ASTR415: Problem Set #1

Curran D. Muhlberger
University of Maryland
(Dated: February 13, 2007)

In order to determine the computational cost of various arithmetic operations, a program was written to time several common integer and floating-point operations. This program was written in three different languages and compiled using multiple compilers, both with and without optimizations, in order to determine the most efficient language and compiler settings for the testing platform. Results from the double-precision addition and multiplication tests were then used to calculate the floating-point performance in Mflops of the system. The square root and power tests indicated that, on most platforms, the `sqrt` function was more efficient than the `pow` function for unoptimized compilers. Finally, the results for all tests were used to guess what optimization techniques each compiler employed to improve its performance.

I. CPU BENCHMARKING

The easiest way to measure the computational cost of arithmetic operations is to perform identical operations repeatedly and time how long it takes to complete. The C program *benchmarker* takes a single argument n and records the CPU time needed to perform a set of operations n times. These operations are:

- integer_addition: integer addition by 1
- integer_subtraction: integer subtraction by 1
- integer_multiplication: integer multiplication by 1
- integer_division: integer division by 1
- double_addition: double-precision addition by 1.0
- double_subtraction: double-precision subtraction by 1.0
- double_multiplication: double-precision multiplication by 1.000001
- double_division: double-precision division by 1.000001
- double_sqrt: double-precision square root finding
- double_pow: double-precision exponentiation by 0.5

A test for each operation was isolated into a separate function taking n as a parameter. The functions initialized a variable, repeatedly performed the target operation n times in a `for`-loop, and printed out the CPU time spent in the loop. The final value of the variable was also printed to prevent optimizing compilers from eliminating the body of the function as "dead code." The argument n was declared `const` in the function's definition so that an optimizing compiler would know that the upper bound of the `for`-loop would not change during execution.

A. Double-Precision Addition and Multiplication

A bash shell script was written to run *benchmarker* for $n \in \{10^6, 3\times10^6, 10^7, 3\times10^7, 10^8, 3\times10^8, 10^9\}$ and to print a table of execution time versus n for a specified test. The results for double-precision addition and multiplication are shown in table I. The executable was compiled using the Sun Studio C compiler without optimizations.

These results are plotted in figure 1. The irregularities for small n are due to the lack of precision (10 ms) of C's `clock` function (times of 0 ms are plotted as the minimum value of the y-axis, as 0 cannot be represented on a logarithmic scale). The relationship between n and execution time is clearly linear. Calculating the slope yields a performance of 169 Mflops for addition and 171 Mflops for multiplication.

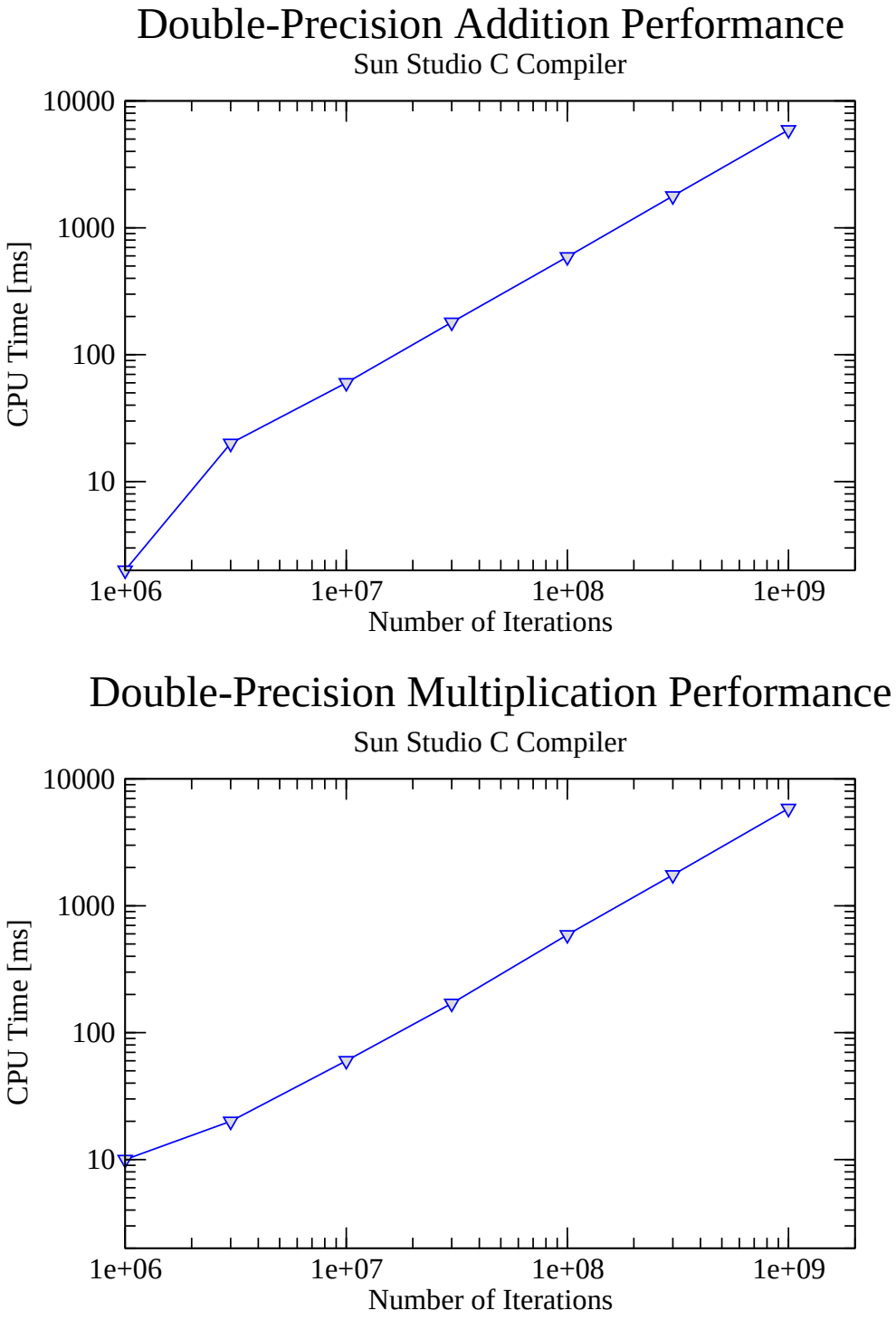

FIG. 1: Double-precision addition and multiplication performance. Executable compiled from C source code using the Sun Studio C compiler without optimizations.

TABLE I: Double-precision addition and multiplication performance. Executable compiled from C source code using the Sun Studio C compiler without optimizations.

n	Addition [ms]	Multiplication [ms]
1×10^6	0	10
3×10^6	20	20
1×10^7	60	60
3×10^7	180	170
1×10^8	590	590
3×10^8	1780	1750
1×10^9	5910	5840

TABLE II: Square root and power tests on a variety of platforms. Each executable was compiled with the system's default C compiler without optimizations.

CPU	Operating System	`sqrt` [ms]	`pow` [ms]	Ratio
AMD Athlon64 3500+	Gentoo Linux (64-bit)	18840	15290	0.81
AMD Opteron 2200SE (dual-core)	Red Hat Enterprise Linux 4 (64-bit)	20430	81570	3.99
Intel Core2 6300 (dual-core)	Ubuntu Linux 6.10 (32-bit)	10220	163640	16.01
2 × Intel Xeon (quad-core)	Red Hat Enterprise Linux 4 (64-bit)	18420	100780	5.47
Intel Pentium 4 3.0 GHz (HT)	Red Hat Enterprise Linux 4 (64-bit)	37900	125610	3.31
Intel Celeron 1.0 GHz	Red Hat Enterprise Linux 3 (32-bit)	57600	321380	5.58
Intel Pentium 4 3.4 GHz	Microsoft Windows XP Professional (32-bit)	27547	75625	2.75
2 × IBM PowerPC G5 2.0 GHz	Mac OS X 10.4.8 (64-bit)	31990	37890	1.18
Sun-4 UltraSPARC-II 450 MHz	Solaris 9 (32-bit)	155320	290910	1.87

B. Square Root and Power Functions

The relative performance of the square root and power functions appears to be dependent on the architecture of the testing system. On the system that was used for all of the other tests, at 10^6 iterations the square root test consumed 28 seconds of CPU time while the power test only consumed 16 seconds to produce an identical result. According to these results, when compiling C source code without optimizations, it is more efficient to use `pow(x, 0.5)` instead of `sqrt(x)` to find the square root of a variable `x`. This is unusual, however, as the `sqrt` function is far less general in purpose than `pow` and should be at least as efficient by implementing the same algorithm as `pow` called with an exponent of 0.5.

Further investigation showed that this was an anomaly. When the square root and power tests were performed on a variety of other architectures, the `sqrt` function was often between 3× and 16× faster than the `pow` function. However, the extent to which `sqrt` outperformed `pow` varied widely even between very similar platforms. Since these routines are done in software, it would seem that their performance is highly dependent on the version of libm on the system. The results of this cross-platform testing are shown in table II.

Having the power function as the slower of the two makes much more sense. The `sqrt` function should perform at least as well as `pow` because at worst it could use the same algorithm as `pow` with the second argument hard-coded as 0.5. More likely the algorithm used by `sqrt` is specialized for finding square roots and is completely different from that used by `pow`. The fact that it only depends on one argument and not two means that more can be assumed at design- and compile-time, eliminating decision-making code.

The anomaly with the Athlon64 processor running Gentoo Linux is very intriguing. When the code is compiled with optimizing compilers, the square root test runs as quickly as the power test, but never faster. Furthermore, AMD provides a library called ACML_MV that includes an accelerated power routine. Linking against this library causes the power test to speed up even more, again significantly outperforming the `sqrt` function for the same calculation. No accelerated `sqrt` routine is provided. Clearly when working with software math routines, it is essential to use the best libraries available for the system, as the differences in efficiently between implementations can be enormous.

TABLE III: The versions of the compilers tested, along with their optimization flags.

Compiler Name	Version	Optimization Flags
GNU C Compiler	4.1.1	-march=k8 -ffast-math -funroll-loops -ftree-vectorize -O3
GNU FORTRAN Compiler	4.1.1	-march=k8 -ffast-math -funroll-loops -ftree-vectorize -O3
Sun Studio Express C Compiler	Build 35_2	-fast -xarch=amd64a -xvector=simd
Sun Studio Express FORTRAN 95 Compiler	Build 35_2	-fast -xarch=amd64a -xvector=simd
Intel C Compiler	9.0	-O3 -xW -ip
Intel FORTRAN Compiler	9.0	-O3 -xW -ip
Sun Java HotSpot Server VM	1.6.0	N/A

TABLE IV: Floating-point performance of various C compilers.

Compiler	Addition [Mflops]	Multiplication [Mflops]
GNU C Compiler	168.122	168.127
GNU C Compiler (optimized)	549.495	549.145
Sun Studio C Compiler	169.173	171.290
Sun Studio C Compiler (optimized)	∞	1093.365
Intel C Compiler	549.600	545.536
Intel C Compiler (optimized)	546.623	1099.245
Java	514.952	515.476

C. Language and Compiler Comparisons

To determine the most efficient language/compiler combination, the *benchmarker* program was ported to Java and FORTRAN. Each version was then compiled with a variety of compilers, both with and without optimizations, and the results were recorded and compared. For double-precision addition and multiplication, the tests were timed for the same series of n as above. For all other tests, times were only recorded for $n = 1 \times 10^9$.

The testing machine consisted of an AMD Athlon64 3500+ CPU with 512 KB of cache running at 2.2 GHz. The system was running 64-bit Gentoo Linux based on a custom 2.6.18 kernel. All system libraries were compiled with gcc-3.4 using -O2 optimizations. The C and FORTRAN compilers tested were 64-bit Linux compilers from the GNU project, Sun Microsystems, and Intel. Their versions and optimization flags are shown in table III. Note that the Intel compiler will disable the execution of vectorized code on AMD chips [1]. This was solved by patching the executables using a script written by Mark Mackey.

The double-precision addition and multiplication times of the C and Java versions of the program are plotted in figure 2. The calculated performance in Mflops is shown in table IV.

Similarly, the double-precision addition and multiplication times for the FORTRAN version are plotted in figure 3, and the performance in Mflops is shown in table V.

The above plots yield a great deal of insight into the optimizations used by the various compilers. For the addition tests, there appear to be only two distinct compilation patterns. The optimized versions likely utilize loop unrolling to more than tripple the execution speed. The optimized Sun compiler, however, was able to determine the end result

TABLE V: Floating-point performance of various FORTRAN compilers.

Compiler	Addition [Mflops]	Multiplication [Mflops]
GNU FORTRAN Compiler	162.965	166.233
GNU FORTRAN Compiler (optimized)	547.154	547.967
Sun Studio F95 Compiler	182.617	172.786
Sun Studio F95 Compiler (optimized)	∞	1095.781
Intel FORTRAN Compiler	544.395	546.944
Intel FORTRAN Compiler (optimized)	546.944	1100.558

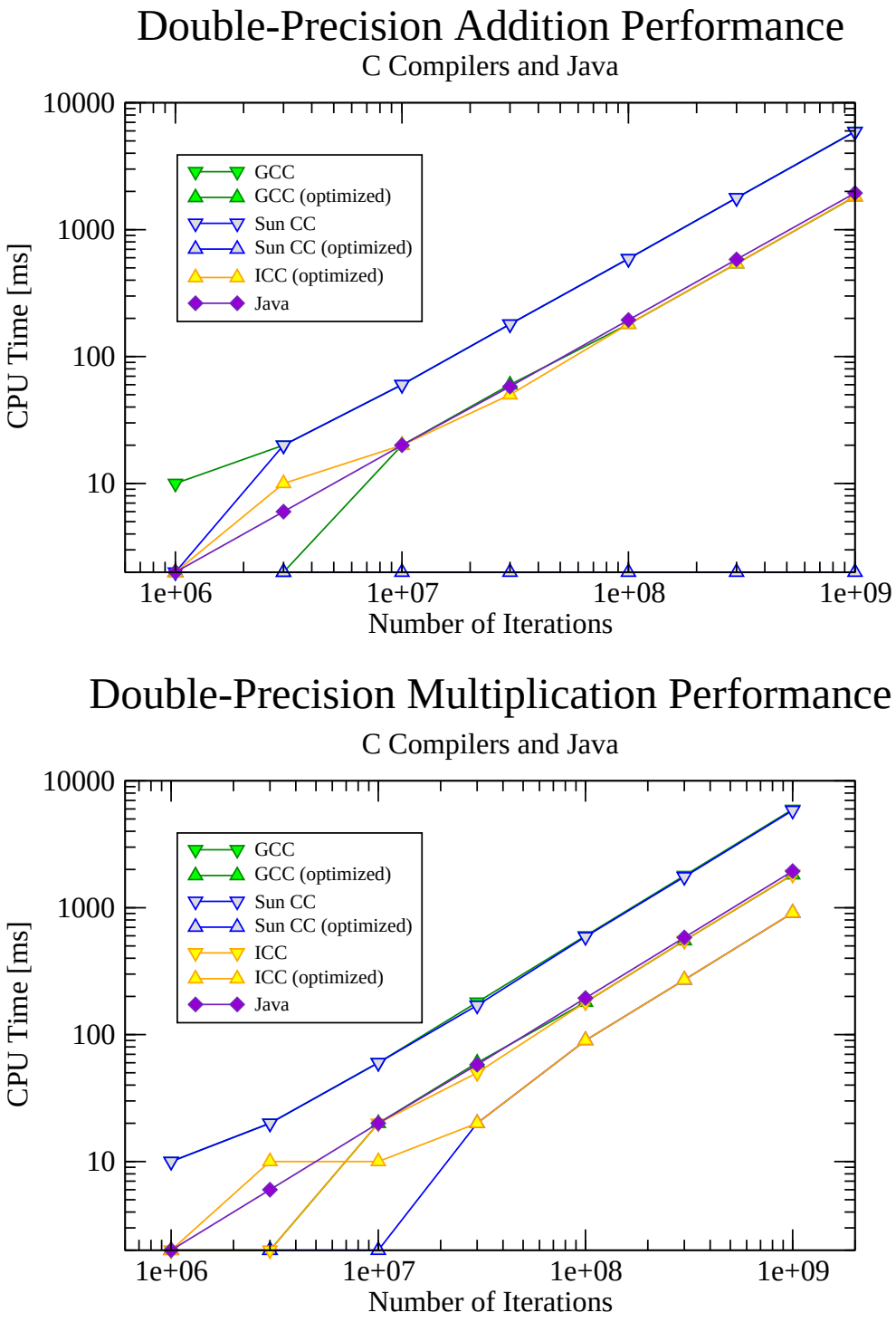

FIG. 2: Double-precision addition and multiplication performance. Executables compiled from C and Java source code using a variety of compilers both with and without optimizations.

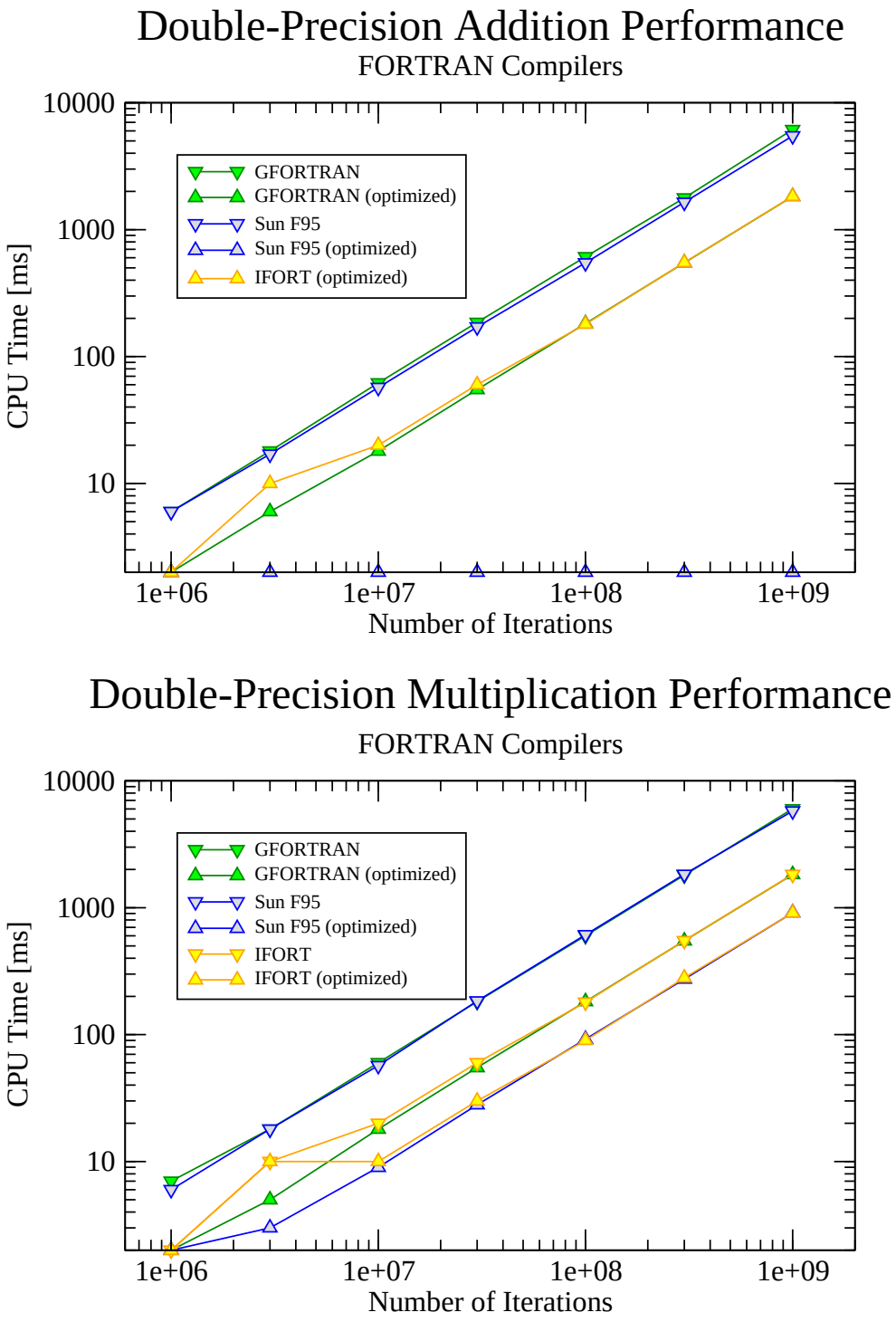

FIG. 3: Double-precision addition and multiplication performance. Executables compiled from FORTRAN source code using a variety of compilers both with and without optimizations.

of the loop, and thus the test ran instantaneously for all n. For both languages the unoptimized and optimized Intel compilers produced nearly identical results, so only the latter is shown.

In the case of multiplication, three distinct compilation patterns are visible. The middle line represents a speedup similar to that for addition, likely achieved by a combination of loop unrolling and fast arithmetic. The bottom line, however, represents vectorized code taking advantage of the processor's SIMD instructions. These allow two double-precision operations to be packed into a single 128-bit XMM register, cutting the execution time in half. It should be noted that an overflow error occurs during the multiplication test, since $1.000001^{10^9} \sim 10^{494}$ is larger than the maximum value for an IEEE double of $(2 - 2^{-52}) \times 2^{1023} \sim 10^{308}$. All executables returned a value of `+inf` for this test.

The Java code performs surprisingly well in both cases. Java is often considered unsuitable for scientific computation because its cross-platform bytecode is interpreted by a virtual machine at runtime. However, modern Java VMs utilize a just-in-time compiler (JIT) to profile an application at runtime and produce optimized machine code. These results show that, for simple double-precision operations, Java is just as fast as an optimized C or FORTRAN compiler that does not utilize SIMD instructions, and it does not require specifying machine-specific optimization flags at compile-time. Furthermore, Java is being used as the basis for the X10 programming language, which is being developed by IBM for the DARPA program on High Productivity Computing Systems [2].

A comparison of Java and C compilers across all tests is shown in figure 4. All optimizing compilers, including Java, are able to predict the outcome of the multiplication and division tests. Furthermore, the Sun Studio C compiler and Java were both able to predict the outcome of the addition and multiplication tests, just as the Sun compiler did for double-precision addition above. Intel's compiler may use loop unrolling to improve its times relative to the other unoptimized compilers, but GCC's optimization approach is a mystery, as it computes the result in a very short but non-zero span of time.

Double-precision subtraction performs similarly to addition, with all optimizing compilers performing relatively equally. The results for division, however, are completely different from those for multiplication. Unoptimized division performs extremely slowly, and even with optimizations it is one of the slowest tests when using Sun's compiler and Java. GCC's optimization translates the division into multiplication by the inverse of the divisor, which can be computed at compile-time since it is constant. Here Intel's technique is a mystery, as it performs far better than Sun's compiler, yet it has clearly not translated the operation into multiplication. Furthermore, it returns a result of 0, while the other compilers return 2.5×10^{-318}. It is possible that Intel is achieving its speedup by sacrificing precision.

Finally, for the square root and power tests, all optimized compilers appear to perform equally well. This is reasonable, as these are software routines that have already been compiled separately from the program. Java's poor performance on the power test is not a defect in the language or the VM. It is consistent with the results found on other platforms, where the square root test typically out-performs the power test. The VM is compiled with its own math library, so it is not using the system library that all of the other compilers are. On this system, it appears that the power library has been highly optimized in a way that is not typical for other Linux systems, so the Java performance is probably a good indicator of the hardware's performance given a more standard set of system libraries.

A similar comparison of FORTRAN compilers is shown in figure 5. The results follow the same patterns as those for C, except that now the Sun Studio FORTRAN 95 compiler also converts the division into multiplication, and furthermore it does this multiplication using SIMD instructions to obtain by far the best performance on this test.

II. DURATION OF WORK

The first version of the code, written in Java, probably took about 15 minutes to write, and it only took about 5 minutes to port it to C. The FORTRAN code probably took about 20 minutes to write, after spending at least 45 minutes searching for timing routines.

Each executable took about 2 minutes to finish 10^9 iterations, so generating the data on double-precision addition and multiplication took about 4 minutes per operation per version. The single-version graphs only took about 15 minutes to create (and much of that was spent learing the interface of Grace). The final graphs comparing all of the compilers and optimizations flags took a couple of hours to create.

Far more time was spent just tweaking the code, the compiler options, and the makefile. The report also took a few hours to draft. However, work proceeded at this pace mostly because so much time was given in which to complete

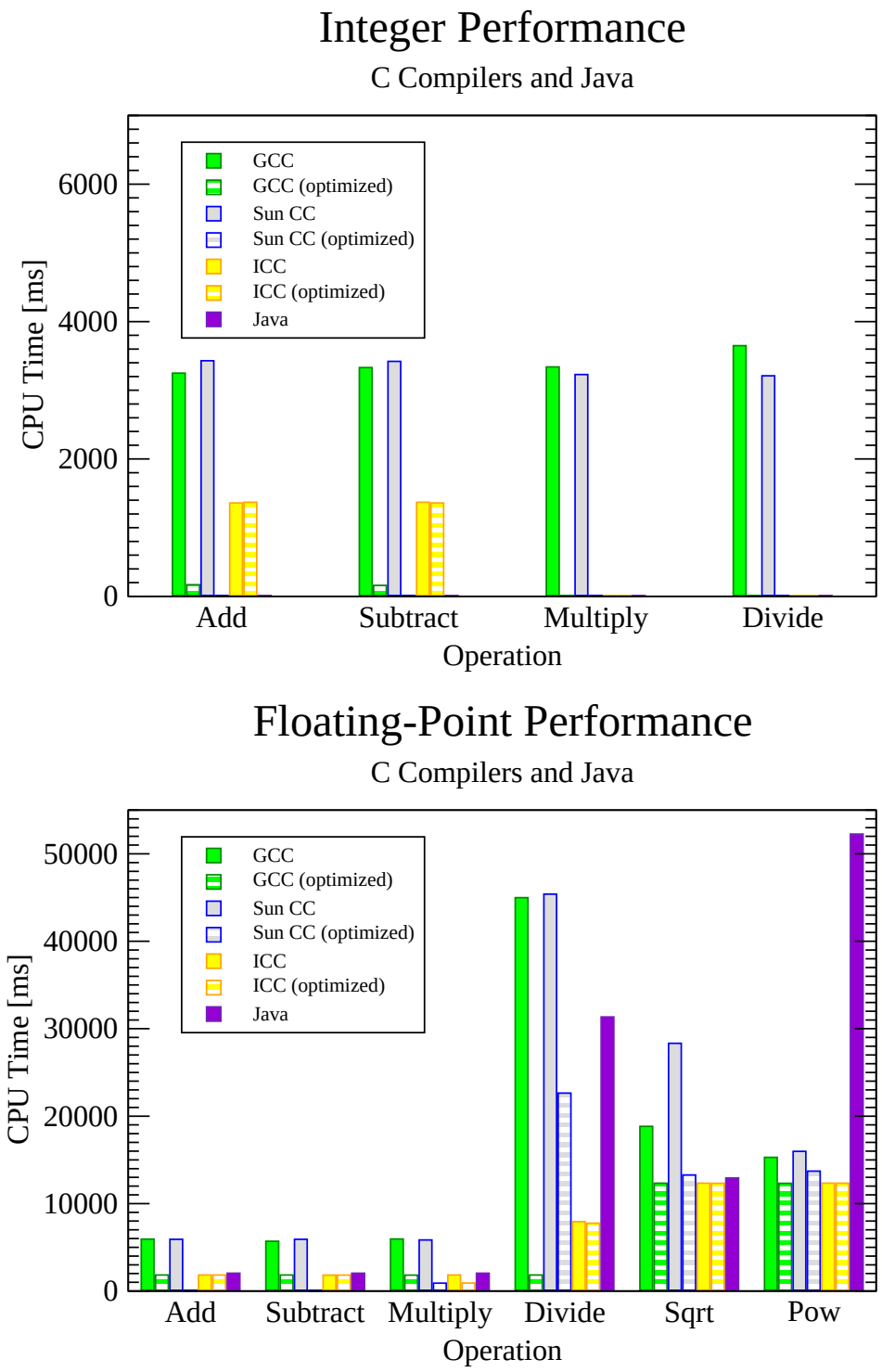

FIG. 4: Integer and floating-point performance for all tested operations. Executables compiled from C and Java source code using a variety of compilers both with and without optimizations.

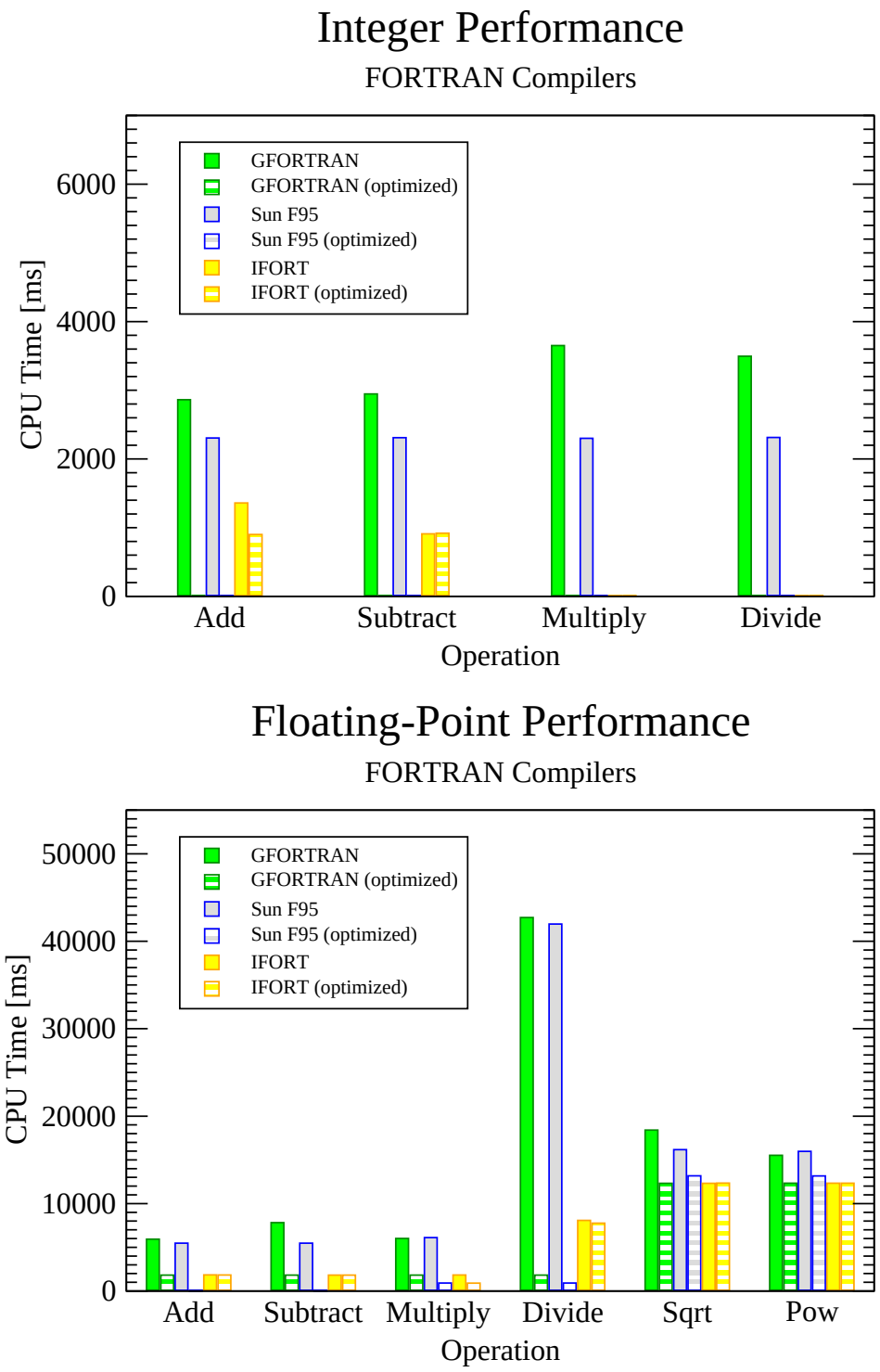

FIG. 5: Integer and floating-point performance for all tested operations. Executables compiled from FORTRAN source code using a variety of compilers both with and without optimizations.

the assignment. The results could have been obtained much more quickly otherwise.

---

[1] Mackey, Mark *Intel's compiler: is crippling the competition acceptable?*. `http://www.swallowtail.org/naughty-intel.html`. Updated 2005-12-08. Accessed 2007-02-11.

[2] IBM Research. *The X10 Programming Language*. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html`. Updated 2006-03-17. Accessed 2007-02-13.