A Specialised Compiler for Reduced Floating-Point Precision in Large-Scale Numerical Models

Andrew Dawson
Atmospheric, Oceanic & Planetary Physics, University of Oxford, Oxford UK

1. The Role of Inexact Computations in Numerical Simulations

Weather forecasting, and many other fields that rely on high performance computing, have enjoyed the benefits of continued supercomputing power increases, allowing model complexity and cost to increase steadily with time. However, the next generation of supercomputing technology, the exascale computer, is out of reach with current technology largely due to the anticipated excessive power demands. Continued improvement in supercomputing performance will require both making more efficient use of the current technologies and the design of new hardware architectures that can avoid the power issues.

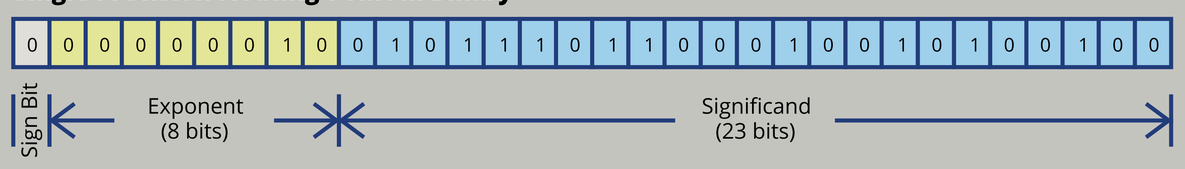
One option for next-generation hardware is to design energy-efficient domain-specific hardware (e.g., FPGAs) or leverage existing technologies (e.g., GPUs) that can perform floating-point calculations with a reduced number of bits, or with errors. Many parts of our numerical models may be over-engineered in terms of floating-point precision, using 64-bit double precision for algorithms where the solution may be able to be computed to acceptable accuracy with many fewer bits, or tolerate errors in the least significant bits. This makes reduced precision or inexact floating-point calculations an attractive option for weather forecast models. Reducing floating-point precision could not only save computing resources (CPU time, memory, communication) but also allow for model components to incorporate added complexity.

2. The Representation of Floating-Point Numbers

Computers use floating-point numbers to represent arbitrary decimal numbers. A floating-point number is represented as a group of 32 bits (or 64 bits for double precision). The bits are divided into groups, each having a different meaning:

- **Sign Bit:** The left-most bit is the sign-bit which determines if the number is positive or negative. If the bit is set (has value 1) then the number is negative, if the bit is not set (has value 0) the number is positive.
- **Exponent:** The next 8 bits (11 for double precision) are called the exponent and determine the magnitude of the number.
- **Significand:** The remaining 23 bits (52 for double precision) are called the significand (also called the mantissa) and determine the digits that make up the number. It is the number of bits in the significand that determine the precision to which decimal numbers can be represented.

Single Precision Floating-Point in Binary

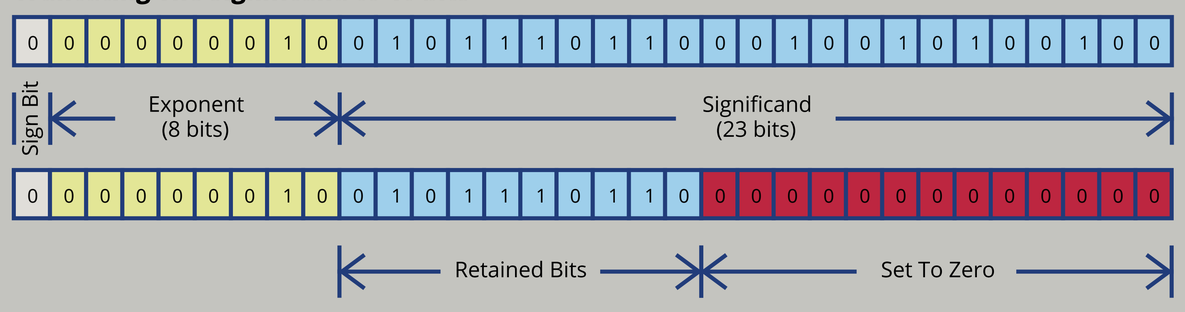


3. Reduced Floating-Point Precision on Standard Hardware

Reducing the precision of a floating-point number can be done simply by reducing the number of bits used to represent it. Conventional CPU hardware typically limits us to using either 32-bit or 64-bit floating-point numbers, meaning support for arbitrary representations is not readily available.

A relatively straightforward way to approach this problem is to emulate the reduced precision in software, whilst still using conventional 32- or 64-bit floating-point representations in hardware. The simplest emulation strategy is to truncate the number of bits in the significand of the floating-point number by setting a given number of the least significant bits to zero.

Truncating The Significand to 10 bits

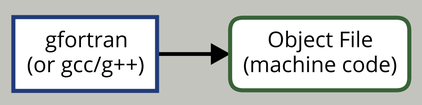


Rounding is performed if the left-most bit to be zeroed is set. This is done to avoid introducing biases caused by always rounding in the same direction.

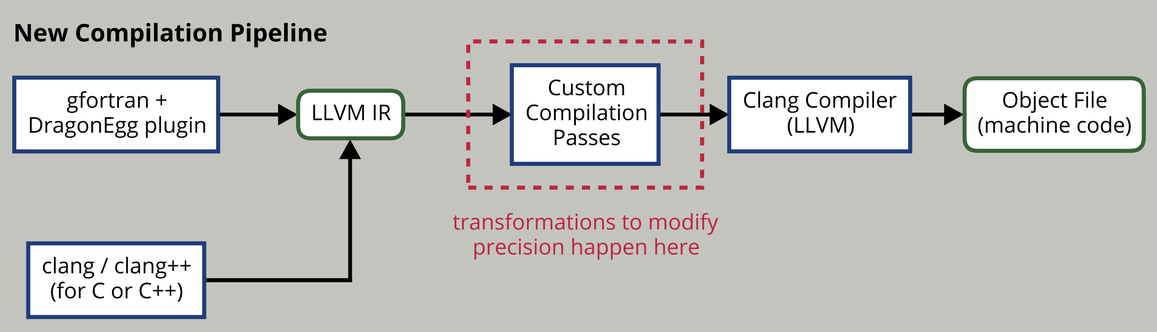
4. Constructing a Reduced-Precision Compiler with LLVM

A specialised compiler has been constructed using the LLVM [1] compiler toolkit. The LLVM framework is a set of modular compiler components and tools that allows us to create custom compilation steps that can be inserted into a standard compilation pipeline. For this research a compiler module that modifies the number of bits in the significand of floating-point numbers was written.

Typical Compilation



New Compilation Pipeline



Fortran source code is first processed by gfortran with the DragonEgg [2] plug-in, which is part of the LLVM project. This process works for code written to any Fortran standard supported by gfortran (i.e., 77/90/95/2003). For C or C++ code the initial compilation can be done with the clang [3] or clang++ compiler front-ends, which are also part of the LLVM project.

The custom compiler pass modifies code after it has been turned into LLVM intermediate representation (IR), which is an accessible yet language agnostic representation of the original source code. The compiler pass introduces truncation in the significand of each floating-point number used in the code.

5. How the Compiler Works

The compilation pass works by inserting and re-writing LLVM IR instructions so that floating point numbers are always represented with the required number of bits in the significand.

The simplest case to consider is how to store a floating-point number in memory with reduced precision. To do this the compiler re-writes storage instructions to ensure all floating-point values are truncated before being stored in memory:

Example: Storing a floating-point number

Consider the IR storage instruction for the expression `a = 1.234324621`:

```
store float 0x3FF3BFCB20000000, float* %a, align 4
```

Our compiler will modify this instruction to apply a reduction of precision before storage:

```
%tmp01 = call float @reduce_precision_float(float 0x3FF3BFCB20000000)
store float %tmp01, float* %a, align 4
```

Compound expressions are represented by multiple instructions in LLVM IR, so a reduction of precision needs to be performed after each operator is applied:

Example: Handling intermediate values

Consider the IR instructions for the expression `b = a * 4 - 1`:

```
%0 = load float* %a, align 4           ; load variable a from memory into register %0
%1 = fmul float %0, 4.000000e+00       ; multiply by 4 and store in register %1
%2 = fsub float %1, 1.000000e+00       ; subtract 1 and store in register %2
store float %2, float* %b, align 4     ; store the result in memory in the variable b
```

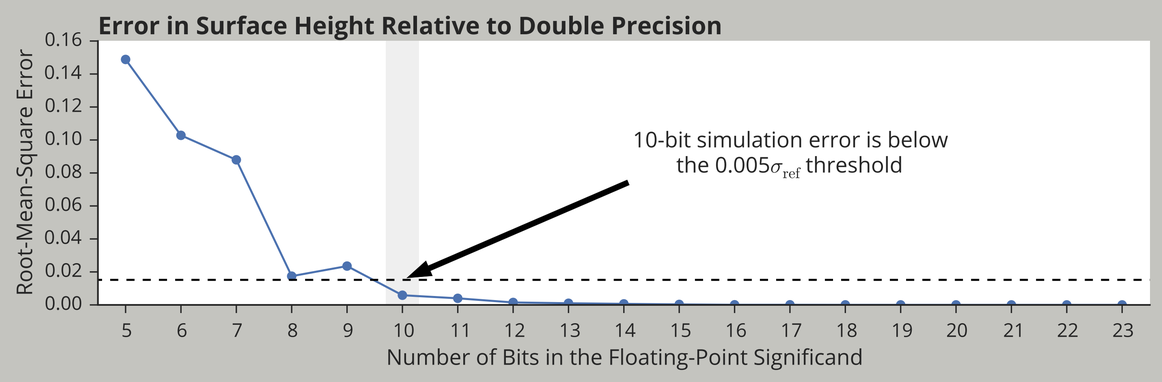
Our compiler will modify these instructions to ensure the result of each instruction in the sequence is represented with reduced precision:

```
%0     = load float* %a, align 4
%1     = fmul float %0, 4.000000e+00
%tmp01 = call float @reduce_precision_float(float %1)
%2     = fsub float %tmp01, 1.000000e+00
%tmp02 = call float @reduce_precision_float(float %2)
store float %tmp02, float* %b, align 4
```

6. A Simple Demonstration

The compiler has been applied to a shallow-water equation model to determine the smallest number of significand bits that can be used whilst still producing a useful simulation. The model is run for 500 time-steps and the error in the height field compared to an accuracy threshold of $0.005\sigma_{\mathrm{ref}}$ (arbitrary but rather strict), where σ_{ref} is the standard deviation of the height field from a double precision reference.

Error in Surface Height Relative to Double Precision



The simulation with only 10 bits in the significand produces errors within tolerance, indicating that it is possible to lose a large number of bits from the significand and see only a relatively small error in the result. The compiler made it trivial to produce this experiment, requiring only 3 lines of code to be added to the original source to control precision.

7. Summary and Further Work

The compiler has the following benefits:

- Makes implementing reduced floating-point precision simple even in a complex model code.
- Works for any language with an LLVM front-end including Fortran (all variants) and C/C++.

However, the compiler is somewhat limited in flexibility, with precision control restricted to setting the number of retained bits in the significand and turning the emulation on and off dynamically.

We have also developed an alternate emulator in the form of a Fortran 90 software library. This can be more flexible than the compiler approach, but is much more tedious to apply even to modest sized codes, and can only be applied to Fortran 90+ codes. The compiler approach can be rapidly applied to large codes to answer broad questions about floating-point precision, whereas the software emulator can be used for more in-depth analysis of floating-point precision.

We intend to apply the compiler described here to large portions of the ECMWF forecast model IFS as part of ongoing research into inexact processing in full-scale NWP models.

References

[1] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, pages 75–88. IEEE Computer Society, 2004.
[2] DragonEgg — using LLVM as a GCC backend. `http://dragonegg.llvm.org`.
[3] clang: A C language family frontend for LLVM. `http://clang.llvm.org`.

`http://www2.physics.ox.ac.uk/contacts/people/dawson`
`andrew.dawson@physics.ox.ac.uk`