Evaluating the Performance of Vectorized Function Approximations

John Doherty
Department of Computer Science
University of Illinois at Urbana-Champaign
Champaign, IL 61801
`jjdoher2@illinois.edu`

December 21, 2017

Abstract

In this paper, a package capable of approximating functions in one dimension and generating vectorized function evaluations is introduced. The performance of the generated code is analyzed and an accurate measure of the GFLOPS and memory bandwidth is found. The performance is found be near peak for floating point operations but not near peak for memory accesses. This is possibly because the actual peak memory bandwidth is lower if the processor is actually accessing the memory of another processor in its NUMA node and not its own. The generated code also has the nice property that the runtime is independent of the error of the approximation. Going forward there are numerous ways to improve the performance of the code, including using fused multiply add instructions.

1 Introduction

One often encounters one dimensional functions in numerical analysis which are slow to evaluate. Special functions often fall in this category. The typical methods to evaluate these involve recurrence methods. This results in functions that can take orders of magnitude longer to evaluate than basic operations. If there is no way to analytically to obtain better speeds, one may be okay using an approximation that can be evaluated quickly. This is the basis of FUNPACK package of special functions [2]. Similarly, this paper will use approximations to create faster evaluations. In this paper, I will introduce a python package that is capable of generating vectorized approximations for smooth, one dimensional functions. The package is open source and freely available under the MIT license on pypi and github. The package is called adaptive-interpolation and is located at github.com/jdoherty7/Adaptive_Interpolation.

The paper is organized as follows. First, the algorithms in the package are discussed. The first is an adaptive interpolation method based on the Remez algorithm which is used to approximate the given function. This part of the package is written in python. The second algorithm in the package is the code generation sequence. This method takes the approximation and generates code which can be quickly evaluated in a specified language (C++, ISPC, or OpenCL). The characteristics of this generated code will be discussed and the performance will be evaluated.

2 Approximation: Adaptive Interpolation

Given an interval [a, b] $a, b \in \mathbb{R}$ and a real, smooth function $f \in \mathbb{C}^{\infty}$ on the interval, we seek to find an approximation, f_{approx} such that the relative error between the two functions is below some predefined tolerance $\frac{||f_{approx}-f||_{\infty}}{||f||_{\infty}} < \epsilon$. The minimax polynomial is the polynomial which minimizes the maximum absolute error on the interval. Since the norm of a function is constant on a constant interval, the relative error will also be minimized. Thus, by finding the minimax polynomial on the interval the error that we are trying to minimize will be. In order to find the minimax polynomial, the Remez Algorithm is used since it is known to converge to the minimax polynomial on a fixed interval [6]. However, given a degree, it may be that the error tolerance cannot be satisfied with the minimax polynomial. There are two options to reduce the error. One is to increase the interpolation degree used and the other is to employ an adaptive scheme which reduces the size of the interval. In this package the second method is chosen because it is less likely to overfit and keeps the flop count low, which is important for the generation section.

We will now describe the full algorithm, it consists of building a binary tree where each node contains a polynomial and the interval on which it is valid. The root of this tree will be the full interval [a, b] and the polynomial is calculated by using polynomial interpolation with a fixed order. The Remez algorithm is not used until closer to convergence since it is more expensive, however, the option is given to use only use the Remez algorithm for interpolation. The polynomial interpolation is done by building a square Vandermonde matrix and solving the linear system, where the right hand side is the function values on the node set. The chebyshev nodes are used so as to avoid runge's phenomenon, which is when the absolute error increases as the order of the interpolant is increased [6]. This occurs because not all node sets satisfy weierstrass theorem, which states that there is an interpolated polynomial which uniformly converges to the underlying funtion [6]. Chebyshev nodes are also used because they have a Lebesgue constant that grows more slowly than equispaced nodes, meaning the polynomial interpolation using them will converge faster with increasing order [5]. Multiple basis functions can be chosen, however the chebyshev basis functions are used in this paper because of their orthogonality, which reduces the conditioning of the Vandermonde matrix and because their recursive function has a low flop count which is useful for evaluation.

Once an interpolant is found on the current interval the relative error is calculated. This is done by using the maximum function value on the full interval and the maximum absolute error on the current interval. If this relative error is too large then the interval is bisected and a new interpolant is calculated on the left sub interval and the right sub interval. This process continues until the relative error is near the specified tolerance. Once it is near this tolerance the Remez Algorithm is used to calculate the interpolant. The adaptive scheme continues until the relative error is below the specified tolerance. Once this occurs the interval is no longer bisected. When algorithm is finished the tree will contain all of the data required to evaluate the approximation.

Algorithm 1: Adaptive Interpolation Method

```
Result: Approximation tree that contains a polynomial for each
        subinterval.
1  [a, b] = interval;
2  n = maximum order of interpolant;
3  Function adapt(a, b, n):
4      Polynomial = solve_linear_system(a, b, n);
5      Error = $\frac{||f-polynomial||_\infty}{||f||_\infty}$;
6      add_to_tree(polynomial);
7      if error too high then
8          adapt(a, (a+b)/2, n); ;          // adapt on left subinterval
9          adapt((a+b)/2, b, n); ;          // adapt on right subinterval
10     end
```

3 Evaluation: Code Generation

Once the approximation is created, we want to quickly evaluate it. Quick evaluation is unlikely in pure python and thus we resort to generating code in other languages in order to achieve the speed desired. This will also allow us to take advantage of vectorization and parallelism in other languages. Specifically, we will generate our approximations in OpenCL and ISPC. ISPC is a language that allows us to take advantage of vectorization on a CPU while OpenCL is a language that allows us to take advantage of the parallelism found in a GPU. PYOPENCL is used to generate the OpenCL code in this application.

The actual code generated for evaluating the approximation works as follows. An input array of points to evaluate the function at are given in an array x and the output of the function is placed in an array y. It is assumed that the points within x lie within the specified interval $[a, b]$, otherwise the evaluation may fail. The data representing the tree built during the approximation will also be given to the function. The evaluation is made by using the data in the approximation tree. This evaluation consists of two parts. The first part is a search to find which leaf in the approximation tree contains the interval which the point being evaluated exists in. The second part is evaluating the polynomial which exists on the leaf for that point.

Given our bisection adaptive scheme there are two methods that can perform this interval search. The first is using a binary search through the approximation tree. If k is the deepest level in the approximation tree, where the root of the tree is at level 0, thne the runtime of a binary search will be $O(k)$. Another method is to use a hash table. The hash table can be created by expanding the approximation tree to a complete tree. Each leaf in the complete tree then contains a pointer to the original leaf. The interval that the point lies on can then be determined using the hashing function $\lfloor\frac{(x-a)(2^k)}{b}\rfloor$. This has the advantage of having a $O(1)$ runtime, which is good for the performance, but has a $O(2^k)$ space complexity. This is also only practical for a bisection adaptive scheme. If splits are made at other points then the space complexity would become too large as each leaf in the hash table represents equispaced intervals. Thus, while the hash table should have far better performance, we would still like to analyze the performance of the binary search tree since it is more general and would be useful if a different adaptive scheme is used.

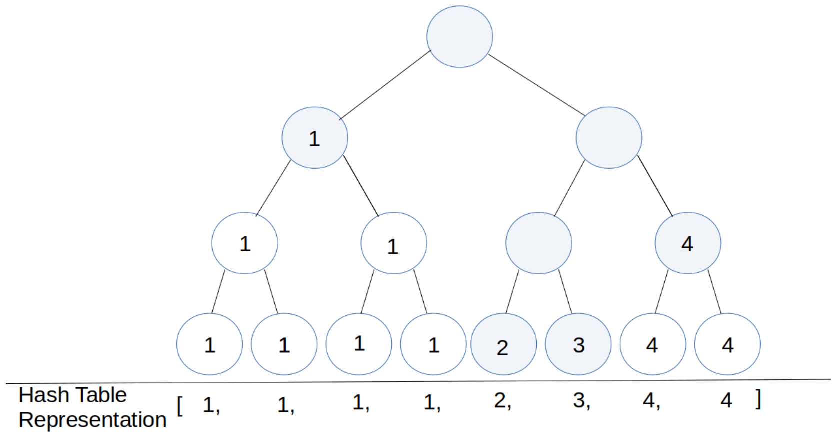


Figure 1: Shaded nodes are part of the actual approximation tree, non shaded nodes are used to expand the tree to be complete in order to create the hash table. The numbers represents the leaf number. The bottom contains the hash table representation of the approximation tree.

Once the correct interval is found the polynomial that exists in the interval must be evaluated. For monomials, Horner's method is used for the evaluation. This is because, if n is the number of terms in the polynomial then only n additions and n multiplications are required to evaluate it. For chebyshev polynomials, which are used in the results section, the recursive formula $T_{n+1} = 2*x*T_n - T_{n-1}$ is used to calculate each term. This is similarly done for other orthogonal polynomials.

Algorithm 2: Pseudo code for generated approximation code

```
Result: Approximate value of function at each point x.
[a, b] = interval;
n = maximum order of interpolant;
k = deepest level in approximation tree;
leaves = hash table containing pointer to polynomial defined at each leaf;
for x do
    coeff = leaves[(int) (x - a)$\frac{2^k}{b}$];
    T0, T1 = 1, x;
    y = coeff[0]*T0;
    y = y + coeff[1]*T1;
    for $1 < i <= n$ do
        Tn = 2*x*T1 - T0;
        y = y + coeff[i]*Tn;
        T0, T1 = T1, Tn;
    end
end
```

4 Characterizing Performance on a Single Threaded CPU

In this section, we will characterize the performance of the generated approximation code. The main goal of this section is to characterize the performance of this code and compare it to the peak machine performance that the code is being run on. This will tell us how well our code is performing. We can then characterize individual components of the evaluation, which will allow us to have a better picture of where performance is being lost and what can be improved.

4.1 Overview

In this experiment, we will test the speed of the generated approximation code using the two interval search methods previously discussed, binary search and the hash table. For each method, we will also individually test the interval search section and polynomial evaluation section of the code. This is done by only generating these sections and then timing them in the same way as the other tests. This will tell us what section of the code that the evaluation spends more time on. We will also test how these runtimes compare when using single precision floating point numbers and double precision floating point numbers, which should differ because the vector width is different for the two data types.

The performance will then be compared to peak machine performance to see how well it performs in an absolute sense. Since the interval search is mainly memory accesses, it is best to compare this to the peak memory bandwidth. The polynomial evaluation is mostly floating point operations so it makes more sense to compare this to the peak GFLOPS possible from the machine.

4.2 Technical Details

The code discussed was generated in ISPC and run on a single thread in a CPU. This is done to make it easier to characterize the performance and diagnose problems. The experiments were run on dunkel, which is a computer maintained by the scientific computing group at the University of Illinois at Urbana-Champaign and the CPU used was Intel E5-2650 v4. It has a base clock speed or 2.2 GHz and a turbo clock speed of 2.9 GHz and has a Broadwell microarchitecture [3]. When the generated code was compiled, the floating point instructions that were generated were vmulps, vaddps, and vsubps. This was verified by performing an objdump on the generated binary file. For the Broadwell microarchitecture, the reciprocal throughput of these instructions is each .5 cycles per instruction. Thus the expected peak performance of the CPU is 4.4 - 5.8 GFLOPS. The other metric used to measure peak performance was the maximum memory bandwidth. This is given in the CPU specs and is 76.8 GB/s for this CPU [3]. However, this is an upper bound. It is possible that the CPU being used is accessing the memory of another CPU in its NUMA node, which would result in a lower peak memory bandwidth.

Once the code was generated a shared library was created and loaded into a python script using the ctypes package. The shared library contained the generated code located within an ISPC function. The compiler flags used with ISPC included disabling fused multiply add instructions, setting the target to i32x8 and specifying the architecture to be x86-64. The FMAs were disabled to ensure that the number of floating point operations in the ISPC code was the same as the assembly code. The target and architecture flags were set because the machine used was 64 bit with a vector width of 8 [3].

After the shared library was created, the function approximation could be called on an array of points. In these tests, this array contained 2^26 equispaced points located within the full interval. The array was this large so that the asymptotic performance of the code could be determined. These points were randomly permuted prior to evaluation to remove spatial locality for memory accesses. This function was called 8 times prior to the timed run. The timings for the function were obtained by running the function 15 times and calculating the average runtime. Additionally, an empty function with no instructions was generated. This empty function was run and had an average runtime of 10^{-5} seconds. This can be used as measure of uncertainty in our measurement and no measurement will be taken to more than 5 digits.

4.3 Results

First, we will analyze the performance of the best generated code in these tests. This is the approximation which utilizes a hash table with single precision floating point numbers. This is found at the bottom of table 1. In this case, the full

Interval Search Type		Average Runtime (seconds)	GFLOPS	Memory Bandwidth (GB/s)
Binary Search Tree	Full Evaluation	1.3996	.1395	7.3235
	Interval Search	1.0522		7.8407
	Polynomial Evaluation	.0616	3.1707	
Hash Table	Full Evaluation	.3337	.6555	6.7426
	Interval Search	.0547		9.1408
	Polynomial Evaluation	.0617	3.2921	

Table 1: Table displaying the performance of an approximation with a relative error of 9.38E-7 which uses single precision floating point numbers in the evaluation.

Interval Search Type		Average Runtime (seconds)	GFLOPS	Memory Bandwidth (GB/s)
Binary Search Tree	Full Evaluation	1.3764	.2838	14.8939
	Interval Search	.9954		16.5763
	Polynomial Evaluation	.1386	2.8184	
Hash Table	Full Evaluation	.4935	.8865	9.1185
	Interval Search	.1515		6.6007
	Polynomial Evaluation	.1379	2.9460	

Table 2: Table displaying the performance of an approximation with a relative error of 9.38E-7 which uses double precision floating point numbers in the evaluation.

Interval Search Type		Average Runtime (seconds)	Average Runtime (seconds)
Binary Search Tree	Full Evaluation	1.3764	2.2373
	Interval Search	.9954	1.8641
	Polynomial Evaluation	.1386	.1385
Hash Table	Full Evaluation	.4935	.4937
	Interval Search	.1515	.1416
	Polynomial Evaluation	.1386	.1385

Table 3: Table comparing the runtimes of evaluations which achieve different relative errors and use double precision floating point. The approximation in the left column attains a relative error of 9.38E-7 and the right column is the approximation that attains a relative error of 9.93E-15

evaluation obtains .6555 GFLOPS and has a memory bandwidth of the 6.7426 GB/s. When directly comparing to the peak machine performance of 4.4 - 5.8 GFLOPs and 78.6 GB/s, this seems to do poorly. However, GFLOPS is a more useful metric for floating point operations while memory bandwidth is a better metric for memory stores and loads. Thus to see how well the evaluation does at each of these metrics, the code is broken up into its individual parts, an interval search and polynomial evaluation, and run separately.

In the interval search, the main cost is the one memory access performed and this achieves a memory bandwidth of 9.1408 GB/s. This is still far below the peak machine memory bandwidth of 78.6 GB/s. It is unlikely that there is any calls to disk, as the size of the hash table is only $2^5 * 4 = .125KB$. So the entire hash table should fit in the L1 cache, which is 32 KB. One possibility is that the memory is being accessed in another processor's memory which is in the same NUMA node. If this is what is occurring then the maximum memory bandwidth would be lower.

In the polynomial evaluation, the main cost is likely to be the floating point operations, however, some memory accesses do occur, so it is possible that these may impede the evaluation time. Thus, we use a tool called perf to determine the ratio of floating point operations being performed in the code. Perf is a tool that lets the user see how much work is spent on each instruction in the assembly. However, it may not be entirely accurate because of skid, but it is useful for our purposes [1]. When using perf on the generated code it was found that approximately 19.9 percent of the cycles are being spent on floating point operations (see the appendix for perf annotations). Since the polynomial evaluation obtains 3.2921 GFLOPS, this tells us that the floating point operations are being performed at a rate of around 15 GFLOPS. This is above the peak performance of the machine, which is 4.4 - 5.8 GFLOPS. However, it is within an order of magnitude and may be able to explained by the inaccuracy of perf.

Another thing to note is that the full evaluation time when using the hash table with single precision is approximately twice as fast as when double precision numbers are used. This is expected because the vector width of the machine when using double precision floating points is half of the vector width of the machine when using single precision floating points.

Another thing to compare is the interval search method used. While the binary search method has a higher memory bandwidth, its runtime is far worse because of the greater number of accesses it must make. The binary search method makes the interval search the dominant cost of the evaluation in every scenario tested, whereas when the hash table is used the polynomial evaluation is approximately the same cost as the interval search. This suggests that there are few scenarios where the binary search would be better suited than the hash table. However, since the cost of the hash table is absorbed by the memory binary search may have a chance at being better if the function being evaluated required high levels of adaptivity but only in select regions, thus forcing the hash table to be large enough to need to be stored on disk and allowing the binary search tree to be small enough (since it is not complete) to fit into memory. However, this is not one of those cases since for the double precision numbers with an error tolerance of 1E-6 creates a tree of 7 levels, as defined earlier, and the double precision numbers with an error tolerance of 1E-14 creates a tree of 9 levels. This creates hash tables of sizes $2^7 * 8 = 1KB$ and $2^9 * 8 = 4KB$, which are both capable of fitting into L1 cache of this processor. Furthermore, the tree is restricted to having a maximum depth of 30 currently, which would mean a maximum hash table of size 4 - 8 GB, which will typically fit into memory and not need to be stored on disk. Thus, for our purposes the hash table will typically retain a runtime constant on similar order magnitude as binary search. A consequence of this fact is that, when using a hash table, there is little effect on the runtime of the approximation evaluation as the error tolerance is decreased. This can be seen in table 3.

Thus, we have shown that the package is capable of generating code which runs floating point operations near peak performance. Furthermore, it is found that the floating point operations consist of a significant portion of the overall evaluation time and improving them can have a sizable impact on the overall performance. One way that can immediately improve the performance is to use FMAs. Unfortunately, the memory bandwidth does not appear to be operating near the expected peak performance. One reason this may be is that memory from another processor in the same NUMA node is being accessed which would have a lower peak performance. Lastly, it was shown that using the hash table appears to allow approximations whose runtimes do not depend on the error tolerance allowed, which is a positive if quick and accurate function approximations are desired.

5 Conclusion

In this paper, a new python package was introduced that generated fast vectorized approximations of functions in one dimension. An adaptive interpolation scheme using the Remez algorithm was used to approximate the functions. Once approximated, fast vectorized code that represented the approximation was generated in ISPC. The performance of this code was analyzed and the GFLOPS and Memory Bandwidth was measured. The measured GFLOPs of the code was found to be accurate representation of the performance since approximately half of the time was being spent on floating point operations and it achieved approximately half of the peak GFLOPs. However, the memory bandwidth measure was significantly under the peak. One possible explanation for this is that the code is accessing memory in the same NUMA node but of a different processor. This would have a lower memory bandwidth and could be the limiting factor. It was also found that in scenarios covered using the current version of the package that the generated code based on the hash table would be better than that based on the binary search and that the runtime was nearly independent of the error tolerance desired.

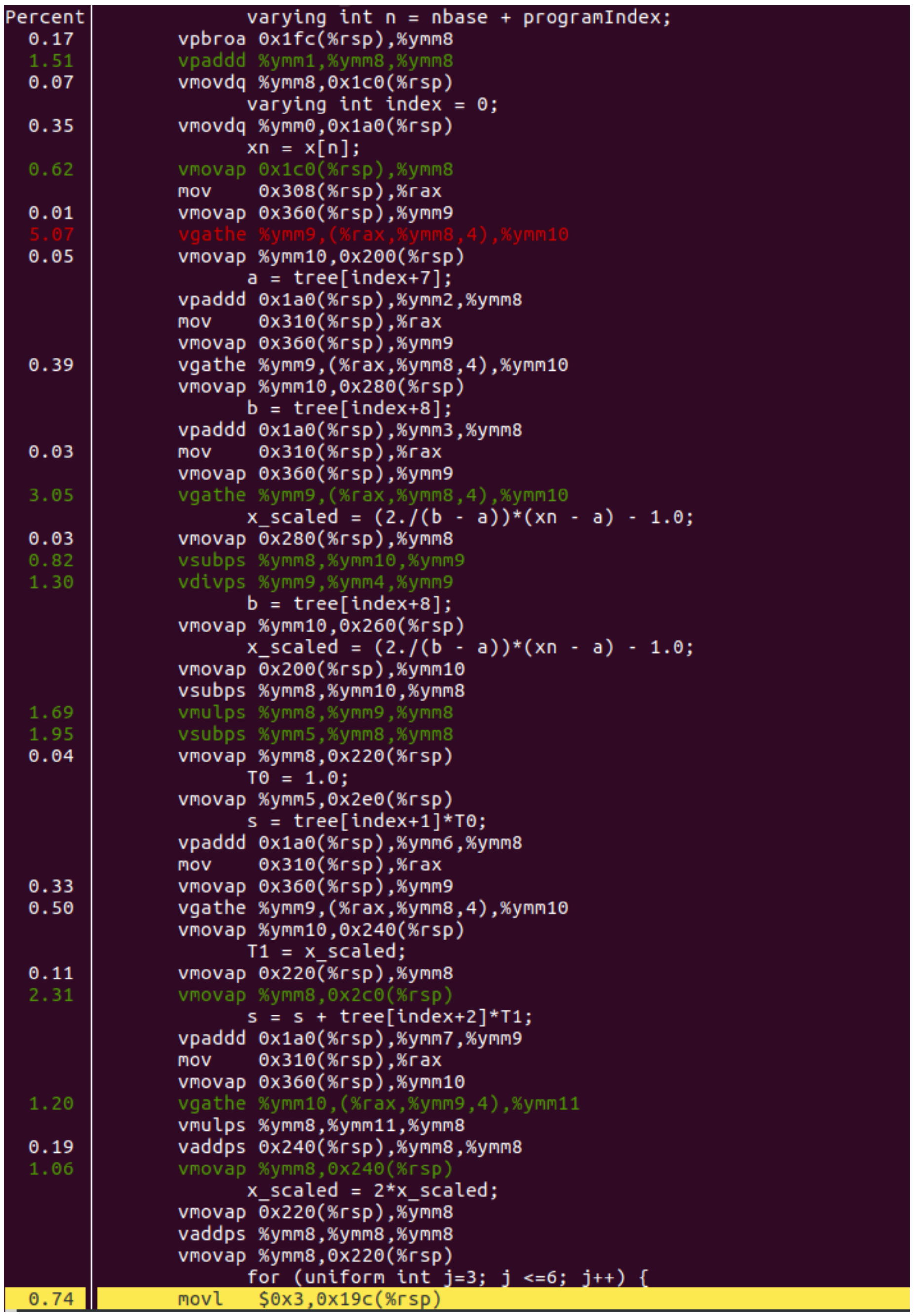
Future research will focus on further characterization of the generated code, including scaling with input size, error tolerance, and order. Relative performance of opencl special functions and these generated special functions can also be measured and used as motivation for integration into the framework of a partial differential equation solving packaged.

6 Appendix

6.1 Analysis with Perf

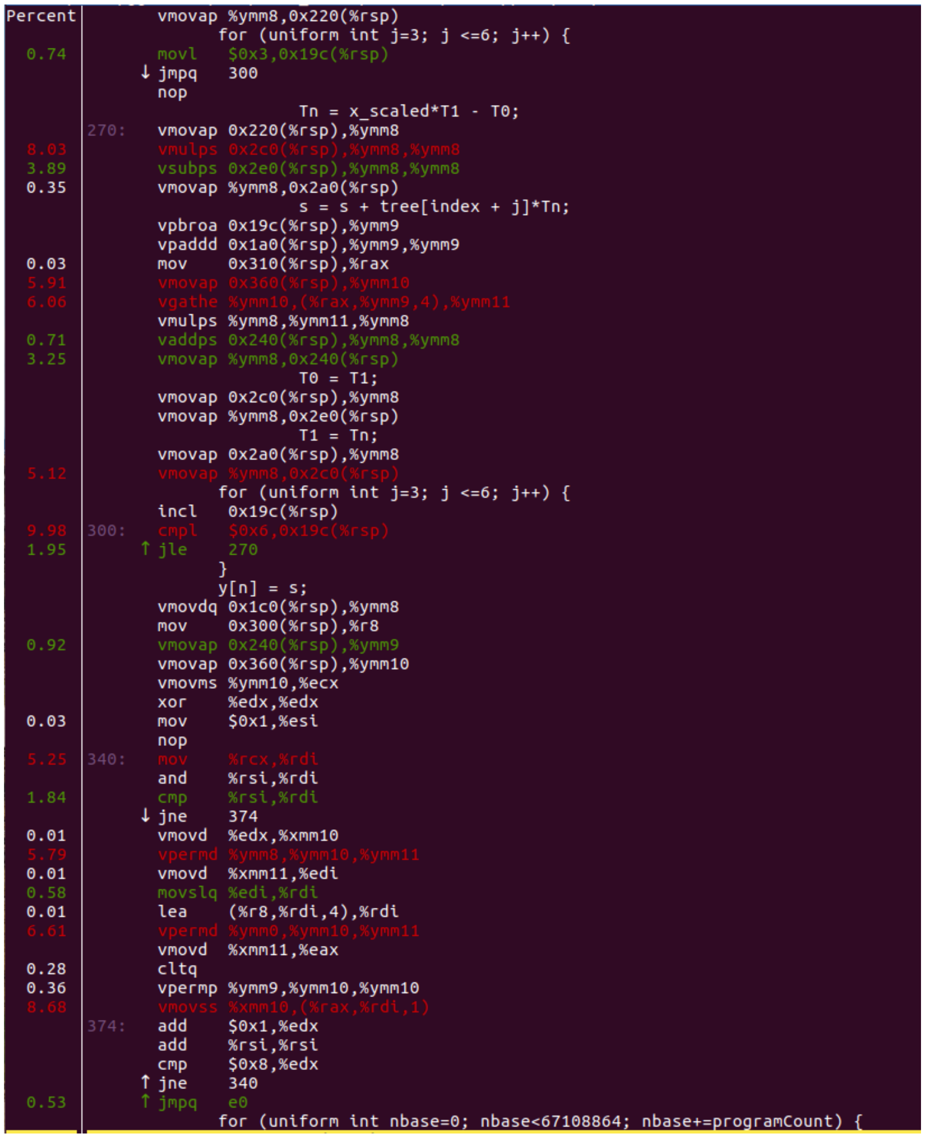
The following commands were run to obtain the perf report. This was called using a helper C++ file. From the images, one could see the that within the actual evaluation of the approximation, $1.51 + .82 + 1.30 + 1.69 + 1.95 + 8.03 + 3.89 + .71 = 19.9$ percent of the cycles were spent on floating point operations. However, this is not a completely accurate representation, due to skid, which causes instructions to be counted when they should not have been [1]. The instructions used to run perf are below. The -c flag is used to specify the sampling period used [1]. The overall runtime of the program was approximately 1 second, as shown in table 1. The report of the sections of code which used the largest percentage of cycles in the code is shown in the images below.

```
Perf record -c 2000 ./evaluator
Perf report
```



```
Percent                    varying int n = nbase + programIndex;
  0.17         vpbroa 0x1fc(%rsp),%ymm8
  1.51         vpaddd %ymm1,%ymm8,%ymm8
  0.07         vmovdq %ymm8,0x1c0(%rsp)
                     varying int index = 0;
  0.35         vmovdq %ymm0,0x1a0(%rsp)
                     xn = x[n];
  0.62         vmovap 0x1c0(%rsp),%ymm8
               mov    0x308(%rsp),%rax
  0.01         vmovap 0x360(%rsp),%ymm9
  5.07         vgathe %ymm9,(%rax,%ymm8,4),%ymm10
  0.05         vmovap %ymm10,0x200(%rsp)
                     a = tree[index+7];
               vpaddd 0x1a0(%rsp),%ymm2,%ymm8
               mov    0x310(%rsp),%rax
               vmovap 0x360(%rsp),%ymm9
  0.39         vgathe %ymm9,(%rax,%ymm8,4),%ymm10
               vmovap %ymm10,0x280(%rsp)
                     b = tree[index+8];
               vpaddd 0x1a0(%rsp),%ymm3,%ymm8
  0.03         mov    0x310(%rsp),%rax
               vmovap 0x360(%rsp),%ymm9
  3.05         vgathe %ymm9,(%rax,%ymm8,4),%ymm10
                     x_scaled = (2./(b - a))*(xn - a) - 1.0;
  0.03         vmovap 0x280(%rsp),%ymm8
  0.82         vsubps %ymm8,%ymm10,%ymm9
  1.30         vdivps %ymm9,%ymm4,%ymm9
                     b = tree[index+8];
               vmovap %ymm10,0x260(%rsp)
                     x_scaled = (2./(b - a))*(xn - a) - 1.0;
               vmovap 0x200(%rsp),%ymm10
               vsubps %ymm8,%ymm10,%ymm8
  1.69         vmulps %ymm8,%ymm9,%ymm8
  1.95         vsubps %ymm5,%ymm8,%ymm8
  0.04         vmovap %ymm8,0x220(%rsp)
                     T0 = 1.0;
               vmovap %ymm5,0x2e0(%rsp)
                     s = tree[index+1]*T0;
               vpaddd 0x1a0(%rsp),%ymm6,%ymm8
               mov    0x310(%rsp),%rax
  0.33         vmovap 0x360(%rsp),%ymm9
  0.50         vgathe %ymm9,(%rax,%ymm8,4),%ymm10
               vmovap %ymm10,0x240(%rsp)
                     T1 = x_scaled;
  0.11         vmovap 0x220(%rsp),%ymm8
  2.31         vmovap %ymm8,0x2c0(%rsp)
                     s = s + tree[index+2]*T1;
               vpaddd 0x1a0(%rsp),%ymm7,%ymm9
               mov    0x310(%rsp),%rax
               vmovap 0x360(%rsp),%ymm10
  1.20         vgathe %ymm10,(%rax,%ymm9,4),%ymm11
               vmulps %ymm8,%ymm11,%ymm8
  0.19         vaddps 0x240(%rsp),%ymm8,%ymm8
  1.06         vmovap %ymm8,0x240(%rsp)
                     x_scaled = 2*x_scaled;
               vmovap 0x220(%rsp),%ymm8
               vaddps %ymm8,%ymm8,%ymm8
               vmovap %ymm8,0x220(%rsp)
                     for (uniform int j=3; j <=6; j++) {
  0.74         movl   $0x3,0x19c(%rsp)
```

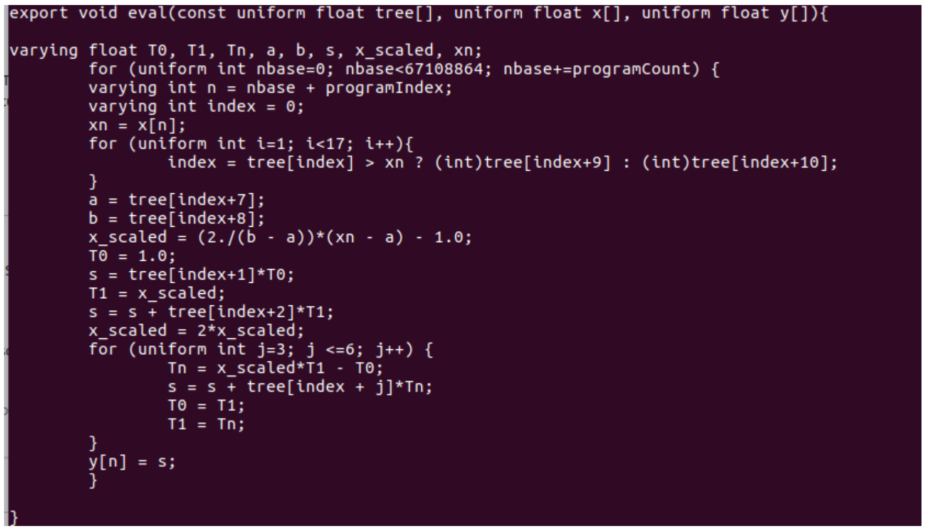
Figure 2: Cycles spent on each instruction in the code. Generated code for this image is the Polynomial Evaluation section of the hash table approximation which uses single precision numbers. It had an error tolerance set to 10^{-6}



```
Percent│       vmovap %ymm8,0x220(%rsp)
       │             for (uniform int j=3; j <=6; j++) {
  0.74 │       movl   $0x3,0x19c(%rsp)
       │     ↓ jmpq   300
       │       nop
       │                     Tn = x_scaled*T1 - T0;
       │270:   vmovap 0x220(%rsp),%ymm8
  8.03 │       vmulps 0x2c0(%rsp),%ymm8,%ymm8
  3.89 │       vsubps 0x2e0(%rsp),%ymm8,%ymm8
  0.35 │       vmovap %ymm8,0x2a0(%rsp)
       │                     s = s + tree[index + j]*Tn;
       │       vpbroa 0x19c(%rsp),%ymm9
       │       vpaddd 0x1a0(%rsp),%ymm9,%ymm9
  0.03 │       mov    0x310(%rsp),%rax
  5.91 │       vmovap 0x360(%rsp),%ymm10
  6.06 │       vgathe %ymm10,(%rax,%ymm9,4),%ymm11
       │       vmulps %ymm8,%ymm11,%ymm8
  0.71 │       vaddps 0x240(%rsp),%ymm8,%ymm8
  3.25 │       vmovap %ymm8,0x240(%rsp)
       │                     T0 = T1;
       │       vmovap 0x2c0(%rsp),%ymm8
       │       vmovap %ymm8,0x2e0(%rsp)
       │                     T1 = Tn;
       │       vmovap 0x2a0(%rsp),%ymm8
  5.12 │       vmovap %ymm8,0x2c0(%rsp)
       │             for (uniform int j=3; j <=6; j++) {
       │       incl   0x19c(%rsp)
  9.98 │300:   cmpl   $0x6,0x19c(%rsp)
  1.95 │     ↑ jle    270
       │             }
       │             y[n] = s;
       │       vmovdq 0x1c0(%rsp),%ymm8
       │       mov    0x300(%rsp),%r8
  0.92 │       vmovap 0x240(%rsp),%ymm9
       │       vmovap 0x360(%rsp),%ymm10
       │       vmovms %ymm10,%ecx
       │       xor    %edx,%edx
  0.03 │       mov    $0x1,%esi
       │       nop
  5.25 │340:   mov    %rcx,%rdi
       │       and    %rsi,%rdi
  1.84 │       cmp    %rsi,%rdi
       │     ↓ jne    374
  0.01 │       vmovd  %edx,%xmm10
  5.79 │       vpermd %ymm8,%ymm10,%ymm11
  0.01 │       vmovd  %xmm11,%edi
  0.58 │       movslq %edi,%rdi
  0.01 │       lea    (%r8,%rdi,4),%rdi
  6.61 │       vpermd %ymm0,%ymm10,%ymm11
       │       vmovd  %xmm11,%eax
  0.28 │       cltq
  0.36 │       vpermp %ymm9,%ymm10,%ymm10
  8.68 │       vmovss %xmm10,(%rax,%rdi,1)
       │374:   add    $0x1,%edx
       │       add    %rsi,%rsi
       │       cmp    $0x8,%edx
       │     ↑ jne    340
  0.53 │     ↑ jmpq   e0
       │             for (uniform int nbase=0; nbase<67108864; nbase+=programCount) {
```

Figure 3: Cycles spent on each instruction in the code. Generated code in this is the Polynomial Evaluation section of the hash table approximation which uses single precision numbers and a hash table. It had an error tolerance set to 10^{-6}

6.2 The Generated Code

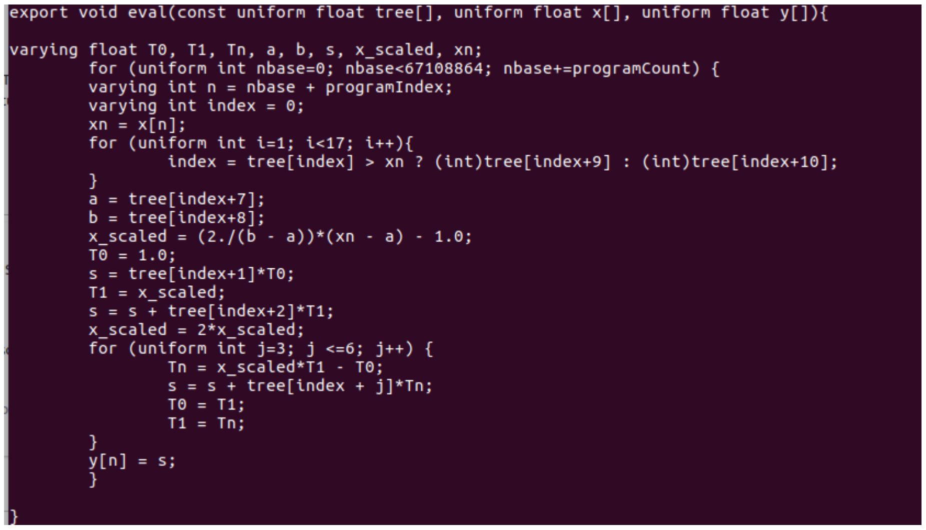


```
export void eval(const uniform float tree[], uniform float x[], uniform float y[]){

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;
        }

}
```

Figure 4: This is an image of the generated ISPC code that evaluates the approximation with an error less than 10^{-6} using single precision numbers and a binary search.



```
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;
        }

}
```

Figure 5: This is an image of the generated ISPC code that evaluates the approximation with an error less than 10^{-6} using single precision numbers and a hash table.

References

[1] Perf tutorial. Available at `https://perf.wiki.kernel.org/index.php/Tutorial`.

[2] Cody, W. The funpack package of special function subroutines. *ACM Transactions on Mathematical Software (TOMS) 1*, 1 (1975), 13–25.

[3] Intel. Specifications for intel® xeon® processor e5-2650 v4. Available at `https://ark.intel.com/products/91767/Intel-Xeon-Processor-E5-2650-v4-30M-Cache-2_20-GHz` (2005/06/12).

[4] Selesnick, I. Lecture notes for el 713: Digital signal processing fir filter, design with the remez algorithm, 2016.

[5] Smith, S. J. Lebesgue constants in polynomial interpolation. In *Annales Mathematicae et Informaticae* (2006), vol. 33, Eszterházy Károly College, Institute of Mathematics and Computer Science, pp. 1787–5021.

[6] TASISSA, A. Function approximation and the remez algorithm.