Performances de schémas d’évaluation polynomiale sur architectures vectorielles
8ème Rencontres Arithmétique de l’Informatique Mathématique
Banyuls

Hugues de Lassus Saint-Geniès et Guillaume Revy

DALI, Université de Perpignan Via Domitia
LIRMM, Université de Montpellier
CNRS UMR 5506

28 – 30 juin 2016
Motivations

Approximation of elementary functions
- Tedious, architecture-dependant and error-prone implementations
- Wide variety of software libraries: performances vs accuracy
- Often involves polynomial evaluations

Goals of the MetaLibm ANR project:
- Automatize code generation for mathematical functions and filters
- Get high performances given a target accuracy and architecture

Our goals:
- Generate vectorizable code for polynomial evaluation
- Use features of the target architectures to improve performances
In this talk:

- How do different polynomial schemes behave on **SIMD FP units**?
- Study of the performance of **classic** and **latency-minimal schemes**
- **Native precision** computations
- No adaptation of coefficients (Knuth & Eve, Paterson & Stockmeyer)
- Impact on **elementary function evaluation** (a vectorized logarithm)

Some conclusions:

- The classic Horner’s rule is quickly **not the best choice** for efficiency.
- **Denormalization** can be fatal to performance.
- **Faster**, more **parallel schemes** can have enough accuracy for **faithfully-rounded** functions.
Outline

1. Reminder on polynomial evaluation schemes
2. Studying the vectorization of polynomial evaluations
3. Impacts on function approximation
4. Conclusions and perspectives
Outline

1. Reminder on polynomial evaluation schemes
2. Studying the vectorization of polynomial evaluations
3. Impacts on function approximation
4. Conclusions and perspectives
Reminder on polynomial evaluation schemes

The most usual polynomial schemes:

- Horner’s rule
- 2nd-order Horner’s rule
- Estrin’s scheme
Latencies: Horner’s rule

Latency = \# cycles required for a complete evaluation
= \# cycles for the critical path on infinite parallelism

Evolution of the latency with Haswell costs for MUL, ADD and FMA

\[ n \text{ ADD} + n \text{ MUL} \]
Latencies: Horner-2

\[ n \text{ ADD} + n + 1 \text{ MUL} \]

Evolution of the latency with Haswell costs for MUL, ADD and FMA
Latencies: Estrin’s scheme

Evolution of the latency with Haswell costs for MUL, ADD and FMA
On pipelined architectures, passed a certain degree, latencies can be lowered with more parallel schemes.
On pipelined architectures, passed a certain degree, latencies can be lowered with more parallel schemes.

How is throughput impacted?

- As instruction parallelism increases, so does the pressure on registers.
- A common “rule of thumb” is: parallel schemes have better latencies but worse throughputs than sequential schemes.
Outline

1. Reminder on polynomial evaluation schemes
2. Studying the vectorization of polynomial evaluations
3. Impacts on function approximation
4. Conclusions and perspectives
Studying the vectorization of polynomial evaluations

Objectives:
- Be able to choose fast polynomial evaluation schemes
- Benefit from vector extensions of our ISA (SWAR)

Protocol:
- Benchmarks of Horner, 2nd-order Horner and Estrin schemes
- Polynomials of degrees ranging from 3 to 32
- Using Taylor(log, 1) for the coefficients
- ISA featuring AVX2, with or without FMA (Haswell, ~Ivy Bridge)
- Single / double precision
- Compiler: GCC 5.2.0
Scalar vs. auto-vectorized code: First observations
Scalar vs. auto-vectorized code: First observations

![Graph 1: Horner vs. vectorized Horner vs. Horner2 vs. vectorized Horner2 vs. Estrin vs. vectorized Estrin (no FMA)]

- Plot shows performance measurements for different polynomial evaluation schemes on vector architectures.
- Comparisons include Horner, vectorized Horner, Horner2, vectorized Horner2, Estrin, and vectorized Estrin.
- Y-axis: Rcpp throughput (no FMA) vs. degree
- X-axis: Degree of the polynomial

![Graph 2: Horner vs. vectorized Horner vs. Horner2 vs. vectorized Horner2 vs. Estrin vs. vectorized Estrin (FMA)]

- Similar setup to Graph 1, but with FMA included.
- Performance metrics updated accordingly.
Impact of denormalization (1)

- Do arithmetic instructions always have the same latency, *regardless of the inputs*?
- For instance, what happens when $x^n$ or a partial result is *denormal*?

Experimental protocol:

- $2^{10}$ samples $x$ between $2^{-12}$ and $2^{-5}$
- Measure the **latency to compute** $x^{16}$ using a naive and a binary approach:
  - naive: $x \otimes x \otimes \cdots \otimes x$
  - binary: $((x \otimes x) \otimes (x \otimes x)) \otimes \ldots$
Impact of denormalization (2)

**120-150 cycles penalty** for MULSS normal, normal → denormal

→ Confirmed by Agner Fog’s optimization manuals
Throughputs with GCC 5.2 for inputs in $[1, 2]$
What do these graphs say?

The obvious:
- FMA is a $\approx 2x$-gain;
- Parallel schemes become more efficient as $n$ increases

The not-so-obvious:
- Denormalization can introduce huge overheads;
- The sixteen 256-bit AVX registers seem to resist well the pressure of moderately high degrees;

Are there even more parallel, more efficient schemes? $\rightarrow$ use CGPE$^1$

---

$^1$http://cgpe.gforge.inria.fr/
Comparisons with other schemes (1)

All schemes of minimal latency (for Haswell without FMA) for the degree 12, compared with classic schemes

... when no denormalization occurs.
Comparisons with other schemes (2)

When $x^{12}$ is a denormal number...

![Graph comparing different schemes]

H. de Lassus Saint-Geniès
Performances de schémas d'évaluation polynomiale sur architectures vectorielles
And when $x^8$ is a denormal number.
Outline

1. Reminder on polynomial evaluation schemes
2. Studying the vectorization of polynomial evaluations
3. Impacts on function approximation
4. Conclusions and perspectives
Implementation of a **faithful vectorized single-precision natural logarithm** for AVX2 architectures, using Intel Intrinsics Instructions:

- Logarithm approximation is quite simple
- Single-precision allows **native precision** computations for faithfulness
- Intrinsics enable **easy access to assembly instructions** and are supported by GCC.

Real world application: high-energy collision of two protons

- log was the 2nd **most-called** function in a Cern’s CMS simulation\(^2\) (\(> 7\) M calls in \(> 4300\) traces)
- Average of 1600 calls/trace

\(^2\)Piparo & Innocente 2016
Implementation details:

- **Evaluates 4 logarithms** at a time in 4 steps:
  1. Special treatment for denormals and inputs $> 2^{126}$
     - $x' = x \cdot 2^{23}$ or $x' = x \cdot 2^{-2}$
     - Then, $\ln(x) = \ln(x') + (-23||2) \cdot \ln 2$
  2. Range reduction using **RCPPS**: $u = x \cdot \circ (\frac{1}{x}) - 1$
  3. Table-based reduction, using **GATHER**: $\log_2(\circ (\frac{1}{x}))$
  4. Polynomial evaluation and reconstruction:

\[
\ln x \approx - \ln 2 \cdot \left( \log_2 \left( \circ \left( \frac{1}{x} \right) \right) + (23||0) - (2||0) \right) + \ln(1 + u)_{\text{poly}}
\]

- **Degree-5 fp-minimax polynomial** beginning with $0 + x + \ldots$
- Used **Intrinsics backend** within CGPE, without FMA (FMA optimization handled by compiler)
Results (1): Throughput and size of logf_4
(throughput in cycles/scalar result):

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Implementation</th>
<th>Interval</th>
<th>Throughput</th>
<th>Code + Data (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC 16.0.1</td>
<td>Intel SVML</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>11</td>
<td>2200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>GCC 6.1.0</td>
<td>GNU Libmvec</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>N/A</td>
<td>522</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CGPE deg-5</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>43</td>
<td>1282</td>
</tr>
<tr>
<td></td>
<td>Horner</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>44</td>
<td>1266</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Horner-2</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>42</td>
<td>1274</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Estrin</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>41</td>
<td>1270</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.4</td>
<td></td>
</tr>
</tbody>
</table>

Caveats:
- Sizes for Intel SVML and GNU Libmvec only for main routines
- GNU Libmvec needs -ffast-math $\Rightarrow$ no denormals, no exceptions, ...
Results (1): Throughput and size of logf_4
(throughput in cycles/scalar result):

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Implementation</th>
<th>Interval</th>
<th>Throughput</th>
<th>Code + Data (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC 16.0.1</td>
<td>Intel SVML</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>11</td>
<td>2200</td>
</tr>
<tr>
<td></td>
<td>GNU Libmvec</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>N/A</td>
<td>522</td>
</tr>
<tr>
<td>GCC 6.1.0</td>
<td>CGPE deg-5</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>43</td>
<td>1282</td>
</tr>
<tr>
<td></td>
<td>Horner</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>44</td>
<td>1266</td>
</tr>
<tr>
<td></td>
<td>Horner-2</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>42</td>
<td>1274</td>
</tr>
<tr>
<td></td>
<td>Estrin</td>
<td>$[2^{-149}, 2^{-126}]$ $[2^{-12}, 2^{12}]$</td>
<td>41</td>
<td>1270</td>
</tr>
</tbody>
</table>

Caveats:
- Sizes for Intel SVML and GNU Libmvec only for main routines
- GNU Libmvec needs `-ffast-math` $\rightarrow$ no denormals, no exceptions, ...

Conclusions
- **Low impact** of the polynomial scheme at degree 5 for our implementation
- **High cost** of `mulps` for renormalization $\rightarrow$ integer ops should be used instead
## Results (2) : Accuracy of $\text{logf}_4$

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Interval</th>
<th>Max. error (ulps)</th>
<th>Incorrect rounding ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel SVML</td>
<td>$[2^{-149}, 2^{128} - 2^{104}]$</td>
<td>1</td>
<td>24%</td>
</tr>
<tr>
<td>GNU Libmvec</td>
<td>$[0.5, 1]$</td>
<td>2.25</td>
<td>67%</td>
</tr>
<tr>
<td>CGPE deg-5</td>
<td></td>
<td>0.997</td>
<td>$2.8 \times 10^{-9}$</td>
</tr>
<tr>
<td>Horner</td>
<td>$[2^{-149}, 2^{128} - 2^{104}]$</td>
<td>0.997</td>
<td>$2.8 \times 10^{-9}$</td>
</tr>
<tr>
<td>Horner-2</td>
<td></td>
<td>0.997</td>
<td>$2.8 \times 10^{-9}$</td>
</tr>
<tr>
<td>Estrin</td>
<td></td>
<td>0.997</td>
<td>$2.8 \times 10^{-9}$</td>
</tr>
</tbody>
</table>

**Caveats:**

- Intel SVML and GNU Libmvec use the **I/O precision** for intermediate computations
Results (2) : Accuracy of \( \logf_4 \)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Interval</th>
<th>Max. error (ulps)</th>
<th>Incorrect rounding ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel SVML</td>
<td>([2^{-149}, 2^{128} - 2^{104}])</td>
<td>1</td>
<td>24%</td>
</tr>
<tr>
<td>GNU Libmvec</td>
<td>([0.5, 1])</td>
<td>2.25</td>
<td>67%</td>
</tr>
<tr>
<td>CGPE deg-5</td>
<td>([2^{-149}, 2^{128} - 2^{104}])</td>
<td>0.997</td>
<td>(2.8 \times 10^{-9})</td>
</tr>
<tr>
<td>Horner</td>
<td>([2^{-149}, 2^{128} - 2^{104}])</td>
<td>0.997</td>
<td>(2.8 \times 10^{-9})</td>
</tr>
<tr>
<td>Horner-2</td>
<td></td>
<td>0.997</td>
<td>(2.8 \times 10^{-9})</td>
</tr>
<tr>
<td>Estrin</td>
<td></td>
<td>0.997</td>
<td>(2.8 \times 10^{-9})</td>
</tr>
</tbody>
</table>

Caveats:
- Intel SVML and GNU Libmvec use the I/O precision for intermediate computations

Conclusions:
- Our \( \logf_4 \) is almost always correctly-rounded (6 results are “only” faithful)
1. Reminder on polynomial evaluation schemes

2. Studying the vectorization of polynomial evaluations

3. Impacts on function approximation

4. Conclusions and perspectives
Conclusions

Automatic generation of polynomial evaluations should take into account

- **low-level, architecture-dependant details**: SIMD/SWAR, FMA available...
- the degree of the polynomial to be implemented
- if input domain and evaluation scheme may introduce **denormalization** (on specific architectures)
- the **compiler** (for vector code and FMA)
Perspectives

- Handle FMA in CGPE directly: ≠ latencies, maybe ≠ low-latency schemes

- **Automate the choice** of efficient schemes with respect to:
  - Floating-point arithmetic (binary32, binary64)
  - (μ-)**Architecture** (extensions, arithmetic units, registers, caches, . . .)
  - Polynomial degree
  - I/O precision/accuracy
  - Maybe more?

- Observe how **extended-precision algorithms** perform
Performances de schémas d’évaluation polynomiale sur architectures vectorielles
8ème Rencontres Arithmétique de l’Informatique Mathématique
Banyuls

Hugues de Lassus Saint-Geniès et Guillaume Revy

DALI, Université de Perpignan Via Domitia
LIRMM, Université de Montpellier
CNRS UMR 5506

28 – 30 juin 2016