Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy
(Revisiting Iterative Refinement for Linear Systems)

Julie Langou[1], Julien Langou[1], Piotr Luszczek[1], Jakub Kurzak[1], Alfredo Buttari[1], and Jack Dongarra[1]

University of Tennessee, Knoxville TN 37996, USA,
`julie,langou,luszczek,kurzak,buttari,dongarra@cs.utk.edu`

Extended abstract: more information, data, codes available at

`http://www.cs.utk.edu/~julie/iter-ref/`

Abstract. Recent versions of microprocessors exhibit performance characteristics for 32 bit floating point arithmetic (single precision) that is substantially higher than 64 bit floating point arithmetic (double precision). Examples include the Intels Pentium IV and M processors, AMDs Opteron architectures and the IBMs Cell processor. When working in single precision, floating point operations can be performed up to two times faster on the Pentium and up to ten times faster on the Cell over double precision. The performance enhancements in these architectures are derived by accessing extensions to the basic architecture, such as SSE2 in the case of the Pentium and the vector functions on the IBM Cell. The motivation for this paper is to exploit single precision operations whenever possible and resort to double precision at critical stages while attempting to provide the full double precision results. The results described here are fairly general and can be applied to various problems in linear algebra such as solving large sparse systems, using direct or iterative methods and some eigenvalue problems. There are limitations to the success of this process, such as when the conditioning of the problem exceeds the reciprocal of the accuracy of the single precision computations. In that case the double precision algorithm should be used.

1 Introduction

The motivation behind this work is the observation that a number of recent processor architectures exhibit single precision performance that is significantly higher than for double precision arithmetic. An example of this include the IBM Cell multiprocessor which was announced with a theoretical peak of 204.8 GFLOPS in single precision (32 bit floating point arithmetic) and a peak of only 20 GFLOPS in double precision (64 bit floating point arithmetic). Even the Intel x87 processor with the use of the Streaming SIMD Extensions (SSE) unit on the Pentium III does 4 flops/cycle for single precision, and SSE2 does 2 flops/cycle for double. Therefore, for any processor with SSE and SSE2 (e.g. Pentium IV), the theoretical peak of single is twice that of double, and on a chip with SSE and without SSE2 (e.g. some Pentium III), the theoretical peak of single is four times that of double. AMD processors share the same relation between SSE and SSE2, the only difference being that their x87 units can do 2 flops/cycle for any precision. Appendix 1 contains additional information on the extensions to the IA-32 instruction set.

Another advantage of computing in single versus double precision is that data movement is cut in half. This helps performance by reducing memory traffic across the bus and enabling larger blocks of users data to fit into cache. In parallel computations, the total volume of communication is reduced by half and the number of initiated communication is reduced as well (if block sizes are doubled). The effect is that the communication behaves as if the bandwidth is multiplied by two and latency halved by two.

The use of extensions to the ISA of x86-x87 has been put into practice in a number of implementations of the BLAS. This provides a speed improvement of a factor of two in single precision compared to double precision for basic operations such as matrix multiply. Some experimental comparisons of SGEMM versus DGEMM on various architectures are given in Table 2.

The motivation for this paper is to exploit single precision operations whenever possible and resort to double precision at critical stages while attempting to provide the full double precision results.

2 Some Numerical Experiments

The first set of experiments show the performance of the sequential algorithm on a number of systems. In the third and fourth columns of Table 1, for each system, we report the ratio of the time to perform SGEMM (Single precision Matrix-Matrix multiply for GEneral matrices) over the time to perform DGEMM (Double precision Matrix-Matrix multiply for GEneral matrices) and the ratio of the time to perform SGETRF (Single precision LU Factorization for GEneral matrices) over the time to perform DGETRF (Double precision LU Factorization for GEneral matrices). As claimed in the introduction this ratio is often 2 (Katmai, Coppermine, Northwood, Prescott, Opteron, UltraSPARC, X1), which means single are twice as fast as double. Then in the fifth and sixth columns we report the results for DGSEV over DSGESV. The results from Table 1 show that this method can be very effective on a number, but not all, architectures. The Intel Pentium, AMD Opteron, Sun UltraSPARC, Cray X1, and IBM Power PC architectures, all exhibit a significant benefit from the use of single precision. Systems such as the Intel Itanium, SGI Octane, and IBM Power3 do not show the benefits.

It is to note that single precision computation is significantly slower than double precision computation on Intel Intanium 2.

The next set of experiments is for a parallel implementation along the lines of ScaLAPACK. In this case n is in general fairly large and, as we can observe in Table 2, the cost of the iterative refinement becomes negligible with respect

Architecture (BLAS)	n	DGEMMR /SGEMM	DGETRF /SGETRF	DGESV /DSGESV	# iter
Intel Pentium III Coppermine (Goto)	3500	2.10	2.24	1.92	4
Intel Pentium III Katmai (Goto)	3000	2.12	2.11	1.79	4
Sun UltraSPARC IIe (Sunperf)	3000	1.45	1.79	1.58	4
Intel Pentium IV Prescott (Goto)	4000	2.00	1.86	1.57	5
Intel Pentium IV-M Northwood (Goto)	4000	2.02	1.98	1.54	5
AMD Opteron (Goto)	4000	1.98	1.93	1.53	5
Cray X1 (libsci)	4000	1.68	1.54	1.38	7
IBM Power PC G5 (2.7 GHz) (VecLib)	5000	2.29	2.05	1.24	5
Compaq Alpha EV6 (CXML)	3000	0.99	1.08	1.01	4
IBM SP Power3 (ESSL)	3000	1.03	1.13	1.00	3
SGI Octane (ATLAS)	2000	1.08	1.13	0.91	4
Intel Itanium 2 (Goto and ATLAS)	1500	0.71			

Table 1. Ratio of execution times (speedup) for DGEMM/SGEMM ($m = n = k$), DGETRF/SGETRF and DGESV/DSGESV on various architectures, the number of iterations of iterative refinement in DSGESV is given in the last column.

to PDGETRF . Using PDSGESV is almost twice (1.83) as fast as opposed to using PDGESV for the same accuracy.

Architecture (BLAS-MPI)	# procs	n	PDGETRF /PSGETRF	PDGESV /PDSGESV	# iter
AMD Opteron (Goto OpenMPI MX)	32	22627	1.85	1.79	6
AMD Opteron (Goto OpenMPI MX)	64	32000	1.90	1.83	6

Table 2. Performance comparison between PDGETRF/PSGETRF and PDGESV/PDSGESV on an AMD Opteron cluster with Myrinet interconnects, the number of iterations of iterative refinement in PDSGESV is given in the last column.

As an extension to this study, we present in this section results for iterative refinement in quadruple precision on an Intel Xeon 3.2GHz. The iterative refinement code computes a condition number estimate; the input matrices are random matrices with uniform distribution. For quadruple precision arithmetic, we use the reference BLAS compiled with ifort -O3 the Intel Fortran compiler (with -O3 optimization flags on) since we do not have an optimized BLAS in quadruple precision. Results are presented in Table 3. The obtained accuracy is of about 10^{-32} for QGETRF and QDGETRF as expected. No more than 3 steps of iterative refinement are needed. The speedup goes from 10 for a matrix of size 100 to 100 for a matrix of size 1000. In Table 4, we give the time for the different kernels used in QGESV and QDGESV. Interestingly enough the time for QDGESV is dominated by QGEMV and not DGETRF!

n	QGESV time (s)	QDGESV time (s)	speedup
100	0.29	0.03	9.5
200	2.27	0.10	20.9
300	7.61	0.24	30.5
400	17.81	0.44	40.4
500	34.71	0.69	49.7
600	60.11	1.01	59.0
700	94.95	1.38	68.7
800	141.75	1.83	77.3
900	201.81	2.33	86.3
1000	276.94	2.92	94.8

Table 3. Iterative refinement in quadruple precision on a Intel Xeon 3.2GHz.

routine	time (s)	kernel name	time (s)
QGESV	201.81	QGETRF	201.1293
		QGETRS	0.6845
QDGESV	2.33	DGETRF	0.3200
		DGETRS	0.0127
		DLANGE	0.0042
		DGECON	0.0363
		ITERREF	1.9258

Table 4. Detailed of the time for the various operations involved in QDGESV and QGESV for a matrix of size $n = 900$.