Bob is right (per usual)

Stewart A. Levin

ABSTRACT

Bob who? —— Bob Clapp. ..... About what? —— Table lookups. When reusable calculations are expensive, it can be advantageous to precalculate results over relevant parameter ranges and subsequently use table lookups to speed up programs that use such calculations repeatedly. Classically, sin/cos tables for fast Fourier transforms have been used to good effect. During recent development of a 3D downward continuation code, I found a range covering over an order of magnitude in the speed of sin/cos computations, with only nearest neighbor table lookup approximation outperforming Intel optimized vector routines.

INTRODUCTION

In October, 2013, I attended a ProMAX^{TM} Users Group meeting in Denver. A presenter at the meeting, Fritz Foss, now a coauthor of our upcoming SEG talk on downward continuation of Mars orbiter radar data, handed out "Martian" glasses and fascinated the audience with a description of how he was working on applying 3D poststack seismic imaging to a collection of radar tracks that had been acquired over the Martian polar icecaps. At the end of his talk, he plead for a way to avoid migrating the 90+% of zeros in his data that corresponded to the 300 km gap between the surface of Mars and the satellite's orbit. I volunteered and a collaboration ensued.

Over the months that followed, I got a slow out-of-core version working properly, but we needed it to be an order of magnitude faster to significantly outperform constant-velocity Stolt migration with the full length traces that included all the zeros. This I did achieve with resort to parallel I/O routines, but along the way I also looked at speeding up the computational downward continuation kernel. The results surprised me and may well surprise you too.

CHASING SPEEDUPS

The original downward continuation kernel was written in C, used the sin() and cos() routines from the -lm math library, and was compiled with gcc version 4.1.2 using -O3 optimization. As the version of ProMAX we were using had been mostly compiled with Intel's icc and ifort compilers, I managed to rebuild my downward continuation

executable with icc version 11.1 using the -fast option. The executable did run faster, though runtime was, as anticipated, still dominated by I/O, not computational speed.

As phase shift downward continuation relies on many evaluations of $\exp(2\pi iq)$ for a limited range of shifts q, it made sense to see if table lookup would provide a performance boost comparable to that seen in fast Fourier transform codes. (Wilcox et al. (2014)) Digging further, I found that ProMAX has a fastcos.h include file that provides macros for sin, cos and a (cos, sin) pair using table lookups. Further digging uncovered the sincos() routine in the GNU math library to produce (sin, cos) pairs.

To assess the various options, including the gcc versus the icc compiler, I wrote the test harness shown in Appendix A. The test carefully excluded parallel loop execution but biased my comparisons slightly in favor of table lookup by permitting the compiler to generate SSE (short vector) instructions to retrieve two floating point table entries, i.e. a sin/cos pair, at a go. The length of the table, that is, the number of pretabulated sin/cos pairs was set at 65536 which is 256 Kb.

Each option was timed for 2 billion random arguments resulting in the timings show in Table 1. In all timings, the minimum elapsed time of 6 trials was used, sufficient to get a reasonable estimate despite running on a computer shared with other users.

sin/cos timings	
Experiment	Δ time (seconds)
gcc -O3 table lookup	22.5
icc -fast table lookup	29.9
gcc -O3 -lm sinf() & cosf()[1]	136.9
gcc -O3 -lm sincosf()	114.8
icc -fast sinf() & cosf()	65.4
icc -fast sincosf()	65.4

Table 1: Elapsed times for 2×10^9 evaluations of single precision sin() and cos() for arguments randomly generated between $-\pi$ and π.

In reality, arguments to sin() and cos() in F-K methods are not randomly distributed but run fairly sequentially. So I retimed each option for 2 billion arguments in increasing order resulting in the timings show in Table 2.

In some ways, these comparisons are unfair. The table lookups are only approximate with a maximum error equivalent to nearly 1/100th of a degree of arc. A fairer comparison is to use slower linear interpolation to reduce the error by another order of magnitude. Timings for this case are given in Tables 3 and 4.

In practice, a well written F-K code will likely arrange for arrays of arguments to sin() and cos(), not just scalars. In this setting, Intel's Math Kernel Library (MKL)

[1]These are the single precision versions.

sin/cos timings	
Experiment	Δ time (seconds)
gcc -O3 table lookup	22.1
icc -fast table lookup	22.0
gcc -O3 -lm sinf() & cosf()	118.6
gcc -O3 -lm sincosf()	96.0
icc -fast sinf() & cosf()	63.6
icc -fast sincosf()	59.4

Table 2: Elapsed times for 2×10^9 evaluations of single precision sin() and cos() for arguments linearly generated between $-\pi$ and π.

sin/cos timings	
Experiment	Δ time (seconds)
gcc -O3 table lookup	35.2
icc -fast table lookup	31.4
gcc -O3 -lm sinf() & cosf()	137.1
gcc -O3 -lm sincosf()	114.3
icc -fast sinf() & cosf()	63.4
icc -fast sincosf()	63.3

Table 3: Elapsed times for 2×10^9 evaluations of single precision sin() and cos() for arguments randomly generated between $-\pi$ and π using linear table interpolation.

provides optimized vector routines, e.g. vsSinCos(), to calculate the sin() and cos() of an array. Of course timings for this option must be compared against array-valued lookups in the sin/cos table. These results are summarized in Table 5. Timings are only shown for the sequential access pattern as, like the scalar icc timings, the times were virtually the same for random argument sequences.

MAKING SENSE OF IT ALL

With a factor of over 20 between the fastest and slowest of the options, some explanations are in order. Certainly one observation is that the Intel icc compiler creates more optimized code that the GNU gcc compiler.

Looking at the GNU math library source code (available for download from `www.gnu.org/software/libc/download.html`), the sin() and cos() calculations involve a careful reduction of the argument to a range between $-\pi/4$ and $\pi/4$, followed by a 13^{th} or 14^{th} order polynomial evaluation involving six tabulated coefficients.

The Intel basic math library cosf() and sinf() routines call an internal library routine __libm_sse2_sincos() which, as its name implies, takes full advantage of the

sin/cos timings	
Experiment	Δ time (seconds)
gcc -O3 table lookup	30.9
icc -fast table lookup	17.8
gcc -O3 -lm sinf() & cosf()	109.7
gcc -O3 -lm sincosf()	86.5
icc -fast sinf() & cosf()	57.6
icc -fast sincosf()	56.6

Table 4: Elapsed times for 2×10^9 evaluations of single precision sin() and cos() for arguments linearly generated between $-\pi$ and π using linear table interpolation.

sin/cos timings	
Experiment	Δ time (seconds)
icc -fast nearest neighbor table lookup	6.2
icc -fast linear interpolated table lookup	40.0
icc -fast sincosf()	16.9

Table 5: Elapsed times for 2×10^9 evaluations of single precision sin() and cos() using MKL and table lookup for arguments evaluated 1000 at a time. Random and sequential argument ordering yielded virtually identical table evaluation runtimes.

short vector SSE instruction and processor instruction-level parallelism. I understand that it, too, evaluates a similar polynomial.

sin() and cos() versus sincos()

The near equality in the Intel compiler and runtime results comparing sincos() with evaluating sin() and cos() separately is striking because the sincos() routine is passed memory addresses for the outputs and so would store the outputs into memory. This tells me that the Intel compiler -fast option has rolled sincos() inline into my test code, allowing the compiler to keep results in registers.

The GNU compiler and math library yield a small improvement in runtime when using the dual sincos() routine versus evaluating sin() and cos() separately. Since evaluating the sin() and cos() separately takes 118.6, each separately takes 118.6/2=59.3. Assuming the sincos() routine only evaluates one and uses sqrt() to get the other, we should expect a timing of about 59.3[sin]+ϵ[cos]+[memory stores] which says that the memory stores cost about half as much as the trigonometric evaluations.

sin() and cos() versus table lookup

The GNU compiler generates table lookups that are about the same speed as those generated by the Intel compiler. Still, this saves about a factor of 3 to 4 compared to separate or joint evaluations of sin() and cos() with their -lm math library. The penalty is a loss of accuracy with a maximum phase error of 0.0028 degrees. (For linear interpolation, the maximim phase error is 0.0000051 degrees.)

With the Intel icc compiler, table lookup was also the winner, by a factor of 2 to 3, in almost all cases. The one exception was, however, an important one: vectorized sincos() versus vectorized table lookup with linear interpolation. The reason linear interpolation is important to consider is twofold:

- The longer the trace, the more small errors in sin/cos approximation degrades the result of phase rotation.
- The 65536 entry sin/cos table I used for these tests was much larger than the 32K fast L1 cache on the machine and, indeed, was exactly the size of the slower 256K L2 cache. Pushing for higher accuracy by using an even more refined table incurs a significant memory performance hit, whereas linear interpolation allows higher accuracy with even shorter tables.

CONCLUSIONS

So what has Bob got to do with this study? When I presented early, albeit incorrect, timing results, he added that processor speeds have continued to grow faster than memory access speeds. Where once it made excellent sense to precompute and tabulate various mathematical functions, the computational load for such a function needed to benefit from tabulation is now sufficiently high that simple trigonometric functions no longer automatically qualify. Bob Clapp emphasized that this trend is accelerating and one will soon need thousands of operations in any function to justify precomputation. In another recent seminar, Bob further made the point that the tradeoffs are even worse, by orders of magnitude, when comparing precomputed migration-inversion operator tables with on-the-fly recomputation. So, in conclusion, Bob is right (per usual).

APPENDIX A

Computer code for timing tests.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#ifdef DOTBLFAST
#define nolinux
#endif
#include "fastcos.h"

int main(int argc, char **argv)
{
  float x=0.5f;
  float sx=0.2f, cx=-0.24f;
  float ex, fx;
  float xscale = 0.999f*M_PI;
  size_t i;
  size_t n=2000000000;

    allocfastcos();
    ex = rand()/(2000000.0f+RAND_MAX);
    for(i=0; i<n; ++i) {
       fx = 1.0f - ex;
#if defined(DORANDACCESS)
       x=(float)(xscale*(fx*(rand()/((double) RAND_MAX))+
                                      ex*(i/((double) n))));
#endif
#if defined(DOSEQACCESS)
       x=(float)(xscale*(ex*(rand()/((double) RAND_MAX))+
                                      fx*(i/((double) n))));
#endif
       ex *= (0.25f*x);
#if defined(DOTBLFAST) || defined(DOTBLSLOW)
       FASTCOSSIN(x,cx,sx);
       ex *= (cx*sx);
#endif
#if defined(DOSINCOSF)
       sincosf(x,&sx,&cx);
       ex *= (cx*sx);
#endif
#if defined(DOSINFCOSF)
       ex *= (cosf(x)*sinf(x));
#endif
    }

/* force loop evaluation by printing out final values */
    printf("x=%g cos(x)=%g sin(x)=%g\n",x,cos(x),sin(x));
#if defined(DOTBLFAST) || defined(DOTBLSLOW)
    printf("x=%g fastcos(x)=%g fastsin(x)=%g\n",x,cx,sx);
```

```
#endif
    printf("ex=%g\n",ex);

    return EXIT_SUCCESS;
}
```

REFERENCES

Wilcox, C., M. M. Strout, and J. M. Bieman, 2014, An optimization-based approach to lookup table program transformations: Journal of Software: Evolution and Process, **26**, 533–551.