NASA Contractor Report 178364

ICASE INTERIM REPORT 4

PERFORMANCE OF FORTRAN FLOATING-POINT OPERATIONS ON THE FLEX/32 MULTICOMPUTER

Thomas W. Crockett

NASA Contract No. NAS1-18107
August 1987

(NASA-CR-178364) PERFORMANCE OF FORTRAN FLOATING-POINT OPERATIONS ON THE FLEX/32 MULTICOMPUTER Final Report (NASA) 11 p
Avail: NTIS HC A02/MF A01 CSCL 09B

N87-30073

Unclas
G3/60 0102693

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographics, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

Performance of FORTRAN Floating-point Operations on the Flex/32 MultiComputer

Thomas W. Crockett

Institute for Computer Applications in Science and Engineering

ABSTRACT

A series of experiments have been run to examine the floating-point performance of FORTRAN programs on the Flex/32™ computer. The experiments are described, and the timing results are presented. The time required to execute a floating-point operation is found to vary considerably depending on a number of factors. One factor of particular interest from an algorithm design standpoint is the difference in speed between common memory accesses and local memory accesses. Common memory accesses were found to be slower, and guidelines are given for determining when it may be cost effective to copy data from common to local memory.

1. Introduction

A series of experiments have been run to determine the approximate execution times of FORTRAN floating-point operations on NASA Langley's Flex/32 MultiComputer. The results obtained are potentially useful for (1) comparing the performance of the Flex/32 with other computers, (2) developing execution time models for FORTRAN programs on the Flex/32, and (3) guiding algorithm design decisions.

Two basic benchmark programs were used. The first benchmark measured a very simple arithmetic statement which could be analyzed in some detail. The second benchmark was based on the Livermore Loops, a set of FORTRAN kernels often used to assess floating-point performance. Variations of these two programs were used to compare single and double precision performance, as well as the relative speeds of common and local memory. In addition, the first benchmark was run using different combinations of compilers and optimization options to compare the quality of code generated.

The two benchmark programs are described first, along with a description of the test conditions. The timing results are then presented, followed by an analysis and discussion.

2. Test Programs

2.1. Benchmark 1

The first test program was developed specifically for this study. It measures the unit of work which is given by the FORTRAN assignment statement

$$c(i) = a(i) \circ b(i)$$

which consists of some simple index operations, two operand fetches, a floating point operation, and a store of the result. *a*, *b*, and *c* are one dimensional arrays of length 30,000 elements each. "∘" represents the operator

This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

+, −, *, or /.

The *a* and *b* arrays are first initialized with random numbers in the range from -100,000 to +100,000. The above assignment statement is then embedded in a doubly-nested **do** loop. The inner loop iterates over every element in the array, while the outer loop serves to lengthen the overall duration of the test in order to minimize timing errors. The total elapsed execution time of the outer loop is measured, and the loop overhead (as determined by timing an empty doubly-nested loop) is deducted from the total. The result is divided by the number of loop iterations to yield an average time for a single execution of the arithmetic assignment statement. The same procedure is repeated for each of the four arithmetic operators. In addition, the entire test is repeated three times using increasing values of the outer loop counts to check for consistency of results. This results in operation counts for each operator of 300,000, 600,000, and 900,000 for the three trials.

Timings were obtained for each of the four operators using all 16 combinations of the following test conditions:

(1) *a*, *b*, and *c* were declared as either **real** or **double precision**;

(2) *a*, *b*, and *c* were allocated in either local or common memory;

(3) two different FORTRAN compilers were used, the AT&T/National Semiconductor compiler (*cf77*) and the Greenhills/Flexible compiler (*cf77 -f*); and

(4) each compiler was run both with and without optimization (*-O* option).

2.2. Benchmark 2

While the simplicity of the statement used in Benchmark 1 makes it useful for comparing the performance of the arithmetic operators and for analyzing the impact of data types and placement, it has significant shortcomings as a predictor of floating-point performance in other applications. FORTRAN arithmetic expressions in real programs are typically more complex, and also occur within a richer context. In particular, the operations in Benchmark 1 tend to be memory bound, with no potential for the accumulation of intermediate results in registers. For this reason, Benchmark 1 was expected to give results which were at the low end of the performance spectrum.

In order to get a better feel for the range of performance which would be found in practice, the Livermore Loops [3] benchmark was also employed. The Livermore Loops consist of 14 computational kernels extracted from actual scientific codes. They have been used in a variety of benchmarking efforts over the last fifteen years or so. The version used here was obtained from Argonne National Laboratory's ***netlib*** software distribution service [1].

Four variations of the Livermore Loops benchmark were used. The first two are single and double precision versions of the standard program. The other two are single and double precision versions which allocate all of their data (other than index variables and loop counters) in common memory instead of local memory. Only minor adjustments to the declarations were needed to accommodate the different versions — no changes to the computational kernels were required. The Livermore Loops were run using only the optimized Greenhills code, which was expected to give the best performance.

2.3. Test Conditions

The same hardware and operating system configurations were used for all runs of both benchmarks. All programs were run on a Flexible Computer Corporation Flex/32 [2] at NASA Langley Research Center. At the time these measurements were made, the Langley Flex contained 20 C1C computer cards (based on the NS32032 microprocessor and NS32081 coprocessor), two of which were dedicated to running UNIX. The remaining 18 were used for parallel programs running under Flexible's MMOS operating system. The Langley machine was configured with 2.25 MB of common (shared) memory, and each of the processors contained either 1 or 4 MB of local (private) memory.

All tests were run on a single 4 MB C1C computer card[1] under the MMOS operating system (Release 1.2.3.1). The processor's memory management unit (MMU) was enabled, which is the normal operating configuration at Langley. In order to avoid potential interference from other jobs, all 18 of the MMOS

[1] Experiments conducted at Langley indicate that floating-point operations are about 1% faster on 1 MB computer cards than on 4 MB cards.

processors were allocated to the benchmark programs, but only a single processor actually executed the tests. Parallelism and the effects of common memory contention were not covered by this study.

All timing measurements use elapsed ("wall clock") time as reported by the MMOS *CCrticks* system call.

3. Results

Table 1 summarizes the results from Benchmark 1. Each entry in the table contains four values, one each for addition, subtraction, multiplication, and division, in that order. The units are microseconds. Table 2 contains the same information expressed as thousands of floating-point operations per second (KFLOPS).

Precision	Memory Type	Operator	Compiler			
			cf77		cf77 -f	
			no opt.	opt.	no opt.	opt.
Single	Local	+	37.7	37.7	40.4	22.0
		-	37.8	37.7	40.3	22.3
		*	38.6	38.6	39.8	20.4
		/	43.3	43.3	42.5	24.6
	Common	+	43.0	43.1	43.0	24.5
		-	43.0	43.1	43.0	25.0
		*	43.8	44.0	42.4	23.2
		/	48.5	48.7	45.2	27.2
Double	Local	+	39.2	39.2	46.3	28.4
		-	42.5	42.5	46.3	28.3
		*	39.8	39.8	47.2	29.0
		/	48.7	48.7	51.2	33.8
	Common	+	46.6	46.9	51.6	33.9
		-	50.7	51.0	51.6	33.9
		*	45.8	46.1	52.5	34.4
		/	54.6	55.0	56.8	39.6

Table 1.

Benchmark 1: Average execution times in µs for the FORTRAN statement **a(i) = b(i) ∘c(i)**.

The results of Benchmark 2 are presented in Table 3. The KFLOP rates of each of the 14 Livermore kernels for all four versions of the test are given, along with some summary statistics.

3.1. Repeatability

Several tests were conducted, primarily using Benchmark 1, to check for consistency and repeatability of results. As described previously, Benchmark 1 used increasing iteration counts to check for timer inconsistencies or other performance discrepancies. None were found. The operation times were always in agreement to within ±0.1 µs in any given run of the program.

The program was also executed several times, at different times of the day and with different loads on the system, to check for variations between runs. These tests produced no more than ±0.1 µs variation between runs for local memory operations, but did show slightly more variability in the common memory tests. The largest observed fluctuation was 0.6 µs, but normal fluctuations were on the order of 0.2-0.3 µs, or less. Single precision operation times showed less variation than double precision times. These timing fluctuations for common memory operations were apparently due to common memory traffic generated by the UNIX processors, which use the common memory for certain operations. In particular, the MMOS job queue is maintained in common memory by the UNIX processors in conjunction with the System Monitor unit.

Precision	Memory Type	Operator	Compiler			
			cf77		cf77 -f	
			no opt.	opt.	no opt.	opt.
Single	Local	+	27	27	25	45
		-	26	27	25	45
		*	26	26	25	49
		/	23	23	24	41
	Common	+	23	23	23	41
		-	23	23	23	40
		*	23	23	24	43
		/	21	21	22	37
Double	Local	+	26	26	22	35
		-	24	24	22	35
		*	25	25	21	34
		/	21	21	20	30
	Common	+	21	21	19	29
		-	20	20	19	29
		*	22	22	19	29
		/	18	18	18	25

Table 2.
Results of Benchmark 1 in KFLOPS.

Experiments were also run to test the sensitivity of the timing results to the values of the operands. In one set of experiments, the seed of the random number generator was varied. As expected with such a large sample size, no changes in the timing results were seen. A second set of experiments varied the range of operand values which were generated. This had little or no effect on multiplication and division times, but addition and subtraction times showed changes of about 2%. Since the range testing was not comprehensive, somewhat larger variations might be seen in practice, especially when there are large differences of magnitude between the two operands of an addition or subtraction.

Benchmark 2 was also run several times to check for repeatability, and the results were the same as for Benchmark 1: local memory operations showed essentially no fluctuations, while small fluctuations were seen for the common memory operations.

4. Analysis and Discussion

Several observations can be made based on the above results. These are discussed in the following paragraphs.

4.1. Compilers.

Benchmark 1 was used to compare the quality of the code generated by the two FORTRAN compilers. As can be seen from Tables 1 and 2, the optimized code produced by the Greenhills compiler (*cf77 -f -O*) is clearly superior to that produced by the other alternatives. In all cases, the optimized Greenhills code was fastest, with speedup factors ranging from 1.33 to 1.95 over the other results. Note that the *unoptimized* Greenhills code performed poorly and was usually slower than the code produced by the AT&T/NSC compiler. This is in agreement with other experiments which have shown that unoptimized Greenhills FORTRAN generates inefficient code for array operations. As can be seen from the table, the AT&T/NSC code was uniformly slow, and the compiler was unable to perform any useful optimizations.

Loop No.	Single Precision		Double Precision	
	Local	Common	Local	Common
1	96	74	72	56
2	78	66	65	52
3	76	57	64	43
4	66	57	48	44
5	51	46	41	35
6	45	42	38	33
7	104	79	81	60
8	52	48	42	39
9	96	86	75	64
10	47	30	38	21
11	39	35	31	26
12	41	39	32	28
13	43	38	34	28
14	52	40	45	32
Min	39	30	31	21
Max	104	86	81	64
Avg	63	53	50	40

Table 3.
Benchmark 2: KFLOP rates for the Livermore Loops.

Examination of the assembly language code produced by the AT&T/NSC compiler also revealed that it follows the C convention of performing all floating-point operations using double precision arithmetic. Single precision (**real**) data is converted as needed before being operated on. Results are then converted back to single precision before being stored. This makes the resulting code somewhat slower than it might otherwise be.

During development and refinement of the code for Benchmark 1, it was noticed that minor changes to the test program could produce significant changes in the timing results, even when the changes did not directly involve the statements being timed. These perturbations were found to be largely due to differences in register allocations and other code optimizations. Experiments showed that even identical blocks of FORTRAN source statements which occur more than once within the same routine can be compiled into different instruction sequences. This dependence on program context was most pronounced with the Greenhills compiler, which performs substantial global optimizations, even without the *-O* option.

This code variability illustrates one of the main difficulties in attempting to generalize benchmark results to other programs, since timings obtained in one situation may be considerably different in another. Thus, the uncertainty principle applies to the measurement of high-level language constructs using software techniques. The inclusion of code to record start and stop times, as well as other test scaffolding (the **do** loops, for example) alters the context of the code being measured and may result in the generation of different instruction sequences than would otherwise be the case.

Because of these considerations, the results presented here should only be used as general indications of expected performance. For detailed analytical models or simulations of program behavior, a range of values may be a more appropriate way to express floating-point operation times than a single number.

4.2. Operation Times

As seen in Table 1, the execution times for addition and subtraction operations are essentially equal, although subtraction is somewhat slower when using the AT&T/NSC compiler for double precision operands. Multiplication is comparable to addition and subtraction, with some variation either plus or minus. Division requires about 3-5 μs longer than the other operations.

The results from Benchmark 2 (Table 3) give a better idea of the potential range of performance which may be encountered in practice. As expected, the results from Benchmark 1 were somewhat slower than the average from Benchmark 2. Expressed as average operation times, the Livermore Loops yielded results ranging from 9.6 to 47.6 μs per floating-point operation. The corresponding range for Benchmark 1 (*cf77 -f -O*) was 20.4 to 39.6 μs.

Generally speaking, the loops in Benchmark 2 which yielded the highest performance were those which contained single assignment statements composed of complex expressions. High performance was also enhanced by frequent use of scalar values and simple subscript operations. The slowest loops were those which contained multiple, simple statements and a higher proportion of integer and subscript operations. Examination of the assembly language code from some of the fastest loops showed that the Greenhills compiler had made very effective use of register operations, reducing the number of memory references needed. The slower loops, because of the number and simplicity of the statements involved, tended to be more memory bound.

4.3. Local vs. common memory.

Results from both benchmarks indicate that placement of data in common memory degrades performance, even in the absence of contention. Table 4 shows the increase in operation times for Benchmark 1 caused by allocating operands and results in common memory instead of local memory. This corresponds to performance degradation ranging from 6-20%. Table 5 shows the percentage performance degradation for Benchmark 2.

The results from Benchmark 1 can be used to estimate the common memory access overhead if we assume a simple timing relationship between local memory operations and common memory operations. Let

t_l = operation time with data in local memory,

t_c = operation time with data in common memory,

C = common memory access overhead, and

n = number of 32-bit common memory accesses per operation.

Then

$$t_c = t_l + nC \tag{1}$$

or

$$C = \frac{t_c - t_l}{n} \tag{2}$$

Note that in this model C could be negative, allowing for the possibility that common memory access might be faster than local memory access, although that has not been observed. For the operations in Benchmark 1, n=3 for the single precision case (2 fetches and a store) and n=6 for the double precision case (4 fetches and two stores).

In principle, the data from Table 1 could be substituted for t_l and t_c in Equation 2 to yield the value of C. Unfortunately, only the Greenhills compiler data can be used. The AT&T/NSC compiler generates extra instructions to compute common memory addresses, so the local memory and common memory codes are not directly comparable, and the data does not fit the simple model above. However, the Greenhills compiler recognizes that these extra address computations can be resolved at compile time, since they involve only constant offsets. The compiler therefore generates identical instruction sequences for the local and common memory operations. Using the Greenhills results gives values for C ranging from 0.83-0.93 μs (avg. 0.88) for single precision operations, and from 0.90-0.97 μs (avg. 0.93) for double precision operations. Thus it appears that a common memory access, in the absence of contention, requires about 0.9 μs longer than an equivalent local memory access. In fact, other experiments indicate that this figure is probably a lower bound. In some situations common memory accesses seem to require substantially longer, implying that the above model is oversimplified. The other factors involved are not well understood at this time.

Increase in Operation Times (μs)					
Precision	Operator	Compiler			
		cf77		cf77 -f	
		no opt.	opt.	no opt.	opt.
Single	+	5.3	5.4	2.6	2.5
	-	5.2	5.4	2.7	2.7
	*	5.2	5.4	2.6	2.8
	/	5.2	5.4	2.7	2.6
Double	+	7.4	7.7	5.3	5.5
	-	8.2	8.5	5.3	5.6
	*	6.0	6.3	5.3	5.4
	/	5.9	6.3	5.6	5.8

Table 4.

Increase in operation times for Benchmark 1 caused by allocating operands and results in common memory.

% Performance Degradation		
Loop No.	Single Precision	Double Precision
1	23	22
2	15	20
3	25	33
4	13	8
5	10	15
6	7	13
7	24	26
8	8	7
9	10	15
10	36	45
11	10	16
12	5	13
13	12	18
14	23	29
Avg	16	24

Table 5.

Percentage performance degradation of the Livermore Loops caused by allocating data in common memory.

Because of the performance penalty for accessing common memory, it may be desirable in some applications to copy data from common memory to local memory before operating on it. Several factors influence the time needed to perform the copy operation, including (1) the data type **(real** or **double precision)**, (2) the dimensionality (scalar or array element), (3) the number of elements copied (a single value or several values in a loop), (4) the addressing modes generated by the compiler, and (5) the surrounding program context. Approximate times for common-to-local copy operations range from 5-15 μs for single precision items, and from 10-20 μs for double precision items.

Given the times for copy operations, as well as the common memory access overhead, estimates can be derived for the number of accesses of a variable needed to justify the expense of copying it to local memory. If the access overhead is assumed to be 0.9 µs, then it can be seen that single precision data must be accessed from 6 to 17 times locally in order to amortize the cost of a copy operation. For double precision data, values must be used from 6 to 12 times before any payoff is observed. If the data must be copied back to common memory afterwards, then roughly twice as many accesses are needed before any performance improvements are realized. Note that these numbers should only be used as general guidelines. Due to the number of factors involved, values both above and below these ranges might be found in practice.

5. Summary

Based on results obtained from the benchmark programs used here, several observations can be made about the performance of floating-point operations in FORTRAN programs on the Flex/32. It was found that floating-point operation times can vary by a factor of 5 or more depending on many factors. The most important factors influencing performance seem to be the extent to which operations are memory bound, the context in which operations occur, and the quality of code generated by the compiler. Allocation of data in common memory also has a distinct negative impact on performance, although the severity of this effect depends on the application. The values of the operands involved also affect addition and subtraction operations, but multiplication and division times appear to be insensitive to the operand values (with the possible exception of zero operands).

Several recommendations can be made which will help to achieve optimum performance. The *-f* and *-O* options of the *cf77* FORTRAN compiler should *both* be used. This results in generation of the highest quality assembly code. Complex arithmetic expressions also seem to be more efficient than very simple ones, and index operations should be kept as simple as practicable. Data should be allocated in local memory whenever possible, and common memory data should be copied to local memory if it will be accessed many times between updates.

Although the results obtained here are specifically for FORTRAN programs running under the MMOS operating system, some of them may generalize to FORTRAN programs under UNIX, and to a lesser extent may also be applicable to C programs. C users should be careful however, since variable scoping rules and the use of pointers will result in substantially different addressing modes being used to access data. Therefore, the apparent common and local memory access times may differ considerably from those found in FORTRAN programs.

Acknowledgements

I would like to thank Vijay Naik for his encouragement of this study and for several helpful suggestions on analyzing the performance of the Flex.

References

[1] Dongarra, J., and Grosse, E. Distribution of Mathematical Software Via Electronic Mail. *Communications of the ACM*, Vol. 30, No. 6, May 1987, pp. 403-407.

[2] Matelan, N. The Flex/32 MultiComputer. *Proceedings of the 12th Annual International Symposium on Computer Architecture* (*Computer Architecture News*, Vol. 13, No. 3), June 1985, pp. 209-213.

[3] Riganati, J., and Schneck, P. Supercomputing. *Computer*, Vol. 17, No. 10, Oct. 1984, pp. 97-113.

NASA
National Aeronautics and
Space Administration

Report Documentation Page

1. Report No. NASA CR-178364	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle

PERFORMANCE OF FORTRAN FLOATING-POINT OPERATIONS
ON THE FLEX/32 MULTICOMPUTER

5. Report Date

August 1987

6. Performing Organization Code

7. Author(s)

Thomas W. Crockett

8. Performing Organization Report No.

Interim Report 4

10. Work Unit No.

505-90-21-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18107

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

16. Abstract

A series of experiments have been run to examine the floating-point performance of FORTRAN programs on the Flex/32 computer. The experiments are described, and the timing results are presented. The time required to execute a floating-point operation is found to vary considerably depending on a number of factors. One factor of particular interest from an algorithm design standpoint is the difference in speed between common memory accesses and local memory accesses. Common memory accesses were found to be slower, and guidelines are given for determining when it may be cost effective to copy data from common to local memory.

17. Key Words (Suggested by Author(s))

benchmark, Flex/32, performance

18. Distribution Statement

60 - Computer Operations and
Hardware
62 - Computer Systems

Unclassified - unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 10	22. Price A02