Multiple Double and Multiword Arithmetic
Part I: Big Integers and Big Reals

Jan Verschelde[†]

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/∼jan
https://github.com/janverschelde
https://www.youtube.com/@janverschelde5226
janv@uic.edu

Ada Europe 2025 tutorial, 10 June, Paris, France.

[†]Supported by a 2023 Simons Travel Award.

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic

2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow

3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic
2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow
3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Overview

beyond hardware arithmetic

1. Hardware arithmetic is often insufficient to get correct results.
2. The cost overhead of software defined arithmetic
   can be compensated by parallel computations.

This tutorial is about

1. the big numbers introduced in Ada 2022, and
2. multiple double arithmetic.

The material is introduced via examples and code in Ada, available at
`https://github.com/janverschelde/Ada-Europe-2025-Tutorial`.

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic
2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow
3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Plan of the Tutorial

1. The big numbers introduced in Ada 2022.
2. Multiple doubles extend the precision with floating-point arithmetic.
3. Parallelism offsets the cost overhead of software arithmetic.
4. Vectorization defines the layout of data for pipelining.

Big Integers and Big Reals

Verifying Floating-Point Arithmetic

- Automatically verifying the correctness of results obtained by floating-point arithmetic remains a research problem.
- A recent preprint posted on the arxiv preprint server:

  David K. Zhang and Alex Aiken:
  Automatic Verification of Floating-Point Accumulation Networks.
  `arXiv:2505.18791v1 [math.NA] 24 May 2025`
  `https://arxiv.org/pdf/2505.18791`
- Ada programmers have their language to verify the correctness.

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic

2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow

3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Big Integers

a first example from `https://learn.adacore.com`

- Introduced in Ada 2022 as type `Big_Integer`.
- Start an Ada program as

```
with Ada.Text_IO;
with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;
```

- Write 2^{256} as follows:

```
Ada.Text_IO.Put_Line(Big_Integer'Image(2 ** 256));
```

- The output is

```
1157920892373161954235709850086879078532699846656405640394
57584007913129639936
```

Binary and Hexidecimal Formats

2^{256} in binary is one 1 followed by 256 zeros.
2^{256} in hexadecimal is one 1 followed by 64 zeros.

- Writing the big integer in hexadecimal and binary format:

```
Ada.Text_IO.Put_Line(To_String(2**256, base=>16));
Ada.Text_IO.Put_Line(To_String(2**256, base=>2));
```

- The output is

```
 16#1 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000#
 2#1 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000#
```

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic
2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow
3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Fibonacci Numbers

As an example where hardware integers are no long sufficient, consider the Fibonacci numbers:

$$f_0 = 0, \quad f_1 = 1, \quad \text{and for } n > 1 : f_n = f_{n-1} + f_{n-2}.$$

Code snippet to compute the *n*-th Fibonacci number:

```
    previous : Big_Integer := To_Big_Integer(0);
    current : Big_Integer := To_Big_Integer(1);
    next : Big_Integer;

  begin
    for i in 1..n loop
      next := previous + current;
      previous := current;
      current := next;
    end loop;
```

the 1000-th Fibonacci Number

The 1000-th Fibonacci number f_{1000} takes 998 additions:

```
 43466557686937456435688527675040625802564660517371780402
481729089536555417949051890403879840079255169295922593080
322634775209689623239873322471161642996440906533187938298
969649928516003704476137795166849228875
```

The output has 210 decimal places.

With very little computations, numbers can grow quickly.

Exact results are not always possible and not always needed.

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic
2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow
3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Floating-Point Numbers

A floating-point number consists of

1. one sign bit,
2. a normalized fraction: the leading bit is nonzero, and
3. an exponent.

Definition (floating-point representation)

The *floating-point representation* $f\ell(x)$ of a real number $x \in \mathbb{R}$ is

$$f\ell(x) = \pm.bb\ldots b \times 2^e,$$

stored compactly as the tuple $(\pm, e, bb\ldots b)$.
The *representation error* is $|f\ell(x) - x|$.

Floating-Point Formats

Hardware supports single precision (32-bit), double precision (64-bit), and long double precision (80-bit), summarized below:

	number of bits			
precision	sign	exponent	fraction	total
single	1	8	23	32
double	1	11	52	64
long double	1	15	64	80

A 64-bit floating-point number has

- 1 sign bit s, 0 for positive, 1 for negative,
- 11 bits e_1, e_2, ..., e_{11} in the exponent, and
- 52 bits f_1, f_2, ..., f_{52} in the fraction, $f_1 \neq 0$.

s	e_1	e_2	$\cdots$	e_{11}	f_1	f_2	$\cdots$	f_{52}

Number Line Example

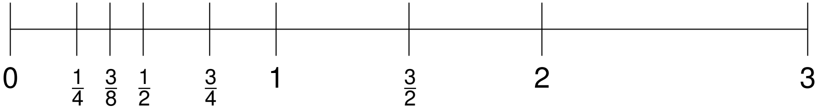
distribution of the floating-point numbers

Consider a floating-point number system with basis 2

1. with two bits in the (normalized) fraction, and
2. with exponents -1, 0, $+1$, $+2$.

We display all positive floating-point numbers in this system:

$$
\begin{array}{lllll@{\qquad}lllll}
.10\ 2^{-1} & = & 0.01 & = & 1/4 & .11\ 2^{-1} & = & 0.011 & = & 3/8 \\
.10\ 2^{0} & = & 0.1 & = & 1/2 & .11\ 2^{0} & = & 0.11 & = & 3/4 \\
.10\ 2^{+1} & = & 1 & & & .11\ 2^{+1} & = & 1.1 & = & 3/2 \\
.10\ 2^{+2} & = & 10 & = & 2 & .11\ 2^{+2} & = & 11 & = & 3
\end{array}
$$



error $|f\ell(x) - x| \leq 1/8$ error $|f\ell(x) - x| \leq 1/2$

Machine Precision

Definition (machine precision)

The number *machine precision* $\epsilon_{\rm mach}$ is the distance between 1 and the smallest floating-point number greater than one.
For basis B and size p of the fraction: $\epsilon_{\rm mach} = B^{-p}$.

For $0 < \epsilon < \epsilon_{\rm mach}$: $(1 + \epsilon) - 1 \neq \epsilon + (1 - 1)$.

The machine precision as supported by hardware single floats (32-bit), double floats (64-bit), and long double floats (80-bit) is below:

precision	number of bits: sign	exponent	fraction	total	machine precision
single	1	8	23	32	$2^{-23} \approx$ 1.192e-07
double	1	11	52	64	$2^{-52} \approx$ 2.220e-16
long double	1	15	64	80	$2^{-64} \approx$ 5.421e-20

the Smallest and Largest Exponent

An exponent $e \in [e_{\min}, e_{\max}]$ where
$e_{\min}$ is the smallest exponent and $e_{\max}$ is the largest exponent.

	number of bits				exponent range	
precision	sign	exponent	fraction	total	$e_{\min}$	$e_{\max}$
single	1	8	23	32	-126	$+127$
double	1	11	52	64	-1022	$+1023$
long double	1	15	64	80	-16382	$+16383$

Special values for the exponent for double precision:

- `111 1111 1111`, nonzero fraction : `-NaN`, not a number;
- `111 1111 1111`, zero fraction : `-Inf`, represents $-\infty$;
- `000 0000 0000` : numbers that are not normalized;
- `011 1111 1111`, zero fraction : `+Inf`, represents $+\infty$.

Extracting Exponent and Fraction

`long_long_integer` is a 64-bit integer, renamed as `integer64`.

```
x : long_float := 0.1;
f : long_float := long_float'fraction(x);
e : integer64 := integer64(long_float'exponent(x));
c : long_float := long_float'compose(f, e);
s : long_float := long_float'compose(f, 52);
m : integer64 := integer64(long_float'truncation(s));
```

The number `c` equals the original `x`,
and `s` is used to turn the fraction into a 64-bit integer.

the Fraction of `0.1`

Writing the fraction in binary and hexadecimal, with the statements

```
integer64_io.put(m,1,base=>2);
integer64_io.put(m,1,base=>16);
```

gives as output

```
2#1100110011001100110011001100110011001100110011001101#
16#CCCCCCCCCCCCD#
```

Working with `0.1` as a `long_float` results in a *representation error*, as `0.1` does not have a finite binary expansion.

$$\texttt{0.1} \neq \frac{1}{10}$$

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic
2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow
3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Big Reals

comparing with `long_float`

- Introduced in Ada 2022 as type `Big_Real`.
- Start an Ada program as

```
with Ada.Numerics.Big_Numbers.Big_Reals;
use  Ada.Numerics.Big_Numbers.Big_Reals;
```

- Comparing `0.1` and $1/10$:

```
 x : constant long_float := 0.1;
 y : constant Big_Real
   := To_Big_Real(1)/To_Big_Real(10);
```

- `Big_Real` arithmetic is exact rational arithmetic.

Comparing `0.1` with $1/10$

The package `conversions` is an instantiation of `Ada.Numerics.Big_Numbers.Big_Reals.Float_Conversions` with the type `long_float`, needed to compute `z`:

```
  z : constant Big_Real := conversions.To_Big_Real(x)

begin
  Put("y : ");
  Put_Line(To_String(y,2,32,0));
  Put("  numerator of y :");
  Put_Line(Big_Integer'Image(Numerator(y)));
  Put("denominator of y :");
  Put_Line(Big_Integer'Image(Denominator(y)));
  Put("x - y : ");
  Put_Line(To_String(z,2,32,0));
```

Comparing `0.1` with $1/10$

avoiding representation errors

The output of the code on the previous slide:

```
y :  0.10000000000000000000000000000000000
  numerator of y : 1
denominator of y : 10
x - y :  0.00000000000000000555111512312578
```

The `y` defined as `To_Big_Real(1)/To_Big_Real(10)` is indeed the rational number $1/10$.

Computing with `Big_Real` *numbers is the same as computing with rational numbers.*

Big Integers and Big Reals

Applications of `Big_Real` Numbers

1. Rational approximations of π.
2. Solve 2-by-2 linear systems exactly.
3. Approximation of square roots.

Rational Approximations of π

The package `conversions` is an instantiation of
`Ada.Numerics.Big_Numbers.Big_Reals.Float_Conversions`
with the type `long_float`, used to compute `y`:

```
x : long_float := Ada.Numerics.Pi;
y : Big_Real := conversions.To_Big_Real(x);

Put(Big_Integer'Image(Numerator(y)));
Put(" /");
Put_Line(Big_Integer'Image(Denominator(y)));
```

showing

```
884279719003555 / 281474976710656
```

A Sequence of Approximations

```
2#1100100000000000000000000000000000000000000000000000#
3.125000000000E+00 25 / 8
2#1100100100000000000000000000000000000000000000000000#
3.140625000000E+00 201 / 64
2#1100100100001000000000000000000000000000000000000000#
3.141113281250E+00 6433 / 2048
2#1100100100001100000000000000000000000000000000000000#
3.141357421875E+00 12867 / 4096
2#1100100100001110000000000000000000000000000000000000#
3.141479492188E+00 25735 / 8192
2#1100100100001111000000000000000000000000000000000000#
3.141540527344E+00 51471 / 16384
2#1100100100001111100000000000000000000000000000000000#
3.141571044922E+00 102943 / 32768
2#1100100100001111110000000000000000000000000000000000#
3.141586303711E+00 205887 / 65536
2#1100100100001111110110000000000000000000000000000000#
3.141592025757E+00 1647099 / 524288
...
```

Steps in the Code

1. `type integer64 is new long_long_integer;`
2. Declare

   `type unsigned_integer64 is mod 2**integer64'size;`

   Take two numbers of type `unsigned_integer64`.
   With `nbr and mask` where `mask` is a bit pattern,
   we select the bits of the fraction `nbr`.
3. Adjust the `mask` to select more leading bits
   of the fraction of π.
4. Write the numerator and denominator of the `Big_Real`
   obtained after converting the composed `long_float`,
   composed with the leading bits of the fraction of π.

Solving 2-by-2 Linear Systems

applying Cramer's rule

Consider a 2-by-2 linear system $Ax = b$:

$$\left[\begin{array}{cc} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right] = \left[\begin{array}{c} b_1 \\ b_2 \end{array}\right].$$

If the determinant $\det(A) = a_{1,1}a_{2,2} - a_{2,1}a_{1,2} \neq 0$, then the solution is

$$x_1 = \frac{\det\left(\left[\begin{array}{cc} b_1 & a_{1,2} \\ b_2 & a_{2,2} \end{array}\right]\right)}{\det(A)} \quad \text{and} \quad x_2 = \frac{\det\left(\left[\begin{array}{cc} a_{1,1} & b_1 \\ a_{2,1} & b_2 \end{array}\right]\right)}{\det(A)}.$$

A Random Instance

```
a coefficient matrix :
 2 / 1   7 / 5
 3 / 2   1 / 4
its determinant : -8 / 5
a right hand side vector :
 9 / 8   1 / 1
the solution :
 179 / 256  -25 / 128
the residual :
 0 / 1   0 / 1
```

Steps in the Code

1. Vector and matrix types of `Big_Real`:

```
subtype Matrix_Range is Integer range 1..2;
type Big_Real_Vector is
  array(Matrix_Range) of Big_Real;
type Big_Real_Matrix is
  array(Matrix_Range, Matrix_Range) of Big_Real;
```

2. Generate random integers in the range from 1 to 9 for the numerator and denominators of the coefficients.
3. Define functions for the determinant, to solve, and to compute the residual vector $b - Ax$.

Approximating Square Roots

The square root of a number n is a solution of

$$x^2 - n = 0.$$

Starting at $x := \sqrt{n}$ (double precision), apply Newton's method:

$$x := x - \frac{x^2 - n}{2x}$$

until $|x^2 - n|$ is smaller than the desired accuracy.

Newton's method converges quadratically:
the number of correct decimal places doubles in each step.

Running Newton's Method with `Big_Real`

Given are two `Big_Real` numbers: `x` and `tol`.

```
  two : constant Big_Real := To_Big_Real(2);
  flx : constant Long_Float
      := conversions.From_Big_Real(x);
  xbr : constant Big_Real
      := conversions.To_Big_Real(SQRT(flx));
  y,z : Big_Real;

begin
  z := xbr;
  y := z*z - x;
  for i in 1..100 loop
    z := z - y/(two*z);
    y := z*z - x;
    exit when (abs(y) < tol);
  end loop;
```

Approximating $\sqrt{5}$

Running with `x` := 5 and `tol` := `1.0E-255` gives

```
  numerator : 198722315814490820940550324768121510508791 47
4526664147032700535852840786587519349416977059375435597922
5749437769845429723293048671021001611336566850938648039607
1368885001714358482097272093314584052162820070128853465636
606660515950399520406721
denominator : 888713213614765926312515801568410129491154 01
3007494916989181911352904768534144466717401786754267481046
0851998064225461709997178176912050977614188447809927517990
3852066582365111217851561947716243292037718152798799007191
18533076467259778531328
```

as the approximation for $\sqrt{5}$ accurate up to 255 decimal places.

Big Integers and Big Reals

1. Overview
   - beyond hardware arithmetic
   - plan of the tutorial
   - verifying floating-point arithmetic
2. Big Integers
   - binary and hexadecimal formats
   - avoiding overflow
3. Big Reals
   - floating-point numbers
   - exact rational aritmetic
   - applications
   - expression swell

Expression Swell and Roundoff Errors

- When the size of intermediate numbers and expressions grow too large, we encounter *expression swell*, a well known problem in computer algebra, and in all exact computations.

  If the end result is also large,
  then there is nothing one can do.

- In many applications, accurate results can be obtained by working in limited precision, if *roundoff errors* remain bounded during the computations, which is in the domain of numerical analysis.

  If roundoff errors cannot be bounded,
  then there is nothing one can do.

Verifying Floating-Point Arithmetic

Ada programmers have their language to automatically verify the correctness of results obtained by floating-point arithmetic:

1. Compute once with `Big_Number` arithmetic,
2. execute then `long_float` arithmetic, and
3. report the difference of the two outcomes.

One could run a program in the `Big_Number` mode, or in the hardware arithmetic mode.

Exercises

1. The Lucas numbers L_n are defined as

$$L_0 = 2, \quad L_1 = 1, \quad \text{for } n > 1 : L_{n-1} + L_{n-2}.$$

   Compute the first 1000 Lucas numbers with big integers.
   What is the largest Lucas number that you can compute?
2. Use the first bits of `Ada.Numerics.e`
   and compute consecutive rational approximations with big reals.
3. Extended the application of Cramer's rule
   to solve 3-by-3 linear systems with rational coefficients.
4. Apply Newton's method to compute cube roots with big reals.

Multiple Double and Multiword Arithmetic
Part II: Double Doubles and Multiple Doubles

Jan Verschelde†

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
`http://www.math.uic.edu/~jan`
`https://github.com/janverschelde`
`https://www.youtube.com/@janverschelde5226`
`janv@uic.edu`

Ada Europe 2025 tutorial, 10 June, Paris, France.

†Supported by a 2023 Simons Travel Award.

Double Doubles and Multiple Doubles

1. Introduction
   - definitions
   - objectives
2. Double Doubles
   - doubling the precision
   - accurate floating-point summation
3. Multiple Doubles
   - multiplying the precision
   - benefits and drawbacks

Double Doubles and Multiple Doubles

1. Introduction
   - definitions
   - objectives
2. Double Doubles
   - doubling the precision
   - accurate floating-point summation
3. Multiple Doubles
   - multiplying the precision
   - benefits and drawbacks

Definitions

extending the precision with multiple doubles

- The *precision* is the smallest positive number we can add to one and obtain a number larger than one.
- A *double* is a 64-bit floating-point number, the precision is $2^{-52} \approx 2.220\mathrm{E}{-16}$.
- A *multiple double* is a nonoverlapping sum of doubles, the precision is $2^{m(-52)}$, for m doubles. The precision of a double double is $2^{-104} \approx 4.930\mathrm{E}{-32}$.
- A *multiword* is a sequence of nonoverlapping hardware numbers, of integers or floating-point numbers, representing one big integer or big real number.

Double Doubles and Multiple Doubles

1. Introduction
   - definitions
   - objectives
2. Double Doubles
   - doubling the precision
   - accurate floating-point summation
3. Multiple Doubles
   - multiplying the precision
   - benefits and drawbacks

Objectives of the Tutorial

1. Explain the basics of the multiple double arithmetic.
2. Understand why the overhead is predictable.

   Working with double double arithmetic $\sim$ complex arithmetic.
3. Know the distinction between
   - when needed to use multiple doubles, and
   - situations where there is nothing one can do.

Context and References

Floating-point arithmetic with 64-bit doubles can be extended to gain more accuracy than what only hardware arithmetic gives.

- Volume 2 of *The Art of Computer Programming* by Knuth details properties of floating-point arithmetic.
- Algorithms to extend 32-bit floating-point arithmetic originated in the late sixties [Dekker, Numerische Mathematik 1971].
- The arithmetic is provided in software packages such as
  - QDlib [Hida, Li, Bailey, 2001], and
  - CAMPARY [Joldes, Muller, Popescu, Tucker, 2016].
- A general reference: "*Handbook of Floating-Point Arithmetic*" by J.-M. Muller et al., Springer-Verlag, 2nd edition, 2018.

The topic is both a classic and a new one: in the context of multithreading and acceleration by Graphics Processing Units (GPUs).

Multiple Double Arithmetic in PHCpack

- PHCpack is software for Polynomial Homotopy Continuation, to solve systems of polynomials.
- GNU GPL license, available at `https://github.com/janverschelde/PHCpack`.
    - multiple double arithmetic is available in the folder `src/Ada/Math_Lib/QD`.
    - multiword arithmetic is available in the folder `src/Ada/Math_Lib/Words`.
    - tasking is available in the folder `src/Ada/Math_Lib/Tasking`.
- All examples in this tutorial are made with this code.

Power Series Arithmetic

motivation for multiple double precision

$$\exp(t) = \sum_{k=0}^{d-1} \frac{t^k}{k!} + O(t^d).$$

Recommended precision to represent the series for $\exp(t)$ correctly:

k	$1/k!$	recommended precision	eps
7	2.0e-004	double precision okay	2.2e-16
15	7.7e-013	use double doubles	4.9e-32
23	3.9e-023	use double doubles	
31	1.2e-034	use quad doubles	6.1e-64
47	3.9e-060	use octo doubles	4.6e-128
63	5.0e-088	use octo doubles	
95	9.7e-149	use hexa doubles	5.3e-256
127	3.3e-214	use hexa doubles	

`eps` is the working precision

Double Doubles and Multiple Doubles

1. Introduction
   - definitions
   - objectives
2. Double Doubles
   - doubling the precision
   - accurate floating-point summation
3. Multiple Doubles
   - multiplying the precision
   - benefits and drawbacks

Doubling the Precision

The double double representation of π is the record with values

```
3.141592653589793116e+00
1.224646799147353207e-16
```

where

- the first double is the high part, and
- the second double is the low part.

We can interpret the low part as the error on the first double.

Error Free Transformations

We have the following property:

If the result can be represented exactly in double precision, then the second double is the error on the result.

Example: $\displaystyle \sqrt{\sum_{i=1}^{64} x^2} = 8$, if $x \in \mathbb{C}$ and $|x| = 1$.

The 2-norm of a vector of 64 complex doubles on the unit circle is 8, computed with double doubles:

```
8.0000000000000E+00 - 4.4681574709783 9E-32
```

This property illustrates also that one does not need a long fraction to represent numbers such as `8 - 4.4E-32` correctly.

The Type `double_double`

At the end of the package `Double_Double_Numbers`:

```
private

  type double_double is record
    hi : double_float;   -- most significant part
    lo : double_float;   -- least significant part
  end record;

end Double_Double_Numbers;
```

where `double_float` renames `long_float`.

1. Basic operations work on a pair of doubles (`hi`, `lo`).
2. Functions that overload the operators call the basic operations.

Faithful Rounding

Let R be a system of floating-point numbers.

- $f\ell(x) \in R$ is the floating-point representation of $x \in \mathbb{R}$.
- For any $x, y \in R$: $x * y$ represents $x + y$, $x - y$, $x \times y$, or x/y.

Definition (faithful and optimal rounding)

The floating-point operation $*$ is *faithful*
if for all $x, y \in R$, $f\ell(x * y)$ equals

- either the largest element of R smaller than or equal to $x * y$,
- or the smallest element of R larger than or equal to $x * y$.

The floating-point operation $*$ is *optimal*
if $f\ell(x * y)$ is nearest to $x * y$ for all $x, y \in R$.

Exact Addition

Let $f\ell(x)$ be the floating-point representation of $x \in \mathbb{R}$.

Theorem (Dekker, 1971)

If floating-point addition is optimal and subtraction faithful, then for $x, y \in R$, $|x| \geq |y|$, and

$$z := f\ell(x + y), \quad w := f\ell(z - x), \quad e := f\ell(y - w),$$

then we have

$$e = y - (z - x),$$

or equivalently: e equals the correction term to the addition.

This implies that the error of a floating-point addition can be computed with floating-point arithmetic.

Then, $x + y$ is represented exactly by the tuple (z, e).

Exact Error Computation

To show that $x + y$ is represented exactly by the tuple (z, e),

$$\text{for any } x, y \in R, |x| \geq |y|,$$

via the computations

$$z := f\ell(x + y), \quad w := f\ell(z - x), \quad e := f\ell(y - w),$$

the result is implied by

1. $z - x \in R$, and
2. $y - w \in R$,

because floating-point subtraction is faithful.

$z - x \in R$

1. Denote $x = f_x 2^{e_x}$, $y = f_y 2^{e_y}$, $z = f_z 2^{e_z}$,
   $$|x| \geq |y| \Rightarrow e_x \geq e_y \text{ and } e_z \leq e_x + 1 \text{ (no overflow).}$$
2. Let $d = e_x - e_y$, then $f_z = \text{round}\left(\frac{f_x}{2} + \frac{f_y}{2^{d+1}}\right)$
   and $z - x = \mu 2^{e_x}$.
3. μ satisfies
   $$\mu \leq \left|2f_z - f_x - \frac{f_y}{2^d}\right| + \left|\frac{f_y}{2^d}\right| < 1 + M$$
   where $M = 2^p$, p is the number of bits in the fraction.

Because μ is an integer, it follows that $z - x \in R$.

$y - w \in R$

We computed

$$z := f\ell(x + y), \quad w := f\ell(z - x), \quad e := f\ell(y - w).$$

Observe the following:

1. $|x| \geq |y| \Rightarrow e_x \geq e_y$ and

   therefore, $y - w$ is an integer times 2^{e_y}.

2. $|y - w| \leq |y|$

   Otherwise, x would be closer to $x + y$ than z, contradicting the optimality of floating-point addition.

Thus, we have $y - w \in R$.

Branchless Sum and Error Computation

The sum and error of a floating-point addition can be computed without an if statement, as defined in the procedure below:

```
procedure two_sum ( a,b : in double_float;
                    s,err : out double_float ) is

  bb : double_float;

begin
  s := a + b;
  bb := s - a;
  err := (a - (s - bb)) + (b - bb);
end two_sum;
```

A proof that this works can be found in "*On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*" by Douglas M. Priest, PhD thesis, UC Berkeley, 1992.

Product and Error Computation

The procedure `split` computes high and low word of a double.

```
procedure two_prod ( a,b : in double_float;
                     p,err : out double_float ) is

  a_hi,a_lo,b_hi,b_lo : double_float;

begin
  p := a*b;
  split(a,a_hi,a_lo);
  split(b,b_hi,b_lo);
  err := ((a_hi*b_hi - p)
       + a_hi*b_lo + a_lo*b_hi) + a_lo*b_lo;
end two_prod;
```

This `two_prod` is applied to the high and low parts of the double doubles in the multiplication.

Division of Double Doubles

On input are two double doubles: $x = (x^{\mathrm{hi}}, x^{\mathrm{lo}})$ and $y = (y^{\mathrm{hi}}, y^{\mathrm{lo}})$.

The instructions to compute $q = x/y$ are as follows:

1. $q_1 := x^{\mathrm{hi}}/y^{\mathrm{hi}}$; $a := q_1 \star y$; $q := x - a$;
2. $q_2 := q^{\mathrm{hi}}/y^{\mathrm{hi}}$; $a := q_2 \star y$; $q := q - a$;
3. $q_3 := q^{\mathrm{hi}}/y^{\mathrm{hi}}$;
4. $q^{\mathrm{hi}} := q_1 + q_2$;
5. $q^{\mathrm{lo}} := q_2 - (q^{\mathrm{hi}} - q_1)$;
6. $q := q + q_3$;

Observe that several of those instructions have double doubles as operands and are thus not elementary hardware functions.

Double Doubles and Multiple Doubles

Accurate Floating-Point Summation

Adding long sequences of real numbers is a basic task.

As test sequences, consider geometric sums:

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1},$$

where r is a constant, independently of i, close to 1,
so the numbers are slowly decaying.

- All numbers in the sequence are positive, $r = 0.999\ldots$.
- The sequence is sorted in decreasing order.

Double versus Double Double Arithmetic

For $n = 1,000,000$ and $r = 0.99999$:

- The output with double precision floating-point arithmetic:

```
the sum :  9.99954602798677E+04
  error :  4.07E-9
```

- The output with double double arithmetic:

```
the sum : 9.999546027987177386064425836022 81E+04
  error : 4.93E-26
```

Adding one million numbers is almost instantaneous,
also in double double arithmetic.

The `Big_Real` arithmetic leads to expression swell,
as shown in the next two slides.

Comparing with `Big_Real` Arithmetic

The numerator of the sum for $n = 61$ and $= 0.99999$ is

```
1052531779735280972469014195849592272797535576087703854 97
```

Comparing with `Big_Real` Arithmetic

The denominator of the sum for $n = 61$ and $= 0.99999$ is

```
169814973489612474139441081825651281718030426146816280037
933005643365757266550038256812824668930364805317063633622
266136128185636774744566930065135761825492050291342758980
061441525226658958450534386596374857381215453584142341197
785345367348888372611546504121811720648862220122496580341
905095616252666232534255473748238197218227457813611196208
829554287323813413048108772281771364102510208942348372351
304271212719880268428734629523577315745168641057824870758
680652668283945036292229372120549145626523959179133873977
208591561079219879705964741781252750741680327546431608989
982147046960242178040358119573046185956958055033708256790
797188394447404812989355185027062728884838683665642462816
885894350930870778656374282896554681073275455210026485104
123460374235800730394334006289252525389544409904660501025
145570306719143855193754141909181801201826795185359155756
691839537111551261842083848171427733274853335095562923040
534914419168697344469066593882861526742571861516556248725
58592
```

Double Doubles and Multiple Doubles

Multiplying the Precision

The octo double representation of π is the record with values

```
 3.1415926535897931E+00
 1.2246467991473532E-16
-2.9947698097183397E-33
 1.1124542208633653E-49
 5.6722319796403157E-66
 1.7449862161352486E-83
 6.0293727322495398E-100
 1.9101235468799890E-116
```

where the parts are ranked from high to low significance.

Observe that the first two doubles of the octo double define the double double representation of π.

The Type `octo_double`

At the end of the package `Octo_Double_Numbers`:

```
private

  type octo_double is record
    hihihi : double_float; -- most significant part
    lohihi : double_float; -- second highest word
    hilohi : double_float; -- third highest word
    lolohi : double_float; -- fourth highest word
    hihilo : double_float; -- fourth lowest word
    lohilo : double_float; -- third lowest word
    hilolo : double_float; -- second lowest word
    lololo : double_float; -- least significant part
  end record;

end Octo_Double_Numbers;
```

Error Free Transformations

Example: $\displaystyle\sqrt{\sum_{i=1}^{64} x^2} = 8$, if $x \in \mathbb{C}$ and $|x| = 1$.

The 2-norm of a vector of 64 complex doubles on the unit circle is 8, computed with multiple doubles:

```
 1d : 8.00000000000000E+00 - 4.46815747097839E-32
 4d : 8.00000000000000E+00 + 8.23258305145073E-65
 8d : 8.00000000000000E+00 - 5.56764060802733E-128
16d : 8.00000000000000E+00 - 1.54394135726410E-257
```

One does not need a long fraction to represent numbers such as `8 - 1.54E-257` correctly.

Cost Overhead

a motivation for parallel computing

The number of floating-point operations, for a multiple double addition `add`, multiplication `mul`, and division `div`, for increasing number *m* of doubles:

m	`add`	`mul`	`div`	`avg`
2	20	23	70	37.7
4	89	336	893	439.3
8	269	1742	5126	2379.0
16	925	11499	33041	15155.0

Observe: while the data is doubled, the average `avg` number of operations increased almost tenfold, which makes the computations much more *compute bound*, rather than *memory bound*.

Double Doubles and Multiple Doubles

1. Introduction
   - definitions
   - objectives
2. Double Doubles
   - doubling the precision
   - accurate floating-point summation
3. Multiple Doubles
   - multiplying the precision
   - benefits and drawbacks

Benefits and Drawbacks

Two apparent disadvantages of multiple double arithmetic:

- Fixed levels of precision, for example:
  working with 73 bits of precision is not possible.
- The exponent size remains fixed,
  working with extremely large or tiny numbers is not possible.

The advantages:

+ Upgrading or downgrading the precision of the numbers simply happens by adding or removing doubles.
+ The cost overhead is predictable.
+ Computations become compute bound for increasing precisions.

Exercises

1. Compute the geometric sum with ratio $r = 1.00001$ instead of 0.99999, so the numbers in the sequence are increasing, for sufficiently large values of n.
2. Write code to apply Newton's method to compute cube roots, using staggered precisions, starting at the cube root of a number in double precision, doubling the precision in each Newton step.
3. Time the code for computing geometric sums, comparing double, double double, and quad double arithmetic. Do the timings agree with the cost overhead factors?

Multiple Double and Multiword Arithmetic
Part III: Multithreading for Quality Up

Jan Verschelde[†]

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/∼jan
https://github.com/janverschelde
https://www.youtube.com/@janverschelde5226
janv@uic.edu

Ada Europe 2025 tutorial, 10 June, Paris, France.

[†]Supported by a 2023 Simons Travel Award.

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking
2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs
3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking
2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs
3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Multithreading, Thread Safety, and Ada Tasking

definition of speedup

- On a shared memory parallel computer,
  all threads have access to the entire main memory.
- A program is *thread safe* if its parallel execution produces the *same* results as a sequential run.
  - Errors occur when two threads alter the same memory locations.
  - The order of computations may change the roundoff.
- Ada tasks are mapped onto kernel threads, enabling speedup:

$$\text{speedup} = \frac{\text{sequential execution time}}{\text{parallel execution time}}.$$

Quality Up

In analogy with speedup, we can define quality up:

$$\text{quality up } Q(p) = \frac{\text{quality on } p \text{ processors}}{\text{quality on 1 processor}}$$

$Q(p)$ measures improvement in quality using p processors, keeping the computational time fixed.

If we can afford to wait the same amount of time on 1 processor, by how much can we improve the quality with p processors?

Confusing precision with accuracy, if the cost overhead of a higher precision is p, then running on p processors offsets the overhead.

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking
2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs
3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Objectives of the Tutorial

1. Apply Ada tasking to achieve quality up.
2. Understand why the multiple double arithmetic is thread safe.
3. How many processors are needed to afford multiple precision?

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking
2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs
3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Hello Tasking!

```
procedure Hello_Tasks ( p : in integer := 4 ) is

  task type worker ( idnbr : integer );

  task body worker is
  begin
    Ada.Text_IO.Put_Line ("Task" & idnbr'Image & " says hello.");
  end worker;

  procedure launch ( i : in integer ) is

    w : worker(i);

  begin
    if i < p
     then launch(i + 1);
    end if;
  end launch;

begin
   launch(1);
end Hello_Tasks;
```

Why Does `hello_tasking` Work?

We consider three stages:

1. Creation: `task type worker ( idnbr : integer );`

   where the identification number `idnbr` is the task discriminant.

   The *i*-th task is created when the variable `w` in

   ```
   w : worker(i);
   ```

   is elaborated.

2. Tasks activate after the elaboration of the declarative part.

3. Tasks execute immediate after a successful activation, where execute means entering the ready state.

The parallelism happens because only the declarative part needs to be elaborated for a task to execute.

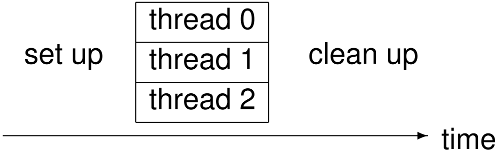
Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking
2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs
3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

The Work Crew Model

Instead of the manager/worker model,
with threads we can apply a more collaborative model.

A computation performed by three threads in a work crew model:



If the computation is divided into many jobs stored in a queue,
then the threads grab the next job, compute the job,
and push the result onto another queue or data structure.

Important for memory management:

- set up: all memory allocations, *before* the run,
- clean up: all memory deallocations, *after* the run.

An Array of Workers

Making an array of workers:

```
Task 1 says hello with id workers(4)_000000030D99360.
Task 2 says hello with id workers(1)_000000030D8F130.
Task 3 says hello with id workers(2)_000000030D92740.
Task 4 says hello with id workers(3)_000000030D95D50.
```

Observe the difference between task number and its entry in the array `workers`.
The last output is obtained via

```
     taskid : constant Ada.Task_Identification.Task_Id
            := Ada.Task_Identification.Current_Task;
```

which returns the identity of the task.

The Code in `hello_task_array`

```
procedure hello_tasks ( p : in integer := 4 ) is

  task type worker;

  task body worker is

    idnbr : constant integer := Identification.Number;
    taskid : constant Ada.Task_Identification.Task_Id
           := Ada.Task_Identification.Current_Task;

  begin
     Ada.Text_IO.Put_Line ("Task" & idnbr'Image
                                  & " says hello with id "
       & Ada.Task_Identification.Image(taskid) & ".");
  end worker;

  workers : array(1..p) of worker;

begin
  null;
end hello_tasks;
```

The `id_generator` Assigns Unique Numbers

```
protected id_generator is

  procedure get ( id : out integer );
  -- returns a unique identification number

private
  next_id : integer := 1;
end id_generator;

protected body id_generator is
  procedure get ( id : out integer ) is
  begin
     id := next_id;
     next_id := next_id + 1;
  end get;
end id_generator;
```

Operations on data encapsulated by a protected object are executed with mutually exclusive access.

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking
2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs
3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Geometric Inner Products

Consider inner products of geometric sequences:

$$\sum_{i=1}^{n} r^i s^i, \quad 0 < r < s, r \approx 1.$$

+ Scalable experiment on vectors, without data arrays.
+ Ratios allow to control the growth of the numbers.

For $n = 10^9$, with ratios $r = 1 + 10^{-10}$ and $s = 1 - 10^{-10}$, we obtain

```
Computing an inner product of size 1000000000
The inner product : 1.00000000000000E+09
```

− The result is wrong: $rs = 1 + 10^{-20} = 1.0$ in double precision.
+ On a Windows 11 Intel i9-13900HZ 2.2Gz: 826 milliseconds.

We compare *wall clock times*, using `Measure-Command` or `time`.

Performance

FLOPS = number of floating-point operations per second

```
    result : long_float := 0.0;
    x : long_float := 1.0;
    y : long_float := 1.0;

  begin
    for i in 0..(dim-1) loop
      result := result + x*y;
      x := x*r;
      y := y*s;
    end loop;
```

In 826 milliseconds, in the body of the the loop we count one addition and three multiplications, which runs one billion times:

$$\frac{4,000,000,000}{0.826} = 4,842,615,012.1 \text{ FLOPS} = 4.8 \text{ GIGAFLOPS}.$$

Running with Double Double Arithmetic

On the Window Subsystem for Linux on the same computer:

```
$ time ts_mtgeoprod 1 2
Running in double double precision ...
Running with 1 threads ...
Computing an inner product of size 1000000000
Task 1 is computing ...
The inner product :
9.99999999950000000050166661016E+08

real    0m17.113s
user    0m17.096s
sys     0m0.004s
```

From 826 milliseconds to 17 seconds: $23.954/0.826 = 20.7$.
Compiled with `-O3 -gnatp -gnatf` flags.

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking

2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs

3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Static Job Scheduling

Distributing the work among three tasks:

$$\underbrace{1 + rs + r^2s^2}_{\text{thread 1}} + \underbrace{r^3s^3 + r^4s^4 + r^5s^5}_{\text{thread 2}} + \underbrace{r^6s^6 + r^7s^7 + r^8s^8}_{\text{thread 3}}$$

Let n be the dimension, and p the number of threads:

$$m = n/p$$

are the number of terms summed up by each thread.

1. Thread i computes start and end index as $(i-1)m$ and $i\,m-1$.
2. The i-th thread writes the result at `a`(i) of array `a`.
3. After all threads are done, the main thread adds up p numbers.

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking

2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs

3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

Parallel Runs

1. The run in double precision took 826 milliseconds.
2. Running with double double arithmetic:
   - With 1 thread: 17 seconds and 113 milliseconds.
   - With 16 threads: 2 seconds and 187 milliseconds.

   Speedup: $17.113/2.187 \approx 7.8$.
3. Running with quad double arithmetic:
   - With 1 thread: 4 minutes, 38 seconds and 65 milliseconds.
   - With 16 threads: 30 seconds and 860 milliseconds.

   Speedup: $278.065/30.860 \approx 9.0$.

To obtain good speedups, increase the size of the problem when increasing the number of threads.

Comparing Running Times

between double and double double arithmetic

A computation in double double arithmetic takes about 20 times longer than the same computation with double arithmetic.

- A sequential run in double precision takes 826 milliseconds.
- The running time in double double arithmetic with 16 threads is 2 seconds and 187 milliseconds.

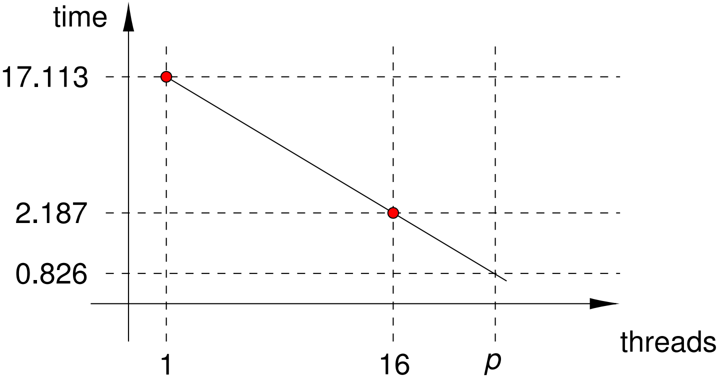
Compare $2.187/0.826 \approx 2.6$.

We doubled the precision in a little over twice the time.

Quality Up

keeping the execution time fixed

$$\text{quality up } Q(p) = \frac{\text{quality on } p \text{ processors}}{\text{quality on 1 processor}} = 2, \quad \text{what is } p?$$



Linear Extrapolation

The line through $(1, 17.113)$ and $(16, 2.187)$ has equation

$$y - 17.113 = \frac{17.113 - 2.187}{1 - 16}(x - 1).$$

This line represents the execution time of parallel runs with x processors in double double arithmetic.

We compute the number p for the execution time to be 0.826.

$$0.826 - 17.113 = \left(\frac{17.113 - 2.187}{1 - 16} \right) (p - 1).$$

or, solving for p, gives

$$p = 1 + \left(\frac{1 - 16}{17.113 - 2.187} \right) (0.826 - 17.113) = 17.368.$$

Multithreading for Quality Up

1. Shared Memory Parallel Programming
   - multithreading and thread safety
   - objectives
   - hello tasking

2. Static Job Scheduling
   - the work crew model
   - inner products of geometric sequences
   - scheduling jobs before the runs

3. Quality Up
   - parallel runs with double doubles and quad doubles
   - scalability

scalability

How many processors do we need to afford multiple double precision?

- The overhead of double double arithmetic can be offset on shared memory parallel computers, using multithreading.
- For quad double arithmetic, teraflop performance is required, as available in graphics processing units.

Multiple Double and Multiword Arithmetic Part IV: Vectorization for Efficient Pipelining

Jan Verschelde†

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/∼jan
https://github.com/janverschelde
https://www.youtube.com/@janverschelde5226
janv@uic.edu

Ada Europe 2025 tutorial, 10 June, Paris, France.

†Supported by a 2023 Simons Travel Award.

Vectorization for Efficient Pipelining

1. Vectors of Multiple Doubles
   - pipelined floating-point addition
   - objectives
   - linearization to compute Taylor series

2. Multiword Arithmetic
   - delaying the normalization
   - vectored inner product

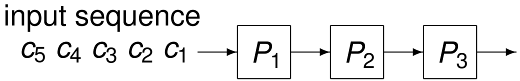
Vectorization for Efficient Pipelining

1. Vectors of Multiple Doubles
   - pipelined floating-point addition
   - objectives
   - linearization to compute Taylor series

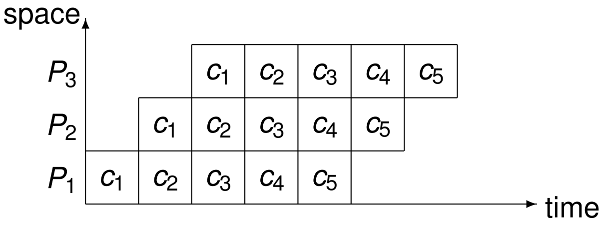
2. Multiword Arithmetic
   - delaying the normalization
   - vectored inner product

Car Manufacturing

Consider a simplified car manufacturing process in three stages:
(1) assemble exterior, (2) fix interior, and (3) paint and finish:



The corresponding *space-time diagram* is below:



After 3 time units, one car per time unit is completed.

Speedup for n Inputs in a p-Stage Pipeline

Consider n inputs for a p-stage pipeline:

$$S(p) = \frac{n \times p}{p + n - 1}.$$

For fixed number p of processors:

$$\lim_{n \to \infty} \frac{p \times n}{n + p - 1} = p.$$

Pipelining speeds up multiple sequences of heterogeneous jobs.

Pipelining is a functional decomposition method
to develop parallel programs.

Floating-Point Addition

A floating-point number consists of a sign bit,
an exponent and a fraction (or mantissa):

$\pm$	e (11 bits)	f (52 bits)

Floating-point addition could be done in 6 cycles:

1. unpack fractions and exponents
2. compare exponents
3. align fractions
4. add fractions
5. normalize result
6. pack fraction and exponent of result

Adding two vectors of n floats with 6-stage pipeline
takes $n + 6 - 1$ pipeline cycles, instead of $6n$ cycles.
$\Rightarrow$ Capable of performing one flop per clock cycle.

Vectorization for Efficient Pipelining

1. Vectors of Multiple Doubles
   - pipelined floating-point addition
   - objectives
   - linearization to compute Taylor series

2. Multiword Arithmetic
   - delaying the normalization
   - vectored inner product

Objectives of the Tutorial

1. Realize that the pipelining is implicit in floating-point computations.
2. Understand to define the data to be simple enough for pipelining, the multiple double arithmetic will remain memory bound.
3. Optimize code by avoiding branching.

Vectors of Double Doubles

A double double x is stored as a record of two doubles: the high part $x^{\rm hi}$ and the low part $x^{\rm lo}$, represented by the tuple $x = \left(x^{\rm hi}, x^{\rm lo}\right)$.

An array of three double doubles is then:

$$\boxed{\left(x_1^{\rm hi}, x_1^{\rm lo}\right)} \boxed{\left(x_2^{\rm hi}, x_2^{\rm lo}\right)} \boxed{\left(x_3^{\rm hi}, x_3^{\rm lo}\right)}$$

An alternative representation is a tuple of two arrays:

$$\left(\boxed{x_1^{\rm hi}} \boxed{x_2^{\rm hi}} \boxed{x_3^{\rm hi}} \, , \boxed{x_1^{\rm lo}} \boxed{x_2^{\rm lo}} \boxed{x_3^{\rm lo}} \right)$$

The tuple of arrays representation has two benefits:

+ convenient to upgrade or downgrade the precision; and
+ enables efficient retrieval of the data arrays,
  as the unpacking of records is avoided.

Turning Inside Arithmetic to the Outside

- To make multiple double arithmetic compute bound, and in this way reduce the cost overhead, when working with arrays, the arithmetic has to applied to the outer levels.
- Consider complex multiplication $\star$, for $i^2 = -1$:

$$(a + b\,i) \star (c + d\,i) = (ac - bd) + (ad + bc)i.$$

For two complex vectors $\mathbf{x} = \mathbf{a} + \mathbf{b}\,i$ and $y = \mathbf{c} + \mathbf{d}\,i$, computing four componentwise products of real vectors $\mathbf{ac}$, $\mathbf{ad}$, $\mathbf{ad}$, and $\mathbf{bc}$ allows for efficient pipelining when computing inner products.

Vectorization for Efficient Pipelining

1. Vectors of Multiple Doubles
   - pipelined floating-point addition
   - objectives
   - linearization to compute Taylor series
2. Multiword Arithmetic
   - delaying the normalization
   - vectored inner product

Linearization

Working with truncated power series, computing modulo $O(t^d)$,
is doing arithmetic over the field of formal series $C[[t]]$.

Linearization: consider $\mathbb{C}^n[[t]]$ instead of $\mathbb{C}[[t]]^n$. Instead of a vector of power series, we consider a power series with vectors as coefficients.

Solve $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{C}^{n \times n}[[t]]$, $\mathbf{b}, \mathbf{x} \in \mathbb{C}^n[[t]]$.

$$
\begin{array}{rcl}
\mathbf{A} & = & A_0 t^a + A_1 t^{a+1} + \cdots , \\
\mathbf{b} & = & \mathbf{b}_0 t^b + \mathbf{b}_1 t^{b+1} + \cdots \\
\mathbf{x} & = & \mathbf{x}_0 t^{b-a} + \mathbf{x}_1 t^{b-a+1} + \cdots
\end{array}
$$

where $A_i \in \mathbb{C}^{n \times n}$ and $\mathbf{b}_i, \mathbf{x}_i \in \mathbb{C}^n$.

Block Linear Algebra

Computing the first d terms of the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$:

$$
\begin{array}{l}
\left(A_0 t^a + A_1 t^{a+1} + A_2 t^{a+2} + \cdots + A_d t^{a+d} \right) \\
\cdot \; \left(\mathbf{x}_0 t^{b-a} + \mathbf{x}_1 t^{b-a+1} + \mathbf{x}_2 t^{b-a+2} + \cdots + \mathbf{x}_d t^{b-a+d} \right) \\
= \mathbf{b}_0 t^b + \mathbf{b}_1 t^{b+1} + \mathbf{b}_2 t^{b+2} + \cdots + \mathbf{b}_d t^{b+d}.
\end{array}
$$

Written in matrix format:

$$
\left[\begin{array}{ccccc}
A_0 & & & & \\
A_1 & A_0 & & & \\
A_2 & A_1 & A_0 & & \\
\vdots & \vdots & \vdots & \ddots & \\
A_d & A_{d-1} & A_{d-2} & \cdots & A_0
\end{array} \right]
\left[\begin{array}{c}
\mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_d
\end{array} \right]
=
\left[\begin{array}{c}
\mathbf{b}_0 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_d
\end{array} \right].
$$

If A_0 is regular, then solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ is straightforward.

Error Analysis

Solving $(A_0 + A_1 t + A_2 t^2 + \cdots + A_d t^d)(x_0 + x_1 t + x_2 t^2 + \cdots + x_d t^d)$
$= (b_0 + b_1 t + b_2 t^2 + \cdots + b_d t^d)$

leads to a lower triangular block system:

$$\begin{bmatrix} A_0 & & & & \\ A_1 & A_0 & & & \\ A_2 & A_1 & A_0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ A_d & A_{d-1} & A_{d-2} & \cdots & A_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_d \end{bmatrix}.$$

Cost to solve: $O(n^3) + O(dn^2)$.

Let κ be the condition number of A_0. Let $\|A_0\| = \|x_0\| = 1$, $\|x_d\| \approx \rho^d$.

In our context, $\rho \approx 1/R$, where R is the convergence radius.

If $\|A_d\| \approx \rho^d$, then $\dfrac{\|\Delta x_d\|}{\|x_d\|} \approx \kappa^{d+1} \epsilon_{\mathrm{mach}}$, and accuracy is lost.

Vectorization for Efficient Pipelining

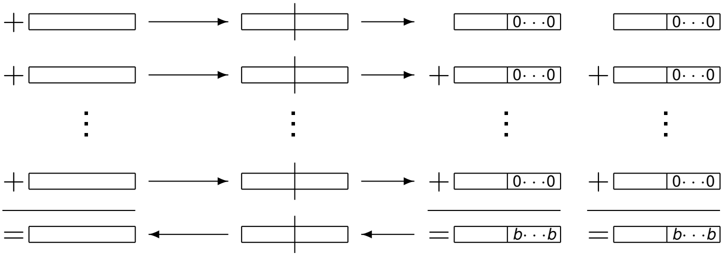
1. Vectors of Multiple Doubles
   - pipelined floating-point addition
   - objectives
   - linearization to compute Taylor series

2. Multiword Arithmetic
   - delaying the normalization
   - vectored inner product

An Error Free Summation

Assuming all 64-bit doubles have the same exponent,
we work with 52-bit integers (fractions of the doubles).

Split a vector of doubles, add the parts, and then fuse the result:



If the number of additions does not exceed some threshold,
then we have sufficiently many zero bits left at the end of the numbers
to represent the result exactly, without any error.

Vectorization for Efficient Pipelining

1. Vectors of Multiple Doubles
   - pipelined floating-point addition
   - objectives
   - linearization to compute Taylor series

2. Multiword Arithmetic
   - delaying the normalization
   - vectored inner product

Vectored Inner Product with Double Double Arithmetic

Given are vectors **x** and **y** both of length n, of double double numbers, we compute $\displaystyle\sum_{k=1}^{n} x_k \star y_k$, where $\star$ is the double double multiplication.

The double double x_k is represented by $(x_k^{\rm hi}, x_k^{\rm lo})$, where the high double $x_k^{\rm hi}$ and the low double $x_k^{\rm lo}$ of x_k are splitted in quarters:

$$(\overbrace{x_{k,0}, x_{k,1}, x_{k,2}, x_{k,3}}^{x_k^{\rm hi}}, \overbrace{x_{k,4}, x_{k,5}, x_{k,6}, x_{k,7}}^{x_k^{\rm lo}}).$$

After splitting also y_k, we compute in double arithmetic:

$$s_0 = \sum_{k=1}^{n} x_{k,0} y_{k,0}, \;\; s_1 = \sum_{k=1}^{n} x_{k,1} y_{k,0} + x_{k,0} y_{k,1}, \;\; s_i = \sum_{k=1}^{n} \sum_{j=0}^{i} x_{k,j} y_{k,i-j},$$

for $i = 2, \ldots, 7$, add $s_0 + s_1 + \cdots + s_7$ in double double arithmetic.

Balanced Quarters of Doubles

To examine the computational efficiency, random 64-bit doubles are generated with a fraction of 52 bits in following pattern:

$$1\,\underbrace{bb\cdots b}_{\text{12 bits}}1\,\underbrace{bb\cdots b}_{\text{12 bits}}1\,\underbrace{bb\cdots b}_{\text{12 bits}}1\,\underbrace{bb\cdots b}_{\text{12 bits}}, \quad b \in \{0,1\}.$$

Splitting such double into four leads to doubles with fractions

$$\begin{array}{l}
1b\cdots b\,00\cdots 0\,00\cdots 0\,00\cdots 0,\\
00\cdots 0\,1b\cdots b\,00\cdots 0\,00\cdots 0,\\
00\cdots 0\,00\cdots 0\,1b\cdots b\,00\cdots 0,\\
00\cdots 0\,00\cdots 0\,00\cdots 0\,1b\cdots b.
\end{array}$$

By virtue of the placement of the ones in the random fractions, all quarters have fixed exponents, e.g.: 0, -13, -26, -39.

All doubles in a multiple double are generated according this pattern.

Computational Results

Computing 1,024 times $\sum_{k=1}^{6144} a_k \star b_k$ in increasing precision:

	ordinary		**speedup**	vectorized	
	cpu time	overhead	$\frac{\text{ordinary}}{\text{vectorized}}$	cpu time	overhead
16d	40s 780ms	6.3x	**4.3x**	9s 491ms	6.2x
8d	6s 428ms	3.3x	**4.2x**	1s 520ms	4.8x
4d	1s 977ms	12.x	**6.2x**	318ms	4.6x
2d	158ms	13.x	**2.3x**	69ms	2.3x
1d	12ms		**0.4x**	30ms	

Ran on an Intel Xeon 5318Y Ice Lake-SP, up to 3.40GHz, 256GB of internal memory at 3200MHz, GNU/Linux, Microway 2024, compiled with GNAT 12.2.0, flags `-O3 -gnatp -gnatf`.

Multithreading to Reduce Overhead

It takes 9 seconds for 1,024 inner products in hexa double precision.

Wall clock time: 9s 308ms, with 85ms for generating the vectors.

In a multithread computation, every thread does one inner product.

On two 24-core Intel Xeon 5318Y Ice Lake-SP, up to 3.40GHz, 256GB of internal memory at 3200MHz, GNU/Linux, Microway 2024, compiled with GNAT 12.2.0, flags `-O3 -gnatp -gnatf`, the wall clock time is 293 milliseconds, using 96 threads.

Comparing the 293 milliseconds to the 318 milliseconds with one thread in quad double precision, we can quadruple the precision and compute as fast as in quad double precision, using 96 threads, achieving *quality up*.

Staging Data for Matrix Multiplications

Postponing renormalizations of multiple doubles benefits the efficiency.

The code is at `https://github.com/janverschelde/PHCpack`.

The convolutions $\displaystyle\sum_{k=1}^{n} \sum_{j=0}^{i} x_{k,j} y_{k,i-j}$ allow to rewrite the inner products in multiple double arithmetic as matrix multiplications in double precision floating-point arithmetic, to prepare for better acceleration with graphics processing units, in particular tensor cores.