Going Fast: Vectorized Special Function Approximations

John Doherty

Fast Algorithms and Integral Equations: Final Project

Motivation

Laplace Kernel $$K(x,y) = \log(||x-y||)$$

Helmholtz Kernel $$K(x,y) = \frac{i}{4} H_{(0)}^{(1)}(||x-y||)$$

$$H_{(0)}^{(1)}(z) = J_{(0)}(z) + iY_{(0)}(z)$$

array size = 2 E16	Laplace Kernel	Bessel Function (1st kind, 0th order)
Runtimes (ms)	1.74	58.82

Bessel Function is 33.7 times slower!

Overview

Approximate Function

Create a fast implementation

Test Performance of implementation

Use implementation in package for solving PDEs

Function Approximations Through Interpolation

Interpolation finds a function to exactly fit your function given a set of nodes.

Error for a fixed order p (Taylor's Remainder Theorem): $O(h^{p+1})$

- Next derivative also affects error which is why we assume smooth

Use Chebyshev nodes to reduce error.

Use an orthogonal polynomial to lower conditioning of Vandermonde matrix

- Chebyshev polynomials are used here.

An Adaptive Method to Improve Accuracy

Use a fixed order interpolation scheme.

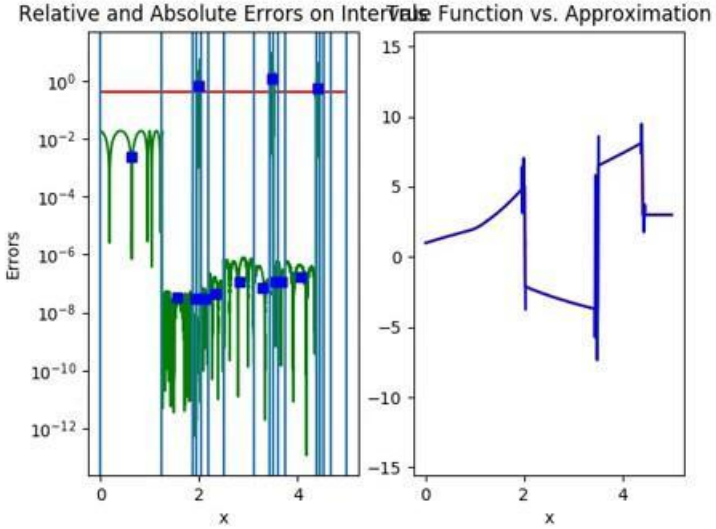
Compute the relative error on the interval.

If it is too large then split the interval and find a new interpolant on each interval.

If the error is still too high, split the interval again.

Run Remez Algorithm

- Minimizes absolute distance between polynomial and function in infinity norm (Interpolation that minimizes error)
- Equioscillation Theorem



Code Generation: Overview

Generate code which will evaluate our approximation at some point, x.

Two parts needed.

Interval traversal to find the interval that it is on

Polynomial evaluation using the coefficients defined on that order.

Multiple Languages and data types

- OpenCL or ISPC in either single or double precision

To improve speed we will generate vectorized code in ispc. The performance will be compared to the scalar version of the code.

Implementation: Scalar Code

Traversal: Binary Search Tree or Hash table

-Binary Search Tree

Runtime: O(log(n))

Memory cost: Low (size of tree)

- Hash table

Runtime: O(1)

Memory cost: O(2**num_levels)

```
export void eval(const uniform float tree[], uniform float x[], uniform float y[]){

varying float T0, T1, Tn, a, b, s, x_scaled, xn;
        for (uniform int nbase=0; nbase<67108864; nbase+=programCount) {
        varying int n = nbase + programIndex;
        varying int index = 0;
        xn = x[n];
        for (uniform int i=1; i<17; i++){
                index = tree[index] > xn ? (int)tree[index+9] : (int)tree[index+10];
        }
        a = tree[index+7];
        b = tree[index+8];
        x_scaled = (2./(b - a))*(xn - a) - 1.0;
        T0 = 1.0;
        s = tree[index+1]*T0;
        T1 = x_scaled;
        s = s + tree[index+2]*T1;
        x_scaled = 2*x_scaled;
        for (uniform int j=3; j <=6; j++) {
                Tn = x_scaled*T1 - T0;
                s = s + tree[index + j]*Tn;
                T0 = T1;
                T1 = Tn;
        }
        y[n] = s;
        }

}
```

Generated BST code

Faster: Vectorization via ISPC

Why not multicore? Why ISPC over OpenCL?

What would vectorization look like?

Where might we lose performance in each method (BST vs. Hash) (Double vs. Single Precision)?

Overview of Results

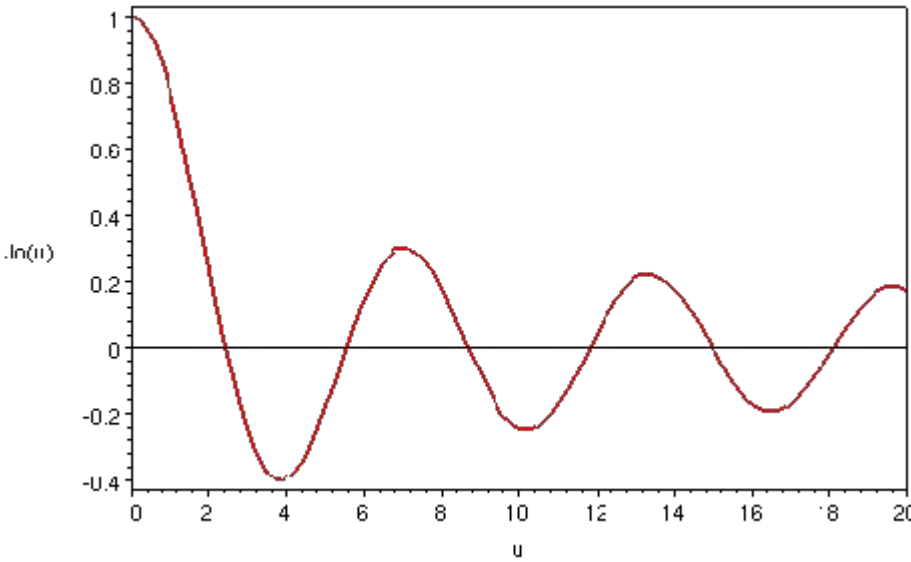
Approximation of 0th order bessel function from 0 to 20.

Order = 5, Error tolerance = 1e-7

Test each traversal method with double vs. single precision. And test each loop timing to see where we are losing performance

ISPC target: avx2-i32x8
-Vector width
- 8 (single precision)
- 4 (double precision)

Ispc arch: x86-64
Ispc opt: disable-fma



Hardware Used

Machine: Dunkel.cs.illinois.edu
CPU: Intel® Xeon® Processor E5-2650 v4
CPU Family: Broadwell
Instruction Set: 64 bit

Base / Turbo clock speed: 2.2 / 2.9 GHz
Max Memory Bandwidth: 76.8 GB/s

Timing Results - Single Precision

Interval Traversal: Using the hash table does give us a constant runtime (saw a 25x speedup).

Hash table: Unexpected, the speedup is 4x more than one would expect. (We should only expect 8 times the performance since we have a vector width of 8)

-Similarly this occurs using doubles

	Scalar Runtime (Sec.)	Vectorized Runtime (Sec.)	Speedup Ratio
Binary Search Tree	4.48	.94	4.8
Interval Traversal	1.55	.61	2.5
Polynomial Evaluation	2.17	.058	37.7
Hash Table	3.13	.30	10.56
Interval Traversal	.38	.051	7.4
Polynomial Evaluation	2.15	.057	37.6

Timing Results - Double Precision

Similar to Single, except that ½ the speedup, which is expected, since the vector width is now 4

Still there is an unexplained 4x speedup

Relative error of 1.2E-8

	Scalar Runtime (Sec.)	Vectorized Runtime (Sec.)	Speedup Ratio
Binary Search Tree	4.36	1.23	3.54
Interval Traversal	1.60	.86	1.86
Polynomial Evaluation	2.08	.13	15.88
Hash Table	2.96	.44	6.69
Interval Traversal	.40	.12	3.05
Polynomial Evaluation	2.08	.14	15.02

Timing Results - Double vs Single

Both are performance of vectorized function (scalar code is approximately equal performance). Both have an error of 1e-7 and order =5

Interval Traversal: Vectorization gives us the expected speedup of 8 for single precision and 4 for double precision.

Hash Table: Reduction in performance by half when using doubles (Expected by vector width being cut in half). Unexpected, the speedup is 4x more than one would expect.

	Single Runtime (Sec.)	Double Runtime (Sec.)	Speedup Ratio of Single
Binary Search Tree	.94	1.23	1.31
Interval Traversal	.61	.86	1.41
Polynomial Evaluation	.058	.13	2.27
Hash Table	.30	.44	1.49
Interval Traversal	.051	.12	2.53
Polynomial Evaluation	.057	.14	2.41

Timing Result - Different Tolerances

Getting more precise results does not require much more work using the Hash Table.

- This is because most of the work is in the Interval traversal (Seen in BST section) and this is a constant using the Hash table

	Error of 1e-7 (Sec.)	Error of 1e-13 (Sec.)
Binary Search Tree	1.23	3.42
Interval Traversal	.86	2.92
Polynomial Evaluation	.13	.15
Hash Table	.44	.62
Interval Traversal	.12	.13
Polynomial Evaluation	.14	.16

Sanity Checking Results - GFLOPS and Memory Bandwidth

Max GFLOPS: 4.4 GFLOPS
- Assuming Instruction throughput (insn/cycle) of 2

Max Memory Bandwidth: 76.8 GB/s
- Actual Peak may be less if they are accessing another processor's RAM

Same order, 5. Error tolerance of 1e-7

Peak GFLOPS reaches near optimum for vectorized.
- Suggests vectorized version is performing near peak (would be concerning if above somehow). Thus likely that scalar is performing more poorly than expected.

	Single GFLOPS	Double GFLOPS	Single Memory Bandwidth (GB/s)	Single Memory Bandwidth (GB/s)
Binary Search Tree	.207	.32	10.88	16.64
Interval Traversal			13.5	9.60
Polynomial Evaluation	3.38	2.98		
Hash Table	.74	.99	7.57	10.17
Interval Traversal			9.75	3.85
Polynomial Evaluation	3.54	2.93		

Sumpy

Python package

- Symbolic code generators for multipole and local expansions and translations
- Creates kernels for Pytential
- Bessel and Hankel Functions evaluated here (uses loopy for the code generation)

Goal is to use the approximation package we just discussed and integrate it into this.

Bessel Functions

How do we handle different order bessel functions?

OpenCL not ISPC

- Expect better performance

Bessel Performance characterization vs Approximation characterization.

Thanks for Listening!

Questions?