Universal SIMD-Mathlibrary

Helmut Dersch
Furtwangen University of Applied Sciences

August 20, 2008

Abstract

Standard functions for single precision floating point vector datatypes are provided for the SIMD-platforms x86 (SSE2), PowerPC and Cell. In most cases, speed and/or accuracy compare favourable with existing SIMD-libraries (MacOS Accelerate Framework, Cell SDK). Most of the algorithms are based on those of the Cephes library, while the implementation is branch-free and parallelized for minimum pipeline stalls. The Universal SIMD Mathlibrary (usm) provides the functions sin, cos, tan, asin, acos, atan, atan2, sqrt, exp, log, pow, abs, ceil, floor, ldexp, and frexp. It is licensed under the GPL3.

Introduction

For general x86-linux I could not find a full implementation of the standard C mathfunctions for floating point vectors, which I needed to port my fast panorama stitcher and blender *PTStitcherNG* [1]. Originally developed for the Cellprocessor of the Sony Playstation 3, I had added a port to MacOS. Both platforms provide builtin support for SIMD-mathfunctions. To unify the codebase of my original project, I wrote versions for all processors, which resulted in this library with a common C-programming interface.

For a general introduction to SIMD see the entry in Wikipedia. This article and software only deal with vectors consisting of four single precision floating point numbers ($4 \cdot 32$ bits = 128 bits). Execution time and accuracy do not depend on the number of floating point elements in the vector; however, I am not aware of existing platforms with more than 4.

Test Platforms

This article is aimed at prospective users so it starts with testresults. For the tests I used all SIMD-capable computers available to me. They coincidentally cover the supported platforms:

- Mac mini (Intel Core 2 Duo, SSE3, 1.83 GHz), MacOS 10.5.4
- Sony Playstation 3 (IBM Cell Processor, 3.2GHz), Yellow Dog Linux 6.
- Asus EeePC (Intel Celeron M, SSE2, 630MHz), Xandros Linux

In the following we compare a total of 11 SIMD-libraries running on these platforms:

- Universal SIMD Mathlibrary (*usm*), compiled for the respective processor.
- Vector Library (*mac*) as part of the MacOS Accelerate Framework.
- Simdmath library as part of the Cell-Development Kit [3]. There are different versions for Altivec instructions (*ps3/ppu*) and the synergestic processing unit (*ps3/spu*).
- The standard scalar math library (*libm*) distributed with the respective operating system. For each float-function `func` a vectorized version `func4` has been compiled, which calculates the `func`-value of the four vector elements.

Test Results

Speed is measured by executing each function 10^6 to 10^7 times on equally spaced values within the function dependend test range. See the source code for details about the test method. In each case, non-inlined versions of the functions are used. The results are provided as time-per-function-evaluation in nanoseconds. Significantly faster (by more than 30%) results are marked in bold. The table speaks for itself.

A few additional notes: the MacOS standard mathlibrary libm is quite fast, and in some cases outperforms the vector libraries (pow and acos, emphasized in the table). See the special note about the pow-function below.

	Core 2 Duo			Cell/PowerPC			Cell/SPU			Celeron M	
	usm	mac	libm	usm	ps3	libm	usm	ps3	libm	usm	libm
sin	**31**	45	74	56	60	905	**30**	43	143	134	750
cos	**31**	46	79	73	61	908	**31**	43	155	148	760
tan	**38**	68	105	**73**	134	786	**40**	59	206	197	1072
asin	36	37	59	**61**	131	750	39	49	347	155	1816
acos	62	67	*57*	121	137	822	57	62	402	275	1806
atan	**27**	40	87	41	45	488	**25**	34	166	169	1078
atan2	50	62	172	**78**	n/a	1142	**43**	n/a	242	289	1222
sqrt	10	10	56	19	20	39	**19**	34	27	70	495
exp	**31**	48	83	40	49	1220	**36**	52	195	155	1550
log	**41**	63	90	46	48	825	29	36	245	140	1094
pow	**93**/ 209	221	*201*	85	96	3337	61	53	255	346	2675

Table 1: Average time (in *ns*) to perform one SIMD-function execution. Bold numbers mark significantly (difference $\geq 30\%$) better results for the respective processor.

Relative errors are calculated by comparing the function results to the results of the corresponding double precision functions, which are assumed to be exact. Again, a large number of function evaluations in the specified testrange are gathered and analyzed. Two types of relative errors are evaluated: Averaged over all data (root-mean-square, RMS) and the maximum error (peak).

Floating point math accuracy is specified by the constant `FLT_EPSILON` which is defined as the smallest number satisfying the relation

$$1.0f + \texttt{FLT\_EPSILON} > 1.0f$$

This constant is $1.19209 \cdot 10^{-7}$ for all tested platforms, and defines an upper limit for the relative error of any floating point operation. For easy comparison all results are normalized to this constant.

Starting with the average-(RMS)-error: Depending on the testrange, the theoretical optimum for a single precision floating point implementation of a contineous function is typically around 0.2 (exceptions like constants etc excluded). The standard mathlibrary of x86-linux (glibc) really shines here in providing this optimum for all functions of the testsuite (see Celeron M/libm column). The libm-results for MacOS and PPU are also close to the optimum, while the SPU-libm

	Core 2 Duo			Cell/PowerPC			Cell/SPU			Celeron M	
	usm	mac	libm	usm	ps3	libm	usm	ps3	libm	usm	libm
sin	0.24	0.24	0.22	**0.27**	1.58	0.22	**0.57**	1.33	1.33	0.24	0.19
cos	0.25	0.25	0.22	0.28	0.33	0.22	**0.60**	1.04	1.04	0.25	0.19
tan	0.35	0.38	0.22	**0.35**	0.47	0.22	0.68	0.66	0.66	0.35	0.21
asin	0.43	0.42	0.26	0.47	**0.27**	0.22	0.49	0.54	0.54	0.43	0.21
acos	0.34	0.34	0.27	**0.37**	0.54	0.23	0.31	0.36	0.36	0.34	0.22
atan	0.34	0.35	0.22	0.40	0.45	0.22	**1.12**	$> 10^6$	1.58	0.34	0.21
atan2	0.36	0.36	0.22	**0.38**	n/a	0.26	**1.41**	n/a	0.70	0.36	0.22
sqrt	0.20	0.20	0.20	**0.31**	1.00	0.20	0.38	0.40	0.26	0.20	0.20
exp	0.23	0.23	0.21	**0.26**	0.44	0.25	0.35	0.43	0.75	0.23	0.21
log	0.19	0.19	0.19	**0.19**	0.39	0.19	**0.38**	1.62	1.56	0.19	0.19
pow	0.42/ 0.30	0.30	0.21	0.46	0.57	0.24	1.35	1.09	1.09	0.42	0.21

Table 2: Average relative error (root-mean-square) of SIMD-functions in units `FLT_EPSILON` ($1.19 \cdot 10^{-7}$). Theoretical lower limit is approximately 0.2.

is much worse. Most of the functions here are duplicates of the respective vector functions.

The vector libraries are generally less accurate. There is little difference beween usm and the MacOS-vector library. The usm implemetation on X86-Linux (Celeron) gives exactly the same results as the one on MacOS. On the PPU and SPU there are some cases with missing (atan2) or buggy (atan) implementations of standard functions in the builtin libraries, which are corrected by usm.

Peak relative error is also provided in units `FLT_EPSILON`, see last paragraph. Depending on the testrange, the theoretical optimum for a single precision floating point implementation of a contineous function is typically 0.5 (exceptions like constants etc excluded). As for the average error, the standard mathlibrary of x86-linux (glibc) provides this optimum for all functions of the testsuite. The libm-results for MacOS and PPU are both worse, while the SPU-libm is much worse.

The results for the vector libraries are similar to the results discussed for the average errors above.

Summing up the comparison: Compared with the Accelerate framework, the usm is faster for almost all functions (often significantly) while providing similar accuracy. Compared with the Cell-SDK functions, the usm is also faster in most

	Core 2 Duo			Cell/PowerPC			Cell/SPU			Celeron M	
	usm	mac	libm	usm	ps3	libm	usm	ps3	libm	usm	libm
sin	0.98	0.98	0.67	**0.96**	490	0.82	**1.84**	275	275	0.98	0.50
cos	0.94	0.94	0.65	**1.07**	4.25	0.82	**1.85**	57.0	57.0	0.94	0.50
tan	1.34	1.66	0.57	**1.38**	2.13	0.57	**1.86**	2.53	2.53	1.34	0.50
asin	2.23	2.33	0.76	2.92	**1.77**	0.79	2.50	**1.78**	1.78	2.23	0.50
acos	1.16	1.15	0.91	**1.53**	3.91	0.83	1.50	**1.03**	1.03	1.16	0.50
atan	1.61	1.77	0.63	1.50	1.94	0.66	**2.61**	$> 10^6$	3.07	1.61	0.50
atan2	1.77	2.23	0.63	**1.70**	n/a	1.00	**3.61**	n/a	2.77	1.77	0.50
sqrt	0.50	0.50	0.50	**1.42**	2.35	0.50	1.48	**1.00**	1.10	0.50	0.50
exp	0.60	0.60	0.50	**1.00**	2.13	0.50	1.03	1.03	2.75	0.60	0.50
log	0.64	0.64	0.49	**0.73**	56.2	0.64	**1.87**	102	63.1	0.64	0.49
pow	4.81/ 0.94	1.32	0.50	**4.42**	6.32	0.73	36.6	**14.4**	14.4	4.81	0.50

Table 3: Peak relative error of SIMD-functions in units `FLT_EPSILON` ($1.19 \cdot 10^{-7}$). Theoretical lower limit is approximately 0.5.

cases. The Cell-SDK functions exhibit a couple of serious errors (RMS-error: atan; peak-error: sin, cos, atan, log) and deficiencies (atan2), which are corrected in usm.

The results are also quite informative regarding platform comparisons. The speed of the Core 2 Duo is very similar to that of the SPU with the exception of the sqrt-function, which is an intrinsic on the SSE-platform. Accuracy is somewhat better for the Core 2 Duo. The seemingly poor performance of the Celeron M is in reality still quite impressive, and a few years ago would have qualified this machine as a workstation.

The libm for PowerPC is very slow, and for some functions slower than the Celeron M version, which are both supposedly based on the same glibc.

Usage

The Universal SIMD-Mathlibrary must be compiled with the GNU C-compiler, version ≥ 4.0. Edit the Makefile and set the `PROCESSOR`-variable to one of `SSE` (for all x86 platforms supporting at least the `SSE2` instruction set, which all current models do), `PPU` (for PowerPC and Cell/PPU supporting the Altivec instruction set) or `SPU` for the Cell's synergestic processing unit. Then type `make`. A version of the library `libsimdmath.a` and two test programs `test` (for the usm), and `tlibm` (for the standard math library libm) are created. On the Macintosh platform you can create an additional testprogram for the builtin vector library (`tlibvecm`) and on the Playstation 3 one for the Cell development kit (`tps3m`), edit the Makefile correspondingly.

To use the library in your program, include the header file `simdmath.h` and link to the library file `libsimdmath.a`. The header file defines the type `vec_float4` for all platforms. All function names are identical to their standard libmath-names, with the character 4 appended (eg `sinf4` for the vector sine function).

Inlined versions of all (and some additional) functions are accessible by including the respective header files. The inline function names contain leading underscores. E.g., to use an inlined version of `sinf4`, include the header file `sincosf4.h` and use the function name `_sinf4`. A few useful additional functions are available as inlined versions only: `_sincosf4`, which calculates sine and cosine of the same argument together as fast as each of them individually, and some alternative test versions of other functions.

Implementation

The base algorithms in most cases are those used in the Cephes [2] library, see the sources for details. Basically, they all consist of three steps

- The argument is reduced into a suitable range with the help of some function specific theorems.
- A Taylor-like approximating polynomial is applied, whose coefficients are carefully optimized.
- The argument reduction step is reversed.

The algorithms are implemented in a branch-free manner and further optimized.

Branches like

```
y = a ? f1(x) : f2(x);
```

are replaced by something similar to

```
y = a & f1(x) | ~a & f2(x);
```

This allows the same code to create several function values at once. The apparent disadvantage of evaluating both cases (f1(x) and f2(x)) is more than offset by the avoidance of pipeline stalls, which makes this version favourable also for scalar functions.

The GNU C-compiler provides standard C-like operators for vector types (`*`, `+`, `-`, `/`), and my original plan was to use these to build a portable C-like implementation, and let the compiler create platform specific code. Unfortunately, this did not work well. While a few missing C-operators (shift, comparison) can be easily provided with C-macros, the differences in processors requires different optimization strategies:

- The ppc and cell can not divide fast. Algorithms with quotients are avoided or treated specially. A different algorithms for atan [4] had to be choosen for these processors.
- ppc and cell processors stall when evaluating long Horner-polynomials:

  ```
  y = (((a4*x+a3)*x+a2)*x+a1)*x+a0;
  ```

  Therefore, these are broken into quadratic factors:

  ```
  y = ((c2*x+c1)*x+c0) * ((x+d1)*x+d0);
  ```

  which can be calculated separately. This version is faster on the cell, but slower on the x86 platform, so I had to provide separate versions. This also explains part of the difference in accuracy.
- Creating vector constants on-the-fly is faster on the cell than loading from memory, and vice-versa on sse-platforms.

Some functions (asin,acos) have not yet been optimized for ppu/spu.

Sqrt is an intrinsic on the SSE-platform, and thus reduces to a single instruction. The implementation for ppu/spu is based on the inverse-square-root estimate intrinsic of these platforms. This instruction is combined with one iteration of the Newton-Raphson algorithm to provide the final result.

The basic pow-function in usm calculates the formula

```
pow(x,y) = exp( log(x)*y);
```

with special treatment of the x=0 case. There is an alternative, more accurate pow-function following the Cephes [2] algorithm, which can be accessed by undefining `FAST_POWF4` in the header file `<powf4.h>`. It is listed in the tables as second entry for pow in the Core-2-Duo/usm column. It roughly provides the slightly improved performance of the mac-pow function. But both are slower and less accurate than the scalar libm-version. This slow (and normally disabled) pow-function is the only usm-function which due to its complexity uses branches.

On the SPU it is not possible to specially treat the x=0 case in pow in a branch-free manner. Therefor, all pow implementations (usm, ps3 and libm) create `NaN` for x=0 on this platform.

The spu truncates results of floating point operations. Its theoretical and practical peak and RMS errors are therefor larger than for the other processors given the same algorithm. In some cases these can be compensated in the algorithm: The exp and sqrt implementations of usm are much simpler (and thus faster) than the ps3 versions, but they provide the same (even somewhat lower) rms-error due to rounding correction.

Behaviour of all functions outside their intended variable range is undefined.

References

[1] PTStitcherNG `http://www.fh-furtwangen.de/~dersch`

[2] Cephes Math Library Release 2.2: June, 1992, by Stephen L. Moshier `http://www.netlib.org/cephes/`

[3] Cell development kit from `http://www-128.ibm.com/developerworks/power/cell/`

[4] B.Carlson, M.Goldstein, Los Alamos Scientific Laboratory 1955