Fourth REPRISES F2F meeting
Amphi Charpak / Jussiu, Paris, April 8, 2022

Analyzing the impact of floating-point precision adaptation in iterative programs

Talk presented at ARITH 2021

Guillaume Revy

Univ Perpignan Via Domitia, DALI, Perpignan, France
LIRMM, Univ Montpellier, CNRS (UMR 5506), Montpellier, France

Context and achievement

Context

- Various floating-point formats exist = different level of accuracy
  - IEEE 754-2019 defines four formats: binary{16, 32, 64, 128}
  - non IEEE formats: BFloat16, Posit, ...
- Floating-point arithmetic is non-intuitive
  - discrete and finite set of values $\rightarrow$ 0.1 not exactly representable
  - loss of arithmetic properties $\rightarrow a+(b+c) \neq (a+b)+c$
- Over-sizing of the computation means $\rightarrow$ binary64 by default
- Precision tuning: technique to improve performance of numerical applications
  - evaluate the impact of modifying the format of certain data

Achievement : a dynamic tool to evaluate the impact of adapting the format of floating-point data in iterative programs

1. instrument programs with multiple-precision computations
2. split the iteration space of loops into several reduced subspaces
3. update the precision of some multiple-precision computations

Motivating example (1/2)

- Approximation of $1/2$ using the Newton-Raphson method

$$u_{i+1} = u_i \cdot (2 - 2 \cdot u_i), \quad u_0 = 0.05$$

```
double ui = .05, tmp1, tmp2;
for(int i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
}
```

(binary64)

Motivating example (1/2)

- Approximation of $1/2$ using the Newton-Raphson method

$$u_{i+1} = u_i \cdot (2 - 2 \cdot u_i), \quad u_0 = 0.05$$

```
double ui = .05, tmp1, tmp2;
for(int i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
}
```

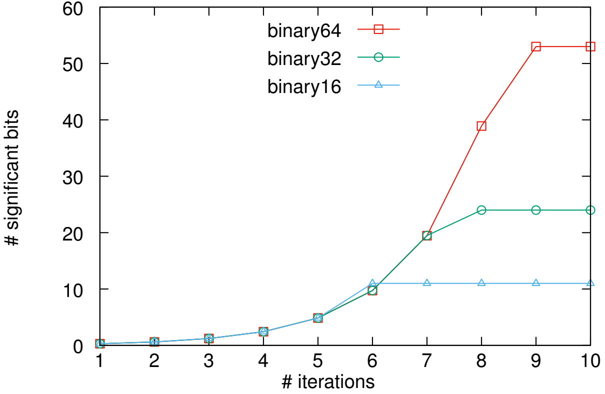
(binary64)

```
tmp1 = 2. * ui;
tmp2 = 2. - tmp1;
ui = ui * tmp2;
```

(binary16)

i	u_i (binary64)	# significant bits	u_i (binary16)	# significant bits
0	0.09500000000000001	0.30	0.09497070312500000	0.30
1	0.17195000000000020	0.61	0.17199707031250000	0.61
2	0.28476639500000010	1.22	0.28491210937500000	1.22
3	0.40734899055740798	2.43	0.40722656250000000	2.43
4	0.48283158089853750	4.86	0.48266601562500000	4.85
5	0.49941049077111310	9.73	0.49975585937500000	11.00
6	0.49999930495773809	19.46	0.49975585937500000	11.00
7	0.49999999999903388	38.91	0.49975585937500000	11.00
8	0.50000000000000000	53.00	0.49975585937500000	11.00

Motivating example (2/2)



How to decide the computation format at each iteration?

Outline of the talk

1. Background on LLVM and MPFR

2. Tool to analyze the impact of adapting data formats

3. Experimental results

4. Concluding remarks

Outline of the talk

1. Background on LLVM and MPFR

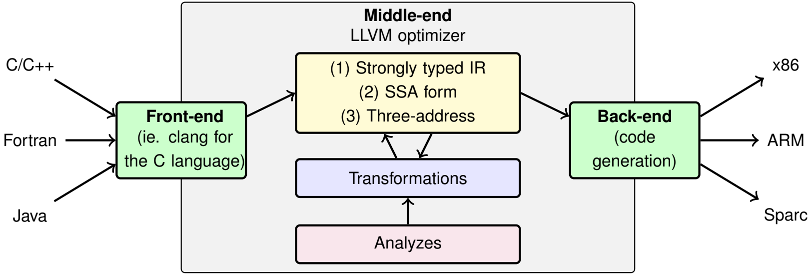
2. Tool to analyze the impact of adapting data formats

3. Experimental results

4. Concluding remarks

LLVM infrastructure

- LLVM = compiler infrastructure and framework



- LLVM optimizer = series of "passes"
  - analysis and optimization passes, run one by one
- LLVM intermediate form = Virtual Instruction Set
  - language- and target-independent form = same passes for all languages and targets

Floating-point arithmetic with MPFR

- Floating-point arithmetic approximates real numbers
- IEEE-754 floating-point number x is represented by a triplet (s, e, m)

$$x = (-1)^s \cdot 2^e \cdot m_0.m_1 \cdots m_{p-1}$$

  - format = exponent range $[e_{\min}, e_{\max}]$ + precision $p \rightarrow$ defined by IEEE standard

- MPFR = library for multiple-precision floating-point computations
  1. a precision p is attached to each MPFR variable
     $\rightarrow$ emulates (non-)standard arithmetic
  2. MPFR functions are of the form `mpfr_op(dst, src1, src2, rnd)`
     $\rightarrow$ fits the 3-address form of LLVM IR

`%c = fadd double %a, %b` $\Rightarrow$ `mpfr_add(c, a, b, MPFR_RNDN)`

Outline of the talk

1. Background on LLVM and MPFR

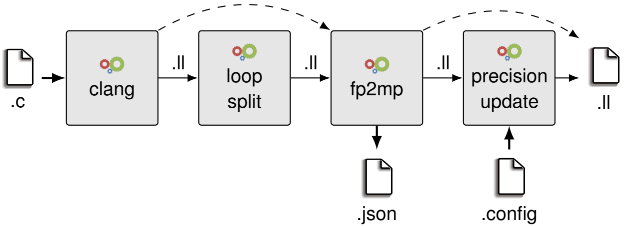
2. Tool to analyze the impact of adapting data formats

3. Experimental results

4. Concluding remarks

Analysis tool workflow

- Tool implemented as a pass in LLVM 10.0.0
- It works on the LLVM IR of a program compiled with the lowest optimization level



- loop split = split the iteration space of loops into several reduced subspaces
- fp2mp = instrument program with multiple-precision computations
- precision update = update the precision of some multiple-precision computations

Back to motivating example

(original)

```
  double ui = .05, tmp1, tmp2;
  int i;

#pragma clang loop split_ratio(25)
  for(i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
  }
```

Back to motivating example

(original)

```
  double ui = .05, tmp1, tmp2;
  int i;

#pragma clang loop split_ratio(25)
  for(i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
  }
```

(instrumented)

```
double ui = .05, tmp1, tmp2;
int i, bnd = floor(9 * 25 / 100);

for(i = 0; i <= bnd; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  ui = ui * tmp2;
}

for(; i < 9; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  ui = ui * tmp2;
}
```

Back to motivating example

(original)

```
  double ui = .05, tmp1, tmp2;
  int i;

#pragma clang loop split_ratio(25)
  for(i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
  }
```

(instrumented)

```
double ui = .05, tmp1, tmp2;
int i, bnd = floor(9 * 25 / 100);

mpfr_t Ui, M1, M2, Tmp1;
mpfr_inits2(53, Ui, Tmp1, M1, M2);

mpfr_set_d(Ui, .05, MPFR_RNDN);

for(i = 0; i <= bnd; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  // ...
  mpfr_mul(M1, Ui, Tmp2, MPFR_RNDN);
  mpfr_set(Ui, M1, MPFR_RNDN);
  ui = ui * tmp2;
}

for(; i < 9; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  // ...
  mpfr_mul(M2, Ui, Tmp2, MPFR_RNDN);
  mpfr_set(Ui, M2, MPFR_RNDN);
  ui = ui * tmp2;
}

mpfr_clears(Ui, Tmp1, M1, M2);
```

Back to motivating example

(original)

```
  double ui = .05, tmp1, tmp2;
  int i;

#pragma clang loop split_ratio(25)
  for(i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
  }
```

(instrumented)

```
double ui = .05, tmp1, tmp2;
int i, bnd = floor(9 * 25 / 100);

mpfr_t Ui, M1, M2, Tmp1;
mpfr_inits2(53, Ui, Tmp1, M2);
mpfr_t C1, C2;
mpfr_inits2(11, M1, C1, C2);

mpfr_set_d(Ui, .05, MPFR_RNDN);

for(i = 0; i <= bnd; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  // ...
  mpfr_set(C1, Ui, MPFR_RNDN);
  mpfr_set(C2, Tmp2, MPFR_RNDN);
  mpfr_mul(M1, C1, C2, MPFR_RNDN);
  mpfr_set(Ui, M1, MPFR_RNDN);
  ui = ui * tmp2;
}

for(; i < 9; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  // ...
  mpfr_mul(M1, Ui, Tmp2, MPFR_RNDN);
  mpfr_set(Ui, M1, MPFR_RNDN);
  ui = ui * tmp2;
}

mpfr_clears(Ui, Tmp1, M1, M2, C1, C2);
```

Back to motivating example

(original)

```
  double ui = .05, tmp1, tmp2;
  int i;

#pragma clang loop split_ratio(25)
  for(i = 0; i < 9; i++) {
    tmp1 = 2. * ui;
    tmp2 = 2. - tmp1;
    ui = ui * tmp2;
  }
```

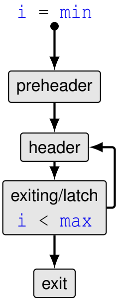
$$\text{rel\_error} \rightarrow \left| \frac{Ui - ui}{ui} \right|$$

(instrumented)

```
double ui = .05, tmp1, tmp2;
int i, bnd = floor(9 * 25 / 100);

mpfr_t Ui, M1, M2, Tmp1;
mpfr_inits2(53, Ui, Tmp1, M2);
mpfr_t C1, C2;
mpfr_inits2(11, M1, C1, C2);

mpfr_set_d(Ui, .05, MPFR_RNDN);

for(i = 0; i <= bnd; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  // ...
  mpfr_set(C1, Ui, MPFR_RNDN);
  mpfr_set(C2, Tmp2, MPFR_RNDN);
  mpfr_mul(M1, C1, C2, MPFR_RNDN);
  mpfr_set(Ui, M1, MPFR_RNDN);
  ui = ui * tmp2;
}

for(; i < 9; i++) {
  tmp1 = 2. * ui;
  tmp2 = 2. - tmp1;
  // ...
  mpfr_mul(M1, Ui, Tmp2, MPFR_RNDN);
  mpfr_set(Ui, M1, MPFR_RNDN);
  ui = ui * tmp2;
}
printf("error= %Le\n", rel_error(ui));
mpfr_clears(Ui, Tmp1, M1, M2, C1, C2);
```

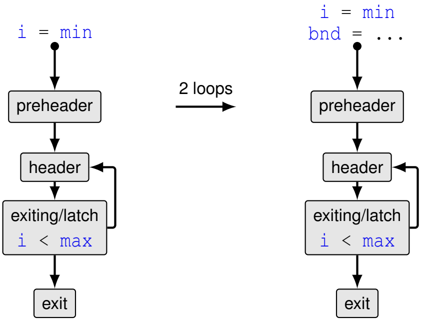
Loop splitting strategy

- In LLVM IR, a loop is represented as a control flow graph
- In the canonical form, a loop is as follows



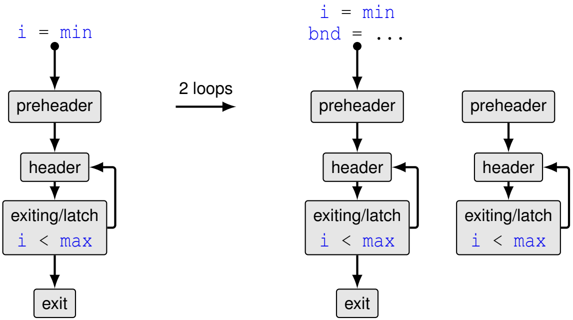
Loop splitting strategy

- In LLVM IR, a loop is represented as a control flow graph
- In the canonical form, a loop is as follows



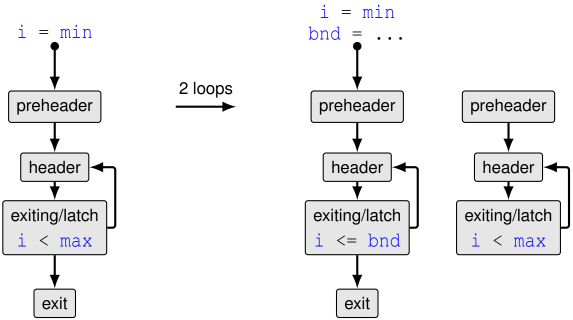
Loop splitting strategy

- In LLVM IR, a loop is represented as a control flow graph
- In the canonical form, a loop is as follows



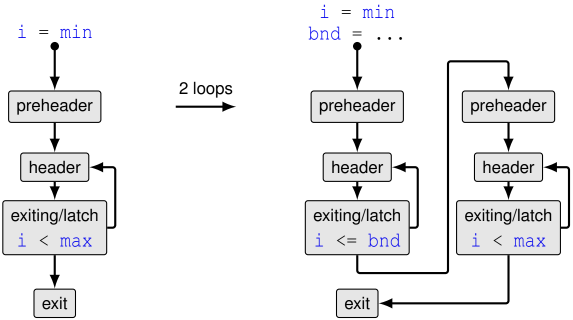
Loop splitting strategy

- In LLVM IR, a loop is represented as a control flow graph
- In the canonical form, a loop is as follows



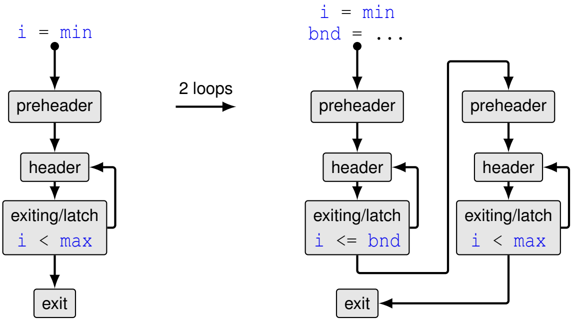
Loop splitting strategy

- In LLVM IR, a loop is represented as a control flow graph
- In the canonical form, a loop is as follows



Loop splitting strategy

- In LLVM IR, a loop is represented as a control flow graph
- In the canonical form, a loop is as follows



- Loops can be split in more than 2 loops
- Loop bounds (i.e. min and max) are not computable in all cases
  - insert counter to count first loop iteration numbers

Outline of the talk

1. Background on LLVM and MPFR
2. Tool to analyze the impact of adapting data formats
3. Experimental results
4. Concluding remarks

Case of bounded loop (1/2)

- Polynomial evaluation using Horner rule
  - number of iterations = degree of polynomial

```
double
evaluate(double *a, int n, double x)
{
  double res = a[n];
#pragma clang loop split_ratio(RATIO)
  for(int i = n-1; i >= 0; i--)
    res = res * x + a[i];
  return res;
}
```

Function	Degree	Interval
$\log_2(1+x)$	31	$[-2^{-2};2^{-2}]$
$\exp(x)$	26	$[-2^{-1};2^{-1}]$
$\sin(x)$	28	$[-\pi/4;\pi/4]$
$\sinh(x)$	30	$[-1;1]$
$\mathrm{erf}(x)-1/2$	32	$[-1/4;1/4]$

Case of bounded loop (1/2)

- Polynomial evaluation using Horner rule
  - number of iterations = degree of polynomial

```
double
evaluate(double *a, int n, double x)
{
  double res = a[n];
#pragma clang loop split_ratio(RATIO)
  for(int i = n-1; i >= 0; i--)
    res = res * x + a[i];
  return res;
}
```

Function	Degree	Interval
$\log_2(1+x)$	31	$[-2^{-2};2^{-2}]$
$\exp(x)$	26	$[-2^{-1};2^{-1}]$
$\sin(x)$	28	$[-\pi/4;\pi/4]$
$\sinh(x)$	30	$[-1;1]$
$\mathrm{erf}(x)-1/2$	32	$[-1/4;1/4]$

- For each RATIO $\in \{0,5,10,\cdots,95,100\}$
  - split the loop into two subloops
  - evaluate the impact of modifying the format of the first subloop

How do evolve the error according to the splitting ratio?

Case of bounded loop (2/2)

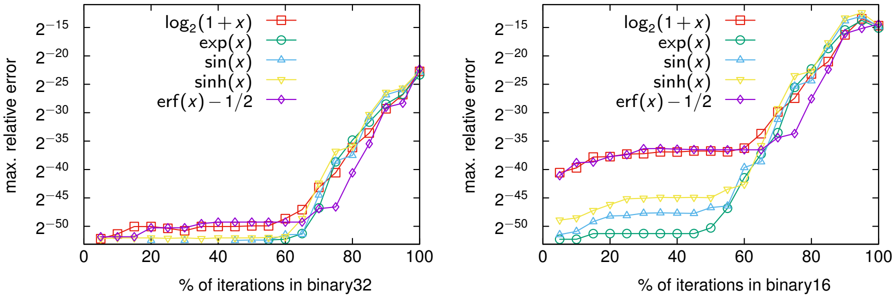


Figure: Maximum relative error according to the percentage of iterations in binary32 or binary16.

Case of unbounded loop (1/2)

- Conjugate Gradient: method to solve the linear system $Ax = b$

$r_0 := p_0 := b - Ax_0$, and $k = 0$
while $\|r_k\| \geq \varepsilon$ and $k <$ maxiter **do**
$\quad \alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
$\quad x_{k+1} := x_k + \alpha_k p_k$
$\quad r_{k+1} := r_k - \alpha_k A p_k$
$\quad \beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
$\quad p_{k+1} := r_{k+1} + \beta_k p_k$
$\quad k = k + 1$
end while

- In exact arithmetic, it converges in n iterations
- But in floating-point arithmetic, the number of iterations is linked to the precision of the computations
- Example: 494_bus matrix (Suite Sparse Matrix Collection)
  - $\varepsilon = 10^{-6}$
  - binary64 = 1315 iterations
  - binary32 = 2494 iterations

How do evolve the number of iterations
when the precision in first subloop is lowered to binary32?

Case of unbounded loop (2/2)

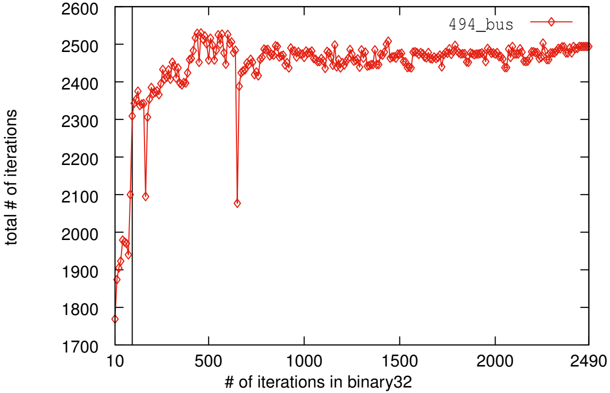


Figure: Total number of iterations according to the number of iterations in binary32, for the conjugate gradient method on the 494_bus matrix of the Suite Sparse Matrix Collection.

Outline of the talk

1. Background on LLVM and MPFR

2. Tool to analyze the impact of adapting data formats

3. Experimental results

4. Concluding remarks

Concluding remarks

Contributions

- Tool to analyze the impact of modifying the format of certain data in iterative programs
  - instrument LLVM IR with MPFR computations
  - split loops to be able to modify the computation precision at certain iterations only
- Current version is an automatic tool to analyze small programs

Future works

- Validate this tool on larger real life applications
- Extend this tool to evaluate the gain of performance of data format modification
- Integrate this tool into a framework for precision tuning