

Compilers, languages, options, and astronomical images

Or

This code is over 1,000,000 times faster than that code^{*}

(*a deliberately attention-grabbing headline claim – not actually untrue, but really rather misleading)

Keith Shortridge – Keith@KnaveAndVarlet.com.au



Introduction: This poster describes a series of tests intended to see how well 12 different programming languages handle access to individual elements of 2-dimensional arrays - the bread and butter of astronomical data. Perhaps the biggest surprise is the range of execution speeds that was found. Some languages are much faster than others. Many compilers do not optimise by default, and there was one case where simply compiling with full optimisation sped up a test program nearly 500 times. The slowest test actually ran over 1.5 million times slower than the fastest, but even without that extreme outlier, the range of speeds covered many orders of magnitude.

Vector instructions: Most modern processors support vector processing instructions, operating on a set of numbers in one go. The X86 processor used for these tests supported the AVX 256-bit (8 ‘floats’ at a time) instructions. Most processors support 128-bit vector instructions, and at levels above -O1 or -O2 the C++ and Fortran compilers tested will use these if possible. The -march=native flag will make a compiler use the best the current processor has, which may speed things up but may not run on lesser processors. I was surprised to see some compilers work out a way to use these on this problem. To see how they did this, look at the comments in the hand-coded assembler, which does the same.

Rectangular multi-dimensional arrays: Disappointingly, few languages really support ‘rectangular’ multi-dimensional arrays. The older languages did (Fortran, PL/1, Algol, Pascal - although Pascal spoilt it by making it hard to pass them to subroutines). The trend for some time has been to support ‘arrays of arrays’, where a 2D array is an array of 1D arrays – not all of which have to have the same number of elements. C/C++ can create ‘rectangular’ arrays, but makes passing them to subroutines awkward. Julia is a recent language that supports rectangular arrays. One feels native rectangular support for 2D arrays should make it easier for compilers to optimise code.

How should I interpret these results? The chart shows the relative execution times for each test, with the fastest scaled to 1. A little arbitrarily, the fastest results are green, the next fastest band are yellow, slower ones are orange and the slowest set are red. The one outlier is purple. This really only tells you how well different languages handled this one, contrived, problem. At high optimisation levels, the C++ and Fortran compilers used vector instructions to address this problem in surprisingly efficient ways, which may not always be possible - all the green entries do this. At -O1 optimisation the C compilers are generating code that really does access individual array elements, but they do so very efficiently. In the chart, these are all in the yellow band. This may be a realistic target for most problems, so yellow may represent a sensible attainable ‘standard’ of sorts. Some orange results, although slower, are quite respectable, and may represent a good trade off for the convenience of the language in question. Red entries indicate combinations not really suited to this sort of problem.

The test: The test used was trivially simple. Pass a 2D array to a subroutine and have that subroutine add the sum of its two index values to each individual element. The C++ code for the body of the subroutine is shown below. The idea was to force the compilers/interpreters to access each element individually, rather than making it easy for them to use built-in vector operations to speed things up. However, some compilers turned out to be really quite ingenious in how they handled this.

```
for (int Iy = 0; Iy < Ny; Iy++) {
    for (int Ix = 0; Ix < Nx; Ix++) {
        Out[Iy][Ix] = In[Iy][Ix] + Ix + Iy;
    }
}
```

Nearly 90 tests, coded in 24 ways in 12 languages, with 13 compilers/interpreters and a variety of optimisation options.

- Which language did you expect to be fastest?
- Did you expect this range of execution times?
- What language do you use?
- What optimisation flags do you use?

- Personal surprises:**
- o The enormous range of speeds measured.
 - o Fortran was marginally slower than C/C++.
 - o The speed of Javascript.
 - o Failing to beat C++ using assembler.
 - o Just how good the C++ compilers are now.
 - o How few compilers optimise by default.
 - o How fast C++ ‘vectors of vectors’ were.
 - o The overheads with R reference objects.
 - o How simple this was to code in Julia (for an ex-Fortran user).
 - o How much memory access can slow things down.
 - o How slow ‘numpy raw’ - one element at a time - was.
 - o Just how sensitive to optimisation level C++ Boost code was.
 - o How long it took to do all this!

Compiler	Assembler	C		C++		C++ : Boost		Fortran	Java	Javascript
		Num. rec.	C raw	vectors	no assert					
clang	1	28.7	28.23	57.15	1177.61	1160.86				
clang -O		1.6	1.63	1.6	18.27	7.93				
clang -O1		6.67	6.68	6.69	625.2	616.77				
clang -O2		1.59	1.63	1.6	18.31	7.94				
clang -O3		1.59	1.62	1.6	15.8	7.92				
clang -O3 native		1	1.01	1.02	14.97	12.02				
g++		40.94	40.78	105.83	1349.95	1319.58				
g++ -O		9.2	7.52	11.07	39.19	19.32				
g++ -O1		9.21	7.51	11.07	39.53	19.34				
g++ -O2		7.51	7.51	7.51	36.66	9.98				
g++ -O3		2.42	2.54	2.44	36.7	7.55				
g++ -O3 native		1.07	1.06	1.09	34.95	7.83				
gfortran							40.73			
gfortran -O							6.7			
gfortran -O1							6.68			
gfortran -O2							6.68			
gfortran -O3							2.43			
gfortran -O3 native							1.06			
Java								7.6		
Node.js									28.62	
Node.js --no-opt										627.21
	Julia	Perl	Perl/PDL arrays	Perl/PDL raw	Perl/PDL vectors	Python arrays	Python lists	Python numpy raw	Python vectors	
julia	10.14									
julia -O0	88.75									
julia -O1	23.43									
julia -O2	10.15									
julia -O3	10.13									
Perl		1283.67								
Perl/PDL			51.35	180443.8	123.02					
Python2						34.14	1438.78	20878.39	55.75	
Python3						47.04	1808.52	9959.94	43.87	
	R : outer	R : raw	R : ref class	Rust	Swift	Tcl : raw				
Rscript	245.12	954.77	1587729.2							
Rustc				1711.89						
Rustc -O				20.29						
Rustc -O3				18.14						
Rustc -O3 native				17.49						
Swiftc					3365.11					
Swiftc -O					15.01					
Swiftc -Onone					3360.25					
Swiftc -Ounchecked					7.11					
tcsh						19703.7				

Some Test-specific notes

Assembler: Two years ago, in assembler I could beat the clang and gcc compilers by about a factor two on this problem. But I can't beat them now. However, being able to read assembler can help; I spotted the C++ -march=native flag through wondering why the C++ code wasn't using wider vectors, and got a factor 2 improvement.

C/C++: C has rectangular multi-dimensional arrays, but you can't pass them easily to subroutines so the subroutine can do the index arithmetic needed. You can do that arithmetic yourself, as the 'C raw' code does, or use the scheme used in 'Numerical Recipes in C' (an ArrayManager class included in the code packages up this NR scheme conveniently). Or use the Boost library, which provides multi-dimensional arrays defined through templates and relies heavily on compiler optimisations for speed. Or use STL vectors of vectors to form 2D arrays.

Fortran: Has always handled multi-dimensional arrays well, but it was only with Fortran 90 that it was able to allocate them dynamically.

Java: Bizarrely, the Java implementation I used generated slower code for to add 'ly + lx' to an element than it did to add 'lx + ly'. Optimisation comes from its JIT interpreter.

Javascript: Not shown in the table, but Javascript ran even faster in Safari.

Perl/PDL: PDL adds IDL-like facilities for Perl, including numpy-like vector and array operations. It's slow at single-element operations, (as is numpy, to a lesser extent).

Python: 'Raw' is using numpy to access individual array elements – something it was not intended for. 'Vectors' uses numpy vector operations to get the required effect.

R: This supplied the outlier result. I tried to bypass an array copy time by using a 'reference object'. A quirk of R's ended up copying the array each time any element of it was modified. A cautionary tale – some attempts to optimise can backfire. 'Outer' is using vector operations rather than individual array access.

Swift: The original Swift compiler was very slow accessing 2D arrays, but the latest versions have improved enormously.

The code: All the code, with comments, used in these tests – and more – is available on GitHub, in the repository: KnaveAndVarlet/ADASS2019. Or send me an e-mail. Or find me and ask about it. There should also be a listings folder close to this poster.

Is this a fair test? Probably not. The test code used here deliberately tried to test how well a system handled access to individual elements of an array, in a way that would be difficult for a compiler to optimise. For example, nested loops that simply copied each element of a 2D array into another array could be turned into a single call to an optimised routine like C's memcpy() that just copies data regardless of its structure, and that wouldn't have tested access to individual array elements. However, it means that the test is probably unrepresentative of many real-world problems, and it certainly allows some languages to shine more than others. In particular, it really penalises interpreted languages like Python, which generally get their speed from packages like numpy, which can work on large blocks of data as quickly as on they do on one element. If your problem is a good match to the bulk processing operations of libraries like numpy, you will find these much more efficient than they may appear here. It's also unfair to new languages – compilers for C/C++ and Fortran have years of development behind them. The evolution of the Swift compiler shows the gains that can come with just a few years of work. For example, optimised Julia is orange in the chart above, but is actually very close to the non-vectorised results from C/C++, as is optimised Swift.

Vector libraries. Some tests ('vectors' for Python, 'outer' for R) use built-in array operations to get the same results as the other tests, but far faster than using access to individual elements. The code looks quite different, and doesn't test access to individual element, which was the original intention. But it shows that use of such facilities can speed up interpretive languages hugely – which is why they exist. How well this works depends on how well the problem can be reformulated to use the array operations available. (Note that the programmer has to solve this problem, which is not the same as having a compiler use vector instructions to speed up code written using single-element access.)

- Summary:**
- The range of execution times is huge.
 - Optimisation options make a big difference.
 - Most compilers don't optimise by default.
 - Template code is very slow if unoptimised.
 - Some obscure optimisations can help.
 - Modern mature compilers are really good.
 - Writing assembler isn't worth the effort.
 - But reading assembler may be useful.
 - Compiled code beats interpreted code. Duh.
 - Libraries like numpy speed things up a lot.
 - Memory access can be a bottleneck.
 - C/C++ is fastest, Fortran a whisker behind, (and most languages can call C code).
 - Your program will have different trade-offs.