y}, \bar{x} \times \bar{y})], \\
\mathbf{x} \div \mathbf{y} &= \begin{cases} [-\infty, +\infty] & \text{if } 0 \in \mathbf{y} \\ \mathbf{x} \times \left[\frac{1}{\bar{y}}, \frac{1}{\underline{y}}\right] & \text{otherwise} \end{cases}.
\end{aligned}$$

The special cases involving ± 0 and $\pm\infty$ for division come from cset arguments and taking the hull, and Hansen lists these in tables in chapter 4 of his book [22]. For example, here is a derivation for $[1, 2]/[0, 0]$: For some given $\alpha > 0$, define

$$f(x) = \frac{\alpha}{x}.$$

Then we can calculate

$$\begin{aligned}
\text{cset}(f, 0) &= \lim_{x \rightarrow 0}, \\
\text{cset}(f, 0) &= \left\{ \lim_{y \rightarrow -0} f(y) \right\} \cup \left\{ \lim_{y \rightarrow +0} f(y) \right\} \\
&= \{-\infty, +\infty\}.
\end{aligned}$$

Since this is true for any $\alpha > 0$, it follows that

$$\begin{aligned}
[1, 2]/[0, 0] &= \square(-\infty, +\infty) \\
&= [-\infty, +\infty].
\end{aligned}$$

For a more complicated case, the input range can be partitioned into subsets that behave in the same way. For example, $[-1, 1]/[0, 0]$ might be split into three cases: $[-1, 0]/[0, 0]$, $[0, 0]/[0, 0]$, and $(0, 1]/[0, 0]$. This stems from the original definition of cset since if an input range $X = X_1 \cup X_2$,

$$\text{cset}(f, X) = \text{cset}(f, X_1) \cup \text{cset}(f, X_2).$$

General Functions

For other functions in general, there are many techniques to derive range extensions that depend on the properties of a function.

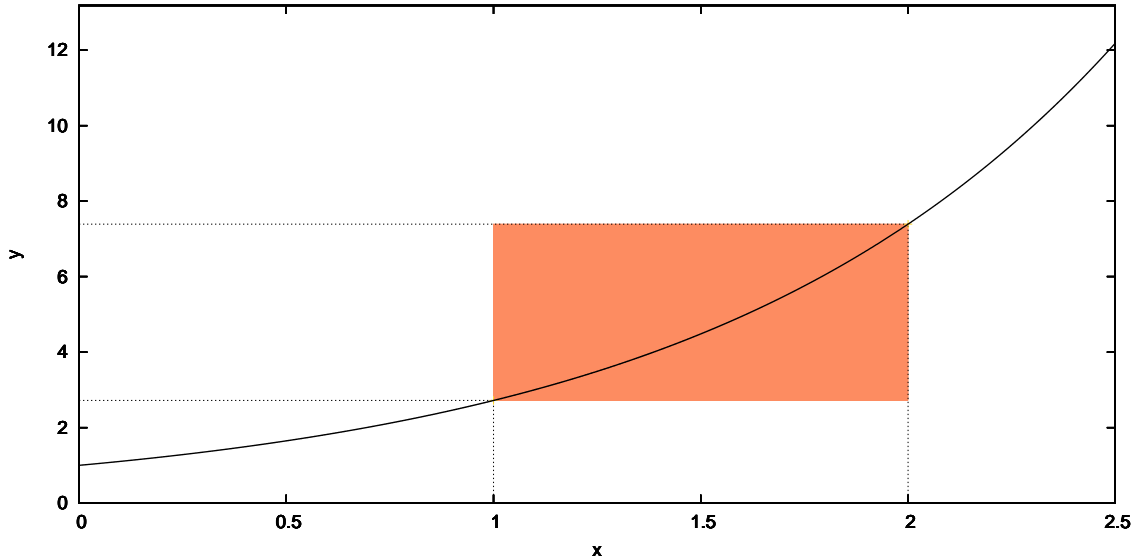


Figure 2.2: Interval arithmetic bounds for $y = \exp(x)$

Monotonic functions are the easiest to deal with. Over an interval domain, the extrema for the function must occur at the boundaries, so for an increasing monotonic function

$$f(\mathbf{x}) = [f(\underline{x}), f(\bar{x})].$$

For example, the exponential function needs only two point evaluations, as shown in Figure 2.2,

$$\exp([1, 2]) = [\exp(1), \exp(2)].$$

Functions that are monotonic over subintervals can be split into subranges. If \mathbf{x} spans more than one monotonic region, then restricting \mathbf{x} to each region and then taking a join of all the partial results works. For example, with $\sin(\mathbf{x})$ in Figure 2.3

$$\begin{aligned} \sin([1, 4]) &= \text{join}(\sin([1, \pi]), \sin([\pi, 4])) \\ &= [\sin(4), \sin(\pi)]. \end{aligned}$$

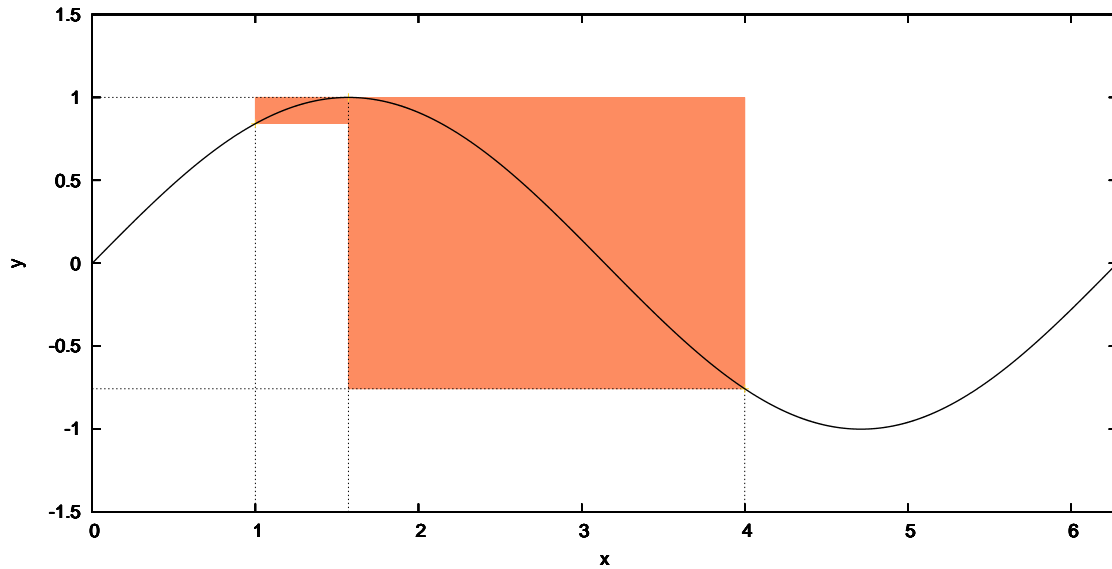


Figure 2.3: Interval arithmetic bounds for $y = \sin(x)$

When a function meets none of the above conditions, perhaps it can be broken down into a composition of simpler operations. If all else fails, the function can be approximated, using a Taylor approximation for example, with any approximation error added to the resulting interval.

Floating Point Arithmetic

So far, all arithmetic has been exact. In a computational context, where numbers are stored with a limited number of bits, issues with arithmetic errors come up. Most modern architectures and programming languages support the IEEE 754 standard [2], which defines bit layout, special values, operator precision, arithmetic exceptions, and rounding modes. The operator precision specification provides bounds on errors due to floating point operations [21]. An interval arithmetic implementation can take these various kinds of uncertainty into account and still maintain the containment constraint.

A *floating point* number takes the form

$$\pm d.dd \cdots d \times \beta^e,$$

where β is the base, e is the exponent, and the number of digits in the significand $d.dd \cdots d$ is p .

The 1985 revision of IEEE 754 uses $\beta = 2$ and defines two basic formats: 32-bit single precision uses $p = 24$, and 64-bit double precision uses $p = 53$. Out of the remaining bits, one is a sign bit and the rest are devoted to e . Additional extended formats with more bits are available, and the upcoming revision of the standard will likely support a 128-bit format along with $\beta = 10$ formats.

While many previous floating point formats used similar representations, the main benefit of the IEEE standard is the definitions for how basic operators should work and user control over rounding and exceptions. All the basic arithmetic operators, remainder, and square root must use exactly rounded arithmetic. This means that the result is computed as if it had arbitrary precision and then rounded to one of the adjacent representable floating point numbers. This user-selectable directed rounding mode can round to nearest or towards one of 0 , $-\infty$, or $+\infty$.

With rounding mode control, it is possible to ensure that a computed floating point interval maintains the containment constraint by rounding the lower bound computation down and upper bound up. However, for some operations such as transcendental functions like $\exp(x)$ the standard does not define a required precision. Also some architectures support the standard half-heartedly. In particular, switching rounding modes may be difficult and basic operations like division may be computed with less precision than mandated by the specification [3]. In these cases, an interval arithmetic implementation needs to spend additional time computing bounds on floating point error to keep the containment constraint.

Issues with Interval Arithmetic

There are several issues with interval arithmetic that made it unpopular for several decades. This was due in part to a misunderstanding of when to use intervals, and also a limitation of treating values as independent quantities. As a result, interval computations often returned $[-\infty, +\infty]$ and interval arithmetic gained a reputation for being unproductive; however, as demand grows for verifiable computing, interval arithmetic and extensions like [affine arithmetic](#) may see more widespread usage.

For some applications, interval arithmetic is too slow because it performs poorly on wide input intervals. For example, let the *approximation error* of the computed interval expression versus the ideal be

$$E[f(\mathbf{x})] = \text{wid}(f(\mathbf{x})) - \text{wid}(\square f(\mathbf{x})).$$

Then Moore showed that for rational functions, if $\text{wid}(\mathbf{x}) = d$ then

$$E[f(\mathbf{x})] = O(d).$$

Except for a few pathological cases, this is also true in general, so subdividing input intervals will eventually give results that are arbitrarily tight. However, overall interval arithmetic converges at a relatively slow linear rate compared to the input range width.

The other issue with interval arithmetic is that treating values as independent quantities causes loose results. Even if the inputs are independent, after some calculation the intermediate values are usually partially correlated. While Moore showed that for a rational function, any expression with no repeated variables has a tight *natural range extension* [40], any repeated variables will cause results to be larger than the *cset*. If the result is much wider, it can require extensive subdivision before settling on usable results.

Consider the subtraction operation

$$\mathbf{x} - \mathbf{y} = [\underline{x} - \bar{y}, \bar{x} - \underline{y}].$$

Given $\mathbf{x} = [-1, 2]$, the operation $\mathbf{x} - \mathbf{x} = [-3, 3]$ is much wider than the expected value $[0, 0]$. To handle this case, a special dependent operation subtraction may be provided:

$$\mathbf{x} \ominus \mathbf{y} = [\underline{x} - \underline{y}, \bar{x} - \bar{y}].$$

This gives the expected result only when it is known that \mathbf{x} and \mathbf{y} are additively dependent. This condition is difficult to check in a long expression and it does not cover the case of partial correlations such as $\mathbf{x} = \frac{1}{2}\mathbf{y} + \frac{1}{3}\mathbf{z}$. Another option is to algebraically manipulate the expression to give fewer repeated variables, but this too is difficult to do in general.

To understand why loose bounds cause an explosion in uncertainty, consider the *joint range* of all variables. If each variable is a subset of \mathbb{R} , then together n variables occupy a joint range that is a subset of \mathbb{R}^n . In interval arithmetic, this subset takes on the shape of an n -dimensional box, since all the variables are assumed to be independent. Because of dependence between variables, the true joint ranges are subsets of these boxes. For example, in the subtraction above, the true joint range of \mathbf{x} and itself is a line segment in \mathbb{R}^2 which is much smaller than the box.

This difference between the multidimensional volume, or *content*, of the computed versus true joint range gives rise to the *wrapping effect*. The computed set is slightly larger, and is like wrapping the true set in a layer of thick wrapping paper. This represents a loss of dependence information because of the independence property of interval arithmetic. In a long computation, each operation adds another layer to the wrapping until the final result is covered in so many layers that it no longer resembles the true *cset*.

2.1.5 Affine Arithmetic

Affine arithmetic (AA) [7] is an extension to interval arithmetic that reduces both problems mentioned above. Like interval arithmetic, it uses a single range of values for each variable, but it also establishes linear correlations between terms. This makes joint ranges tighter. Also, since each value in affine arithmetic is a linear approximation, the approximation error is quadratic instead of linear. This means that as inputs are subdivided, algorithms converge at a faster rate than with interval arithmetic and require fewer range function evaluations. However, the tradeoff is that each affine arithmetic operation is slower, and the space needed for correlation information grows with the number of operations in a function. Here is a summary of the values, basic arithmetic operations, and general functions in affine arithmetic.

A value in affine arithmetic, called an *affine form* is a first degree polynomial:

$$\hat{\mathbf{x}} = x_0 + \sum_{i=1 \dots n} x_i \varepsilon_i.$$

Each $\varepsilon_i \in U = [-1, +1]$ is a *noise symbol* that represents an independent source of uncertainty, and each $x_i \in \mathbb{R}^*$ is a *partial deviation* that shows how much $\hat{\mathbf{x}}$ varies relative to a particular noise symbol. This affine form represents a range of values between the following bounds:

$$\begin{aligned} \underline{x} &= x_0 - \text{rad}(\hat{\mathbf{x}}), \\ \bar{x} &= x_0 + \text{rad}(\hat{\mathbf{x}}), \\ \text{where } \text{rad}(\hat{\mathbf{x}}) &= \sum_{i=1 \dots n} |x_i|. \end{aligned}$$

For any value in the range, there are one or more choices of $\varepsilon_i \in U$ that produce that value. The formula above also shows how to convert from an affine arithmetic value to an interval arithmetic value.

$$\hat{\mathbf{x}} \rightarrow [\underline{x}, \bar{x}] = [x_0 - \text{rad}(\hat{\mathbf{x}}), x_0 + \text{rad}(\hat{\mathbf{x}})]$$

To convert from an interval $\mathbf{y} = [\underline{y}, \bar{y}]$ to an affine form, a new, independent noise symbol ε_k is added.

$$[\underline{y}, \bar{y}] \rightarrow \text{mid}(\mathbf{y}) + \text{rad}(\mathbf{y})\varepsilon_k$$

Different variables can share noise symbols to indicate correlation between their values. For example, let

$$\begin{aligned} \hat{\mathbf{x}} &= 1 + 1\varepsilon_1 + 0.5\varepsilon_2, \\ \hat{\mathbf{y}} &= 2 + 0.5\varepsilon_1. \end{aligned}$$

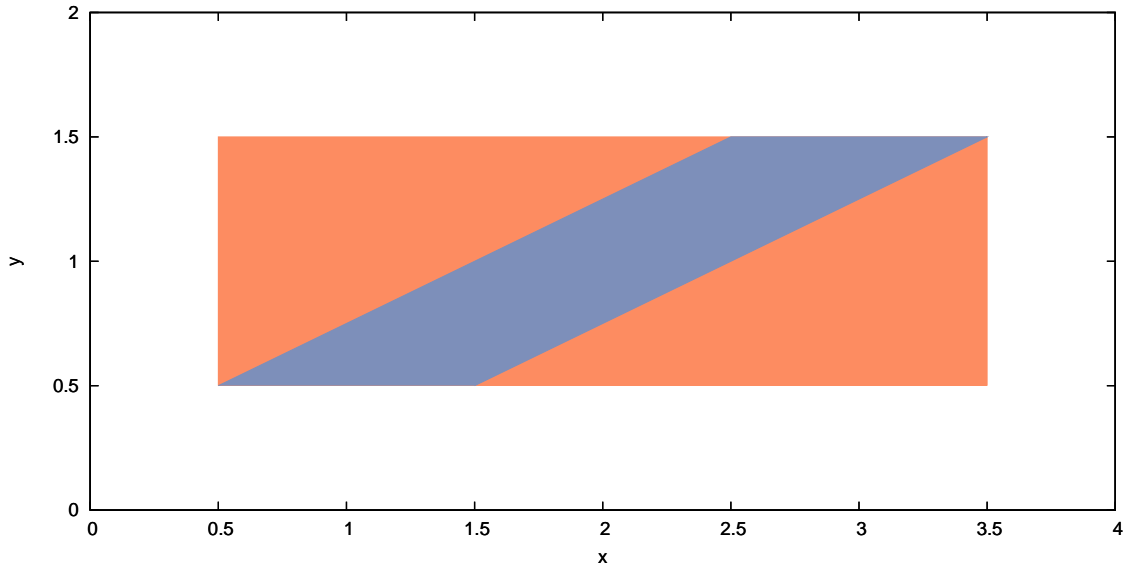


Figure 2.4: Joint range in affine arithmetic ■ versus interval arithmetic ■ of $\hat{\mathbf{x}} = 1 + 1\varepsilon_1 + 0.5\varepsilon_2$ and $\hat{\mathbf{y}} = 2 + 0.5\varepsilon_1$.

This highlights one of the advantages of using affine arithmetic. Converted to intervals $\mathbf{x} = [0.5, 2.5]$ and $\mathbf{y} = [1.5, 2.5]$, and their joint range is a box. Figure 2.4 shows that the parallelogram joint range of $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$. In general, the joint range of a set of affine forms is a *zonotope*, a center-symmetric polytope. This zonotope can be thought of as the Minkowski sum of the vectors corresponding to each noise symbol, in this case $(1, 0.5)$ and $(0.5, 0)$ translated by the central values $(1, 2)$. When some of the variables are correlated, the zonotope sits inside the box representing the joint range of the variables computed using standard intervals. Since the content of this polytope is smaller than the box, the wrapping effect is significantly reduced.

Typically each input to a calculation has one or more noise symbols representing uncertainty in the input, with correlated inputs sharing noise symbols. Then during calculation, new noise symbols must be added each time an operation introduces additional uncertainty that cannot be expressed as a linear combination of existing partial deviations.

Functions in Affine Arithmetic

As in interval arithmetic, basic arithmetic operations and other functions can be extended to affine arithmetic based on their properties. This is a summary from the monograph by Stolfi and Figueiredo, which covers the math library operations more comprehensively [52].

Affine operations can be handled directly. Given affine forms $\hat{\mathbf{x}}, \hat{\mathbf{y}}$ and $\alpha \in \mathbb{R}^*$,

$$\begin{aligned}\alpha \hat{\mathbf{x}} &= (\alpha x_0) + \sum_{i=1 \dots n} \alpha x_i \varepsilon_i, \\ \hat{\mathbf{x}} + \alpha &= (x_0 + \alpha) + \sum_{i=1 \dots n} x_i \varepsilon_i, \\ \hat{\mathbf{x}} + \hat{\mathbf{y}} &= (x_0 + y_0) + \sum_{i=1 \dots n} (x_i + y_i) \varepsilon_i.\end{aligned}$$

For non-affine functions, an affine approximation can be used. That is, given real $f(x)$ and an affine form $\hat{\mathbf{x}}$, some affine approximation exists

$$f^a(\varepsilon_1, \dots, \varepsilon_n) = \sum_{i=1 \dots n} \alpha_i \varepsilon_i + \beta,$$

with approximation error

$$\delta = \max_{\varepsilon_i \in U} \left| f^a(\varepsilon_1, \dots, \varepsilon_n) - f \left(x_0 + \sum_{i=1 \dots n} x_i \varepsilon_i \right) \right|.$$

Then the range function $f(\hat{\mathbf{x}})$ can be computed as

$$\hat{\mathbf{y}} = f(\hat{\mathbf{x}}) = f^a(\varepsilon_1, \dots, \varepsilon_n) + \delta \varepsilon_k,$$

where ε_k is a new noise symbol representing the approximation error as an independent source of uncertainty. Usually the following simplified form is used for f^a since it is more efficient to pick two constants instead of n .

$$f^a(\varepsilon_1, \dots, \varepsilon_n) = \alpha \hat{\mathbf{x}} + \beta$$

In the univariate case, as long as none of the noise symbols are distinguished for a special reason, this simplified form is equivalent to the previous version above. It reduces to finding a linear approximation

$$p(x) = \alpha x + \beta,$$

for $f(x)$ over the interval $[\underline{x}, \bar{x}]$.

There are two common ways of forming $p(x)$: Chebyshev approximation and min-range approximation. *Chebyshev approximation* (minimax approximation) gives the smallest possible δ . *Min-range approximation* gives a result that minimizes radius.

For Chebyshev approximation of univariate functions, the alternation theorem [11] is convenient. It implies that the best degree n polynomial approximation p of a function f over a range $[a, b]$ has $n + 2$ alternations in its approximation error function

$$r(x) = f(x) - p(x).$$

This means that there exists a sequence of $n + 2$ values, called the points of alternation

$$a \leq x_1 < x_2 < \cdots < x_{n+2} \leq b,$$

so that the following condition on alternating signs holds:

$$r(x_i) = -r(x_{i+1}) = \max_{x \in [a, b]} |r(x)|.$$

Since affine arithmetic approximates with linear polynomials, p takes on the form

$$p(x) = \alpha x + \beta,$$

and at least three alternations occur in the best Chebyshev approximation. When a twice differentiable function f is convex or concave, meaning that f'' does not change sign over $[a, b]$, this gives a convenient algorithm for computing the approximation. In this case, there are exactly three alternation points, and two of them occur at the endpoints a, b .

$$\begin{aligned} x_1 &= a \\ x_3 &= b \\ r(a) &= r(b) \end{aligned}$$

From this it can be deduced that

$$\alpha = \frac{f(b) - f(a)}{b - a}.$$

Now only one alternation point, x_2 , remains somewhere in between a and b . Since $r(x)$ reaches a maximum or minimum at x_2 , solving for $r'(x_2) = 0$ gives the required answer. After some manipulation, the result is

$$x_2 = (f')^{-1}(\alpha).$$

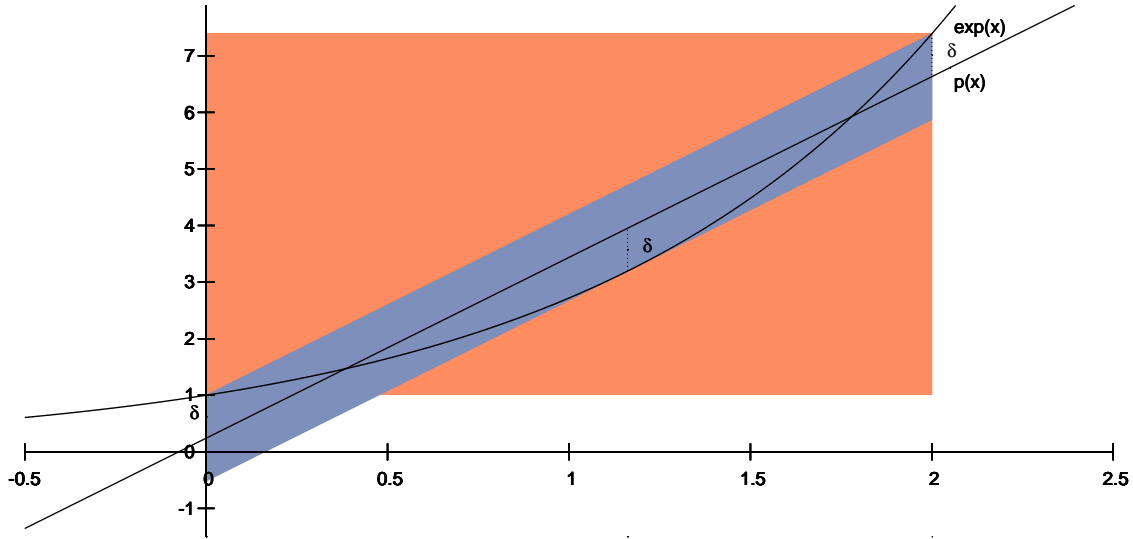


Figure 2.5: Chebyshev approximation of $\exp(x)$ over $[0, 2]$ used in affine arithmetic ■ versus the bounds computed for interval arithmetic ■

This gives the following linear equations, from which it is possible to solve for β and δ

$$\begin{aligned}\delta &= r(x_1) = f(x_1) - \alpha x_1 - \beta \\ \delta &= -r(x_2) = \alpha x_2 + \beta - f(x_2).\end{aligned}$$

Here is an example showing the calculation for $\exp(x)$ over $[0, 2]$ in Figure 2.5:

$$\begin{aligned}\alpha &= \frac{e^b - e^a}{b - a} = \frac{e^2 - 1}{2} \\ x_2 &= \log(\alpha) \\ \beta &= \frac{e^a - \alpha a + e^b - \alpha b}{2} = \frac{1 + e^2 - 2\alpha}{2} \\ \delta &= \frac{e^a - \alpha a - e^b + \alpha b}{2} = \frac{1 - e^2 + 2\alpha}{2}.\end{aligned}$$

The min-range approximation is often easier to compute; however, it is only useful for monotonic functions. For continuous but non-monotonic functions, the result is the same as doing an interval arithmetic evaluation, losing all dependence information. The cases where min-range

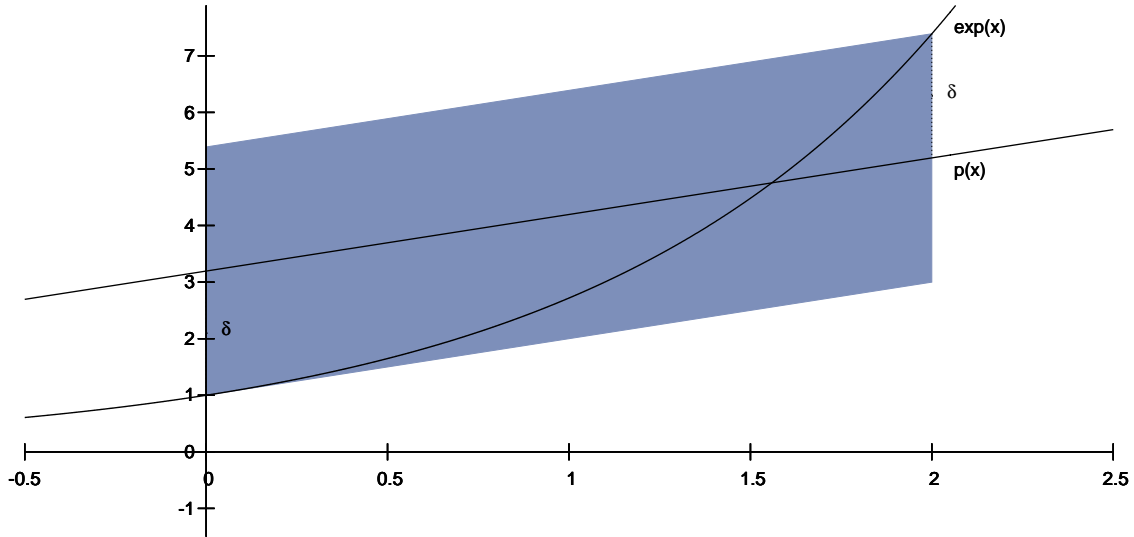


Figure 2.6: Min-range approximation of $\exp(x)$ over $[0, 2]$

works particularly well are when f is monotonic and either convex or concave. Then the resulting zonotope is tangent to f at one endpoint, and the calculation is much easier than the Chebyshev approximation. For example, here is a repeat of the $\exp(x)$ over $[a, b] = [0, 2]$ using min-range instead. The result appears in Figure 2.6. The zonotope in this case must be tangent to the curve at the lower bound.

$$\begin{aligned}
 \alpha &= e^a = 1 \\
 \beta &= e^a - a\alpha = 1 \\
 \delta &= \frac{r(b)}{2} \\
 &= \frac{e^b - (\alpha b + \beta)}{2} \\
 &= \frac{e^2 - 3}{2}
 \end{aligned}$$

There are several reasons for choosing either Chebyshev or min-range. Usually the Chebyshev approximation is the best since it keeps as much dependence information as possible. This gives a zonotope with the smallest possible content and reduces the wrapping effect. On the other hand,

the min-range calculation is often more numerically stable and efficient to compute. Also the minimum radius constraint ensures that if the input range is a subset of the domain of f , then the output range is as well. The Chebyshev approximation does not satisfy this property and can “overshoot” or “undershoot”, as seen in Figure 2.5 – part of the result lies below 0, outside of the range of $\exp(x)$). In fact, the width of the result is greater than in the standard interval approach.

Multivariate functions are more difficult to handle, since properties assumed above like the alternation theorem no longer apply. In general, such functions should be broken down into a composition of univariate or affine functions, which gives a workable but not optimal approximation. For example, the bivariate min function can be expressed using the univariate $\text{pos}(z) = \max(z, 0)$:

$$\min(\hat{\mathbf{x}}, \hat{\mathbf{y}}) = \hat{\mathbf{x}} - \text{pos}(\hat{\mathbf{x}} - \hat{\mathbf{y}}).$$

For some commonly used operations, though, like $\hat{\mathbf{x}} \times \hat{\mathbf{y}}$, it may be helpful to derive an approximation symbolically.

$$\begin{aligned} \hat{\mathbf{x}} \times \hat{\mathbf{y}} &= (x_0 + \sum_{i=1 \dots n} x_i \varepsilon_i)(y_0 + \sum_{i=1 \dots n} y_i \varepsilon_i) \\ &= x_0 y_0 + \sum_{i=1 \dots n} (y_0 x_i + x_0 y_i) \varepsilon_i + \sum_{i=1 \dots n} \sum_{j=1 \dots n, j \neq i} x_i y_j \varepsilon_i \varepsilon_j \end{aligned}$$

In this derivation, all but the summation are affine operations. While the best possible affine approximation to this final term can be computed exactly in $O(n \log n)$ time, using a slightly looser simplification for the partial deviation on a new noise symbol ε_k is more practical [52]:

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 y_0 + \sum_{i=1 \dots n} (y_0 x_i + x_0 y_i) \varepsilon_i \text{rad}(\hat{\mathbf{x}}) \text{rad}(\hat{\mathbf{y}}) \varepsilon_k.$$

Again the presentation so far has ignored floating point error. In this case, to maintain the containment constraint, floating point error induced while calculating an approximation can be added to δ , with the new noise symbol. If available, directed rounding helps to ensure that this computation does not end up with a range that is too small.

This summary of affine arithmetic suggests that dealing with noise symbol sets and computing function approximations makes it more expensive per operation than real arithmetic. A function with n inputs and m operations takes $O(m(n+m))$ space and time, as each non-affine operation adds a new symbol to its result.

2.2 Applications

Range arithmetic is useful in many areas whenever there is a need to understand how intermediate or output variables in a function behave over some set of inputs.

Many numerical applications are well-suited to range arithmetic, from root finding to global optimization. One of the algorithms often used for solving such problems is *branch-and-bound*, shown in pseudocode below:

```

 $S = \mathbf{x}$ 
while  $S$  is not empty:
  remove  $\mathbf{y}$  from  $S$  // (A)
  subdivide  $\mathbf{y}$  into  $\mathbf{y}_1, \dots, \mathbf{y}_n$  // (B)
  compute each  $z_i = f(\mathbf{y}_i)$ 
  add  $\mathbf{y}_i$  to  $S$  if  $z_i$  fits some criteria // (C)
  filter all ranges in  $S$  // (D)
  check if done // (E)

```

Starting with some real function $f(x)$ and initial interval of interest \mathbf{x} , this algorithm recursively subdivides \mathbf{x} . By computing bounds on subintervals and filtering out uninteresting ones, the eventual result is a set of small subintervals matching some criteria. This could be, for example, the roots of f , or the global minimum of f . Many variations on this algorithm exist, depending on factors like the what order to pull elements from S (A), how to generate subintervals (B), how to filter (C, D), and when to stop (E). Filtering eliminates ranges from consideration, for example if the range extension computed for the current \mathbf{y} proves there are no solutions in another candidate range. A common stopping point is when the intervals in S all have width less than some ε or enough is known about the function that a more efficient algorithm can be used instead. During root-finding, for example, if a range evaluation of the derivative shows the function near a root is monotonic, then a real arithmetic technique can be used instead like the bisection method or regula falsi.

The main benefit of this algorithm is that it never misses any solutions, as long as the $f(\mathbf{y}_i)$ evaluation meets the containment constraint.

There are many applications that use this basic algorithm. For 2D graphing without misrepresentation, Fateman used interval arithmetic to plot “honest” graphs [16], and Tupper uses a generalization of interval arithmetic in GrafEq to draw 2D functions in bitmaps [55]. For rendering surface displacement, Moule used interval textures to control the level of detail for mesh

subdivision [41], and Heidrich used *affine arithmetic* to control quadtree subdivision for raytracing displacement shaders [23]. Snyder shows how solving constrained nonlinear optimization using interval arithmetic can be used in graphics for a wide range of generative surface modeling and rendering techniques including raytracing implicit surfaces, intersecting parametric surfaces, and finding minimum distance between shapes [49, 50]. Later work demonstrates a similar set of applications using affine arithmetic [7, 13].

Range arithmetic is useful in program analysis as well. Heidrich used affine arithmetic (AA) to compute error bounds for sampled procedural shaders, programs that describe surface materials for rendering [25]. In many cases in graphics where computing a shader is too expensive, it may be desirable to use a sampled version of the function instead. Without an error analysis, though, the sampling may unknowingly miss important features. AA has been used for similar error analysis of the outputs of digital signal processing methods [15]. In addition to error analysis of program outputs, range arithmetic can also give bounds on intermediate variables. For integer or fixed-point types, this gives conservative bounds on the number of bits needed to represent each variable. This can produce more efficient code for chips that natively support more than one number format. For silicon compilation, interval arithmetic was used to minimize bit usage in integer and pointer variables [53].

In a more general context, Hansen’s book [22] describes how to use interval arithmetic to solve a variety of constrained and unconstrained global optimization problems. It also discusses other techniques like solving linear equations, and root finding with interval Newton’s method.

2.3 Stream Processing

Media applications are one of the main drivers behind *stream processor* development [46]. Tasks like video compression, audio filtering, and 3D rendering share several important characteristics. They deal with large sets of similar data elements and require high throughput, approaching hundreds of GFLOPs. Fortunately processing on these elements is often independent, so these tasks are highly parallelizable as they require little global communication. Not only is the throughput high, but also per element operations tend to be expensive. This requires high *arithmetic intensity*, defined as the ratio of computation to memory bandwidth used. Numerical methods in areas like linear algebra, global optimization, physical simulation, and the solution of differential equations have similar computational requirements. The same is true for *range arithmetic* techniques like those described in the last section, so all of these general-purpose applications make

good candidates for stream processors as well. These types of parallel programming problems inspired the design of the stream programming model and stream architectures.

2.3.1 Stream Programming Model

The two building blocks in the stream programming model are the stream and kernel. A *stream* is a set of similar data elements, such as integers, floats, or records like an RGB colour. These elements can be unordered, or they may be ordered, for instance in a sequence or 2D array. A *kernel* is a function that maps a set of input streams to a set of output streams. Programming with these abstractions works by connecting kernel outputs to the inputs of other kernels, producing a *stream graph*. This graph provides locality and concurrency information at different granularities that allows efficient execution by a stream processor.

In general, kernels fall into several basic categories. A *map* produces one output element corresponding to each input element, much like the map operation on lists in Scheme. A *reduction* takes several input elements to produce one output element; for example, finding the largest element in a stream is a reduction operation. The opposite of a reduction is an *expansion*, which gives multiple output elements for a single input element. An example would be subdivision schemes where one piece of an input domain, or a segment of geometry is recursively split into smaller portions. The *filter* operation copies selected input stream elements to an output stream based on some criteria.

In addition to the basic kernel types, memory access patterns for streams can also be classified. There are the basic sequential operations of reading a stream in order and copying one stream to another. Also a random access read is called a *gather* and random write is a *scatter*.

Here is an example of a stream program. A common operation in image processing is deciding how similar two images are. Given two $m \times n$ greyscale images $A_{i,j}$ and $B_{i,j}$, a simple measure for similarity is the root-mean-square difference.

$$R(A, B) = \sqrt{\frac{\sum_{\text{all } i,j} (A_{i,j} - B_{i,j})^2}{mn}}$$

This can be expressed as a stream program composed of a map and a reduction. Figure 2.7 shows the corresponding stream graph that takes the two images as input streams and returns one scalar result.

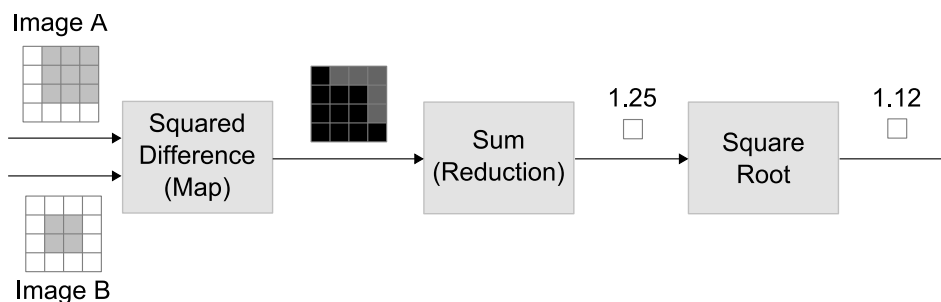


Figure 2.7: Root-mean-square image difference stream program

Parallelism and Locality

The stream model provides many opportunities for efficient execution by explicitly separating the levels of concurrency and locality.

A stream program exposes three different levels of locality: kernel, producer-consumer, and global. While running a kernel on an input element, temporaries are needed to store intermediate results. In general a kernel operates independently on different elements, so temporaries are local to one invocation of a kernel. This is the finest level of locality, *kernel locality*. *Producer-consumer locality* occurs between kernels in the stream graph, where the output stream of one kernel, the producer, passes to the input of another kernel, the consumer. Finally some values that need to be shared across kernels and input/output quantities sit at the global level.

A stream program also exposes several kinds of concurrency. *Instruction-level parallelism* occurs in a kernel where multiple instructions can execute together. A kernel repeating a computation over a stream of elements is *data parallel*, since several copies of the kernel can execute at once over different elements in the stream. Finally, in the stream graph there is *task parallelism* between kernels that can execute concurrently; however, not all implementations exploit this form of parallelism.

2.3.2 Stream Architectures

Chip design is a balancing act between competing factors. Die space on a chip is limited, and the available transistors must be split between three parts: control for instruction decoding, ALUs for performing arithmetic, and storage for holding values [43].

Chip manufacturing improvements continue to change how to efficiently partition a chip. As

clock speed grows faster, the speed of light in silicon limits how far information can travel in a single clock cycle. On current chips with gigahertz clock frequencies, this wire delay means that signals can take several cycles to cross the chip. This makes communication increasingly more expensive than computation. The additional density allowed by shrinking chip lithography processes also affects this cost of communication versus computation ratio. Bandwidth to external memory is related to chip perimeter, which grows more slowly than the chip area useful for computation. Due to these limitations, a single serial processor with one complex controller and single processing pipeline with a small number of ALUs cannot efficiently use available chip area, since communication time between the various parts overshadows computation [46].

Parallel Architecture Taxonomy

Placing many processors on chip is a better way to take advantage of available parallelism. As manufacturing improves, processors, memory, and interconnection networks can all be laid out in a single chip. There are many ways to arrange these parts, and **stream processors** rely on the same principles used to build larger scale parallel machines.

The most relevant design parameters when discussing stream processors are instruction/data handling and memory organization. Current examples of architectures that work well as a stream processor include the *graphics processing unit* (GPU), the Cell BE processor [17], and ClearSpeed CSX [4]. Although they are different in other respects, these processors all share a memory hierarchy that fits the different levels of locality exposed by stream programming and many processing elements for handling massive data-parallelism.

To categorize how machines run instruction streams on streams of data, Flynn defined four basic categories [18]. Ignoring the rarely used MISD, they are

- *Single instruction, multiple data stream (SIMD)*
A controller sends the same instructions to several processors acting on different data.
- *Multiple instruction, multiple data stream (MIMD)*
Each processor has a separate controller generating an instruction stream.
- *Single instruction, single data stream (SISD)*
A single instruction stream drives a single processor.

Modern architectures are often hybrids; for example, current general-purpose CPUs from Intel and AMD support SIMD vector instructions, but overall they are MIMD with a small multiple M

for multi-core CPUs, or SISD for single-core CPU. They devote significant die space to handling task and instruction-level parallelism.

Stream processors fall into the SIMD and MIMD categories, with larger multiples than CPUs for a given chip manufacturing process. They put more focus on ALUs, and less on intelligent control. The ClearSpeed CX600 is 96-way SIMD, with 96 “poly execution units”, each containing only ALUs and I/O, that follow a single instruction stream generated by one “mono execution unit”. The Cell BE processor and the latest generation GPUs, the NVIDIA Geforce 8800 and AMD HD 2900, are MIMD at the outer level and SIMD at a lower level. The Cell BE processor contains one “power processor element” and eight simpler 4-way SIMD “synergistic processor elements”, all running on different instruction streams.

A memory arrangement suitable for stream processing takes advantage of the different levels of locality. Since communication cost is a limiting factor, data should be kept as close to each processing element as appropriate. With this in mind, the stream processors above implement a three level hierarchy. Each processing element has local registers that handle kernel locality, larger buffers for passing streams between kernels, and external memory off-chip.

Affine Arithmetic Issues on GPUs

Implementing affine arithmetic on the GPU in the traditional way would require dynamically allocated sparse vectors. This poses problems with control flow and memory management.

Managing the propagation of noise symbols requires control flow. While complex control flow will have an impact on all architectures, it is particularly hard to deal with on SIMD processors. As a result, programs filled with control flow will not run efficiently on these processors. However, in an application with regular control flow, a SIMD processor can achieve higher arithmetic intensity than a purely MIMD processor of similar complexity, since it can devote more space to ALUs instead of control. For example the designers of the Imagine, one of the earlier stream processor designs, found that switching from SIMD to MIMD ALU clusters would take 1.6 times more die space, significantly reducing the number of clusters that can fit on a chip [28].

Control flow has historically been a difficult problem for SIMD processors. Since all processing elements execute the same instruction stream, it is not possible for one to take a different branch. There are several common ways to deal with this, but all add some overhead. On an if-branch, eventually all branches taken by any processor must be executed one after another by all processors. Whether processors sit partially idle or compute multiple branches and discard results, there is some overhead involved. Conditional streams reduce some of this cost by separating a

stream into multiple dense substreams; however, current techniques still fall well below available peak performance [29].

The second issue is that dynamic memory allocation within a kernel is generally too costly and is usually not supported in stream programming languages; however, the number of noise symbols required depends on the contents of a program, so it is difficult for a human to place tight limits on the amount of space required.

These are the issues that this thesis aims to solve by statically determining the noise symbols needed in *affine forms*. This sidesteps both the control flow and memory management issues of previous implementations of affine arithmetic. Our approach requires only statically allocated vectors containing just enough space to hold the *partial deviations*.

2.3.3 Stream Programming Languages

There are a number of languages based on the stream programming model. StreamC/KernelC is a C++ syntax-based stream language that targets the Imagine processor [30]. Also developed at Stanford, Brook is a more architecture-independent open source language with support currently for both GPU and CPU computation. It implements reductions, n -dimensional array streams, iterator streams, and scatter/gather operations [10]. StreaMIT [38] works with filter-style kernels joined using a fixed set of communication patterns rather than an arbitrary graph.

Finally, Sh is a C++ based stream metaprogramming language [37] that has recently developed into the RapidMind platform [1]. The original Sh forms the basis for the implementation in this thesis, so a more detailed discussion about this language and relevant internal structures continues below.

Floating Point Support

One issue to worry about is that current stream languages (and often *stream processors*) have no support for IEEE rounding modes. For now, this is understandable since in many media applications, a little imprecision is imperceptible and numerical instability is not life threatening.

This may also be an underlying problem with current hardware, which has incomplete rounding mode control and most operations have worse precision than IEEE 754 requires. In a throwback to the early days of *floating point*, this precision is undocumented and changes depending on manufacturer and hardware generation [3].

The lack of proper rounding and precision issues make it more difficult to efficiently implement

conservative range arithmetics. As stream processor usage increases, hopefully both languages and hardware will provide better floating point support. Because of this, the implementation for this thesis ignores the issue of floating point error completely. This breaks the `containment constraint`, but in many graphics applications the floating point error remains insignificant and small containment failures still give usable results.

Sh Metaprogramming Language

Although it is missing some features supported by the other languages mentioned above, Sh has several advantages.

At its core, Sh is a C++ matrix-vector library, unlike the others which are standalone languages. There is a rich set of tuple types like normals, points, colours, and texture coordinates. In immediate mode, operations using these types happen right away. In retained mode, indicated by enclosing operations between `SH_BEGIN` and `SH_END` macros, operations are stored into a `ShProgram` kernel that can take part in stream computation.

In Sh each kernel has one input and one output `ShStream`. Each `ShStream` is built by combining several `ShChannel` objects, which are sequences of a single tuple type, using the `&` operator as shown in the following code fragment:

```
ShChannel<ShVector3f> a, b;  
ShStream s = a & b;
```

Sh handles the memory allocation and manages transfers for the channels, hiding any architecture-specific complexity from the user, just as Brook and StreamIT do. There is currently no way to explicitly build a `stream graph` out of `ShProgram` and `ShStream` objects, but kernels can be invoked by binding an input stream using the `<<` operator and assigning the result to another stream using `=`. In other words, the schedule of kernel execution is specified explicitly in an imperative style. Listing 2.1 shows how this syntax comes together in an implementation of the stream graph from Figure 2.7.

There are several interesting features unique to Sh that make it a programmer-friendly stream language. Since it is a C++ library, all the data abstraction capabilities of an object-oriented language are available for building kernels. In addition, kernels are built and compiled at runtime, so it is possible to use powerful metaprogramming techniques. These include the shader algebra operators that manipulate programs as first class objects, to form new programs [36].

```

ShChannel<ShColor1f> ImageA;
ShChannel<ShColor1f> ImageB;
ShChannel<ShColor1f> ImageDiff;
ShColor1f result;

/* load images into channels */

ShProgram squared_difference = SH_BEGIN("stream") {
    ShInputColor1f a;
    ShInputColor1f b;
    ShOutputColor1f result;
    result = (a - b);
    result = result * result;
} SH_END;

ImageDiff = squared_difference << (ImageA & ImageB);

result = sum(ImageDiff); // reduction sums together elements of stream
result = sqrt(result);

```

Listing 2.1: Image difference stream program

Specifically for this thesis, one of the main advantages to using Sh is that it has a machine-independent intermediate representation (IR) and optimization framework that are both well documented in the Sh book [35].

Figure 2.8 shows the path a kernel takes from specification to execution. Internally a kernel built in retained mode is parsed into the IR, a *control flow graph* (CFG) denoted by

$$G = (N, E, \text{entry}, \text{exit}).$$

This graph consists of sequential basic blocks of code $B \in N$ as the vertices and directed edges E indicating jumps, conditional or unconditional, between basic blocks. There are also two distinguished nodes ($\text{entry}, \text{exit} \in N$) representing the beginning and end of the program. Some useful functions on graph nodes are the *predecessor function* ($\text{Pred}(B)$), the set of immediate predecessors with edges to B in the graph, and the *successor function* ($\text{Succ}(B)$), the set of immediate successors with edges from B .

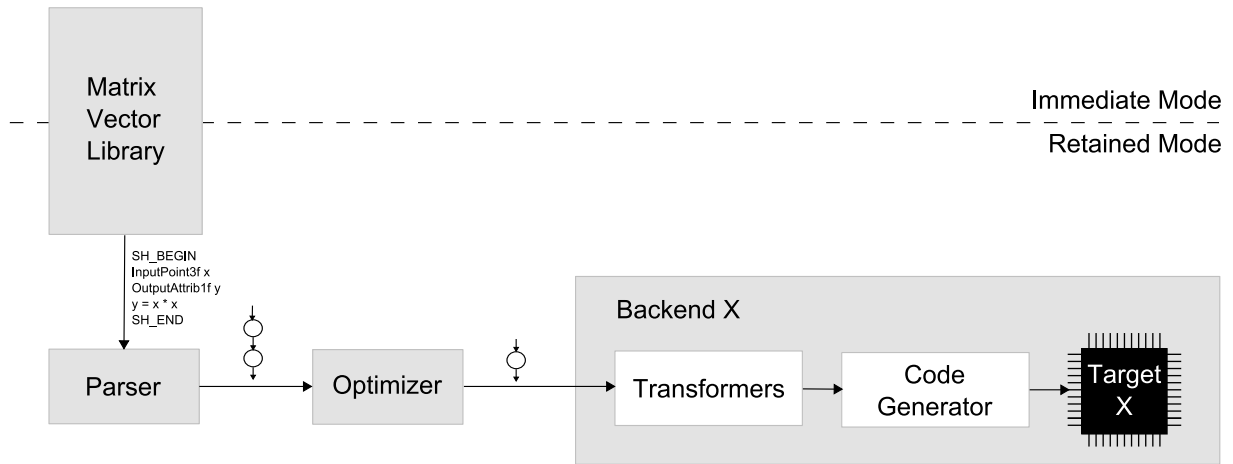


Figure 2.8: Overview of compilation in Sh

All the steps from this point on until code generation use the same IR, rearranging it and attaching additional information. The first stage after parsing is a set of general optimizations useful across all backends. These include several *data flow* analyses on the graph including building *def-use chains*, eliminating dead code, and value numbering for folding constants and lifting uniforms. After this general pass, the graph passes to the backend, which turns the IR into a form executable on a certain target.

When the CFG reaches a backend, the backend requests appropriate *transformers* that turn operations and types that a target architecture does not support into ones that it does. For example, there is a transformer that turns unsupported trigonometry functions into sequences of basic operations and one that simulates integers and fractional types using floats. After this point, there may be another pass through the optimizer (not shown) to take advantage of further optimization opportunities. Then a linear register allocator and code generator produce source code in a language suitable for the given target. In the CPU backend, for example, Sh generates C++ code and invokes the `cc` compiler. Code generation for GPUs supports GLSL and ARB vertex/fragment programs.

This framework made writing the prototype code for this thesis much more palatable. With def-use chains already in place and existing utilities for performing structural analysis, we were able to concentrate on data flow problems specific to affine arithmetic.

Several steps for handling affine arithmetic depend on *structural analysis*, so here is a quick

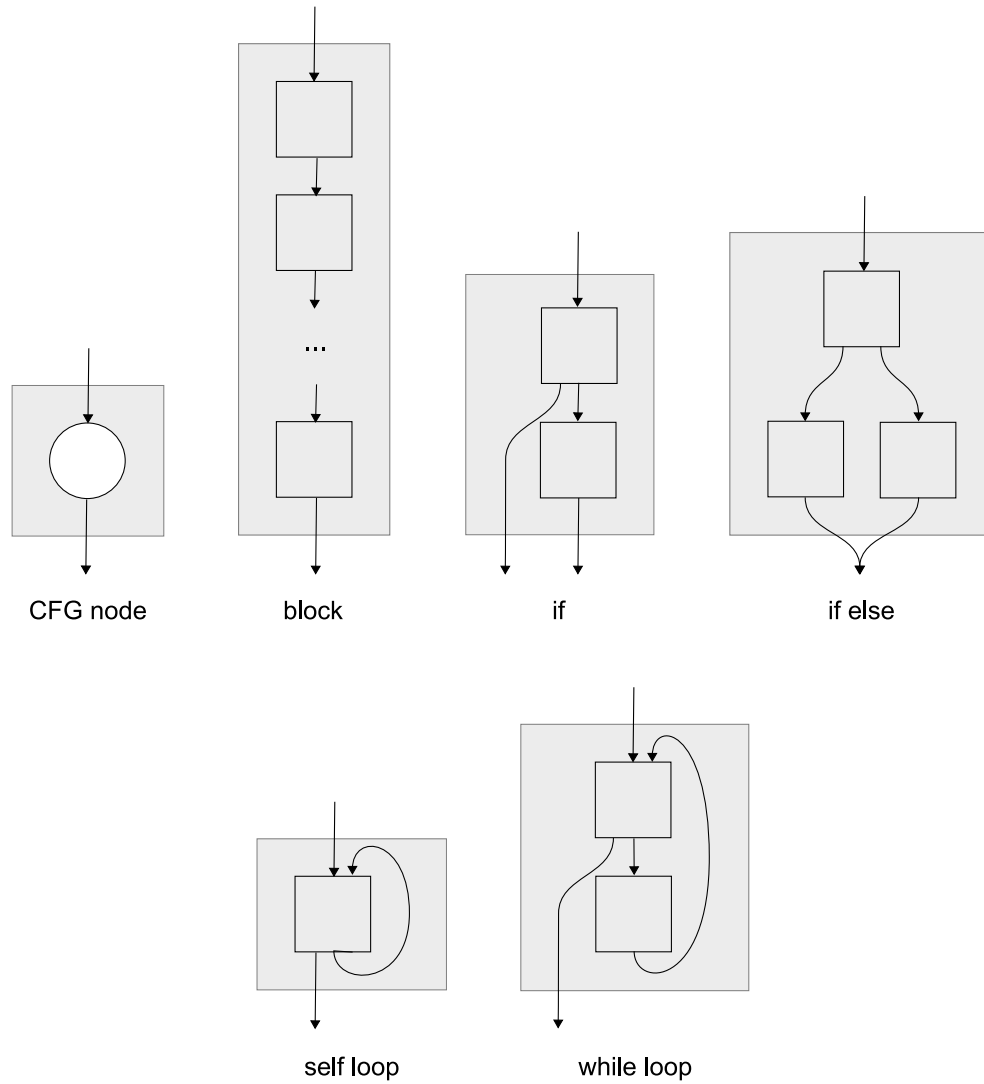


Figure 2.9: Structures extracted during structural analysis in Sh

overview of how this happens in Sh. A *structure* is a connected region in the control flow graph that matches some control flow pattern, such as an if-else branch or while loop. Structural analysis builds a hierarchy of nested structures by finding matching patterns in a bottom-up manner, starting from individual CFG nodes. The kinds of structures supported in Sh are shown in Figure 2.9. While the Sh language supports all of these control flow constructs, they are not recorded explicitly in the intermediate CFG, so structural analysis makes it possible to recover structures from the CFG later in the compilation process. Also, structures that are formed during compilation by transformations in the CFG can also be identified later on.

2.4 Previous Work

We will now survey previous work including implementations of range arithmetic, outlines for optimizing affine arithmetic, and techniques in data flow analysis.

2.4.1 Range Arithmetic Libraries and Compilers

While support for interval arithmetic is widespread, there are only a few public affine arithmetic implementations.

Libraries for standard interval arithmetic exist for most common programming languages and mathematical software packages [34]. Using operator overloading, it is easy to present an interface similar to real arithmetic, and this is the same approach we use for our immediate mode affine arithmetic implementation.

Libraries can add significant function call overhead, though, which can be removed with compiler support for interval types. Examples like CONDOR [31] automatically produce natural range extensions and transform interval computations into sequences of floating point operations. This is how we implemented the retained mode support for affine arithmetic.

Since affine arithmetic is relatively new and not as popular as interval arithmetic, there are only a few publicly available C++ implementations [20, 51]. Most researchers applying affine arithmetic have relied on one of these packages or developed their own implementations.

2.4.2 Affine Arithmetic Optimizations

The originators of affine arithmetic outlined several optimizations that form the basis for this thesis.

When all of a *range function*'s inputs are standard intervals and a program has only if-then-else branches, it is possible to statically determine all *noise symbols* in a program, and hence allocate space for all affine forms at compile time [7], an optimization called *static storage allocation*.

A later paper [52] mentions several techniques for reducing the number of noise symbols. As a computation grows longer, the final result of an affine analysis gathers more and more noise symbols, each of which is less significant on average. The process of taking m noise symbols and replacing them in the k variables by a smaller set of n noise symbols is called *condensation*. Some correlation information may be lost, but reducing the amount of computation at the cost of increasing the *wrapping effect* can be beneficial up to a certain point. For an algorithm like branch-and-bound, condensation can speed up each range function evaluation. This is offset by the cost of having to try more input ranges, as the wrapping effect makes less information available for deciding what to filter. When a loop occurs within a range function, condensation may also be necessary to keep space usage reasonable.

Schwarz et al. gave an algorithm for the $k = 2$ and $n = 2$ case, which reduces to the problem of finding the minimal area enclosing parallelogram. This can be done in $O(m \log m)$ time by a rotating caliper method [47]. The $k > 2$ case remains a difficult problem.

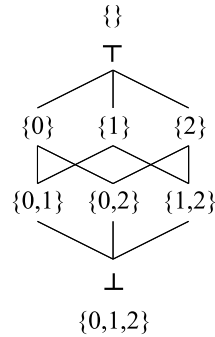
When the m selected noise symbols have partial deviations with relatively small magnitudes, the merging operation will have little impact on the results, unless larger *partial deviations* eventually cancel out.

In one case where noise symbols are redundant, condensation makes no difference to the results other than *floating point* rounding errors. During execution, if several noise symbols appear in only one live affine form, then they can be combined into a single noise symbol without any loss of information [52]. For the rest of this thesis, we will refer to this as *unique condensation*.

In this thesis we deal with static storage allocation and unique condensation as *data flow* problems. The reduction in control flow helps *stream processors* approach peak performance, and the elimination of redundant symbols makes the computation more effective. We will ignore the more general condensation methods, and instead present an attempt at condensing using program structure that fits well into the framework of static analysis.

2.4.3 Data Flow Analysis

Data flow analysis is used to discover information about the variables in a program. This information is propagated through the program to give a conservative idea about how data behaves during execution. Here are a few examples of common data flow analyses: *constant propagation*

Figure 2.10: Lattice diagram of \mathbb{Z}_3

discovers when variables hold constant values; *dead-code elimination* checks for code that is never executed; and *live variable analysis* finds variables that are live, meaning the values they hold might be needed later in the program. Muchnik [42] and Aho et al. [5] provide comprehensive surveys of the data flow framework and its applications.

In the usual formulation, information is represented by values in a *lattice* L , a partially ordered set with two associative and commutative operators: meet \sqcap and join \sqcup .

For every $x, y \in L$ there must be a unique greatest lower bound $z = x \sqcap y$ and least upper bound $z = x \sqcup y$. There are also two special values, top \top and bottom \perp , where $\forall x \in L$

$$\top \sqcup x = x$$

$$\perp \sqcap x = x.$$

The symbol \top represents a complete lack of information and \perp is best described as an overabundance of information.

An example of a lattice in Figure 2.10 is the power set of \mathbb{Z}_3 , which can be used to represent the subsets of three noise symbols occurring in a set of affine forms. In this diagram, the meet of two elements occurs by following lines leading down from the elements until they intersect. Similarly the join can be found by following lines upwards. The height of a lattice is the maximum path length from top to bottom. The power set of any set can form a lattice by using set union for \sqcup and intersection for \sqcap .

In a data flow framework, a set of lattice values V represents information available at various points in a program. A set of transfer functions, F , propagates information by transforming elements in V , representing the data flow from executing some part of the program.

A typical arrangement when propagating along control flow edges is to have a lattice value $\text{in}(B)$ represent the state at the entry to CFG node B . Then a transfer function $f_B \in F$ computes the state when exiting the node

$$\text{out}(B) = f_B(\text{in}(B)).$$

A transfer function also propagates information along control flow edges by using the exit states of predecessor:

$$\text{in}(B) = \bigcap_{A \in \text{Pred}(B)} \text{out}(A).$$

The basic *iterative data flow analysis* algorithm [33] follows a worklist approach as shown in the following pseudocode. This algorithm starts with only known information typically at the entry node, and then propagates that information through the rest of the graph.

```

initialize in(B) =  $\top$ 
initialize in(entry)
initialize out(B) using propagation
W = entry
while W is not empty
  remove B from W
  compute out(B) =  $f_B(\text{in}(B))$ 
  if out(B) changes
    propagate to in(A) for each  $A \in \text{Succ}(B)$ 
    if in(A) changes, add A to W

```

Without the worklist, a variation of this algorithm exists that repeatedly traverses the CFG in depth-first order from entry until there are no further changes. When a lattice has a finite height, and the transfer functions are monotonic, then the iterative method above will terminate with a minimum fixed point solution.

Other methods for doing data flow also exist, differing mostly in the placement of lattice values and consequently the associated transfer functions. Variants that reduce the CFG to a nested hierarchy first, like Allen-Cocke interval analysis [6] can be more efficient than the iterative method; however, they are more difficult to implement and do not work as well for backward flow analyses. Backward analysis with an iterative method is easy, simply by flipping the algorithm, starting the worklist from exit and propagating along edges to predecessors. For these reasons,

we stay with iterative methods for the implementation in this thesis, even though they may not be state of the art.

An example of the transfer functions used for live variable analysis [5] are shown below:

$$\begin{aligned} \text{use}(B) &= \text{variables used in } B, \\ \text{def}(B) &= \text{variables (re)defined in } B, \\ \text{out}(B) &= \cup_{S \in \text{Succ}(B)} \text{in}(S), \\ \text{in}(B) &= \text{use}(B) \cup (\text{out}(B) - \text{def}(B)). \end{aligned}$$

One difficulty with data flow analysis within the Sh intermediate representation is that all values are tuples. While recent shading language extensions (and the RapidMind platform) support run-time indices for tuples, these make data flow analysis more difficult, and we do not yet handle this. With indices that are compile-time constants, it is possible to track data flow information per tuple element instead of per variable. The Sh optimizer does this by building *def-use chains* using fieldwise value tracking. These chains link where a the value for a variable is defined, to where that value might be used later in a program.

2.5 Contributions

In the context of previous work, here are the main contributions of this thesis. While we present no huge leaps compared to existing research in [affine arithmetic](#) or [data flow analysis](#), the novel aspect of this thesis is how we combine the two to allow affine arithmetic to run more efficiently.

First, we show how to use static analysis to turn dynamically managed partial deviations in [affine forms](#) into fixed-length vectors. Then we extend the static analysis to improve efficiency by [condensing](#) away unneeded noise symbols from the affine forms. This second stage goes beyond the kinds transformations available to a general-purpose code optimizer, since it reduces the number of outputs from a function, without changing the meaning of the output affine forms.

Finally, we show how well the analysis and condensation works on several sample graphics applications, using intermediate results from the data flow analysis to show how condensation affects the code size and noise symbol distribution.

Chapter 3

Static Analysis of Noise Symbols

This chapter starts with the [data flow](#) analysis used to statically assign and then condense noise symbols. Details about the implementation particulars in [Sh](#) follow.

It is only possible to statically determine all the noise symbols under certain conditions. We deal with a subset of these, where noise symbols are known for all program inputs. This means they must either be intervals or an [affine form](#) with user specified noise symbols. The first case occurs often when initial values are numerical ranges, while the second case happens with chains of [kernels](#) that pass results from one to another. For example, the [branch-and-bound](#) algorithm uses interval inputs.

3.1 Data Flow Analysis of Noise Symbols

The goal of the data flow analysis is to find what noise symbols each operation needs in its result variables. The [lattice](#) and transfer functions for solving the static storage allocation problem are relatively straightforward. Later optimizations that shorten the affine forms will build on this information.

3.1.1 Lattice for AA

Ignoring loops for now, the maximum length path through the program from entry to exit is finite. Since each operation generates some constant number of new noise symbols, this means that there is also some upper bound, T , on the total number of noise symbols used. Loops are

handled by the [hierarchical condensation](#) technique described later in subsection 3.2.2 that keeps the number of noise symbols finite.

Given some value T , the affine form’s noise symbol indices can be taken from \mathbb{Z}_T , so the most convenient lattice to use to represent all live noise symbol indices is the subsets of \mathbb{Z}_T , $L = 2^{\mathbb{Z}_T}$. For tuples, it also makes sense to track each element separately, since in a sequence of componentwise operations each component may have different noise symbols. Thus for an n -tuple, the lattice value is in L^n , and the [meet](#) and [join](#) operations are defined componentwise.

At the start, we can be optimistic and set all lattice values to the empty set and initialize based on the input lattices. Since the lattice is finite, as long as the transfer functions are monotonic, the iterative data flow process will terminate with a solution.

3.1.2 Static Storage Allocation

In the data flow analysis for static storage allocation, symbols propagate along [def-use chains](#), and the transfer functions encode how source symbols are mapped to the destination in each operation. Operations fall into three cases depending on what kind of affine approximation fits best.

Let N be the noise symbols present in any input variable for a statement S . Then the transfer function that gives the noise symbols in the destination variable of S is one of the following:

- $h_{\text{keep}}(S) = N$, when S is an [affine operation](#)
- $h_{\text{approx}}(S) = N \cup \{t_i\}$, when S is approximated by an affine function
- $h_{\text{discard}}(S) = \{t_i\}$, in special cases

Affine functions keep the existing noise symbols; non-affine functions may need one or more new noise symbols t_i to represent [approximation error](#); and in special cases, it may make sense to discard all symbols. For instance, one special function is

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases} .$$

The best possible approximation here is always a constant, either $-1, 0$, or 1 , with the approximation error also a constant.

Because the transfer functions above are monotonic and associative, the data flow computation is guaranteed to find a solution in finite time [27].

To see the transfer functions in action, consider the multiplication

$$\hat{\mathbf{z}} = \hat{\mathbf{x}}\hat{\mathbf{y}},$$

with $\hat{\mathbf{x}} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2$ and $\hat{\mathbf{y}} = y_0 + y_1\varepsilon_1 + y_3\varepsilon_3$.

In this case, the noise symbols in the two inputs are $N = \{\varepsilon_1, \varepsilon_2, \varepsilon_3\}$. Since multiplication is not affine, it uses the $h_{\text{approx}}(S)$ transfer function, and the noise symbols in $\hat{\mathbf{z}}$ will be

$$N \cup \{\varepsilon_4\} = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4\},$$

where ε_4 is a new noise symbol.

As a larger example of how the analysis works across a program, consider the following function f_{\circ} , where the implicit equation for $f_{\circ} = 0$ is a circle with radius r

```
ShInput1f x, y, r;
ShOutput1f f○;
f○ = x * x + y * y - r * r;
```

As shown in Table 3.1, the input variables are given noise symbols, with one independent noise symbol per input. Then the noise symbols propagate through each operation, with affine functions like addition passing through noise symbols and non-affine functions like multiplication adding new noise symbols.

3.2 Merging Symbols

In long affine computations, the number of noise symbols will increase as the number of operations increase, so pruning away redundant or insignificant symbols becomes important.

The general method of *condensing* symbols is to pick a point in the program, choose n noise symbols, and use $k < n$ new noise symbols to represent the uncertainty in the original n . Then if the original n noise symbols appear in m live variables, each one of these variables must be updated to use the new symbols.

Some condensation methods depend on peeking at the actual values of the *partial deviations*, such as merging those with small magnitudes. This is difficult in a static context, so the techniques explored here make merge decisions only on information available during static analysis: the noise symbols in each variable and where they appear in the program.

		Noise Symbols					
		ε_1	ε_2	ε_3	ε_4	ε_5	ε_6
Inputs	x	new symbol					
	y		new symbol				
	r			new symbol			
Body	$x2 = x * x$	existing symbol			new symbol		
	$y2 = y * y$		existing symbol			new symbol	
	$r2 = r * r$			existing symbol			new symbol
	$xy = x2 + y2$	existing symbol			existing symbol		
	$f = xy - r2$	existing symbol			existing symbol		
Output	f_{\circ}	existing symbol					

Table 3.1: Static analysis of f_{circle} showing the noise symbols for the result of each operation

The first technique, merging of unique symbols, handles the case when all n noise symbols appear only in one live variable, or $m = 1$. Then all n noise symbols can be replaced by a single new noise symbol without changing the joint range of all variables.

The second technique uses a hierarchical structural analysis. Any symbols generated in a structure in the control flow graph are condensed as execution leaves that structure. This is used as a simple solution to ensure loop bodies produce a fixed number of symbols. In this case, there are different ways to pick the k new noise symbols. We use the simplest alternative: m new symbols, one for each live variable leaving the structure.

Some symbols may have a special meaning, and it is possible to mark and ignore these during condensation. For example, noise symbols given in the inputs may be used by an external agent to determine the correlation between the outputs and inputs, so these symbols should not be considered for elimination.

3.2.1 Unique Condensation

Noise symbols that appear in only one live tuple element represent a source of uncertainty that is irrelevant elsewhere. If several symbols like this appear in only one affine form, they can be merged together. In a geometric sense, this is special case of combining collinear generating vectors of the joint range's zonotope.

To pick candidates for this kind of merging, we need to find out which noise symbols are always unique at a point in the program, no matter what execution path. Then we can insert merges at such safe points and propagate the new arrangement of noise symbols. This process can be broken down into two parts: finding unique symbols and merging symbols.

Finding Unique Symbols

Since variables hold the noise symbols, finding out the set of live variable definitions gives the live noise symbol information we need. This is a small variation on the classic data flow approach for finding live variables. It can be solved using a lattice that represents the set of live definitions at each statement and the following transfer functions:

$$\begin{aligned}
 \text{use}(S) &= \text{variables used in } S, \\
 \text{def}(S) &= S.\text{dest} - \text{use}(S), \\
 \text{out}(S) &= \text{rchout}(S) \cap \bigcup_{X \in \text{Succ}(S)} \text{in}(X), \\
 \text{in}(S) &= \text{use}(S) \cup (\text{out}(S) - \text{def}(S)).
 \end{aligned}$$

The only change from the usual live variable analysis is the intersection with $\text{rchout}(S)$, the set of definitions that exist after S . This helps when there is a path in the CFG to some successor, X , that does not go through S . In that case, some definitions in $\text{in}(X)$ may not be in $\text{rchout}(S)$.

Then after each operation in the program, counting how many live variables a noise symbol appears in reveals the unique symbols. Without using static single assignment or SSA form, though, a live variable v may be holding one of several definitions, denoted by the set $\text{defs}(v)$. The count for how often a noise symbol n appears in a variable v is then

$$\text{count}(v, n) = \begin{cases} 1 & \text{if } n \in \text{syms}(d) \text{ for some } d \in \text{defs}(v) \\ 0 & \text{otherwise} \end{cases},$$

and the overall count for all live variables is the sum

$$\text{count}(S, n) = \sum_{\{v:\text{defs}(v)\cap\text{out}(S)\neq\emptyset\}} \text{count}(v, n).$$

When this count is 1 for n , then it means ε_n must be unique after statement S no matter what subset of the definitions in $\text{out}(S)$ are actually live during a particular execution of the program. So we can let

$$\text{unique}(S) = \{n \mid \text{count}(S, n) = 1\}.$$

Merging Unique Symbols

Now that we know where certain symbols become unique, they can be merged together.

Since merging involves only summing together the unique symbols to give a new symbol, the cost is low. It makes sense to merge whenever there is more than one unique symbol in a variable.

Ideally, a merge algorithm that does not add any new variable definitions would work best, since any new definitions would require rebuilding some [def-use chains](#). In fact, if there are no operations with a discarding transfer function (see [subsection 3.1.2](#)), it is possible to show that unique symbols will only appear in the destination variable of an operation.

This is true because noise symbols become unique when previously live definitions die, and definitions die upon their last use as a source variable in an operation.

Unless the operation throws away noise symbols, symbols that become unique as a result of the dead source variables will also appear in the destination variable.

Since operations that discard all symbols are usually rare, it is reasonable to make the simplifying assumption that merging unique variables only affects noise symbols in the destination of an operation. While some unique symbols may slip through at discarding operations, the moment such missed symbols are involved in a non-discarding operation, they will be caught and merged.

Here are the changes to the transfer functions required for handling merges. While originally, only a single lattice value `dest` represented all the noise symbols present in the destination variable of an operation, now we add on two more sets `rep` and `unique`, and define a new lattice

$$\text{mergeDest} = (\text{dest} - \text{unique}) \cup \text{rep}.$$

In other words, `dest` is still computed oblivious of any merging, and then to produce the lattice value after merging, we find the set of unique symbols to merge and choose $\text{rep} \in \text{unique}$ as the

		Noise Symbols					
		ϵ_1	ϵ_2	ϵ_3	ϵ_4	ϵ_5	ϵ_6
Inputs	x	new symbol					
	y		new symbol				
	r			new symbol			
Body	$x2 = x * x$	existing symbol			new symbol		
	$y2 = y * y$		existing symbol			new symbol	
	$r2 = r * r$			existing symbol			new symbol
	$xy = x2 + y2$				merge representative	merged symbol	
	$f_{\circ} = xy - r2$						merged symbol
Output	f_{\circ}						existing symbol

Table 3.2: Static analysis of f_{circle} showing the noise symbols after unique condensation, respecting input symbols.

single representative noise symbol. The only other change is that other transfer functions need to be retooled to use `mergeDest` instead of `dest`.

When there are loops involved, as more unique symbols are merged, the unique set may change and it could be that $\text{rep} \notin \text{unique}$. However, even if this happens, the representative can remain the same. Since it is unique, one name is as good as any other unused name, and it can be thought of as recycling an old unused name for a new symbol.

Similarly, other merges earlier in the program may reduce the unique set to a single element. In this special case, merging is no longer necessary and both `unique` and `rep` can be safely cleared.

Going back to the f_{\circ} circle example, Table 3.2 shows the result after unique condensation, where input symbols are never involved in any merging. There is some improvement in the number of noise symbols near the end; however, this example is too short to show the benefits possible on larger, more typical cases with more non-affine operations.

3.2.2 Hierarchical Condensation

This second condensation technique merges symbols generated in a single structure. We will call this *hierarchical condensation* since it starts from the nested hierarchy of structures given by structural analysis. This condensation method is used currently only as a way to keep the number of noise symbols in a loop finite. Otherwise each loop iteration would add extra noise symbols and static analysis would never terminate.

While this technique can work on structures other than loops, it is too crude to apply in general for now. To see why, consider how such a merge happens. Using the same form introduced at the start of the section, merging involves replacing a zonotope generated by n vectors by another bounding zonotope generated by k vectors. All of this happens in a space with m dimensions.

Suppose a structure adds n new symbols, and these appear in m variables upon leaving the structure. The joint range for the n noise symbols in m variables is a m -dimensional zonotope.

Picking k new noise symbols to replace the n is then a matter of making a new m -dimensional zonotope. This new zonotope must include the existing one while keeping the wrapping effect small. There is no known method to do this efficiently, except when m is small (e.g. finding the least bounding parallelogram for $m = 2$).

Instead, we apply the simple method of using $k = m$ new noise symbols, one for each variable. Unfortunately, with no shared noise symbols, this loses all correlations from the original n and gives a m -dimensional box for the resulting zonotope.

Adding support for this form of condensation requires only a few changes to the program right before static analysis. Our approach depends on a structural analysis preprocess, which builds a hierarchical tree of the loop structures. Noise symbols are then tagged with the depth in the tree where they were generated.

1. Detect the m variables that are both written to inside a structure and still alive upon leaving the structure.
2. Add one ESCJOIN operation per variable as a placeholder
3. Compute nesting depth for each structure
4. Tag symbols with the nesting depth where they were born
5. Use a special transfer function for ESCJOIN operations that combines noise symbols nested more deeply than the current structure

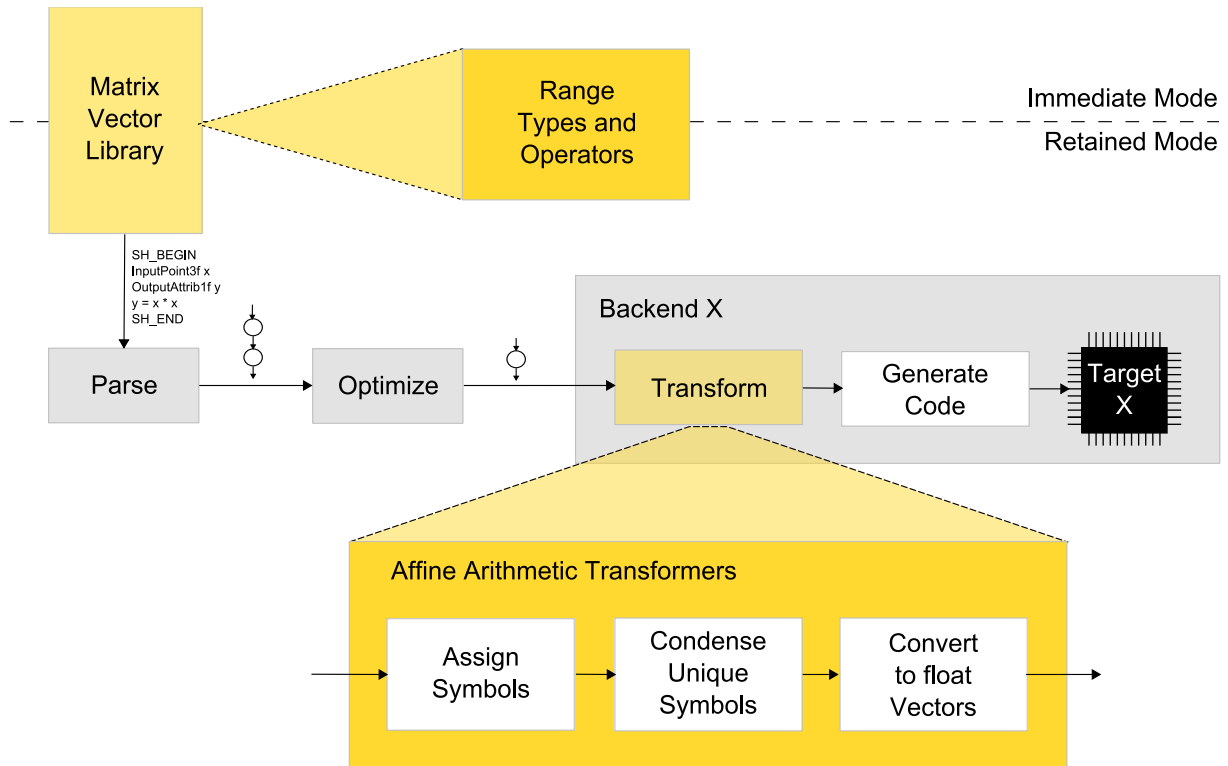


Figure 3.1: Adding support for affine arithmetic to Sh.

3.3 Implementation

The analyses discussed so far can benefit an implementer of [affine arithmetic](#) targeting any language or architecture. We will focus now on our particular implementation mapping [range arithmetic](#) computation to stream processors using the Sh system as a foundation.

Adding range arithmetic types to Sh required two main changes, shown in [Figure 3.1](#): new types and operations in the matrix vector interface to represent ranges and [affine forms](#) and additional [transformers](#) for turning affine forms into regular floating point operations during compilation.

3.3.1 Additional types and operators

Support for range arithmetic starts by updating the types and operators at interface library level. This allows range types to work anywhere existing types do, in both immediate and retained modes. For example, code that originally used a regular float 3-vector `ShVector3f` can use instead a `ShVector3a_f` for affine arithmetic using floating point partial deviations.

Few changes were required for immediate mode, since Sh vectors and matrices are wrappers around C++ types. Thus templated C++ classes were added for both interval and affine arithmetic, where the affine arithmetic class dynamically generates new noise symbols as needed.

Retained mode support was slightly more involved. Internally, Sh must be aware of how a new type relates to existing types and how the type is encoded in memory for streams and textures. To handle these behaviours, Sh was updated to allow developers to specify what implicit conversions to use in addition to existing standard C++ conversions and how to perform those casting operations.

In addition, several new functions were added to the library that apply only to range types; for example, operations like `width`, `midpoint`, and construction from a pair of bounding values. One unusual operator that proved useful was

$$\text{errfrom}(\hat{\mathbf{x}}, \hat{\mathbf{y}}) = 0 + \sum_{i, \varepsilon_i \in S} x_i \varepsilon_i,$$

where $S = \text{syms}(\hat{\mathbf{x}}) \cup \text{syms}(\hat{\mathbf{y}})$ and $\text{syms}(\hat{\mathbf{x}})$ is the set of noise symbols appearing in the affine form $\hat{\mathbf{x}}$. This finds the sensitivity of $\hat{\mathbf{x}}$ relative to $\hat{\mathbf{y}}$, or in other words, gives the uncertainty in $\hat{\mathbf{x}}$ that traces its lineage back to $\hat{\mathbf{y}}$.

Finally methods were added for building range extensions of `ShProgram` objects. In the intermediate representation, the types recorded in the sequence of statements for the `ShProgram` are replaced with the corresponding interval or affine type. Dealing with a program that has recorded operations using these types in retained mode is discussed in the next section.

3.3.2 From affine forms to float vectors

The next stage describes how variables tagged as affine forms in a program are turned into fixed-length float vectors that GPUs and stream processors can easily operate on. There are several stages to the conversion, including the static analyses presented earlier in this chapter.

Control-flow handling

Control constructs need to be altered, since if a condition variable is an affine form, it may include both true and false in its range. As a result, execution may need to run through both sides of a branch.

This happens, for example, when $\hat{x} = \varepsilon_1 = [-1, 1]$ on the following if-branch

```
ShAttrib1f foo;
SH_IF( $\hat{x} > 0$ ) {
    foo = 1;
} SH_ELSE {
    foo = 2;
} SH_ENDIF;
```

Fixing this in general requires modifying the control flow graph. The transformation used is a variation of the one used by Heidrich et al. [24]. Although in the implementation the following transformations happen directly on the control flow graph after structural analysis, the examples here show the transformations using Sh pseudocode for clarity.

```
SH_IF(cond) {
    then-block; // some statements in here
} SH_ELSE {
    else-block; // a few statements here, or empty
} SH_ENDIF;
```

After detecting and removing the affine form as a condition variable *cond*, the result looks like the following:

```
ShAttrib1f condMaybeTrue = hi(cond);
ShAttrib1f condMaybeFalse = lo(cond);

savedW = W; // (A)
SH_IF(condMaybeTrue) {
    then-block;
}

SH_IF(condMaybeFalse) {
```

```

    swap(W, savedW); // (B)
    else-block;
}

// (C)

SH_IF(condMaybeTrue && condMaybeFalse) {
    W = join(W, savedW)
}

```

Here W represents all variables written to in either the `then-block` `else-block`. Since the `else-block` should not be affected by any changes to variables in the `then-block`, the original values for W are saved at point (A) and restored at (B).

If only one branch executes, then at (C), everything is fine since the result already sits in W ; however, if the condition may be both true or false, then a `join` needs to be performed as a final step to place the union of both results into W .

This transformation for handling branches is similar to how some SIMD architectures emulate branching. Both branches are executed one after another, and a mask selects which result to keep.

Static Analysis

The static analysis step includes both static storage allocation and condensation, and happens in two parts. First one pass of static storage allocation is run to completion, and then another pass happens with `unique condensation` enabled. Only after the first pass completes can we be sure that symbols that appear unique are truly unique.

The implementation of the `iterative data flow` analysis uses bit-vectors to store lattices and a worklist approach for keeping track of statements that still need to be updated:

```

worklist = all  $h_{\text{approx}}$  and  $h_{\text{discard}}$  statements
while worklist is not empty:
    remove stmt from worklist
    compute transfer function for stmt
    if stmt.dest lattice changes:
        add all statements using the stmt result to worklist

```

During the unique condensation phase, live definitions and $\text{count}(S, n)$ are found for each statement S . Unlike the noise symbol allocations, which remain useful later for converting the affine forms into regular floating point values, these counts are only used once for finding unique symbols.

As a result, we used the more space-efficient method of storing lattices for live definitions only at the granularity of blocks of statements, and recomputing intermediate live definitions and counts on the fly.

Any unique symbols found are added to the unique lattice for a statement, and the above worklist propagation is re-computed for just that statement, using the extended transfer function with `unique`, `rep` and `mergeDest`.

```

findUnique(block):
  initialize count using live definitions in( $B$ ) (i.e. start of block)
  for each stmt in block:
    increase count for the new definition stmt.dest
    decrease count for dead definitions
    if |unique(stmt) > 1| :
      perform merge
      worklist = stmt
      propagate using extended transfer function

```

Operator Replacement

Once we know the sets of noise symbols per variable, the final step is to turn the affine forms and computations into a format that the target architecture can handle. Since no machines support affine arithmetic directly, in practice this means converting back to standard floating point arithmetic.

In *SIMD within a register* (SWAR) architectures like GPUs and Cell BE, where registers are float vectors, the range extension of a program contains vectors of affine forms. There are different ways in which the resulting centers and partial deviation values can be partitioned once again into fixed length float vectors, and the right splitting strategy depends on the target architecture.

Some of the most common operations in affine arithmetic are finding the `width` and adding a multiple of one affine form to another. Since these are supported efficiently on GPUs by dot products and hardware-supported swizzling, the best layout in this case is to group all the partial

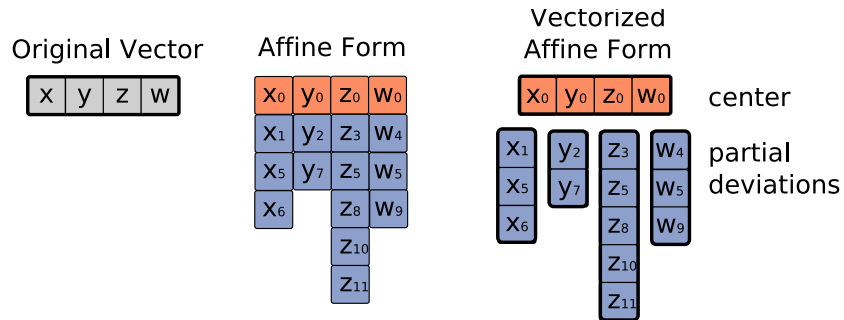


Figure 3.2: Vectorizing affine forms

deviations from an affine form into one vector. (Figure 3.2).

Since large programs can produce long sequences of partial deviations, the resulting vector might exceed the maximum supported vector size, which is typically four in current GPUs. Such long vectors are strip-mined into several shorter vectors.

However, the current trend in GPUs is moving towards scalar ALUs, so there is often little performance difference between running one operation on a 4-vector and repeating the same operation on four scalars [3]. In such GPUs, like the NVIDIA Geforce 8800, the only improvement from generating vector code is keeping code size small. However, vectorization does expose parallelism that might be useful on VLIW GPUs like the ATI 2900 architecture and on other SWAR architectures like x86 CPUs with SSE.

As a final step each statement in the program using affine arithmetic is replaced by a new sequence of statements that implements the affine arithmetic operation using the float vectors.

We have now covered the entire process starting from extensions that allows an Sh user to program explicitly in range arithmetic, through static analysis with condensation, to code generation for floating point vectors on the GPU. The next section continues with applications and performance numbers that test this implementation.

Chapter 4

Applications

To demonstrate the performance of the [static analysis](#) and [condensation](#) techniques, we look at two applications: an [implicit surface raytracer](#), and a [parametric surface toolkit](#). While the focus will be on computer graphics, each application contains more widely applicable numerical nuggets: [root finding](#) and [constrained global optimization](#).

While the examples have similar foundations, each covers slightly different ground. Both use variations of the [branch-and-bound](#) algorithm and can handle arbitrary functions. The implicit raytracer is an interactive application aimed at shorter functions and runs the entire branch-and-bound loop as a single-pass GPU shader. Looking at simple functions makes it easier to tease out the factors affecting performance.

The global optimization example, on the other hand, is a stream program that depends on cooperation between the host CPU and the GPU, with the GPU handling all the [affine arithmetic](#) computation on each iteration. This example uses affine arithmetic to analyze more complicated functions with loops that require [hierarchical condensation](#).

All of the experiments in this section were done on an NVIDIA Geforce 8800 GTS 320MB (Linux x64 driver version 100.14.19) plugged into a 2.4Ghz Intel Core 2 Duo PC with 3GB RAM.

4.1 Performance Evaluation

The goals are to show that affine arithmetic works well on GPUs and to understand how [unique condensation](#) improves efficiency.

The main metric for evaluating performance is the total execution time needed to find a result

with a given accuracy. In addition, *GIPS* (billions of scalar instructions per second), will be used to give a rough idea of how well-suited affine arithmetic is for the GPU by comparing to peak GIPS. For a given algorithm, these two measures are related by

$$\text{execution time} = \frac{\text{instruction count} \times 10^9}{\text{GIPS}}.$$

It is important when comparing different algorithms for arriving at the same result to focus on execution time. GIPS alone does not show the full benefit, since changes in the algorithm, like using unique condensation, can give results to the same accuracy using fewer instructions; however, each instruction is doing more work than before.

To better understand how unique condensation affects execution time, there will be an additional breakdown into the two main factors that unique condensation changes: the instruction count and the number of temporary registers. Reducing either of these is generally a good thing on most hardware; however, the size of the effect depends on the architecture. In particular, here is a quick overview of what happens on the 8800 that will help to explain results in the rest of the applications section.

Programs consisting entirely of MAD instructions (multiply-add, or $a = b * c + d$) were used to test the 8800. As shown in Figure 4.1, execution time increases linearly with instruction count when keeping the number of temporary registers the same. In other words, when all the instructions are MADs, GIPS remains relatively constant as code size changes. Most other operations have a similar GIPS to MAD [3], so this constant relationship to code size holds in general even with a mix of different instructions.

The relationship between GIPS and number of temporaries is less straightforward and depends on the hardware and driver implementation. Modern GPUs typically keep hundreds of threads (copies of a kernel) in flight at a time to hide latency from any single thread. All the threads share some fixed number of registers, though, so if each thread needs more registers, the GPU scheduler can keep fewer threads in flight without resorting to slower memory. As the number of required registers increases even more, values may need to spill into memory for a program to run at all [8]

Figure 4.2 is a graph of these effects on the 8800. When using up to 128 scalar registers, the number of threads in flight decreases in steps, at roughly power of two boundaries. Above 128 registers, GIPS decreases rapidly as values begin to spill into memory and levels out once most values live in memory.

One side note about performance is that we made no attempt at improving compilation

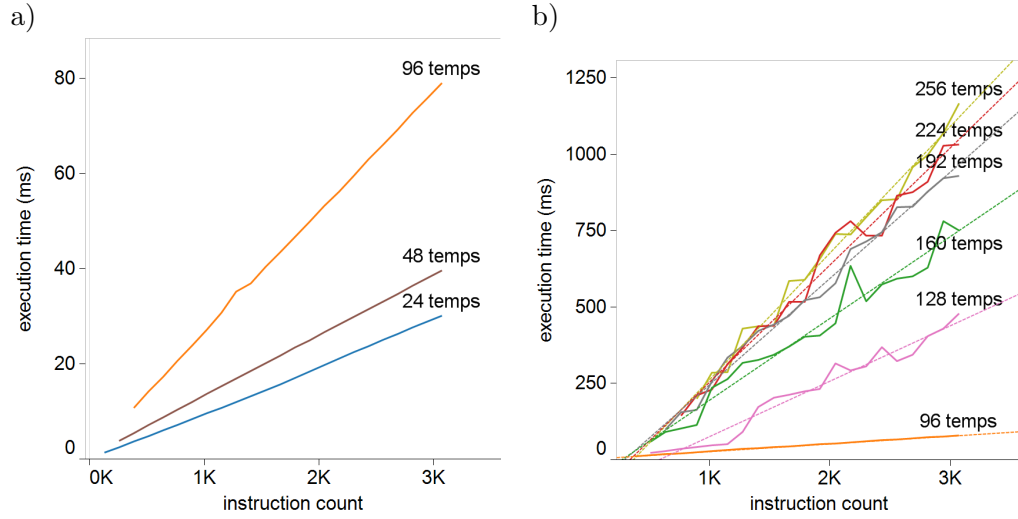


Figure 4.1: Execution time versus number of instructions on an 8800GTS for different fixed numbers of temporary registers. a) 96 registers and below b) 96 registers and above

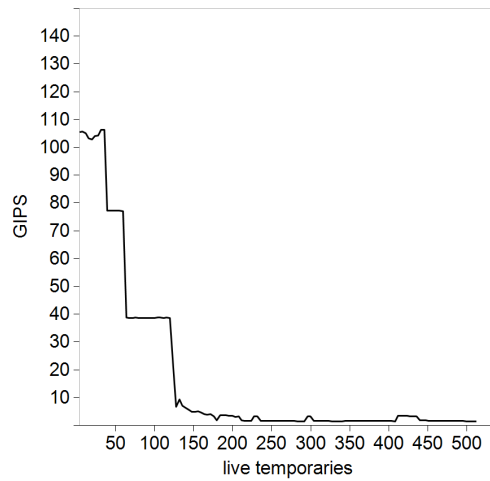


Figure 4.2: GIPS versus number of temporaries for a program with 2048 MAD instructions

time. In fact, as the control flow graph makes its way through the backend `transformers`, many steps like building use-def chains and `structural analysis` are repeated several times for the sake of simple, correct implementation over compilation speed. Stream and shader programs are typically compiled once and used many times, so compilation speed is not a pressing issue. In any case, all the examples below take from about one second to around a dozen seconds to compile.

4.2 Raytracing Implicit Surfaces

Implicit surfaces, defined by the level set of a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}^1$, are commonly used for shape modeling and displaying scientific and medical data. Raytracing is one way to render these surfaces. It produces an image by casting rays from a viewpoint and finding the first intersection with the level surface (Figure 4.3). This corresponds to finding the first root of f , parametrized by distance along the ray.

There are many ways to find the roots, and we will use an `affine arithmetic` version of the bisection method [12], which is relatively simple and guarantees finding the first root.

```
// find first root of f in range x
firstRoot(f, x):
  push x on stack

  while stack is not empty:
    pop y from stack
    ẑ = f(y) // (A)

    if ẑ contains 0:
      if wid(y) < ε: // (B)
        return mid(y)
      push [mid(y), ȳ] on stack // (C)
      push [y, mid(y)] on stack
  return no root
```

This algorithm starts with a `interval arithmetic` range x over which we want to find a root, and repeatedly splits this into smaller sub-ranges. For each range, the `affine arithmetic range extension` of f is computed to determine whether that range may contain a zero. Ranges that

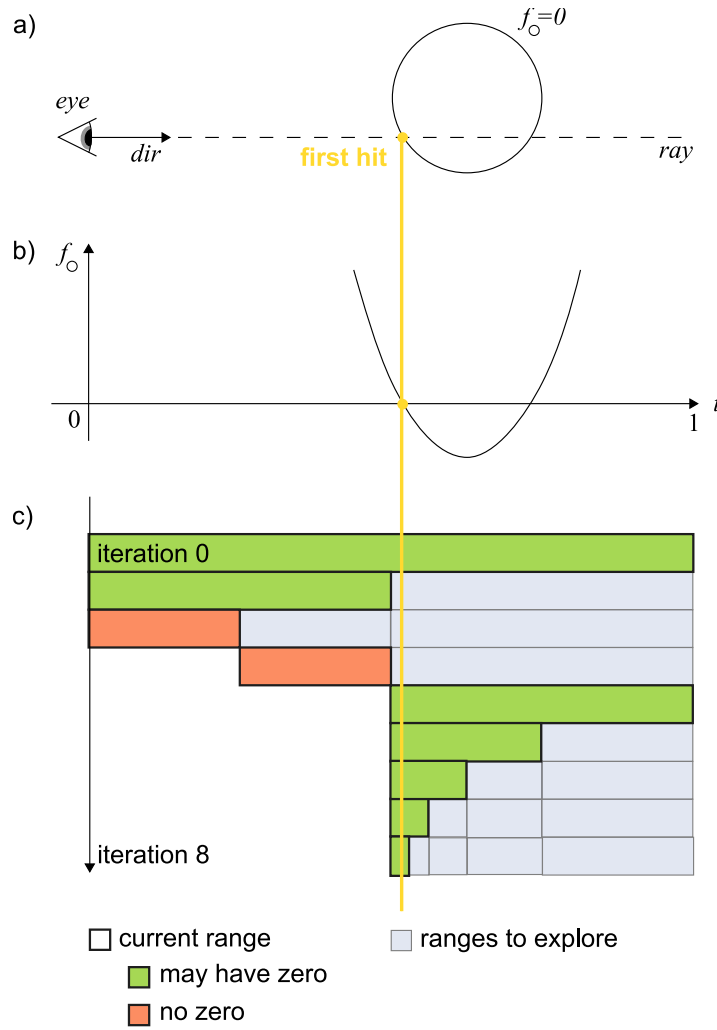


Figure 4.3: Raytracing the implicit function f_O , with $ray = eye + t \cdot dir$ involves finding (b) the first root of function f_O along the ray. Starting from the top to (c) shows a possible sequence of intervals explored by bisection approach. In each iteration, one candidate interval is checked for a possible zero.





generator	squared distance function	surface
 point	$x^2 + y^2 + z^2$	
 circle	$x^2 + y^2 + z^2 - 2\sqrt{x^2 + y^2} + 1$	

Table 4.1: Implicit functions using squared Euclidean distance from a generator curve.

might contain a root are returned if we have reached the desired degree of accuracy (B), or explored further by splitting them in half (C).

Figure 4.3 (c) shows a trace of this algorithm in action. Because of the *wrapping effect*, the range extension is not a perfect predictor of where roots occur, so sometimes ranges without a root are considered; however, because *range arithmetic* is conservative, no root is ever missed.

4.2.1 Test Cases

For test cases, we picked a handful of examples with different levels of complexity using blobby shapes introduced by Blinn [9]. These are adaptations of electron density functions that represent a field strength f at a point $p = (x, y, z)$ based on the squared Euclidean distance $d(p)$ from some generator (an atom in the real world):

$$f(p) = e^{-ad(p)}.$$

A sample of generators is shown in Table 4.1, along with the resulting surfaces.

More complicated test cases were built by summing several functions and adding deformations. Summing functions is an affine operation:

$$g(p) = \sum_{i=1 \dots n} f_i(p).$$

Adding affine operations increases program length without increasing the number of noise symbols while computing each f_i .

Using deformations, on the other hand, produces longer sequences of non-affine operations, leading to variables with more noise symbols. Two deformations used in the examples are twisting around the y -axis and adding sinusoidal bumps in the y direction. α and β are small constants used to control the magnitude and period of the deformation.

$$\text{twist}_y(p) = (x \cos(\alpha y) - z \sin(\alpha y), y, x \sin(\alpha y) + z \cos(\alpha y))$$

$$\text{bump}_y(p) = (x, \alpha \sin(\beta x) \sin(\beta z)y, z)$$

Similar deformations can be defined for the x and z axes. The twist deformation is particularly ill-suited for unique condensation, as it spreads noise symbols out to both the x and z coordinates and prevents them from becoming unique.

4.2.2 Results

All the test cases were run at 512×512 resolution using $\varepsilon = 10^{-5}$ during the trace. Figure 4.4 shows the first set of simple functions tested: spheres with bump deformations. Figure 4.5 shows the resulting measurements.

Unique condensation reduces both the maximum number of required temporaries and instruction count. Without the bumps, where there are few non-affine operations, it does no worse, but also not much better in any measurement. As more bumps are added, longer sequences of non-affine operations produce more unique symbols. This leads to greater reductions in both the number of temporaries and instruction count, to about a 30% reduction with two bumps.

Reductions in the number of temporaries in turn improves GIPS. Because of the relationship shown in Figure 4.2, sometimes small changes in number of temporaries can give huge GIPS benefits, while other changes have little effect on GIPS.

The final speedup is equal to

$$\text{speedup} = \frac{\text{execution time}_{\text{AA}}}{\text{execution time}_{\text{AAUC}}} = \frac{\text{instruction count}_{\text{AA}}}{\text{instruction count}_{\text{AAUC}}} \times \frac{\text{GIPS}_{\text{AAUC}}}{\text{GIPS}_{\text{AA}}}.$$

Having many non-affine operations that give many noise symbols may not always improve by as much as the bump examples. As mentioned before, if symbols from the non-affine operations do not become unique, as with the twist, then unique condensation also has less effect.

Consider the torii in Figure 4.6 with three bumps and three twists:

$$\text{torus}(\text{bump}_z(\text{bump}_y(\text{bump}_x(p))))),$$

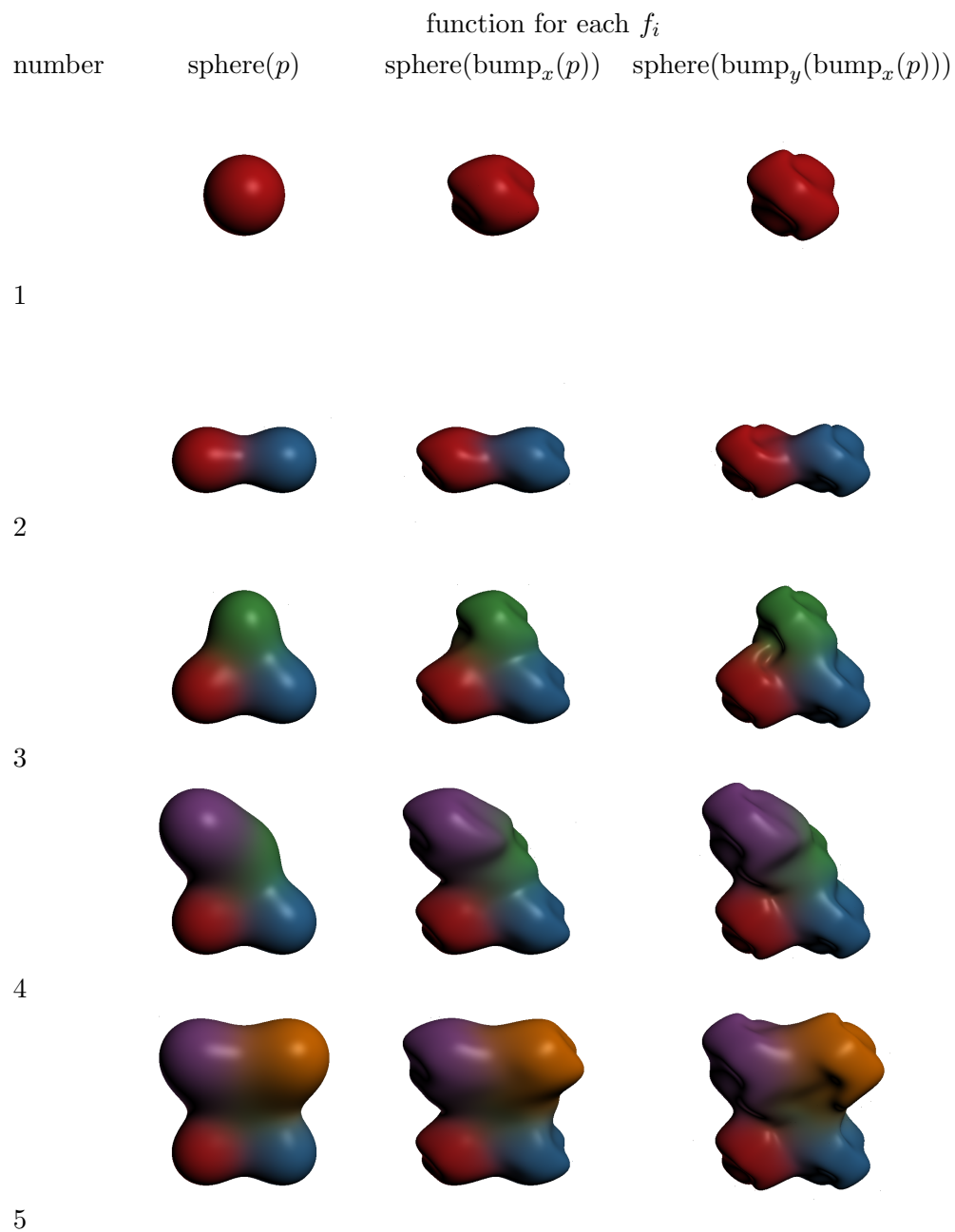


Figure 4.4: Sphere functions varying in number of spheres and bump deformations applied

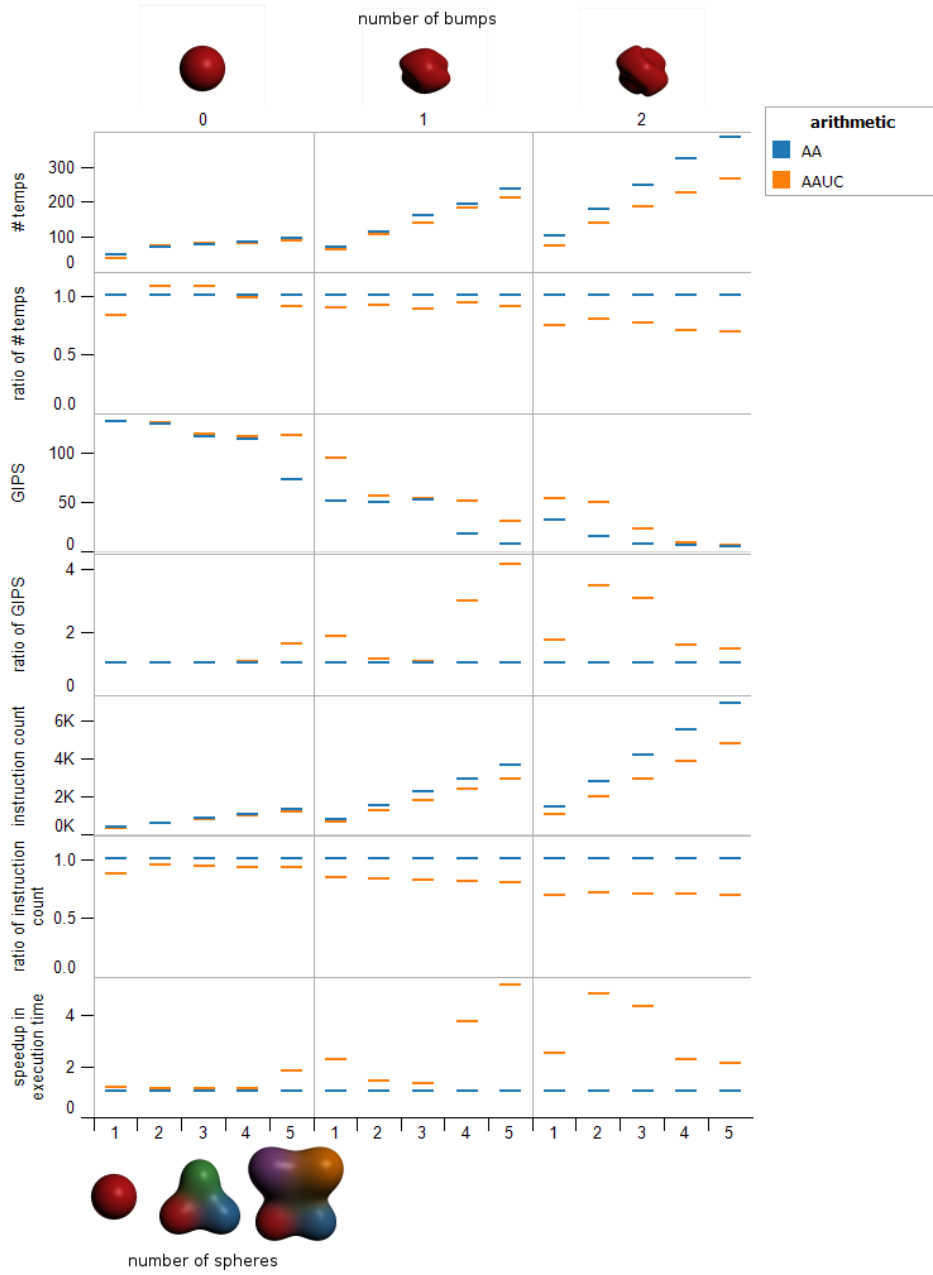


Figure 4.5: Performance results for sphere functions. From top to bottom, the number of temporaries affects GIPS. Together GIPS and instruction count affect execution time.

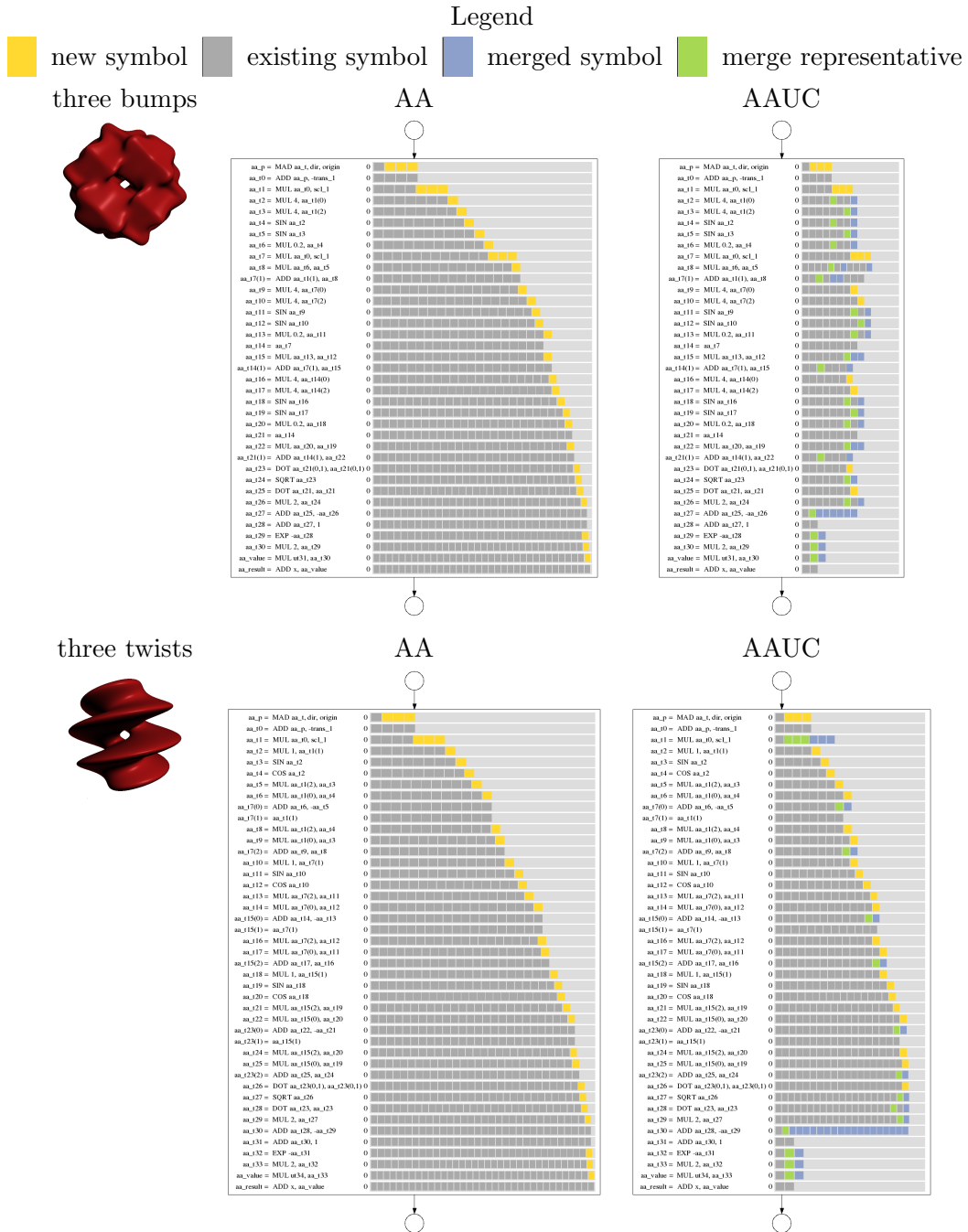


Figure 4.6: Each diagram is like Table 3.2, with the spaces removed so each line shows only the number of live noise symbols after an instruction. Comparing the effects of unique condensation in each case, there is a much greater reduction throughout the entire program for the bumps than there is with the twists.

$$\text{torus}(\text{twist}_y(\text{twist}_y(\text{twist}_y(p))))).$$

Both deformations lead to programs of about similar complexity in terms of instruction count and number of noise symbols before unique condensation. After condensation, though, there is a much greater reduction in live noise symbols for the bumps than for the twists.

From these simple examples, affine arithmetic with unique condensation never hurts and helps significantly as the number of noise symbols increase. Different functions benefit to different degrees, though, depending on how many symbols become unique.

4.3 Global Optimization on Generative Models

The next sample application focuses on analyzing more complicated functions with control flow. Functions are built in this case using generative modeling [48]. Generative modeling is a way of representing geometric shapes by functions and forming functions for complex shapes by using operators to combine simpler shapes.

Each shape is a parametric function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$, which is implemented in Sh as an `ShProgram` object. There are also special variables $x_0, x_1, \dots \in [0, 1]$ for the parametric coordinates, and a function for x_k can be built using $m_x(k) : \mathbb{R}^k \rightarrow \mathbb{R}$ that returns just the k^{th} input.

Supported operators include the usual operations available on Sh variables like $+$, $-$, \sin , lerp as well as affine transformations.

For cleaner notation, assume that if an operation combines two functions $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $g : \mathbb{R}^p \rightarrow \mathbb{R}^q$ such as $f + g$, the number of inputs and outputs are magically increased to make the operation work out. For example, if $n < q$, f is padded with $q - n$ extra outputs with value 0. Similarly, if $m < p$, f gains $p - m$ inputs that are ignored.

Sample functions

Below are a few example functions.

A point $P \in \mathbb{R}^n$ is a function from $\mathbb{R}^0 \rightarrow \mathbb{R}^n$

$$f = P$$

.

We will use round () parentheses for function inputs and square [] to index on function outputs. For instance, with the point above, $f[0] = P[0]$ is the first coordinate in the output.

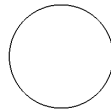
A line segment from point A to point B is a function from $\mathbb{R}^1 \rightarrow \mathbb{R}^n$, and it can be represented by a linear interpolation function:

$$\text{lerp}(x_0, A, B) = x_0A + (1 - x_0)B$$

Here x_0 can be replaced by any other function with one output.

This is a 2D unit circle

$$m_circle(x_0) = (\cos(2\pi x_0), \sin(2\pi x_0))$$



Both the line and circle are curves, functions with one input parameter. In addition, there is also support for piecewise cubic Bézier curves, read in from a path specification in an SVG (Scalable Vector Graphics) file:

$$m_svg_curve(filename, u).$$

The implementation for these curves in Sh stores the control points in a texture map, and then looks up the control points for the i^{th} Bézier curve when the input parameter $u \in [\frac{i}{n}, \frac{i+1}{n}]$. For now, a direct lookup into the texture map is not possible. This is because `range arithmetic` indices are not yet supported in texture lookup, and u will be a range value when we start analyzing functions with affine arithmetic. Texture lookup techniques for standard `interval arithmetic` exist [41] and can be modified for affine arithmetic; however, we have not implemented this yet. Instead, a for loop is used to iterate until finding the right set of control points.

`m_svg_curve(filename, u)`

Read in control points $P_{i,j}$ for $0 \leq i < n$,

where $P_{i,0}, P_{i,1}, P_{i,2}, P_{i,3}$ are the control points for `\mbox{B\'{e}zier}_\square curve_\square i`

`result = P_{0,0}`

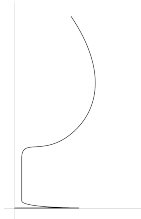

```

SH_FOR(i = 0; i < n; ++i) {
  SH_IF(i ≤ nu && nu < i + 1) {
    result = ∑j=0...3 Pi,jBj3 // Bj3 are the cubic Bernstein polynomials
  } SH_ENDIF
} SH_ENDFOR
return result

```

This is a SVG piecewise Bézier curve for wine glass profile, used later to build a wine glass:

`m_svg_curve("wineglass_profile.svg", x0)`



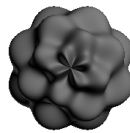
Surfaces are functions from $\mathbb{R}^2 \rightarrow \mathbb{R}^n$, such as a sphere:

`m_sphere(x0, x1) = m_circle(x0) sin(πx1) + (0, 0, cos(πx1))`



Here is a bumpy sphere, similar to the one in Snyder's examples:

`m_sphere(x0, x1)(1 + $\frac{1}{2}$ sin(8πx0) * sin(8πx1))`

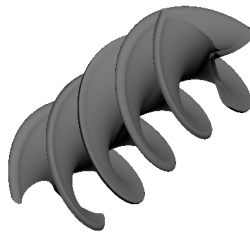


Additional operations supported include affine transformations. `m_translate_z(m, dz)`, for example, translates function m by amount dz along the z -axis. Similarly, `m_rotate_z(m, angle)` rotates m by $angle$ radians around the z -axis.

Using these two operations and an SVG curve, we can create a piece of fusilli. Here is the SVG cross section:

$$m_svg_curve("fusilli_cross.svg", x_0)$$

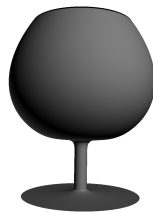

Then this is translated and rotated along the z-axis to give

$$m_rotate_z(m_translate_z(m_svg_curve("fusilli_cross.svg", x_0), 3x_1), 3\pi x_1)$$


One interesting combination of these affine transformations is the profile product. This sweeps and scales a 2D cross section curve along a third axis to give a 3D surface, where the scaling and translation each depend on one coordinate of a 2D profile curve.

$$m_profile(cross_section, profile) = m_translate_z(cross_section * profile[0], profile[1])$$

The profile product can produce shapes like a wine glass:

$$m_profile(m_circle(x_0), m_svg_curve("wineglass_profile.svg", x_1))$$


4.3.1 Global Optimization Algorithm

The global optimization used below is a simpler version of the method found in Snyder's book. The algorithm was changed to run all of the range arithmetic as a GPU stream program, so instead of considering one candidate range at a time, it looks at large batches of ranges in parallel.

The algorithm follows a branch-and-bound approach that breaks the domain into smaller and smaller subranges until finding a subrange that holds the global minimum. The function is sampled at each iteration to find a bound f^* on the minimum value achieved so far. This way, subranges in the domain where the function is definitely greater than f^* can be dropped from future exploration.

Here is pseudocode for the algorithm. Each of the lines marked with ■ is a separate stream program run on the GPU. In the whole function, affine arithmetic is only involved during the range extension computation on the first marked line.

The unmarked lines that perform no range arithmetic could also be run on the GPU, but they require support for expansion, reduction, and filter operations on streams that were not readily available in Sh when this application was originally implemented. Detailed analysis of how affine arithmetic performs will focus in any case on the GPU portion. Other timings for the real arithmetic sections on CPU, and overhead for moving data between CPU and GPU will be covered more briefly.

```

// Find  $y \in \mathbf{x}$  which minimizes the objective function  $f(y)$ 
// Lines marked ■ run on the GPU and the rest run on the CPU.
minimize( $f$ ,  $\mathbf{x}$ ):
     $D = \mathbf{x}$  // candidate boxes in the domain
     $fmax = \infty$  // upper bound on the minimum value  $f(x)$ 

    do
        replace each range  $\mathbf{x} \in D$  by splitting it into subranges
        compute the range extension  $R_i = f(D_i)$  ■
        sample the function value  $sample_i = f(\text{mid}(D_i))$  ■
        find  $f^* = \min(f^*, \min(sample_i))$ 
        filter out  $D_i$  for which  $R_i > f^*$ 
        check if finished  $\max(\text{wid}(D)) < \varepsilon$ 
    return an answer  $\text{mid}(D_0)$ 

```

Since in the version of Sh used for this thesis, there is still significant overhead per iteration in the non-GPU sections, it helps to ensure that the GPU works with large batches at a time. As a result, the splitting step produces at least 256×256 subranges. It also evaluates no more than 2048×2048 subranges at a time due to size limits on the GPU.

All of the examples below use this global optimization algorithm to find where the minimum Euclidean distance between two shapes A, B occurs.

$$\text{dist}(A, B) = \sqrt{\sum_{i=1}^n (A[i] - B[i])^2}$$

Figure 4.7 shows the result on a simple test case finding the distance between a point and 2D teapot curve.

4.3.2 Results

The minimum distance global optimization was run on several pairs of shapes, as seen in Figure 4.8.

The bumpy spheres use no control flow, while the fusilli and wineglass both use loops and if-else branches to generate Bézier curves. The results in Table 4.2 show the same set of performance counters as the implicit raytracing example. With the bumpy spheres, the performance improvement from adding `unique condensation` is relatively large. Like the similar bumpy spheres from the previous example, there are many non-affine operations that generate good opportunities for `unique condensation`.

Table 4.3 describes the execution times for all parts of the algorithm, on and off the GPU. Compared to the time actually spent running on the GPU, the overhead of transferring data to and from GPU as well as the range splitting are all relatively large. However, by moving the entire algorithm to the GPU the transfer overhead can be eliminated, and the other computation times likely reduced.

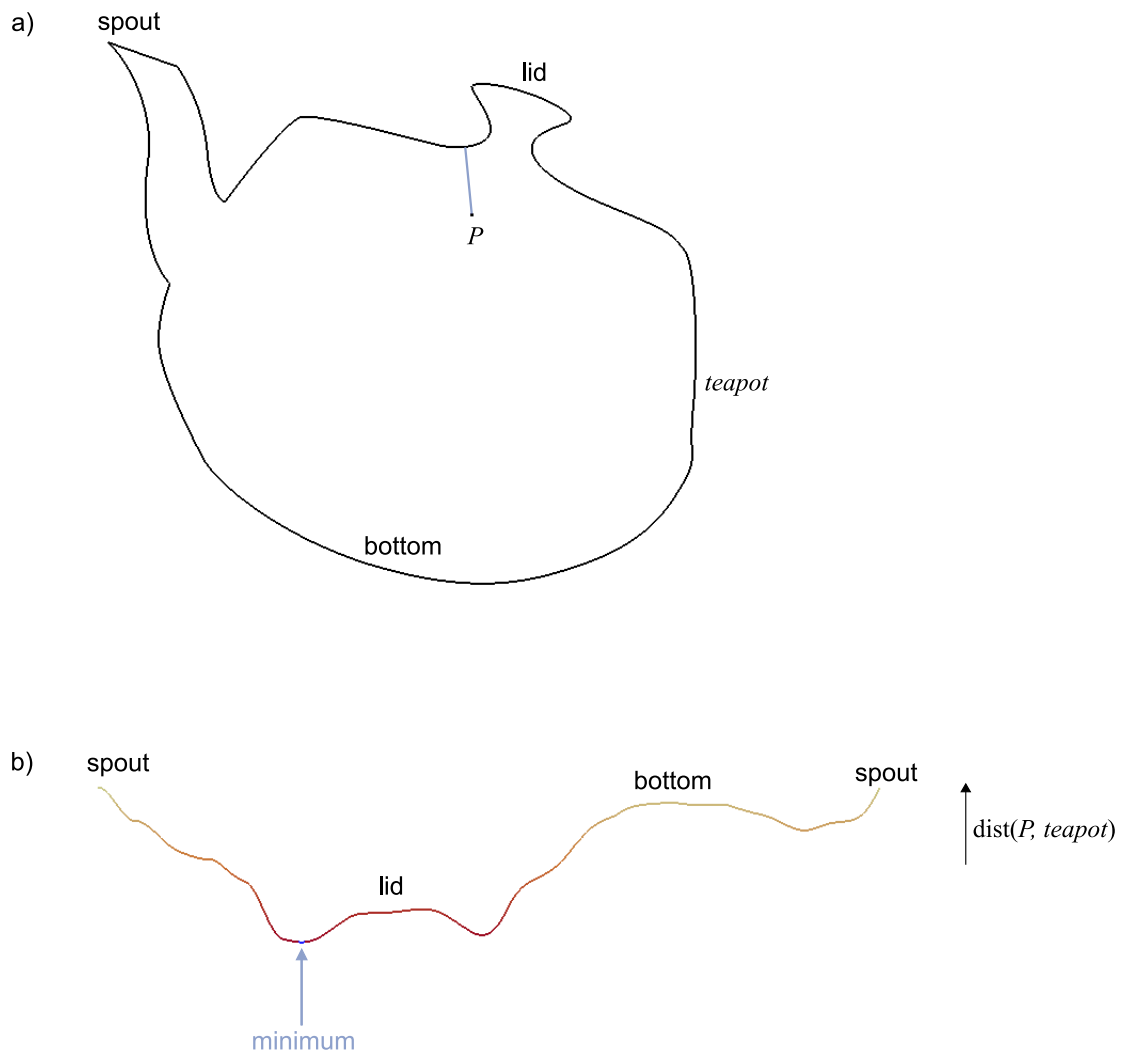


Figure 4.7: Minimum distance between point and teapot. a) shows where the minimum occurs and b) plots the corresponding distance function being minimized

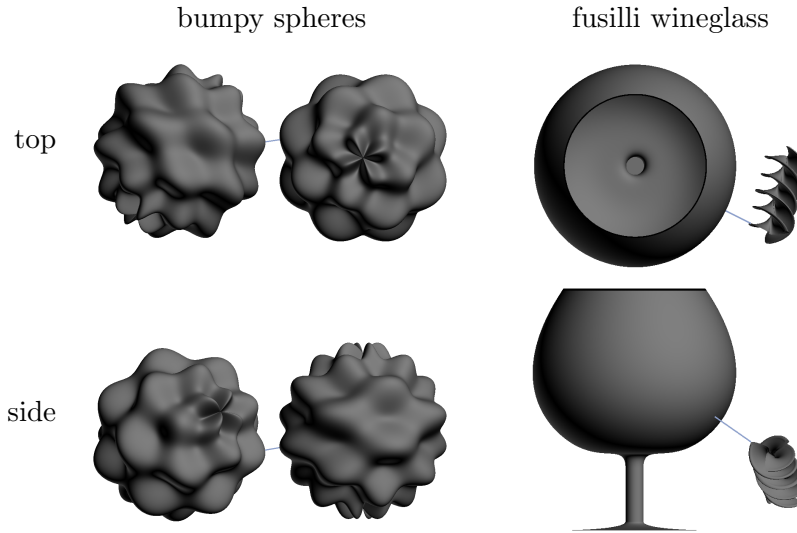


Figure 4.8: Top and side views showing the minimum distance between a) bumpy spheres b) fusilli and wine glass c) bumpy wine glass and wine bottle

case	bumpy spheres			fusilli wineglass		
	AA	AAUC	ratio	AA	AAUC	ratio
number of temps	211	113	1.8	123	96	1.3
instruction count	4728	2637	1.7	4578	3199	1.4
total render time (ms)	490	150	3.3	890	420	2.1

Table 4.2: Performance measures for the minimum distance examples and ratios showing the improvement of AAUC over AA. Total render time is over all iterations.

		bumpy spheres	fusilli wineglass
split ranges		260	1600
compute range extension	transfer inputs to GPU	600	1400
	render ■	150	420
	transfer outputs from GPU	380	690
sample function value	transfer inputs to GPU	0.4	1
	render ■	140	320
	transfer outputs from GPU	80	590
find f^* , filter, check if done		6	130

Table 4.3: Total times for each step in the global optimization algorithm using AA with unique condensation. ■ marks the parts run on the GPU.

Chapter 5

Conclusions

This thesis shows how static analysis enables *affine arithmetic* to run efficiently on GPUs. As a result, applications using affine arithmetic can harness the computational power afforded by GPUs to improve interactivity or to tackle more complex functions.

Classic *data flow* analysis is used to statically determine the *noise symbols* in an affine arithmetic computation. This lets GPUs operate on a dense vector representation for affine arithmetic instead of an expensive and less regular sparse vector approach. In addition, static analysis provides useful information for *condensing* noise symbols, a shortening of vectors that gains efficiency by computing results that are slightly different, but still correct. Two simple condensation techniques are explored: *unique condensation* for redundant noise symbols and *hierarchical condensation* for managing the number of noise symbols in loops.

Two graphics applications, rendering implicit surfaces and global optimization on parametric surfaces show the performance of the implementation and condensation techniques.

5.1 Future Work

While the current static analysis works well and the condensation methods give significant speedups, this is only a tentative first step. Trying more general applications and improving condensation are both good directions to pursue.

5.1.1 More General Applications

We have applied decades old compiler technology to run a small set of decades old graphics applications using affine arithmetic on a single architecture, the GPU. How will the techniques in this thesis work with more general applications and on other stream processors? Are there other arithmetics that work well with a similar approach?

So far, performance on the two applications we tried is reasonable but does suffer with more complex functions. It is important to test a wider variety applications and understand which ones run well using affine arithmetic on the GPU. To support even more applications, we can extend the static analysis to allow textures lookups and to provide better programmer feedback.

Texture lookup is a fundamental operation in graphics, useful in many areas from displacement mapping to lighting surfaces. While there has been some work on using interval arithmetic textures for displacement mapping [41], using affine arithmetic instead leads to new problems: efficient texture representations and texture lookups that take advantage of the non-rectangular nature of zonotopes.

Making affine arithmetic easier to use for programmers will also help encourage more applications. Affine arithmetic behaviour depends on the expression used to compute a function. Changing the order of operations or using discontinuous operations can increase the wrapping effect drastically with little feedback to the programmer. Perhaps static analysis can help the programmer identify and change sequences of operations that lead to problems.

Turning sparse vectors into simpler dense vectors should improve performance on all architectures. How much impact does our approach have on other processors like the Cell BE and multicore CPU? Is there a significant change compared to the best available conventional implementations of affine arithmetic?

In addition to affine arithmetic, there are many other arithmetics. We should compare performance on stream processors with other simpler range arithmetics like interval arithmetic and reduced affine arithmetic [19]. Beyond ranges, arithmetics like dual numbers used in automatic differentiation and geometric algebra also depend on sparse vectors. These may benefit from static analysis as well.

5.1.2 Smarter Condensation

Currently, hierarchical condensation decisions and zonotope merging methods are quite crude. More advanced methods from compilers and computational geometry could lead to better con-

densation.

This thesis does not solve the problem of when to merge in long blocks of computation and how aggressively to merge. Picking merge points should depend on a some metric that takes into account both storage limits of an architecture and measures of noise symbol significance. In particular, based on the formulation of m live variables containing n noise symbols condensing into $k < n$ noise symbols, we know details about m and n from static analysis. Can this information, along with other program structure, help automatically choose good merge points and deciding what k to use?

Handling $n > 2$ or $k > 2$ well also depend on finding fast algorithms for approximating high-dimensional zonotopes with low-dimensional ones that are easy to implement on GPU. While there has been some progress on approximation by 2D and 3D parallelepipeds, more general approximations are needed.

Interaction with more advanced compiler optimizations would also be fruitful to explore. For example, loop unrolling might be an alternative to hierarchical condensation. Using single static assignment (SSA) form during static analysis could reduce space overhead at the expense of additional joins wherever φ functions are needed. Running common subexpression elimination before building a range extension prevents new noise symbols for the same source of uncertainty [52].

Bibliography

- [1] Data-parallel programming on the cell be and the gpu using the rapidmind development platform. 28
- [2] *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, ansi/ieee standard 754-1985 edition, 1985. 12
- [3] Gpubench. <http://gpubench.sourceforge.net>, 2004. 13, 28, 51, 53
- [4] Csx processor architecture. Technical report, ClearSpeed, February 2007. 26
- [5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986. 35, 37
- [6] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976. 36
- [7] J. ao Luiz Dihl Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. Oct. 1993. 2, 8, 15, 23, 34
- [8] ATI. *ATI Radeon HD 2000 programming guide*, 2007. 53
- [9] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982. 57
- [10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, 23(3), August 2004. 28
- [11] E. W. Cheney. *Introduction to Approximation Theory*. McGraw-Hill, New York, 1966. 18

- [12] A. de Cusatis Jr., L. H. de Figueiredo, and M. Gattass. Interval methods for ray casting surfaces with affine arithmetic. In *Proceedings of SIBGRAP'99*, pages 65–71, 1999. 1, 55
- [13] L. H. de Figueiredo. Surface intersection using affine arithmetic. In *Proceedings of Graphics Interface '96*, pages 168–175, 1996. 23
- [14] P. S. Dwyer. *Linear Computations*. John Wiley & Sons, New York, 1951. 5
- [15] C. F. Fang, R. A. Rutenbar, and T. Chen. Fast, accurate static analysis for fixed-point finite-precision effects in dsp design. In *Proc. ICCAD'03*, pages 275–282, 2003. 23
- [16] R. Fateman. Honest plotting, global extrema, and interval arithmetic. In *Proceedings of ISSAC-92*, pages 216–223, 1992. 22
- [17] B. Flachs, S. Asano, S. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a cell processor. 26
- [18] M. J. Flynn and K. W. Rudd. Parallel architectures. *ACM Computing Surveys*, 28(1):67–70, March 1996. 26
- [19] M. N. Gamito and S. C. Maddock. Ray casting implicit fractal surfaces with reduced affine arithmetic. *The Visual Computer*, 23(3):155–165, March 2007. 72
- [20] O. Gay. *Libaa: Une librairie C++ d’Affine Arithmetic*. EPFL, 2003. 33
- [21] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991. 12
- [22] E. R. Hansen and G. W. Walster. *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 2004. 10, 23
- [23] W. Heidrich and H.-P. Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface*, pages 8–16, 1998. 1, 23
- [24] W. Heidrich and H.-P. Seidel. View-independent environment maps. In *Eurographics/SIG-GRAPH Workshop on Graphics Hardware*, pages 39–45, 1998. 48
- [25] W. Heidrich, P. Slusallik, and H. Seidel. Sampling procedural shaders using affine arithmetic. *ACM Trans. on Graphics*, 17(3):158–176, July 1998. 1, 23

- [26] T. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, September 2001. 6
- [27] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, pages 305–317. 39
- [28] U. J. Kapasi. *Conditional Techniques for Stream Processing Kernels*. PhD thesis, Stanford University, March 2004. 27
- [29] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of MICRO*, pages 159–170, 2000. 28
- [30] U. J. Kapasi, W. J. Daly, S. Rixner, J. D. Owens, and B. Khailany. The imagine stream processor. In *Proceedings of the International Conference on Computer Design*, August 2003. 28
- [31] M. Kass. CONDOR: Constraint-based dataflow. In *Proc. SIGGRAPH*, pages 321–330, July 1992. 33
- [32] R. B. Kearfott, M. T. Nakao, A. Neumaier, S. M. Rump, S. P. Shary, and P. van Hentenryck. Standardized notation in interval analysis. *Submitted to Reliable Computing*, 2002. 8
- [33] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL '73*, pages 194–206, 1973. 36
- [34] V. Kreinovich, D. J. Berleant, and M. Kosheleve. Interval and related software. <http://www.cs.utep.edu/interval-comp/intsoft.html>, 2005. 33
- [35] M. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. AK Peters, 2004. 30
- [36] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader Algebra. In *ACM Trans. on Graphics (Proc. SIGGRAPH)*, volume 23, pages 787–795, Aug. 2004. 29
- [37] M. D. McCool, Z. Qin, and T. S. Popa. Shader Metaprogramming. In *Proc. Graphics Hardware*, pages 57–68, Sept. 2002. 28
- [38] MIT Computer Architecture Group. *StreamIt Language Specification*, version 2.0 edition, October 2003. 28

- [39] D. P. Mitchell. Robust ray intersection with interval arithmetic. In *Proc. Graphics Interface*, pages 68–74, May 1990. 1
- [40] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1966. 1, 5, 6, 14
- [41] K. Moule and M. McCool. Efficient bounded adaptive tessellation of displacement maps. In *Graphics Interface*. 1, 23, 63, 72
- [42] S. S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. 35
- [43] J. D. Owens. *GPU Gems 2*, chapter Streaming Architectures and Technology Trends, pages 457–470. Addison-Wesley, 2005. 25
- [44] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. 2
- [45] D. Ratz. Inclusion isotonic extended real interval arithmetic. Technical Report D-76128, Universität Karlsruhe, May 1996. 6
- [46] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. 23, 26
- [47] C. Schwarz, J. Teich, E. Welzl, and B. Evans. On finding the minimum enclosing parallelogram. Technical report, ICSI Berkeley. 34
- [48] J. M. Snyder. *Generative Modeling for Computer Graphics and CAD*. Academic Press, Boston, 1992. 62
- [49] J. M. Snyder. Interval analysis for computer graphics. *Computer Graphics (Proc. SIGGRAPH)*, 26(2):121–130, July 1992. 23
- [50] J. M. Snyder and J. T. Kajiya. Generative modeling: A symbolic system for geometric modeling. *Computer Graphics (Proc. SIGGRAPH)*, 26(2):369–378, July 1992. 1, 23
- [51] J. Stolfi. Libaa. <http://www.ic.unicamp.br/~stolfi/EXPORT/software/c/libaa>, 1994. 33

- [52] J. Stolfi and L. H. de Figueiredo. Self-validated numerical methods and applications. *Brazilian Mathematics Colloquium monograph*, IMPA, 1997. 17, 21, 34, 73
- [53] M. Stphenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon computation. In *Proc. of PLDI 2000*, pages 108–120, 2000. 23
- [54] T. Sunaga. Theory of an interval algebra and its application to numerical analysis. *RAAG Memoirs*, 2:547–564, 1958. 5
- [55] J. Tupper. Reliable two-dimensional graphing methods for mathematical formulae with two free variables. In *Proceedings of SIGGRAPH 2001*, pages 77–86, 2001. 1, 22
- [56] G. W. Walster. Compiler support of interval arithmetic with inline code generation and nonstop exception handling. Technical report, Sun Microsystems, 1998. 6
- [57] G. W. Walster, E. R. Hansen, and J. D. Pryce. Extended real intervals and the topological closure of extended real relations. Technical report, Sun Microsystems, March 2002. 6
- [58] M. Warmus. Calculus of approximations. *Bulletin de L'Academie Polonaise des Sciences*, IV(5):253–259, 1956. 5
- [59] R. C. Young. The algebra of many-valued quantities. *Mathematische Annalen*, 104:260–290, 1931. 5

Index

- AA, 15
- affine form, 15
- approximation error, 13
- arithmetic intensity, 23

- branch-and-bound, 22

- CFG, 30
- Chebyshev approximation, 18
- condensation, 34
- constant propagation, 34
- containment constraint, 7
- containment set, 6
- content, 14

- data flow analysis, 34
- data parallelism, 25
- dead-code elimination, 35
- def-use chains, 37
- definition of a definition, viii
- degenerate interval, 9

- expansion, 24

- filter, 24
- floating point, 12
- fundamental invariant, 7

- gather, 24

- GIPS, 53
- GPU, 26

- hierarchical condensation, 45
- hull, 9

- IA, 9
- Instruction-level parallelism, 25
- interval hull (\square), 9
- interval arithmetic (IA), 9
- iterative data flow analysis, 36

- join, 9
- joint range, 14

- kernel, 24
- kernel locality, 25

- lattice, 35
- live variable analysis, 35
- lower bound, 8

- map, 24
- meet, 9
- midpoint, 9
- MIMD, 26
- min-range approximation, 18

- natural range extension, 7
- noise symbol, 15

partial deviation, 15
predecessor function, 30
producer-consumer locality, 25

radius, 9
range arithmetic, 1
range extension, 6
range function, 6
reduction, 24

scatter, 24
significant figures, 5
SIMD, 26
SISD, 26
static storage allocation, 34
stream, 24
stream graph, 24
stream processor, 23
structural analysis, 31
structure, 33
successor function, 30
SWAR, 51

task parallelism, 25
transformer, 31

unique condensation, 34
upper bound, 8

width, 9
wrapping effect, 14

zonotope, 16