Less Hazardous and More Scientific Research for Summation Algorithm Computing Times

Philippe Langlois, David Parello, Bernard Goossens, Kathy Porada

Less Hazardous and More Scientific Research for Summation Algorithm Computing Times

Philippe Langlois, David Parello,
Bernard Goossens, Kathy Porada *

September, 29 2012

Abstract

Several accurate algorithms to sum IEEE-754 floating-point numbers have been recently published. The recent contributions by Rump, Ogita and Oishi and the newest ones proposed by Zhu and Hayes are examples of accurate summation algorithms. Some of these even compute the faithful or the correct rounding of the exact sum, *i.e.* the most accurate value with respect to the finite precision of the floating-point arithmetic. This computed sum does not suffer anymore from the condition number of the summation. In such cases, the run-time performances and the memory prints become the discriminant properties to decide which algorithm is best.

In this paper we focus on the reliability of the run-time performance measure of such core algorithms. We explain how right Rump when he writes "*Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research.*" Neither the classical flop count nor hardware counter based measures are satisfactory here. We propose to analyze the instruction level parallelism of these algorithms to reliably evaluate their performance potential. We use PerPI, a software

*Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan , France. Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier, France. CNRS, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier, France. first_name.last_name@univ-perp.fr

tool that automatizes an almost machine independent instruction-level parallelism analysis. We study recent accurate summation algorithms with a detailed focus on the two newest faithful ones. We illustrate and discuss why PerPI provides a more reliable performance analysis, the remaining weakness and how to improve confidence for future contributions in this area.

Keywords: floating-point arithmetic, accurate summation, faithful summation, performance evaluation, reproducibility, instruction level parallelism, PerPI.

The summation of an arbitrary number of floating-point values is a basic but no so simple challenge for numerical software experts. Since 1965 numerous papers have introduced new summation algorithms and at least one new "better" algorithm has been published every year since 1999 – see Table 1.

The challenge is how to manage both the accuracy of the computed sum and its computation speed?

This paper begins presenting why and how to measure the performance of summation algorithms. In Section 2, we give an account of how difficult it is to measure —and to publish— trustful run-times for such core algorithms. Section 3 illustrates how an instruction level parallelism (ILP) analysis provides an insightful evaluation of their performance potential. In Section 4, we use PerPI, a software tool for the ILP analysis of x86 binary codes, to measure and analyze recent accurate and faithful summation algorithms.

1 Why should we measure the summation algorithm performance?

In the following, $\mathbf{u}$ denotes the floating-point arithmetic precision. For instance, $\mathbf{u} = 2^{-53} \approx 10^{-16}$ in the binary64 format of the IEEE-754 standard [6]. As usual and for any available rounding mode, $fl(x)$ represents the floating-point value which rounds the real value x at precision $\mathbf{u}$ —the overflow case is excluded.

1965	Møller, Ross
1969	Babuska, Knuth
1970	Linz, Nickel
1971	Dekker, Malcolm
1972	Kahan, Pichat
1974	Neumaier
1977	Bohlender
1981	Linnainmaa
1982	Leuprecht/Oberaigner
1983	Jankowski/Smoktunowicz/Wozniakowski
1985	Jankowski/Wozniakowski
1988	Pichat
1987	Kahan
1991	Priest
1992	Clarkson, Priest
1993	Higham
1997	Shewchuk
1999	Anderson
2001	Hlavacs/Ueberhuber
2002	Li *et al.*(XBLAS)
2003	Demmel/Hida, Nievergelt, Zielke/Drygalla
2005	Ogita/Rump/Oishi, Zhu/Yong/Zeng
2006	Eisinberg/Fedele, Klein, Zhu/Hayes
2008	Rump/Ogita/Oishi
2009	Rump, Zhu/Hayes
2010	Zhu/Hayes

Table 1: A search for optimal accuracy vs. speed trade off explains the numerous summation algorithms published since 1965

1.1 floating-point summation accuracy

Classical recursive summation is a backward stable algorithm to compute $\sum_{i=1}^{n} x_i$. The computed sum accuracy is bounded by $(n-1) \times cond \times \mathbf{u}$. This bound mainly depends on the condition number $cond = \sum |x_i| / |\sum x_i|$. No more significant digits may be returned for large condition numbers. For instance, this is the case when $n \times cond$ is larger than 10^{16} in the binary64 format.

Two straightforward solutions can be applied to improve accuracy. Firstly, a larger precision can be simulated with a multiple precision arithmetic library. Twice ($\mathbf{u}^2$) or four times the $\mathbf{u}$ precision are implemented by the double-double or the quad-double libraries [4]; arbitrary precision libraries ($\mathbf{u}^K$) also exist, *e.g.* ARPREC [4] or MPFR [13]. Secondly, we can compensate. Compensated algorithms calculate every intermediate sum rounding error to improve the final result accuracy. Several compensation strategies have been proposed since the famous Kahan's one – see the survey [5, chap. 4]. We further pay more attention to the recent Sum2 algorithm and its SumK generalisation proposed by Rump *et al.*[15].

Multiprecision arithmetic and compensation both lead to the same kind of accuracy improvement: the computed sum error is now bounded by $\mathbf{u} + \mathcal{O}(n^\alpha) \times cond \times \mathbf{u}^K$, with $\alpha \in [1, K]$. This bound exhibits a better accuracy than the previous one: the precision $\mathbf{u}$ is now an achievable accuracy. Nevertheless this best accuracy is still constrained by the summation condition number.

Much effort has been done to skip over this condition number limitation. The major issue is to compute a faithfully or an exactly rounded sum for any entry vector (of any reasonable length n).

1.2 Faithfully and exactly rounded sums

A faithful result is one of the two consecutive floating-point numbers enclosing the exact sum, or the sum itself if it is a floating-point number. An exactly (or correctly) rounded sum generalises to $\sum_i$ the rounding model that the IEEE-754 standard defines for the arithmetic operations $+, -, \ldots$. The exact sum is rounded according to a given rounding mode, *i.e.* as if the result would be computed with an infinite precision and an unlimited range and eventually rounded [14]. A significant over-cost should be paid to guarantee

an exactly rounded result when the sum is close to a rounding breakpoint. As this difficult case is rather specific we will focus on the faithfully rounded sums.

The algorithms presented next compute this faithful sum independently of the entries (of any reasonable length n) condition number.

Distillation is an iterative technique which converges to the faithful (or the exact) rounded value. The entry vector $[x]$ is error free transformed into vector iterates $[x^{(1)}], [x^{(2)}], \cdots, [x^*]$ such that $\sum x_i = \sum x_i^*$ and $[x^*]$ provides the expected rounded value. Zhu and Hayes illustrate the distillation technique clearly and apply it to their iFastSum algorithm [23]. They prove the following property, which is useful to derive such an $[x^*]$: x_i^* are the non-overlapping components of $[x^*]$ verifying $|x_1^*| < |x_2^*| < \cdots < |x_n^*|$, and $fl(x_i^* + x_{i+1}^*) = x_{i+1}^*$.

Another way to reach faithfulness is to enlarge the working space and keep all the significant parts of the sum. Classical examples are Malcolm's or Kulish's long accumulators [12, 7]. We may also cut the summands as proposed in Rump *et al.*'s AccSum or FastAccSum algorithms [18, 17], or sum exponent based partitions as done by Zhu and Hayes'HybridSum and OnLineExact [23, 24]. These approaches all compute error-free partial sums.

1.3 Run-time efficiency becomes the discriminant factor

The previous algorithms all return the same best accurate computed sum for a given computing precision. This property is hard to obtain and even harder to prove: it definitely remains the main published contribution. Accuracy is no more the discriminant feature between these summation algorithms. A reliable evaluation of their run time and memory cost becomes here more important than before. In practice, this issue helps to choose between these accurately equivalent algorithms and in absolutes, it gives an appreciation on how a new proposal improves the state of the art.

For these simple algorithms a reliable memory cost analysis is easy to derive and present. In the next Section we show that we should not be so confident with the published run-time efficiency analysis.

2 How to measure the summation algorithm performance?

The classical way to evaluate the time complexity of a numerical algorithm is a two step process. First, the number of floating-point arithmetic operations is counted (with no distinction of their elementary cost). This yields a complexity exhibited as a function of the entry size[1] taking care of the multiplicative constant in the highest degree terms. Accurate and faithful summation algorithms are linear. For instance, Rump's AccSum and FastAccSum respectively cost $7n$ and $6n$ flops to return the faithful sum of n entries with a given range of conditioning. Second, this theoretical count is faced to run-time experimental measures; the run-time unit being seconds or machine cycles. This experimental part gets more tedious as the computing environment complexity increases.

Conclusions stated after such an analysis method are far from reliable.

2.1 Theoretical flop count vs. measured run-time

Theoretical flop count and measured run-time are not always proportional and sometimes even not correlated.

We illustrate this first weakness with the two summation algorithms DDSum and Sum2. DDSum is the classical recursive summation implemented using double-double arithmetic [21, 10]. Sum2 is the compensated summation by Rump *et al.*[15]. These algorithms yield results of similar accuracy simulating twice the computing precision, *i.e.* $\mathbf{u} + \mathcal{O}(n^\alpha) \times cond \times \mathbf{u}^2$, with α equals to 1 or 2 for DDSum and Sum2 respectively. Table 2 presents the theoretical flop counts and the measured numbers of cycles for these two accurate algorithms. We compare them to the flops and cycles measured for the classical recursive summation Sum run in precision $\mathbf{u}$. The presented ratios can be interpreted as the over-cost to double a length n computed sum accuracy. The matching codes are presented in Section 3.2. The cycle numbers are measured as detailed in A. To summarize these measures, we extract a value exhibiting the average behavior of some considered computing environments. In the next Section we detail what these measures depend on and how they may vary according to many parameters such as hardware, compiler, sum length ...

[1] Other parameters such as *e.g.* accuracy are not considered here.

Metric	Sum	Sum2	DDSum
Flop count	$n-1$	$7n$	$10n$
Flop count (approx. ratio)	1	7	10
Measured #cycles (approx. ratio)	1	2.5	7.5
Flop count / measured #cycles (approx.)	1	2.8	1.4

Table 2: Flop counts and measured run-times are not proportional

Table 2 shows that the time cost to be paid for accuracy is less than we would expect from theoretical flop count. This has been identified in some previous publications, *e.g.* [18], [24]. This gap from measured run-times to flops increases as computing environments improve —compare for instance [15] to [18].

Beside Sum, Sum2 and DDSum run respectively about 3 and 1.5 times faster than what the flop count announces. Sum2 is three times faster than DDSum, one half only coming from the flop count. The flop count is appropriate neither to anticipate the improvement accuracy over-cost nor to compare the two challenging accurate algorithms.

The last line in Table 2 exhibits that the flop counts and the measured run-times are not proportional. Sum2 executes twice more floating-point additions than DDSum and about three times more than Sum in a same time unit. We claim that this is the significant property that reliably explains the actual behavior of these implementations. Before describing how to provide an automatic and reliable evaluation of this property we emphasize why measuring run-times is a very difficult experimental process.

2.2 Timing measure is an ultimately difficult task

Measuring time has ever been a part of the numerical software developer's job. This task became ultimately difficult as the computing chain complexity increased. Architecture, micro-architecture, compiler and operating system interact and are the main parameters of this complexity.

We have already mentioned that the run-time measures yield non-reproducible results. The execution time of a binary program varies, even using the same data input and the same execution environment. This uncertainty has numerous causes. Spoiling events such as background tasks, concurrent jobs or OS

interrupts are well-known factors. Modern processors include parts built to run with a non-deterministic behavior: instruction scheduler, branch predictor, cache management are examples of components implementing randomized algorithms. External conditions such as the room temperature modify the clock frequency which invalidates second-based timings. Time measure based on cycles is tricky with fixed duration bus cycles and variable duration core cycles. Some timing functions provided by the OS are bus clock based and others are core clock based. Both mix cycles coming from computation instructions which are core clock based and memory access instructions which are bus clock based. The cycle can hardly be considered as a constant duration unit on modern processors, *e.g.* Intel i7,...

It is wrong to assume that the hardware performance counters included in these units manage this complexity and provide accurate and reliable measures. Recent work from software and system performance experts emphasizes very well this bad news. Zaparanuks *et al.* caution performance analysts to be suspicious of cycle counts gathered with performance counters [22]. For example they exhibit that the cycle count depends on the measured code memory placement. PAPI is one well-known interface library to these hardware counters. Members of the PAPI team show that in practice counters that should be deterministic display variation from run to run on the widely used x86_64 architecture. They also show variations from machine to machine even when sharing the same architecture. Moreover, there is no standard way to count floating-point or SSE instructions. Even the retired instructions counters which only depend on the instruction set architecture exhibit variations while they should keep stable from run to run and across machines. In [20] they conclude that it is difficult to determine known "good" reference counts for comparison.

Instruction and cycle counts are the key measures for performance evaluation in our scope. Cycle count is a non-deterministic value on the current computing environments. Even the deterministic instruction count is hard to measure with reliability on available hardware. Not surprisingly, it is very hard to reproduce timings announced in publications.

2.3 How to trust not reproducible experimental results?

Run-time measures are mostly not reproducible even when the experimenters do their best.

This important issue is also observable in a publication sequence about summation by the same group of authors. Table 3 presents the amount of numerical experiments in Rump *et al.* contributions between 2005 and 2009.

Year and Ref.	2005 [15]	2008 [18]	2008 [19]	2009 [17]
Algorithms	Sum2	AccSum	AccSumK	FastAccSum
	SumK		NearSum	FastPrecSum
			AccSumHugeN	
			AccSign	
% of numerical exper.	26%	20%	20%	$< 3\%$
Pages	9/34	7/35	6/30	1/37
% of timings	11%	100%	100%	100%
Pages	1/9	7/7	6/6	1/1
Tested environments	2	3	3	1
Timing tables	1	8	6	1
Timing figures	0	6	2	0

Table 3: Proportion of the numerical experiments reports in Rump *et al.* publications. 2005: a picture, 2008: two movies, 2009: a slogan.

We identify two periods. First three articles are composed of 20-25% of numerical experiments reports. These numerical experiments aim to illustrate the proposed algorithms accuracy and timings. As mentioned, these experimental sections only countain timings for the faithful and correct sums [18, 19]. The number of tested environments and reported tables and figures increases significantly as shown with the last three lines of Table 3. Next period: the latest published result also concerns mainly a faithful algorithm [17]. The experimental part only includes timings. Less than 3% of this paper is devoted to the experimental issue reduced to its minimum: a single set of measures for a single computing environment. This brusque change is indeed justified by the author: "Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research." Yet, this paper is entitled *Ultimately Fast* Accurate Summation.

In the recent papers by Zhu and Hayes, the experiments also suffer from this unavoidable hazard. Some timing tables presented in [23] and [24] share

the same test environment (same computing system and input data). However the presented run-times vary significantly between the papers for some algorithms: from -13% to +20% (Data no.3) and from -1.5% to +13% (Data no.1) for iFastSum and HybridSum. It is also confusing that the ordinary recursive summation Sum applied to several sums of the same length present 20% of run-time variation in the same result table [23, Table 4.1, Data no.3].

It is an open problem to measure and publish trustful run-times at least in the area we study here. Theoretical flop count is indisputable but not significant of the actual run-time efficiency on current computing units. Conversely the experimental measure of computing run-times is a painful task yielding at best a blurred picture of the performance limitations.

3 ILP and the PerPI tool

Langlois and Louvet showed in [9] that the instruction level parallelism (ILP) explains the gap between the theoretical flop count and the running times for this kind of core algorithms. PerPI is a software tool that automatizes this ILP analysis proposed by Goossens *et al.* in [2]. In this Section we recall the main principles of this ILP analysis and illustrate it with accurate algorithms Sum2 and DDSum. We also briefly present the PerPI software tool we further use to evaluate the performances of some accurate and faithful summation algorithms.

3.1 ILP and the performance potential of an algorithm

We analyze the instruction level parallelism of a program run by simulating it with a Hennessy-Patterson ideal machine [3]. This ideal machine executes every program instruction as soon as possible, *i.e.* one cycle immediately after the execution of the producers it depends on.

An ideal machine has infinite resources: infinite number of renaming registers, perfect branch predictor, perfect memory disambiguation. As a result, running a code on an ideal machine is like having at hand the full execution trace and picking up from this trace instructions as soon as their sources are available. In such a way, the run is ordered according to the only producer-consumer dependencies: the Read After Write true instruction dependency.

ILP represents the instruction potential of a program run to be executed

simultaneously. Every current processor, when running a program applied to a data set, exploits a part of the ILP thanks to well-known techniques such as pipelining, superscalar execution, out-of-order and speculative execution, renaming, dynamic branch prediction or address speculation... In contrast, the ideal machine exploits all the ILP.

Our main measure is the number C of cycles of such an ideal run. It describes the best possible performance of the run when constrained by the producer-consumer dependencies.

This measure does not depend anymore either on the machine or on the conditions of the run (except the given program and data set). However, this model still depends on the instruction set architecture (ISA) from which is derived the cycle granularity, being the atomic time to execute a single instruction. For instance, an ISA including a fused multiply and add (FMA) instruction would count one less cycle to run a sequence of linked multiplication and addition (as in $a \times b + c$) than an ISA without an FMA. This ISA dependency will be presented and discussed in Section 4.3.

We also count the number I of executed instructions. The average number of instructions executed per cycle I/C is the classically denoted ILP of a run. In a way, ILP helps to get rid of the mentioned architectural dependency. For instance in the previous FMA example, the ILP is the same in both architectures. It may be used to compare programs rather than runs when the data set which is used has no influence either on the number of instructions run or on the number of cycles of the run.

In practice we count C and I for a given machine language. The smaller C the potentially faster the algorithm thus explains the actually measured runtimes. The PerPI tool provides these measures for any program compiled to run on a x86_64 machine. Before we start at the algorithmic level to illustrate the principle of the analysis.

3.2 ILP analysis of Sum, Sum2 and DDSum

We study the ILP of the Sum, Sum2 and DDSum algorithms.

3.2.1 Sum

The following tables present the ideal executions of these algorithms, *i.e.* their executions on the Hennessy-Patterson ideal machine. We apply the previ-

ously described model to identify the execution cycle k of every instruction and exhibit the total number of cycles $C = \max k$. Instructions are presented in the second leftmost column. The main part of these three sum algorithms is a loop iterated $n-1$ times. We denote every iteration index `i` with `i=1..n-1`. The $n-1$ rightmost columns contain the cycle number k of the line-corresponding instruction for the column-corresponding iteration. Cycles start at 0 executing all the possible initialisations.

Sum		iter.	1	2	3	...	$n-1$
	`s = x[0];`		0				
	`for(i=1; i<n; i++)`						
a	`  s = s + x[i];`		1	2	3	$\cdots$	n-1
	`return(s);`						n

Table 4: The ideal execution of Sum takes n cycles

Table 4 shows the well-known sequential behavior of the classical recursive summation. The core loop contains only one instruction. Iteration i has to wait for the value of `s` computed at the previous iteration $i-1$ to be executed. Sum takes $n-1$ cycles to complete its loop and returns its result at cycle n. The total cycle count for its ideal execution is $C_{\mathsf{Sum}} = n$.

During these n cycles, $I = n$ instructions are executed. Hence the average ILP of the classical recursive summation equals 1 confirming that this algorithm has no instruction parallelism.

3.2.2 Sum2

The intermediate variables of the Sum2 and DDSum algorithms are explicit in Tables 5 and 7. We do not mentioned associated initialisations.

The core loop of Sum2 contains 8 instructions executable in 6 cycles: as `t2` and `t3` are independent variables instructions d and e are executed in the same cycle. Two consecutive iterations of the core loop are linked by the value of `s` (instruction b) and `c` (instruction h). Square boxed values in Table 5 exhibit these two iteration dependencies. As for Sum, these two accumulations are executed one cycle after the availability of the previous iteration value. Yet, the whole iteration critical path does not suffer from these dependencies: a new iteration starts every cycle and two consecutive iterations overlap during 6 cycles. This is illustrated by Table 6. The $n-1$

Sum2		iter.	1	2	3	...	$n-1$
	`s = x[0];`		0				
	`for(i=1; i<n; i++){`						
a	`  s_ = s;`		1	2	3	$\cdots$	n-1
b	`  s = s + x[i];`		1	2	3	$\cdots$	n-1
c	`  t = s - s_;`		2	3	4	$\cdots$	n
d	`  t2 = s - t ;`		3	4	5	$\cdots$	n+1
e	`  t3 = x[i] - t;`		3	4	5	$\cdots$	n+1
f	`  t4 = s_ - t2;`		4	5	6	$\cdots$	n+2
g	`  t5 = t4 + t3;`		5	6	7	$\cdots$	n+3
h	`  c = c + t5;`		6	7	8	$\cdots$	n+4
	`}`						
	`return(s+c);`						n+6

Table 5: The ideal execution of Sum2 takes $n+7$ cycles. The rightmost columns display the execution cycle of the corresponding line instruction for iterations $1, 2, \cdots, n-1$.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
						6_a	7_a	8_a	9_a	10_a	11_a	12_a
					5_a	6_b	7_b	8_b	9_b	10_b	11_b	12_b
				4_a	5_b	5_c	6_c	7_c	8_c	9_c	10_c	11_c
			3_a	4_b	4_c	4_d	5_d	6_d	7_d	8_d	9_d	10_d
			3_b	3_c	3_d	4_e	5_e	6_e	7_e	8_e	9_e	10_e
		2_a	2_c	2_d	3_e	3_f	4_f	5_f	6_f	7_f	8_f	9_f
	1_a	2_b	1_d	2_e	2_f	2_g	3_g	4_g	5_g	6_g	7_g	8_g
	1_b	1_c	1_e	1_f	1_g	1_h	2_h	3_h	4_h	5_h	6_h	7_h

Table 6: Sum2 loop iterations during the 13 first cycles of its ideal execution (including starting cycle 0). Instruction α of iteration i is i_α ($\alpha = a, b, \cdots, h$). 7 iterations over the 12 started ones are completed. Sum2 ideally runs 8 instructions per cycle – after 6 initialising iterations.

iterations complete in $n+4$ cycles. The final correction `s+c` adds one cycle

to the return statement. The total cycle count of the Sum2 ideal execution is $C_{\mathsf{Sum2}} = n + 7$ including the initialisation step.

Sum2 executes $I = 8n - 5$ instructions. The ILP tends to 8 executed instructions per cycle for large n. Compared to Sum, Sum2 executes 8 times more instructions (to double the accuracy of the computed result) but without introducing any significant cycle over-cost on the ideal machine.

3.2.3 DDSum

DDSum		iter.	1	2	3	...	$n-1$
	`s = x[0];`		0				
	`for(i=1; i<n; i++){`						
a	`  s_ = s;`		1	8	15	$\cdots$	7n-13
b	`  s = s + x[i];`		1	8	15	$\cdots$	7n-13
c	`  t = s - s_;`		2	9	16	$\cdots$	7n-12
d	`  t2 = s - t ;`		3	10	17	$\cdots$	7n-11
e	`  t3 = x[i] - t;`		3	10	17	$\cdots$	7n-11
f	`  t4 = s_ - t2;`		4	11	18	$\cdots$	7n-10
g	`  t5 = t4 + t3;`		5	12	19	$\cdots$	7n-9
h	`  s_l = s_l + t5;`		6	13	20	$\cdots$	7n-8
i	`  s_ = s;`		2	9	16	$\cdots$	7n-12
j	`  s = s + s_l;`		7	14	21	$\cdots$	7n-7
k	`  e = s_ - s;`		8	15	22	$\cdots$	7n-6
l	`  s_l = s_l + e;`		9	16	23	$\cdots$	7n-5
	`}`						
	`return(s);`						7n-4

Table 7: The ideal execution of DDSum takes $7n - 5$ cycles. The rightmost columns display the execution cycle of the corresponding line instruction for iterations $1, 2, \cdots, n-1$.

The core loop of DDSum contains 12 instructions executable in 9 cycles: `t2` and `t3` are again independent variables and the two copies of `s` (instructions a and i) can be executed as soon as the sources are updated (resp. instructions j and b). Two consecutive iterations of the core loop are linked by the values of `s` and `s_l` (resp. instructions j and l). Square boxed values in Table 5 exhibit the longest iteration dependency on `s`. This dependency

introduces a 7 cycles delay between two consecutive iterations (while only 4 cycles are introduced by the `s_l` update). A new iteration starts every 7 cycles and two consecutive iterations overlap during 2 cycles.

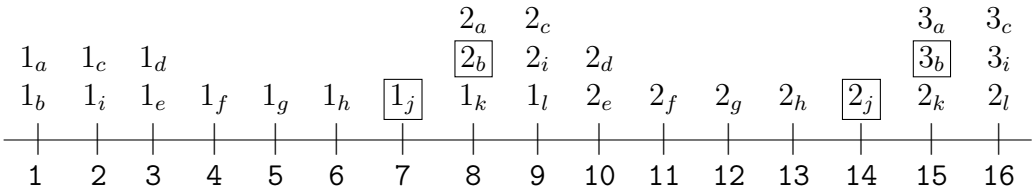

Table 8: DDSum loop iterations during the 17 first cycles of its ideal execution (including starting cycle 0). Instruction α of iteration i is i_α ($\alpha = a, b, \cdots, l$). 2 iterations over the 3 started are completed. DDSum ideally runs about 1.7 instructions per cycle.

DDSum consists in $n-1$ loop iterations. Hence the total cycle count for its ideal execution is $C_{\mathsf{DDSum}} = 7n - 5$. Compared to Sum, the accuracy improvement of DDSum increases the cycle count by a factor of 7.

DDSum executes $I = 12n - 10$ instructions. Its ILP tends to about 1.7 executed instructions per cycle for large n. DDSum exhibits about 4.5 less instruction level parallelism than Sum2 as illustrated by Table 8.

3.2.4 ILP explains Table 2

In next Table 9, we recall the measures from Table 2 and summarize the previous ILP analysis for Sum, Sum2 and DDSum programs.

The DDSum ILP analysis exhibits nearly the same cycle over-cost than the measured one (resp. 7.5 and 7). It means that in the real machine used for the cycle measure, the available computing units have been able to fully exploit the DDSum ILP, running it at more than 90% of its performance potential. However, DDSum ideally runs 7 times slower than Sum since its ILP is low. Sum2 exhibits more ILP than DDSum, explaining its better measured performances on current computing units. Sum2 presents such a good ILP that it has the potential to run as fast as the original Sum program while computing at a doubled precision. Table 6 exhibits the very good

Metric	Sum	Sum2	DDSum
Flop count (approx. ratio)	1	7	10
Measured #cycles (approx. ratio)	1	2.5	7.5
Flop count / measured #cycles (approx.)	1	2.8	1.4
Ideal C (approx. ratio)	1	1	7
Ideal I / C (approx.)	1	8	1.7

Table 9: ILP analysis explains Table 2 measures

regularity of its execution parallel flow: 8 floating-point additions can be performed at every cycle. This parallelism is exploitable only when enough floating-point units are provided. As this is not yet true on actual processors, only 30% of the ILP is exploited by current SIMD units (SSE, AVX).

In both cases, the ideal cycle ratio C (compared to Sum) is more significant than the theoretical flop count to explain and strengthen the experimental measures.

3.3 The PerPI tool automatizes this ILP analysis

In [2], Goossens *et al.* present the software tool PerPI which automatizes the ILP analysis. PerPI measures and visualises the ILP of x86-coded algorithms.

The measuring part of PerPI is a Pin tool [16]. Pin [11] is an Intel (R) free programmable tool. Pin consists of an engine which instruments any code at run time with user-defined measurement routines. PerPI is a set of routines aiming at computing the run code ILP. The ILP is computed while the real code is run. The examining routine gives the control to the examined code for a single instruction run and recovers control to update its examination statistics. This back and forth execution is continued until the examined code has been fully scanned. At each step of the examination, PerPI computes the run cycle of the examined instruction, increments the number of instructions run so far and possibly updates the highest run cycle. In such a way, PerPI computes the ratio I/C, where I is the number of machine instructions run, and C is the number of cycles needed to complete the run.

In practice, any x86 binary file may be measured with PerPI. PerPI also outputs histograms of instructions per cycle and data-dependency graphs. Histograms are similar to Tables 6 and 8. Measured instructions are x86

machine code ones. More details are presented in [2].

PerPI provides reproducible results for a given x86 binary file run on any sample of a given data set. Only one run is sufficient to reliably measure the performance potential of one implementation, and the analysis is also automatic. For instance one run on one summand vector of a given length and condition number yields the expected measure for (most of) the summation algorithms. This is the main and significant contribution this tool allows us. This simplifies very much the workload and avoids the previously described experimental uncertainties.

PerPI measures the Henessy-Patterson ideal run of a x86 binary file. This ideal run may suffer from some compiler choices. We have sometimes observed that some compiler options or versions could degrade the instruction level parallelism of the binary program. This allows the user to focus and to understand how the algorithm, the compiler and the architecture interact. Hence the implementation's experimental measures are also explained and strengthened, or put into perspective.

The next Section is devoted to this reliable run-time analysis for recent accurate and faithful summation algorithms.

4 ILP analysis of recent summation algorithms

We consider seven recent and fast summation algorithms. Two are the previous Sum2 and DDSum accurate ones —the returned sum is as accurate as if computed with twice the precision. The five faithful summation algorithms are iFastSum, AccSum, FastAccSum, HybridSum and OnLineExact. We briefly describe the principles of these latter ones to justify how to choose the test data sets.

4.1 Some recent faithful and fast summation algorithms

We already mentioned Zhu and Haye's distillation iFastSum which is a SumK algorithm with a clever dynamic error control [23].

AccSum and FastAccSum are two adaptative summation algorithms proposed by Rump *et al.* in [18, 17]. They split the summands into chunks that sum exactly and then they carefully sum as many chunks as necessary. The

chunk width depends on the total number of summands n. To satisfy the targeted accuracy constraint, this latter n has to be less than about the square root of the precision, *e.g.* $2^{26} \approx 6.7 \cdot 10^7$ in the binary64 format. The chunk position is set in AccSum, depending on the maximum absolute value of the summands; it is more dynamic in FastAccSum. The latter choice also reduces the number of floating-point operations by 30% which explains Rump's denomination. Rump measures better than expected run-times he attributes to ILP quoting previous results of one of this paper's author [8, 9]. The computational effort of these three algorithms is proportional to the conditioning of the sum.

Zhu and Hayes' HybridSum and OnLineExact rely on the exponent extraction of the summands [23, 24]. This allows to carefully accumulate the summands in one or two short vectors (resp. for HybridSum and OnLineExact) whose length depends on the exponent range, *e.g.* 2048 in the binary64 format. So the initial sum of length n is transformed into the distillation of this short length vector (with iFastSum).

Zhu and Hayes optimize the final distillation step in OnLineExact, removing all the zero components of the exponent indexed vector. The new computational effort is mainly the n summands exponent extraction step. For large n, Zhu and Hayes claim that neither the short vector distillation, nor the actual exponent range of the summands significantly over-cost the first step [24, p.7]. They also observe a significant difference between the asymptotic flop count ($5n$) and the running times they measure for OnLineExact. The authors argue that OnLineExact improves HybridSum thanks to more ILP: two sequences of 4 floating-point operations seem to be optimized for the considered implementation [24, p.4]. PerPI will automatically highligth and justify these claims.

To conclude, the computational cost of these algorithms at least depends on three parameters: the sum length n, its condition number *cond* and its summand exponent range δ. Corresponding test data are detailed in A.1.

4.2 PerPI measures of summation algorithms

We mentioned that PerPI allows us to produce reliable results only running the program once for a given value set of the parameters.

Figure 1 is a screen copy of PerPI results. It presents three runs of two sample summation algorithms for a given vector of summands. First three

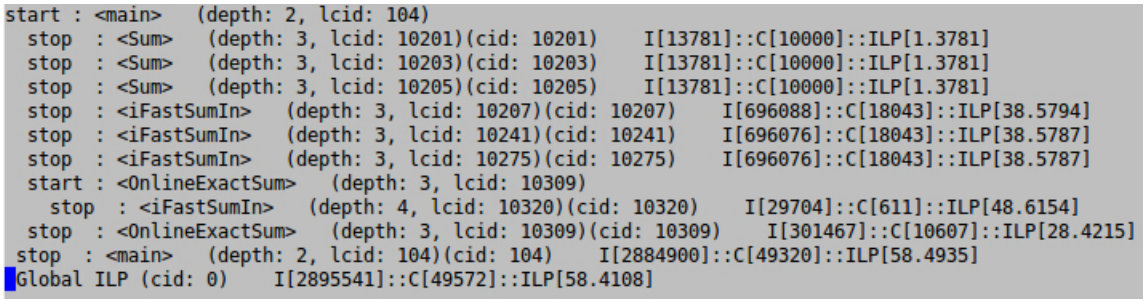

Figure 1: Reproducibility: one run is enough

runs of Sum share the same number of instructions I and cycles C, resp. $I = 13781$ and $C = 10000$. Thanks to the previous ILP analysis of Sum presented in Section 3.2.1, this latter value exhibits that the tested sum length was $n = 10000$ here.

Next three runs are iFastSum ones. Surprisingly, the number of instructions run is not constant, ranging from 696076 to 696088. We do not have the exact explanation of such a variation. However, it is always tiny (0.0012% here) and never impacts the number of cycles C. Neither PerPI nor Pin are responsible for this. The program applied to a unique data set really ran 12 more x86 machine instructions in the first run than in the two next ones.

The last lines in Figure 1 illustrate two nested measures: OnLineExact calls iFastSum to distillate the short vector after the exponent extraction step. PerPI measures OnLineExact whole execution and also provides the local iFastSum internal run measure.

This reproducibility improves the reliability of the testing step and simplifies it.

4.3 PerPI result analysis for the summation algorithms

For the considered summation algorithms we measure the minimum number of cycles C running binaries generated by three recent versions of gcc described in Figure A.2. In the next figures, we present it as ratios compared to Sum, *i.e.* $C_{\mathsf{Sum}} = 1$; in other words the over-cost for accuracy or faithfulness measured as a multiplicative factor of the number of cycles. The (absolute) PerPI measure for the classic n length summation is $C^{abs}_{\mathsf{Sum}} = n$. Hence, these ratios also equal C/n, the average number of cycles for accurately or faithfully add each summand of a n length entry.

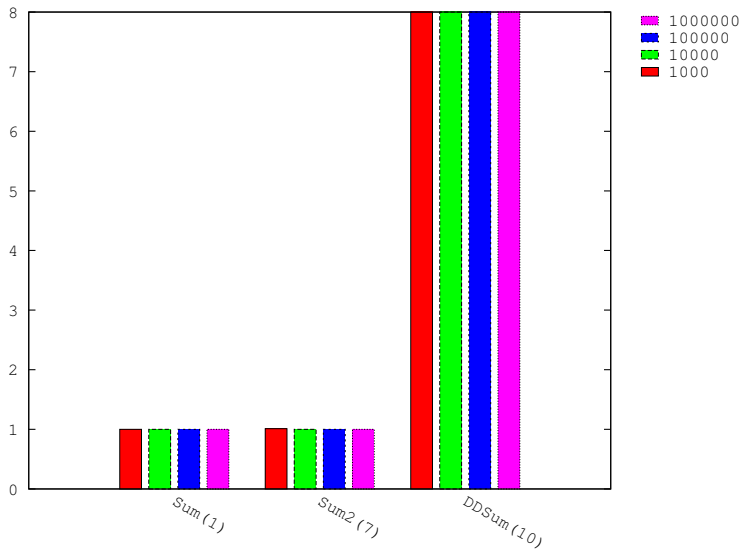

Figure 2: Number of cycles: ratios vs. Sum for $cond = 10^{32}$ and $n = 10^3, 10^4, 10^5, 10^6$. (x-axis value in parenthesis is the flop count over n)

4.3.1 Accurate summations

Accurate Sum2 and DDSum are presented in Figure 2 while the five faithful algorithms in next Figure 3 for varying n length sums of the same condition number $1/\mathbf{u}^2$. To compare the PerPI measures with the classic flop count, the flop count ratio over n is the value in parenthesis recalled for every algorithm.

Compensated Sum2 is definitely the proper choice to compute a twice more accurate sum. As expected from the algorithmic level ILP analysis presented in Section 3.2, Sum2 benefits from a high ILP and so (ideally) introduces no significant timing over-cost compared to original Sum. On the contrary, DDSum takes 8 times more cycles to (ideally and actually) return the accurate sum —which here nearly equals the full computing precision $\mathbf{u}$. Neither n nor *cond* impact these over-costs.

The ratio PerPI measure for DDSum is one unit larger than the one in Table 9 derived from our algorithmic level analysis —we also recall that the measured one is 7.5 in Table 9. As already mentioned it is not surprising since PerPI measures the code actually generated by the compiler: the one which is actually run by the computer. This one unit difference comes from

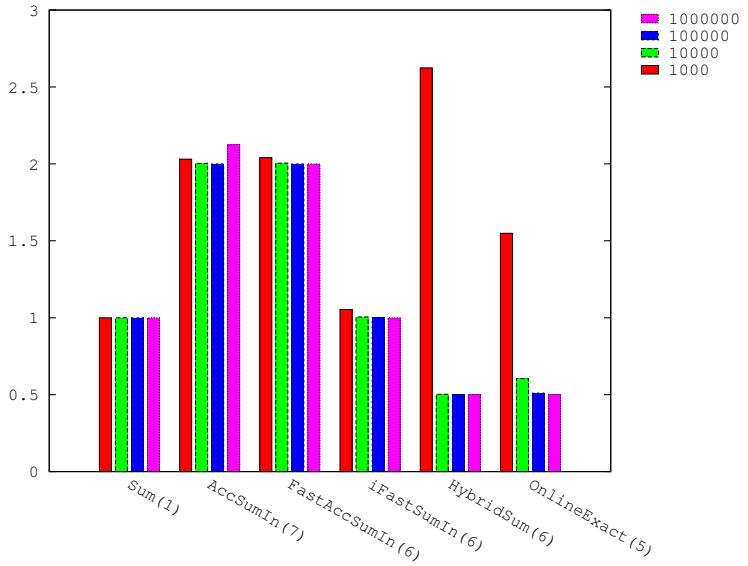

Figure 3: Number of cycles: ratios vs. Sum for $cond = 10^{32}$ and $n = 10^3, 10^4, 10^5, 10^6$. (x-axis value in parenthesis is the flop count over n)

the x86 instruction structure and so illustrates the ISA dependency. In next three lines from Figure 7, `s` value is used twice and `s_` has to be conserved until instruction *f*. The compiler introduces one intermediate copy after instruction *c* to fit the x86 instruction structure that modifies one of the two source registers.

```
c  t  = s - s_;
d  t2 = s - t ;
f  t4 = s_ - t2;
```

To conclude about accurate summations, PerPI results are broadly consistent with the algorithmic level ones and confirm the analysis of Section 3.2.

4.3.2 Faithful summations

The ratios for the faithfully rounded sums are presented on Figure 3. All of them are significantly smaller than the flop count ones. These algorithms benefit from an important ILP that justifies better than expected measured timings. The rightmost algorithms —the newest ones— are the potentially fastest ones. Oldest FastAccSum and AccSum roughly double the cycle over-

cost compared to the newest ones. Nevertheless, the FastAccSum flop count improvement ($3n$ flop inner loop) does not provide a faster ideal execution than AccSum($4n$ flop inner loop). The ideal execution of both iFastSum and these two algorithms does not significantly depend on the sum length n.

4.3.3 A focus on HybridSum and OnLineExact

HybridSum and OnLineExact exhibit a very impressive and interesting behavior. For larger n their ideal executions return a faithfully rounded value in about half less cycles than the original Sum (which here returns a computed sum with no significant digit). PerPI measure highlights this property but how justify it?

The first step of HybridSum and OnLineExact is the extraction of the summand exponents that generates a shorter vector whose length is bounded by the exponent range (which is independent of n). It is clear that this extraction part represents the main computational cost of their executions. The assembly code study provides a two step explanation.

The first point comes from Sum that ideally runs in about $C = n$ cycles. The compiler unrolled the Sum inner-loop, *e.g.* gcc did it 8 times, and so improves its control part: the number of control instructions is roughly divided by 8. Nevertheless this loop unrolling does not shorten the longest dependency chain made of the sequence of floating-point accumulations (actually a scalar addition performed by the vector unit). We assume that no new intermediate variable has been introduced by the compiler to hold the initial order of the floating-point operations[2]. Hence the ideal execution of Sum for a length n summation is forced to n cycles. On the contrary HybridSum and OnLineExact do not suffer from such a long constraint since they only sum the short vector(s) of length set produced by the exponent extraction step. This step is basically one run through the n length entry vector (to extract every summand exponent and actualize the corresponding cell of the short vector). This extration step fully benefits from loop unrolling the compiler introduces: here gcc unrolls it twice and so reduces by a factor 2 the run through the n length vector. This explains that PerPI mesures about twice less cycles than Sum for both HybridSum and OnLineExact (as soon as n is large enough compared to the exponent range).

[2]In the integer summation case, the loop unroll is fully beneficial. The length of the longest dependency chain is divided by the unrolling factor.

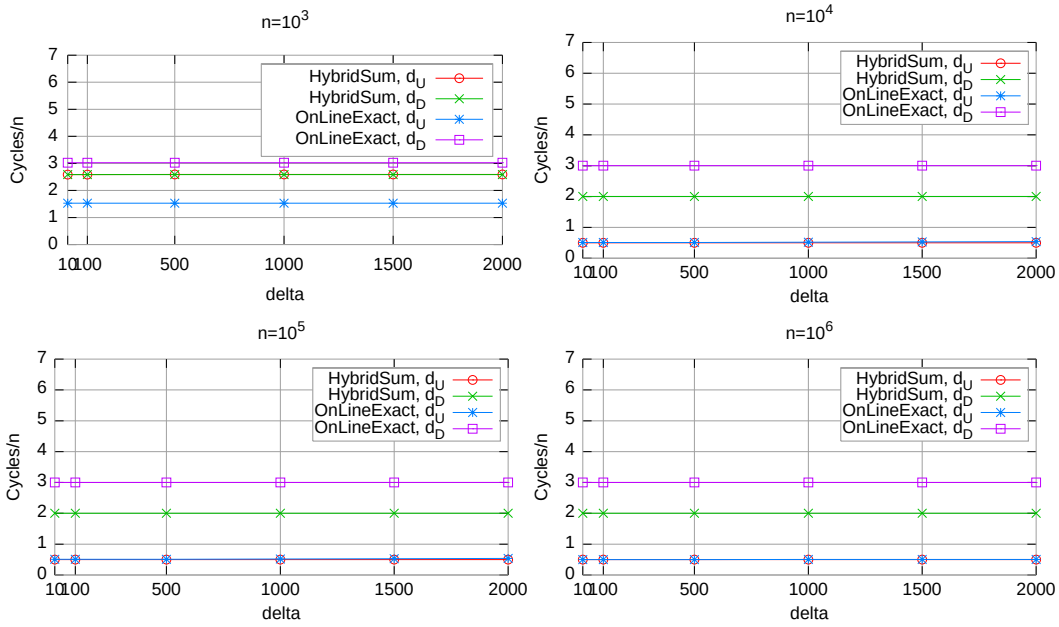

Figure 4: The d_D exponent distribution impacts more OnLineExact than HybridSum whereas it behaves similarly for d_U and $n \geq 10^4$.

Let us mention that in our case, the gcc compiler was able to automatically transform HybridSum but not OnLineExact. It has been necessary to rewrite OnLineExact in a form inspired by the HybridSum one to finally benefit from the same compiler optimization.

How does the actual exponent range of the entry vector, *i.e.* the difference between the larger and the smaller exponent, affect HybridSum and OnLineExact? Zhu and Hayes denote it δ and mention that it "does not influence the running time much except ... for real sum equals zero" [24, p.7]. A larger exponent range increases the short vector length and the final distillation too. We have already mentioned that this step computational cost is not significant compared to the extraction one. Indeed Figure 4 illustrates that the execution lengths of HybridSum and OnLineExact (here normalized by n) are constant while delta varies.

Nevertheless these ideal timings actually vary with factors of 4 and 6 (resp.) for two data despite sharing the same exponent range δ. More explanation is necessary.

```
start : <HybridSum>
  start : <iFastSumIn>
  stop  : <iFastSumIn>     I[62719]::C[2580]::ILP[24.3097]
stop  : <HybridSum>      I[267980]::C[20020]::ILP[13.3856]
start : <OnlineExact>
  start : <iFastSumIn>
  stop  : <iFastSumIn>     I[334]::C[32]::ILP[10.4375]
stop  : <OnlineExact>    I[229263]::C[30026]::ILP[7.63548]
```

Figure 5: Final iFastSum runs more cycles in HybridSum than in OnLineExact (2580 vs. 32) for data d_D contrary to the whole summation process (20020 vs. 30026) —here $n = 10^4$ and $\delta = 500$.

The exponents of the length n entry vector d_U are uniformely randomized in the range $[-\delta/2, \delta/2]$ —U stands for uniform. Conversely vector d_D has $n-1$ entries with the same exponent $-\delta/2$, and the last one with the exponent $\delta/2$ —D stands for Dirac. The exponent distribution modifies the instruction level parallelism of HybridSum and OnLineExact. HybridSum and OnLineExact have very similar execution lengths for data with uniformly distributed exponents while OnLineExact suffers more than HybridSum from highly repetitive exponents. This point is surprising since OnLineExact includes a concatenation step that produces a very short vector in the latter case (actually about twice the number of different value exponents, *e.g.* about 4 for d_D). Hence the final distillation step (with iFastSum) is definitely reduced to the minimum. In comparison this final step in HybridSum distills one vector whose length is set to the exponent range (independently of δ, *e.g.* 2048 here). PerPi actually measures it as presented in Figure 5. Nevertheless these two treatemnts actually contribute very similarly to the ideal execution length of the whole sum: added cycles are of the order of the non zero summand number. In the ideal model every instruction is executed as soon as its sources are available. Since the zero components of d_D do not contribute to the final step sum all the instructions using it are run very early in the ideal computation chain. The difference comes from the extraction step loop as we explain it now.

Figure 6 is HybridSum extraction core loop. It contains 8 instructions `(a)`,...,`(h)` run in 6 cycles. Two consecutive iterations are linked only if they contain the same index j_h or j_l: accumulation is sequential in the matching component $a[\,]$. Hence these iterations are highly parallel in general (thanks to index loop k independency), and for d_U in particular. In this case the

Cycle				
1	(a)	$a_s = x[k] \times \text{split\_constant}$		
2	(b)	$t = a_s - x[k]$		
3	(c)	$h = a_s - t$		
4	(d)	$j_h = \text{exponent}(h)$	(e)	$l = x[k] - h$
5	(f)	$a[j_h] = a[j_h] + h$	(g)	$j_l = \text{exponent}(l)$
6			(h)	$a[j_l] = a[j_l] + l$

Figure 6: HybridSum extraction step loop: only one cycle delay from the $a[j_h]$ and $a[j_l]$ iteration dependency.

critical path becomes the loop control and so the d_U extration step takes about $n/2$ cycles since the loop is actually unrolled twice —see Figure 4. On the contrary for data d_D this extraction loop is fully sequential: exponent j_h is shared by $n-1$ iterations. Since two linked iterations could start every cycle, the d_D extraction step should take about n cycles. It is necessary to study the assembly code to justify why we measure about $2n$ cycles for this latter case —see Figure 4 or Figure 5. Two consecutive linked iterations actually start every two cycles. Every accumulation `(f)` and `(h)` ($a[j_z] = a[j_z] + z$, with z equals h or l resp.) is translated in two consecutive and linked assembly instructions: *i)* store in a temporary register the result of one add between one register value (h or l) and one memory value (matching $a[\,]$), then *ii)* move the temporary register value back to the memory ($a[\,]$). The availability of an instruction that could add register and memory values and store the result back in the memory would overcome this ISA limitation. This explains the previously mentioned 4 factor between d_U and d_D with a same exponent range δ summed with HybridSum.

Figure 7 exhibits more dependency between the iteration loops through linked accesses of table a_1 produced by the extraction step in OnLineExact. Instructions `(c-c')` rewrite the table update $a_1[j] = a_1[j] + x[k]$ in such a way that its values remains in c_u —and so avoid the table access, *i.e.* memory access, in next instructions `(d-e)`. Let us remark it minimizes the delay between the implicit read-update-store within $a_1[j] = a_1[j] + x[k]$. The linked instruction chain `(b-c-c')` introduces a 3 cycles delay between two consecutive starts of iterations that share the same $a_1[j]$. The whole computing chain while summing d_U is not significantly enlarged by such a delay: the high parallelism (with respect to the index loop k) of the whole extraction step is dominant compared to few $a_1[j]$ conflicts. Again the d_U extraction

Cycle				
1	(a)	j =exponent(x[k])		
2	(b)	$c = a_1[j]$		
3	(c)	$c_u = c + x[k]$		
4	(d)	$h = c_u - c$	(c')	$a_1[j] = c_u$
5	(e)	$t_1 = c_u - h$	(f)	$t_3 = x[k] - h$
6	(g)	$t_2 = c - t_1$		
7	(h)	$l = t_2 + t_3$		
8	(i)	$a_2[j] = a_2[j] + l$		

Figure 7: OnLineExact extraction step loop: instruction chain (b-c-c'), *i.e.* $a_1[j] = a_1[j] + x[k]$, introduces a 3 cycles delay between two consecutive starts of iterations that share the same $a_1[j]$.

step by OnLineExact takes about $n/2$ cycles for the same reasons than for HybridSum. Conversely the critical computing chain is significantly impacted by the 3 cycles delay when summing d_D with OnLineExact. Now the d_D extraction step needs about $3n$ cycles to be completed. So OnLineExact algorithmic constraints explain the previously mentioned 6 factor between summing d_U and d_D having the same exponent range δ.

To finish we mention that OnLineExact extraction step loop in Figure 7 inlines Knuth's 2Sum (instructions (d-e-f-g-h). This latter can be replaced by Dekker's Fast2Sum, a similar error-free transformation of the floating-point addition. It does not improve OnLineExact here. Fast2Sum also suffers from the previously identified 3 cycles delay and includes branches that prevent the compiler to unroll the extraction loop. In this case PerPI measures more than n cycles for d_U (compared to $n/2$), and the same $3n$ for d_D since loop unroll does not improve this summation case.

Figure 8 shows the PerPI histograms of the ideal runs of HybridSum and OnLineExact. It display the instructions (y-axis) run for every cycles (x-axis); colors differ according to the instruction type, *e.g.* SSE instructions are orange while data transfer ones are purple. This illustrates again the exponent distribution effect while summing d_U and d_D. The d_D case run exhibits two phases. In the first phase HybridSum and OnLineExact runs start globally replicating the whole d_U run, resp. about the first 5000 and 6000 cycles. The exponent extraction step runs during this $n/2$ length step: shift (cyan) and logical (blue) instructions applied to identify the binary exponent field

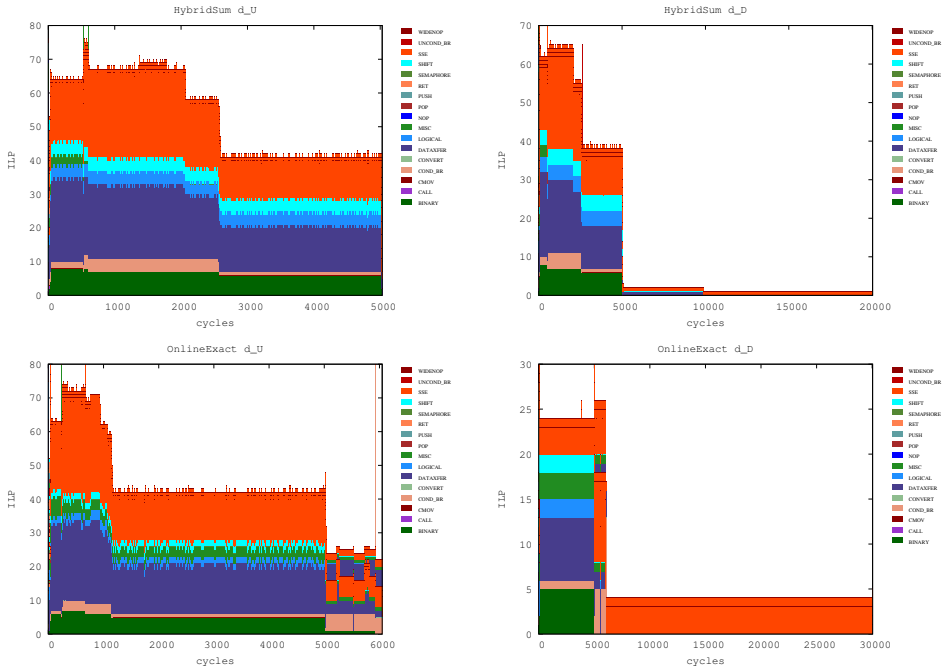

Figure 8: Ideal run histograms of HybridSum (top) and OnLineExact (down) for d_U (left) and d_D (right), $n = 10^4$.

disapear after 5000 cycles. HybridSum ends in few more cycles in the d_U case: every summation of the exponent indexed table entries starts as soon as all extractions and accumulations of the matching entries have been processed. Since d_U exponents are uniformely distributed the whole final summation step (with iFastSum) runs actually in parallel with the extraction step. For d_D, the HybridSum histogram second phase, *i.e.* after cycle 5000, is the remaining part of the $2n$ long extraction step previously identified. It contains only SSE and data transfert instructions. Similarly the last histogram exhibits the $3n$ long extraction step in OnLineExact for data d_D.

To conclude HybridSum and OnLineExact both benefit from about the same ILP in general. We have explained how the exponent distribution, more than the exponent range, may degrade this performance potential. OnLineExact suffers more than HybridSum from algorithmic constraint which may benefit the latter in practice. Nevertheless these faithful algorithms both exhibit significantly better ideal performance than the classic inaccu-

rate summation thanks to a highly parallel extraction step that yields a very short final faithful sum step.

This long discussion illustrates how tedious is the detailed analysis of these performance issues, even in the simplified ideal model PerPI implements. It highlights how the compiler and the targeted ISA condition these measures and the benefit of analyzing both the algorithmic and assembly code levels.

5 Conclusion

Highly accurate algorithms need reliable performance evaluation. The classical flop count is no more significant enough for core algorithms and modern computing units. The measures based on the hardware counters suffer from uncertainty and non reproducibility. This important issue is reported in recent publications proposing accurate summation algorithms aiming to always faster run.

In this paper we detailed how the analysis of the instruction level parallelism (ILP) of these algorithms provides a reliable and significant measure of their performance potential. PerPI is a software platform to analyze and visualise the ILP of programs compiled for the very common x86 based computing units. We used PerPI to analyze recent accurate and faithful summation algorithms. Presented results are reliable and reproducible both in time and location. The ideal execution analysis helps the user to draw up realistic correlations with the measured ones. PerPI automatically justifies the ILP effect some authors previously claim in this scope. It also gives a better understanding of the intrinsic behavior of the algorithm, how it interacts with the compiler and the ISA and how it benefits from the underlying microarchitecture. This analysis may provide optimisation ideas as illustrated for AccSum and FastAccSum in [2].

The paper is a first step toward more science and less hazard when analysing the computing time of such core numerical algorithms. In this scope PerPI results are still not always abstract enough. We have shown that they may depend both on the instruction set and on the compiler, its versions and its options. Defining the good abstraction level for such an analysis is not an easy task. How choose between the algorithmic level analyis and the assembly language level one? The former is more abstract and

generic, the latter is more pragmatic and realistic. In our experience both are fruitful and have been necessary to yield presented results.

While being reliable and reproducible these results suffer from one analysis bias. Our approach is rather significant of the summation algorithm developer work. We focus these algorithms in themselves, *i.e.* without encompassing them in applications using them, *e.g.* integration schema, linear algebra subroutines... Such larger evaluation frames may reduce the global gain the previously identified fastest summation algorithms introduce. Choosing a significant benchmark set for these accurate summation algorithms is a first issue to consider in the future.

We aim to guarantee reproducibility to improve the reliability of accurate floating-point summation. The PerPI results presented here provide a reference for evaluating future contributions. The next step is the development of an open and dynamic website repository satisfying the two main following characteristics. First the website gathers and shares all the resources that yield to a result it presents: tested and test files, make and program source files, data files and data generators, real and ideal associated measures, ... The website also provides an open interaction that can be described such as: upload your new algorithm, upload your new data, run them, compare it to the state of the art and so let's reliably contribute.

Acknowledgment. Authors thank Dr. Nicolas Louvet for his very first and significant contributions to this work. Authors also thank Prof. Siegfried M. Rump and Dr. Takeshi Ogita for their fruitful and incentive discussions. Thanks to Prof. Denis Bonnecase for having checked the english language used in this paper.

A The measure experimental process

A.1 How to choose the test data?

As mentioned in Section 3.2, the time complexity parameters vary according to the summation algorithms. These parameters and corresponding tested values are presented in next Tables A.1 and A.1. Parameter n is the sum length and *cond* is the condition number of the summation as defined in Section 1.1. The difference between the maximum and the minimum exponent of the summands is denoted δ as in [23].

Algorithm	Parameters	Accuracy
Sum	n	Classic summation
Sum2, DDSum	n, *cond*	Twice more precision
AccSum, FastAccSum, iFastSum	n, *cond*	Faithful summation
HybridSum, OnLineExact	n, *cond*, δ	Faithful summation

Table 10: Timing parameters of the sum algorithms. n: sum length, *cond*: condition number, δ: summand exponent length.

Parameter	Tested values
n	$10^3, 10^4, 10^5, 10^6$
cond	$[10^8, 10^{40}] \approx [\mathbf{u}^{-1/2}, \mathbf{u}^{-2.5}]$
δ	$10, 100, 500, 1000, 1500, 2000$

Table 11: Parameters value set.

We use the Rump *et al.*'s generator of arbitrary ill-conditioned dot product [15] we modified such that it generates summands that cover an arbitrary exponent range $[-\delta/2, \delta/2]$. Special entries d_D of Section 4.3 are derived from this generator slightly modified to match the desired data shape.

A.2 Our fuzzy PAPI picture

As explained in Section 2, we present as few as possible results of measured run-times. Table A.2 only aim to present the general tendency of observed results. They should be analysed more qualitatively than quantitatively. These

cond	Sum2	DDSum	AccS.	FastAccS.	iFastS.	HybridS.	OnLineE.
10^{8}	2–3	7–8	4–5	4–5	7–8	5*	4*
10^{16}	2–3	7–8	5–6	5–6	7–8	5*	4*
10^{24}	–	–	7–8	7	13	5*	4*
10^{32}	–	–	8–9	7–8	18	5*	4*
10^{40}	–	–	10-12	9–10	18+	5*	4*

Table 12: Tendencies of the measured run-time ratios (Sum=1); * for $n > 10^5$.

Intel(R) Core(TM) i7 CPU870 2.93GHz, x86_64
GNU/Linux kernel 2.6.38-8-generic
gcc (4.6.3) -std=c99 -march=corei7 -mfpmath=sse -O3 -funroll-all-loops
icc (12.0.4_20110427) -std=c99 -mtune=corei7 -xSSE -axsse4.2 -O3 -funroll-all-loops

Figure 9: Computing environment for measured run-times

tendencies summarize several set of experiments that have been realised as carefully as possible.

Timings rely on the hardware performance counters mainly using the PAPI library. In this case, the counter delays have been evaluated and taken into account. Direct access to some hardware performance counter (*e.g.* the time-stamp counter) with assembly language call gives the same tendencies. For every set of parameter values, timing is the average of reliable 50 runs.

Binary code were generated by two compilers with options that provide as fast as possible reliable runs. These computing environment details are presented in Figures A.2 and A.2.

Intel(R) Core(TM)2 Duo CPU P8800 @ 2.66GHz, x86_64
Linux version 3.2.0-3-amd64 (Debian 3.2.23-1)
gcc-4.7 (Debian 4.7.1-7) 4.7.1
gcc-4.6 (Debian 4.6.3-8) 4.6.3
gcc-4.5 (Debian 4.5.3-12) 4.5.3
gcc -std=c99 -march=core2 -msse2 -mfpmath=sse -O3 -funroll-all-loops

Figure 10: Computing environments for PerPI measures

References

[1] D. H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, pages 54–55, Aug. 1991.

[2] B. Goossens, P. Langlois, D. Parello, and E. Petit. PerPI: A tool to measure instruction level parallelism. In K. Jónasson, editor, *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part I*, volume 7133 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 2012.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 2003.

[4] URL = `http://crd.lbl.gov/~dhbailey/mpdist`, .

[5] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

[6] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, New York, Aug. 2008.

[7] U. W. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and in Practice*. Academic Press, New York, 1981.

[8] P. Langlois. Compensated algorithms in floating point arithmetic. In *12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics, Duisburg, Germany*, Sept. 2006. (Invited plenary speaker).

[9] P. Langlois and N. Louvet. More instruction level parallelism explains the actual efficiency of compensated algorithms. Research Report hal-00165020, DALI Research Team, July 2007. http://hal.archives-ouvertes.fr/hal-00165020.

[10] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, implementation and testing of extended

and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002.

[11] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation,*, pages 190–200. ACM, 2005.

[12] M. A. Malcolm. On accurate floating-point summation. *Comm. ACM*, 14(11):731–736, 1971.

[13] The MPFR library. URL = `http://www.mpfr.org/`, .

[14] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[15] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

[16] URL = `http:www.pintool.org`, .

[17] S. M. Rump. Ultimately fast accurate summation. *SIAM J. Sci. Comput.*, 31(5):3466–3502, 2009.

[18] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation – part I: Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[19] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation – part II: Sign k-fold faithful and rounding to nearest. *SIAM J. Sci. Comput.*, 31(2):1269–1302, 2008.

[20] V. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results? In *3rd Workshop on Functionality of Hardware Performance Monitoring, 2010*, pages 1–11, Atlanta, USA, 2010.

[21] A reference implementation for extended and mixed precision BLAS. `http://crd.lbl.gov/~xiaoye/XBLAS`, .

[22] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA*, pages 23–32, 2009.

[23] Y.-K. Zhu and W. B. Hayes. Correct rounding and hybrid approach to exact floating-point summation. *SIAM J. Sci. Comput.*, 31(4):2981–3001, 2009.

[24] Y.-K. Zhu and W. B. Hayes. Algorithm 908: Online exact summation of floating-point streams. *ACM Transactions on Mathematical Software*, 37(3):37:1–37:13, Sept. 2010.