Validated performance of accurate algorithms

Bernard Goossens, Philippe Langlois, David Parello

DALI Research Project, University of Perpignan Via Domitia
LIRMM Laboratory, CNRS – University Montpellier 2, France.
Context and motivation

Context: Floating point computation using IEEE-754 arithmetic (64 bits)

Aim: Improve and validate the accuracy of numerical algorithms . . .
. . . without sacrificing the running-time performances

Improving accuracy:
Why ? result accuracy $\approx$ condition number $\times$ machine precision
How ? more bits

- double-double (128) or quad-double libraries (256)
- MPFR (arbitrary # bits, fast for 256+)
- Compensated algorithms
Computed accuracy is constrained by the condition number.

Backward stable algorithms

Compensated algorithms

Highly accurate Faithful algorithms
Compensated algorithms

- summation and dot product: Knuth (65), Kahan (66), . . . , Ogita-Rump-Oishi (05,08)
- polynomial evaluation: Horner (Langlois-Louvet, 07), Clenshaw, De Casteljau (Hao et al., 11)
- triangular linear systems: (Langlois-Louvet, 08)

These algorithms are fast in terms of measured computing time

- Faster than other existing solutions: double-double, quad-double, MPFR
  Question: how to trust such claim?
- Faster than the theoretical complexity that counts floating-point operations
  Question: how to explain and verify such claim —at least illustrate?
A classic problem: I want to double the accuracy of a computed result while running as fast as possible?

A classic answer:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Eval</th>
<th>AccEval1</th>
<th>AccEval2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flop count</td>
<td>2n</td>
<td>22n + 5</td>
<td>28n + 4</td>
</tr>
<tr>
<td>Flop count ratio</td>
<td>1</td>
<td>≈ 11</td>
<td>≈ 14</td>
</tr>
<tr>
<td>Measured #cycles ratio</td>
<td>1</td>
<td>2.8 – 3.2</td>
<td>8.7 – 9.7</td>
</tr>
</tbody>
</table>

Flop counts and running-times are not proportional. Why? Which one trust?
Running-time measures: details

Average ratios for polynomials of degree 5 to 200
Working precision: IEEE-754 double precision

<table>
<thead>
<tr>
<th></th>
<th>CompHorner</th>
<th>DDHorner</th>
<th>DDHorner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Horner</td>
<td>Horner</td>
<td>CompHorner</td>
</tr>
<tr>
<td>Pentium 4, 3.00 GHz</td>
<td>2.8</td>
<td>8.5</td>
<td>3.0</td>
</tr>
<tr>
<td>(x87 fp unit)</td>
<td>2.7</td>
<td>9.0</td>
<td>3.4</td>
</tr>
<tr>
<td>(sse2 fp unit)</td>
<td>3.0</td>
<td>8.9</td>
<td>3.0</td>
</tr>
<tr>
<td>(sse2 fp unit)</td>
<td>3.2</td>
<td>9.7</td>
<td>3.4</td>
</tr>
<tr>
<td>Athlon 64, 2.00 GHz</td>
<td>3.2</td>
<td>8.7</td>
<td>3.0</td>
</tr>
<tr>
<td>Itanium 2, 1.4 GHz</td>
<td>2.9</td>
<td>7.0</td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>5.9</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Results vary with a factor of 2
Life-period for the significance of these computing environments?
How to trust non-reproducible experiment results?

Measures are mostly non-reproducible

- The execution time of a binary program varies, even using the same data input and the same execution environment.

Why? Experimental uncertainties

- spoiling events: background tasks, concurrent jobs, OS interrupts
- non deterministic issues: instruction scheduler, branch predictor
- external conditions: temperature of the room (!)
- timing accuracy: no constant cycle period on modern processors (i7, . . .)

Uncertainty increases as computer system complexity does

- architecture issues: multicore, many/multicore, hybrid architectures
- compiler options and its effects
How to read the current literature?

Lack of proof, or at least of reproducibility

*Measuring the computing time of summation algorithms in a high-level language on today’s architectures is more of a hazard than scientific research.*

S.M. Rump (SISC, 2009)

The picture is blurred: the computing chain is wobbling around

*If we combine all the published speedups (accelerations) on the well known public benchmarks since four decades, why don’t we observe execution times approaching to zero?*

S. Touati (2009)
Outline

1. Accurate algorithms: why? how? which ones?

2. How to choose the fastest algorithm?

3. The PerPI Tool
   - Goals and principles
   - What is ILP?

4. The PerPI Tool: outputs and first examples

5. Conclusion
Highlight the potential of performance

General goals

- Understand the algorithm and architecture interaction
- Explain the set of measured running-times of its implementations
- Abstraction \( w.r.t. \) the computing system for performance prediction and optimization
- Reproducible results in time and in location
- Automatic analysis

Our context

- Objects: accurate and core-level algorithms: XBLAS, polynomial evaluation
- Tasks: compare algorithms, improve the algorithm while designing it, chose algorithms \( \rightarrow \) architecture, optimize algorithm \( \rightarrow \) architecture
Abstract metric: Instruction Level Parallelism

- ILP: the potential of the instructions of a program that can be executed simultaneously
- #IPC for the Hennessy-Patterson ideal machine
- Compilers and processors exploits ILP: superscalar out-of-order execution
- Thin grain parallelism suitable for single node analysis
What is ILP?

A synthetic sample: \( e = (a+b) + (c+d) \)

**x86 binary**

```
...  

i1  mov  eax,DWP[ebp-16]  
i2  mov  edx,DWP[ebp-20]  
i3  add  edx,eax     
i4  mov  ebx,DWP[ebp-8]  
i5  add  ebx,DWP[ebp-12]  
i6  add  edx,ebx  
...  
```
What is ILP?

A synthetic sample: \( e = (a+b) + (c+d) \)

### x86 binary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td><code>mov eax, DWP[ebp-16]</code></td>
</tr>
<tr>
<td>i2</td>
<td><code>mov edx, DWP[ebp-20]</code></td>
</tr>
<tr>
<td>i3</td>
<td><code>add edx, eax</code></td>
</tr>
<tr>
<td>i4</td>
<td><code>mov ebx, DWP[ebp-8]</code></td>
</tr>
<tr>
<td>i5</td>
<td><code>add ebx, DWP[ebp-12]</code></td>
</tr>
<tr>
<td>i6</td>
<td><code>add edx, ebx</code></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction and cycle counting
What is ILP?

A synthetic sample: \( e = (a+b) + (c+d) \)

### x86 binary

```
... 
```

### Instruction and cycle counting

Cycle 0: i1 i2 i4

```
i1 mov eax,DWP[ebp-16]
i2 mov edx,DWP[ebp-20]
i3 add edx,eax 
i4 mov ebx,DWP[ebp-8]
i5 add ebx,DWP[ebp-12]
i6 add edx,ebx 
... `
What is ILP?

A synthetic sample: \( e = (a+b) + (c+d) \)
What is ILP?

A synthetic sample: $e = (a+b) + (c+d)$

x86 binary

```
... i1
mov eax, DWP[ebp-16] i2
mov edx, DWP[ebp-20] i3
add edx, eax i4
mov ebx, DWP[ebp-8] i5
mov ebx, DWP[ebp-12] i6
add edx, ebx
add ebx, DWP[ebp-12]...
```

Instruction and cycle counting

Cycle 0: i1 → i2 → i4

Cycle 1: i3 → i5

Cycle 2: i6
What is ILP?

A synthetic sample: \( e = (a+b) + (c+d) \)

X86 binary

\[
\begin{align*}
&\ldots \\
&i1: \text{mov} \ eax, \text{DWP}[ebp-16] \\
&i2: \text{mov} \ edx, \text{DWP}[ebp-20] \\
&i3: \text{add} \ edx, \ eax \\
&i4: \text{mov} \ ebx, \text{DWP}[ebp-8] \\
&i5: \text{add} \ ebx, \text{DWP}[ebp-12] \\
&i6: \text{add} \ edx, \ ebx \\
&\ldots
\end{align*}
\]

Instruction and cycle counting

Cycle 0: \( i1 \rightarrow i2 \rightarrow i4 \)

Cycle 1: \( i3 \rightarrow i5 \)

Cycle 2: \( i6 \)

Number of instructions = 6, Number of cycles = 3

ILP = \# of instructions / \# of cycles = 2
ILP explains why compensated algorithms are fast

ILP:

AccEval

\[ \approx 11 \]

AccEval2

1.65
The PerPI Tool: principles

From ILP analysis to the PerPI tool

- 2008: prototype within a processor simulation platform (PPC asm)
- 2009: PerPI to analyse and visualise the ILP of x86-coded algorithms

PerPI

- Pintool (http://www.pintool.org)
- Input: x86 binary file
- Outputs: ILP measure, IPC histogram, data-dependency graph
1 Accurate algorithms: why? how? which ones?

2 How to choose the fastest algorithm?

3 The PerPI Tool

4 The PerPI Tool: outputs and first examples

5 Conclusion
Simulation produces reproducible results

start: _start
start: .plt
start: __libc_csu_init
start: _init
start: call_gmon_start
stop: call_gmon_start::I[13]:C[9]:ILP[1.44444]
start: frame_dummy
stop: frame_dummy::I[7]:C[3]:ILP[2.33333]
start: __do_global_ctors_aux
stop: __do_global_ctors_aux::I[11]:C[6]:ILP[1.83333]
stop: _init::I[41]:C[26]:ILP[1.57692]
stop: __libc_csu_init::I[63]:C[39]:ILP[1.61538]
start: main
start: .plt
start: .plt
start: Horner
stop: Horner::I[5015]:C[2005]:ILP[2.50125]
start: Horner
stop: Horner::I[5015]:C[2005]:ILP[2.50125]
start: Horner
stop: Horner::I[5015]:C[2005]:ILP[2.50125]
stop: main::I[20129]:C[7012]:ILP[2.87065]
start: _fini
start: __do_global_dtors_aux
stop: __do_global_dtors_aux::I[11]:C[4]:ILP[2.75]
stop: _fini::I[23]:C[13]:ILP[1.76923]

Global ILP ::I[20236]:C[7065]:ILP[2.86426]
Profile results to compare two algorithms

start :   _start   (depth: 1 rtn_s_d: 0)
start :   __libc_csu_init   (depth: 2 rtn_s_d: 0)
 start :    _init   (depth: 3 rtn_s_d: 0)
start :   call_gmon_start   (depth: 4 rtn_s_d: 0)
 stop :  call_gmon_start   (depth: 4 rtn_s_d: 0)    I[13]:C[9]:ILP[1.44444]
start :   frame_dummy   (depth: 4 rtn_s_d: 0)
 stop :   frame_dummy   (depth: 4 rtn_s_d: 0)    I[7]:C[3]:ILP[2.33333]
start :   __do_global_ctors_aux   (depth: 4 rtn_s_d: 0)
 stop :  __do_global_ctors_aux   (depth: 4 rtn_s_d: 0)    I[11]:C[6]:ILP[1.8]
stop :  __do_global_ctors_aux   (depth: 4 rtn_s_d: 0)    I[41]:C[26]:ILP[1.57692]
stop :  __libc_csu_init   (depth: 2 rtn_s_d: 0)    I[63]:C[39]:ILP[1.61538]
start :   main   (depth: 2 rtn_s_d: 0)
 start :    Horner   (depth: 3 rtn_s_d: 0)
stop :     Horner   (depth: 3 rtn_s_d: 0)    I[519]:C[206]:ILP[2.51942]
start :   CompHorner   (depth: 3 rtn_s_d: 0)
 stop :  CompHorner   (depth: 3 rtn_s_d: 0)    I[3732]:C[318]:ILP[11.7358]
start :   DDHorner   (depth: 3 rtn_s_d: 0)
 stop :  DDHorner   (depth: 3 rtn_s_d: 0)    I[4229]:C[2106]:ILP[2.00807]
stop :  main   (depth: 2 rtn_s_d: 0)    I[9062]:C[2509]:ILP[3.6118]
start :   _fini   (depth: 2 rtn_s_d: 0)
 start :    __do_global_dtors_aux   (depth: 3 rtn_s_d: 0)
stop :  __do_global_dtors_aux   (depth: 3 rtn_s_d: 0)    I[11]:C[4]:ILP[2.75]
stop :    __do_global_dtors_aux   (depth: 3 rtn_s_d: 0)    I[23]:C[13]:ILP[1.76923]

Global ILP    I[9169]:C[2562]:ILP[3.57884]
Histograms to compare two algorithms

compensated summation

double-double summation
Visualisation of the instruction dependence graph
Instruction dependence analysis to compare two algorithms

*Ultimatlly Fast Accurate Summation*. S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:
- $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]

<table>
<thead>
<tr>
<th>cond \ n</th>
<th>100</th>
<th>300</th>
<th>1000</th>
<th>3000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^6$</td>
<td>1.09</td>
<td>1.18</td>
<td>1.30</td>
<td>1.35</td>
<td>1.33</td>
</tr>
<tr>
<td>$10^{16}$</td>
<td>1.22</td>
<td>1.22</td>
<td>1.29</td>
<td>1.30</td>
<td>1.88</td>
</tr>
<tr>
<td>$10^{32}$</td>
<td>1.33</td>
<td>1.27</td>
<td>1.45</td>
<td>1.25</td>
<td>1.38</td>
</tr>
<tr>
<td>$10^{48}$</td>
<td>1.35</td>
<td>1.43</td>
<td>1.38</td>
<td>1.33</td>
<td>1.47</td>
</tr>
<tr>
<td>$10^{60}$</td>
<td>1.25</td>
<td>1.33</td>
<td>1.29</td>
<td>1.27</td>
<td>1.40</td>
</tr>
</tbody>
</table>
Instruction dependence analysis to compare two algorithms

**Ultimatly Fast Accurate Summation.** S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:
- $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]
Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation.* S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:
- 3n vs. 4n flop ($\times m$ outer iterations) [SISC,2009]
- but AccSum benefits for more ILP: PerPI outputs
Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation.* S.M. Rump. [SISC,2009]

- New `FastAccSum` is announced to be faster than `AccSum`:
  - $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]
- but `AccSum` benefits for more ILP: PerPI outputs
- Let’s exploit it!

[Graph showing performance comparison between `AccSumVect/FastAccSumUnrolled` for different data sizes and condition numbers.]
Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation.* S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:

- S.M. Rump is right!

6. **Timing.** In this section we briefly report on some timings. We do this with great hesitation: Measuring the computing time of summation algorithms in a high-level language on today’s architectures is more of a hazard than scientific research. The results are hardly predictable and often do not reflect the actual performance.
Accurate algorithms: why? how? which ones?

How to choose the fastest algorithm?

The PerPI Tool

The PerPI Tool: outputs and first examples

Conclusion
Conclusions

PerPI: a software platform to analyze and visualise ILP

- Useful: a detailed picture of the intrinsic behavior of the algorithm
- Reliable: reproducibility both in time and location
- Realistic: correlation with measured ones
- Exploratory tool: gives us the taste of the behavior of our algorithms within “tomorrow” processors
- Optimisation tool: analyse the effect of some hardware constraints

Cons . . . at the current state

- Work in progress
- Not abstract enough: instruction set dependence (RISC vs. CISC, 3-operand instructions, . . .
- Assembler program or high level programming language? IPC vs. FloPC ?
Current working list

- Improving the post-processing visualisation
- Make PerPI available on-line and usable as black-box