Multiword Arithmetic and Parallel Computing

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 devroom, FOSDEM 2025, 2 February

Multiword Arithmetic and Parallel Computing

1. Introduction
   - robust numerical algorithms
   - multiple double arithmetic

2. Multiword Arithmetic
   - an error free summation
   - vectored inner product

3. Parallel Computation
   - multithreading to reduce overhead
   - staging data for matrix multiplications

introduction

An algorithm is *robust* if it does not fail for small perturbations of degenerate inputs.

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

- 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].

Point of this talk: define reference code for GPU acceleration.

multiple double arithmetic

A *multiple double* is an unevaluated sum of nonoverlapping doubles.

Take 64 random complex numbers on the unit circle.
The 2-norm of this vector is 8, computed with multiple doubles:

```
double double : 8.00000000000000E+00 - 4.46815747097839E-32
  quad double : 8.00000000000000E+00 + 8.23258305145073E-65
  octo double : 8.00000000000000E+00 - 5.56764060802733E-128
  hexa double : 8.00000000000000E+00 - 1.54394135726410E-257
```

Cost overhead:

	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

The table lists the number of operations with doubles for a multiple double addition (`add`), multiplication (`mul`), and division (`div`).

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:

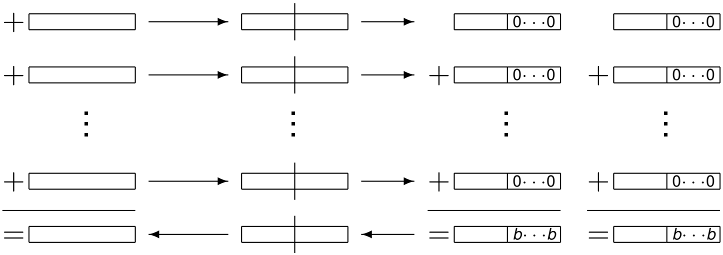
k	$1/k!$	recommended precision	`eps`
7	`2.0e-004`	double precision okay	`2.2e-16`
15	`7.7e-013`	use double doubles	`4.9e-32`
23	`3.9e-023`	use double doubles	
31	`1.2e-034`	use quad doubles	`6.1e-64`
47	`3.9e-060`	use octo doubles	`4.6e-128`
63	`5.0e-088`	use octo doubles	
95	`9.7e-149`	use hexa doubles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

`eps` is the multiple double precision

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.

vectored inner product with double double arithmetic

Given are vectors $\mathbf{x}$ and $\mathbf{y}$ 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.

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`.

a high level parallel computation

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

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

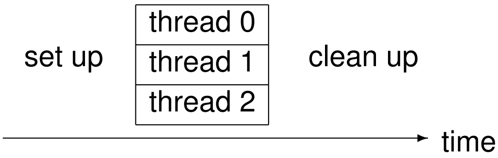
In a multithread computation, every thread does one inner product.

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*.

the work crew 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 and compute the job.

The jobs are define by by tasks. Updating the index to the current job happens in a critical section, implemented in the `GNAT.Semaphores` package, see AdaCore Gem #81 by Pat Rogers.

conclusions

Postponing renormalizations of multiple doubles benefits the efficiency.

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

PHCpack is a software package to solve polynomial systems with homotopy continuation methods, available as an alire crate.

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.