Using Fixed Precision Adder Class Templates for Better Reproducibility in Parallel Applications

(or: An Elegant C++-Solution to a Problem You Probably Didn't Know You Had, but Should at Least Be Aware Of)

Elmar Westphal - PGI/JCNS-TA, Scientific IT Systems, Forschungszentrum Jülich GmbH, Germany

JÜLICH Forschungszentrum

Motivation

The results of series of floating point operations may be order-dependent:

```
float a=10000000,b=0.5;
assert(a+(b+b)==(a+b)+b);
```

```
Assertion failed: (a+(b+b)==(a+b)+b), function main…
```

- Differences are (usually) small
- Results may (or may not) be good enough

```
Generating 100000 random numbers
sum for "float, original order" is   15.815774
sum for "float, ascending order" is  15.811079
sum for "float, descending order" is 15.775952
```

- In parallel codes, grouping may influence the order of operations
- Number of parallel threads may influence grouping and results

```
sum for "float, original order, OpenMP, 2 thread(s)" is 15.815803
sum for "float, original order, OpenMP, 4 thread(s)" is 15.815580
```

- Problematic to debug
- Even more problematic to explain to customers or "management"

Explanation

- During additions, mantissae of floating point numbers may be shifted
  - ☞ Numbers may lose significant bits
    - ☞ Number of possibly lost bits depends on differences in orders of magnitude
      - ☞ Orders of magnitude may progress differently with different orders of operations

Using Higher Precision as a (non-)Fix

- Double precision floating point numbers produce better (but not perfect) results:

```
sum for "double, original order" is   15.81579850049832
sum for "double, ascending order" is  15.81579850052003
sum for "double, descending order" is 15.81579850047638
```

- Also, sooner rather than later we would run out of higher precision data types

OpenMP Reductions and Custom Types in C++

- By default, OpenMP will not be able to perform i.e. reduction operations on custom types
- Custom reduction operations can be defined, also on type template parameters

```
template<typename Tsum, typename T>
Tsum vector_sum_omp(std::vector<T>& v) {
  Tsum sum(0);
#pragma omp declare reduction(Tsum_plus: Tsum: omp_out += omp_in )
#pragma omp parallel for reduction(Tsum_plus:sum)
  for(size_t i=0;i<v.size();++i)
     sum+=v[i];
   return sum;
}
```

Performance Considerations

- More complex than simple addition
  - Additional multiplication and type conversion
  - Integer and floating point arithmetic may differ in performance
- Latency for fetching uncached data is significant
  - Overhead may hide additional runtime
- Real world applications with heavy use show no significant performance impact
- **Your mileage may vary!**

Fixed Precision Adders

Number of digits for integral and fraction part is fixed:

IIII.FFFFF

- The maximum range must be set before use (here at compile time)
- FPAs only consist of a running sum, usually stored in an integer type
  - ratio between range and the type's maximum value is used as normalisation factor (bias)
- Operands are normalised and added to the sum
- **Lost bits (if any) are the same, regardless of order**
- For reading, sum is denormalised and returned

A decimal example:

`123.456789 * bias = 12345679 (bias=100000` for 5 fractional digits)

Binary Fixed Precision Adders

- Running sum is stored in an integer of type Tsum, composed of
  - 1 sign-bit
  - N_{IBits}=ceil(log2(range)) bits for the integral part
  - N_{FBits}=8*sizeof(Tsum)-N_{IBits}-1 for the fractional part
  - bias is $2^{N_{FBits}}$
- An FPA with a range of 1000 using 32-bit integers would have N_{IBits}=10, N_{FBits}=21, bias=2^{21}:

  `SIIIIIIIIIIFFFFFFFFFFFFFFFFFFFFF`

Example: adding π

```
123.456789 *2^21=258907652= 00001111011011101001111000000100
  3.1415927*2^21=   6588397= 00000000011001001000011111101101
                            --------------------------------
                            00001111110100110010010111110001

                          = 265496049 = 126.59838*2^21
```

Using C++--Templates for Fixed Precision Adders

- Configurable templates help cover a wide range of use cases
  - The internal type of the running sum can be set to **adjust precision and memory footprint**
  - Setting the range allows **maximizing precision for the given type**
  - A default type for conversion allows **nearly seamless integration**
- All important conversion values and factors can be **calculated at compile time:**
  - The number of bits available for integral and fractional part
    - Needs a constexpr log2()
  - The (un)bias value for (de)normalising operands
  - This **minimizes the impact on performance**

A constexpr log2 for integral operands and return values

- Modern C++ offers the means to calculate to calculate even not so obvious stuff at compile time

```
static constexpr int log2constexpr(size_t n) {
  return n<=1 ? 0 : 1+log2constexpr(n/2);
} // returns 0 for n=0, mathematically, it should be -inf or undefined
```

Fixed precicion adder caveats

- Results are reproducible, but not exact and/or perfect
  - Obtaining perfect results from usually imperfect input data is tricky, anyway...
- Range of operations must be known in advance
  - **Exceeding the preset range is undefined behavior**
    - This also applies to intermediary results!
- FPAs usually trade precision for reproducility
  - Precision loss depends on the ratio of preset range and actually used numbers
  - **For small ratios, precision might actually be better**

Sample Implementation

```
// Range is the range of numbers to be processed,
// Tout is the type presented to the adder-agnostic world,
// Tsum is the internal adder type
// Exceeding the range is undefined behavior

template<       size_t Range,
                typename Tout=double,
                typename Tsum=typename std::conditional<(sizeof(Tout)>4),int64_t,int32_t>::type>
class fixed_precision_adder {

        // Sanity checks
        static_assert(Range>0,"Adder range can't be zero!");
        static_assert(std::is_integral<Tsum>::value,"Internal adder type must be integral!");
        static_assert(std::is_signed<Tsum>::value,"Internal adder type must be signed!");

        // The size of our internal type minus the signbit
        static constexpr int available_bits() { return 8*sizeof(Tsum)-1; }

        // The number of bits we need for the integral part of our sum
        static constexpr int bits_for_range() { return log2constexpr(Range)+1; }

        // Another important sanity check
        static_assert(bits_for_range()<=available_bits(),"Range too big for internal adder type!");

        // All input types except float are cast up to double to avoid loss of precision

        template<typename T>
        using Tbias=typename std::conditional<std::is_same<T,float>::value,float,double>::type;

        // The factor we need to multiply our input with to properly fill our internal sum
        template<typename T, typename Tret=Tbias<T>>
        static constexpr Tret bias() { return Tret(Tsum(1)<<(available_bits()-bits_for_range())); }

        // The opposite of bias()
        template<typename T, typename Tret=Tbias<T>>
        static constexpr Tret unbias() { return Tret(1)/bias<Tret>(); }

        // Apply bias and round/convert to integer of matching size
        // Order of multiplication and conversion is important

        template<typename Tval>
        Tsum convert(const Tval& val) {
                if (sizeof(Tsum)<=4)
                        return Tsum(llrint(val*bias<Tval>()));
                else
                        return Tsum(lrint(val*bias<Tval>()));
        }

        // The one and only piece of runtime data
        Tsum sum;

public:

        // Constructors. Empty default constructors gives wrong results in OpenMP

        fixed_precision_adder() : sum(0) {}

        template<typename T>
        fixed_precision_adder(const T& val) : sum(convert(val)) {}

        // The actual work, add the biased integral value to the sum

        template<typename Tval>
        fixed_precision_adder& operator+=(const Tval& val) {
                sum+=convert(val);
                return *this;
        }

        // If the other one is a fixed precision adder of the same flavor, just add the sums directly
        // Could be extended for different fixed precision adders

        fixed_precision_adder& operator+=(const fixed_precision_adder& other) {
                sum+=other.sum;
                return *this;
        }

        // std::accumulate needs operator+

        template<typename Trhs>
        fixed_precision_adder operator+(const Trhs& rhs) {
                return fixed_precision_adder(*this)+=rhs;
        }

        // If necessary, disguise as Tout (usually float or double)
        operator Tout() const { return float(sum)*unbias<double>(); }

        // Don't disguise as anything else, unless someone asks
        template<typename T>
        explicit operator T() const { return T(sum)*unbias<T>(); }
};
```

Disclaimer: On this poster, `float`, `double` and "floating point"-numbers in general are assumed to be numerical data types as described in IEEE standard 754-1985. It is also assumed that no over- or underflows occur.