30th IEEE Symposium on Computer Arithmetic, ARITH 2023
Portland, OR, USA, September 4-6 2023

Using loop transformations for precision tuning in iterative programs

Youssef Fakhreddine and 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 → 0.1 not exactly representable
  - loss of arithmetic properties → $a+(b+c) \neq (a+b)+c$
- Over-sizing of the computation means → higher precision by default
- Precision tuning: technique to improve performance of numerical applications

⚠ Most existing tools do not consider iterative nature of programs ⚠

Achievement : a dynamic auto-tuning tool, targeting iterative routines

- reduce the precision of certain instructions at the iteration level,
- to the detriment of an increase of the time of tuning process.

Motivating example (1/2)

- Objective : calculate the sum $\sum_{i=1}^{1000} 0.01 = 10$

```
double s_b64 = 0.;

for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;

printf("s_b64 = %.20lf", s_b64);
```

```
s_b64 = 9.99999999999983124610
```

Motivating example (1/2)

- Objective : calculate the sum $\sum_{i=1}^{1000} 0.01 = 10$

```c
double s_b64 = 0.;

for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;

printf("s_b64 = %.20lf", s_b64);
```

```c
float s_b32 = 0.f;

for (int i=1; i<=1000; i++)
  s_b32 = s_b32 + 0.01f;

printf("s_b32 = %.20f", s_b32);
```

```
s_b64 = 9.99999999999983124610
```

```
s_b32 = 10.00013351440429687500
```

$$\frac{|\texttt{s\_b64}-\texttt{s\_b32}|}{\texttt{s\_b64}} \approx 10^{-5}$$

Motivating example (1/2)

- Objective : calculate the sum $\sum_{i=1}^{1000} 0.01 = 10$

```c
double s_b64 = 0.;

for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;

printf("s_b64 = %.20lf", s_b64);
```

```c
float s_b32 = 0.f;

for (int i=1; i<=1000; i++)
  s_b32 = s_b32 + 0.01f;

printf("s_b32 = %.20f", s_b32);
```

```
s_b64 = 9.99999999999983124610
```

```
s_b32 = 10.00013351440429687500
```

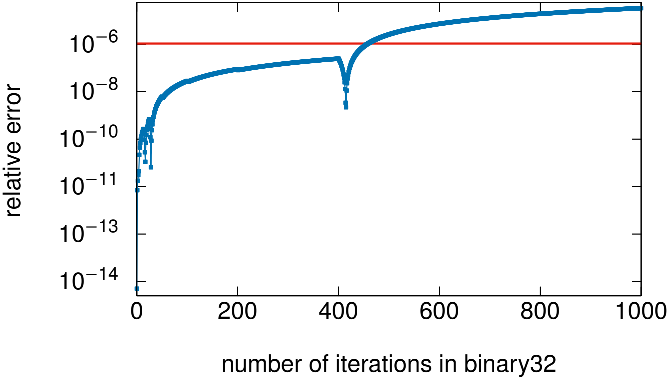
$$\frac{|\mathtt{s\_b64} - \mathtt{s\_b32}|}{\mathtt{s\_b64}} \approx 10^{-5}$$

- Only 1 instruction (addition): which format per iteration to have

$$\frac{|\mathtt{s\_b64} - \mathtt{s\_optim}|}{\mathtt{s\_b64}} < 10^{-6}?$$

☹ Existing approaches → no solution

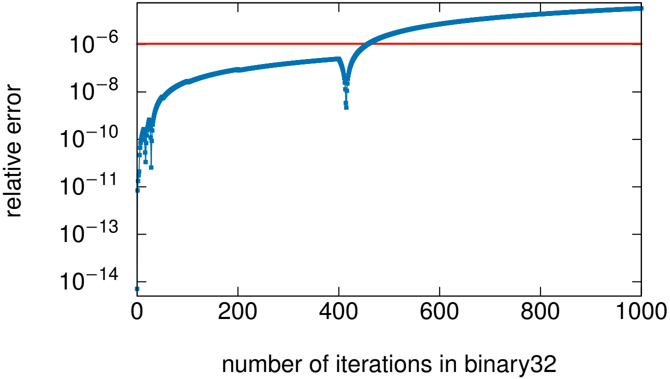
Motivating example (2/2)



- Target threshold = 10^{-6} → 458 iterations in binary32

☺ Our approach → 450 iterations in binary32

Motivating example (2/2)



- Target threshold = 10^{-6} → 458 iterations in binary32

☺ Our approach → 450 iterations in binary32

How to isolate these iterations?

Outline of the talk

1. Auto-tuning of iterative routines

2. Experimental results

3. Conclusion and perspectives

Outline of the talk

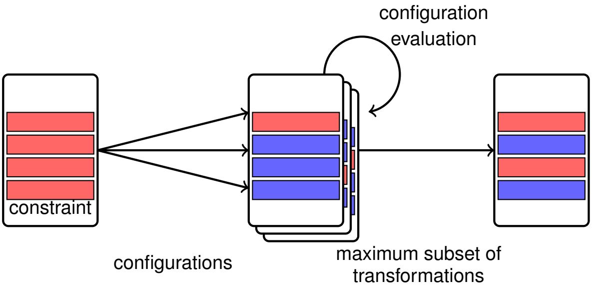
1. Auto-tuning of iterative routines

2. Experimental results

3. Conclusion and perspectives

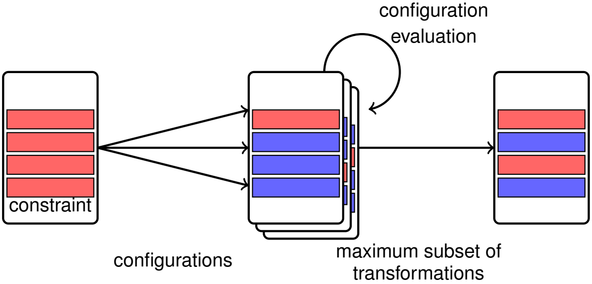
Main flow of dynamic tools

- Most dynamic tools use a trial-and-error strategy
    1. explore a set of possible transformations (configurations)
    2. evaluate the impact of each transformation (eg. accuracy)



Main flow of dynamic tools

- Most dynamic tools use a trial-and-error strategy
  1. explore a set of possible transformations (configurations)
  2. evaluate the impact of each transformation (eg. accuracy)



How to adapt this process to the tuning of iterative programs?

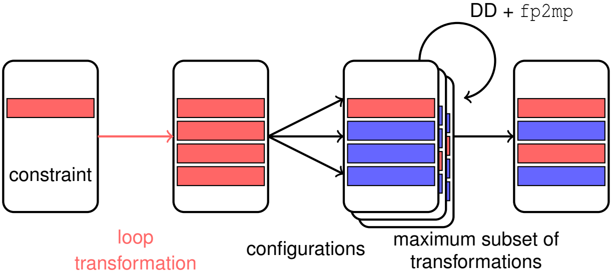
Outline of our project

Originality of the proposed approach

- change combinatorics by targeting instructions in loop bodies
- use compilation techniques on loop: loop splitting and unrolling

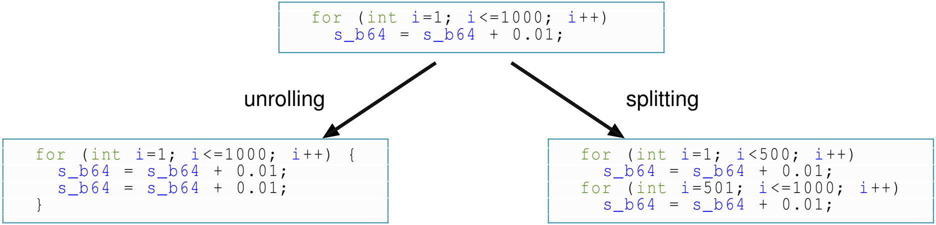
Main steps

- loop transformation (splitting, unrolling)
- configuration evaluation → fp2mp
- building of maximum subset of transformations → delta-debugging



Static loop transformation

- Objective: increase the number of possible transformations
  - leverage the LLVM capabilities of transforming programs



```
for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;
```

unrolling

```
for (int i=1; i<=1000; i++) {
  s_b64 = s_b64 + 0.01;
  s_b64 = s_b64 + 0.01;
}
```

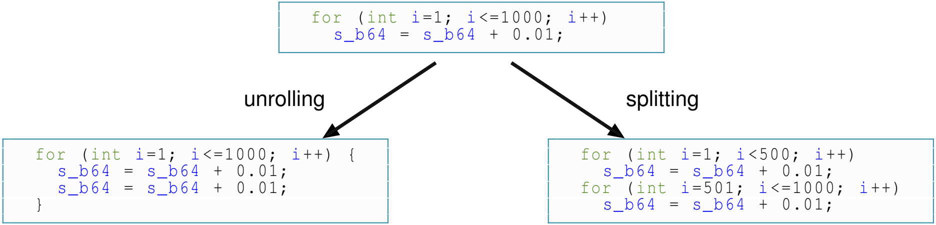
splitting

```
for (int i=1; i<500; i++)
  s_b64 = s_b64 + 0.01;
for (int i=501; i<=1000; i++)
  s_b64 = s_b64 + 0.01;
```

  - do not modify the semantics of the program
  - allow to detect two different patterns of transformations

Static loop transformation

- Objective: increase the number of possible transformations
  - leverage the LLVM capabilities of transforming programs



```
for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;
```

unrolling

```
for (int i=1; i<=1000; i++) {
  s_b64 = s_b64 + 0.01;
  s_b64 = s_b64 + 0.01;
}
```

splitting

```
for (int i=1; i<500; i++)
  s_b64 = s_b64 + 0.01;
for (int i=501; i<=1000; i++)
  s_b64 = s_b64 + 0.01;
```

  - do not modify the semantics of the program
  - allow to detect two different patterns of transformations

⚠ Approach antagonistic to existing ones

  - current trend: reduce the combinatorics to speedup the process
  - our approach: increase the combinatorics → ☹ increase the tuning process time
    ☺ improve the quality of the tuning

Evaluate the impact of transformations

- Objective: check if the constraint is still satisfied
- Rely on fp2mp: LLVM instrumentation tool
  - duplicate floating-point instructions into their MPFR equivalent instructions
  - and allow to compute the result using a desired precision

```
double s_b64 = 0.;

for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;
printf("s_b64 = %.20lf", s_b64);

// |s_b64 - s_mpfr|/|s_b64| < 1e-6 ?
check_reverse_rel_error(s_b64, 1e-6);
```

```
double s_b64 = 0.;
// ...
mpfr_t s_mpfr, C, S;
mpfr_init2(s_mpfr, 53);
mpfr_init2(C, 53);
mpfr_init2(S, 53);
mpfr_set_d(C, 0.01, MPFR_RNDN);

for (int i=1; i<=1000; i++) {
  s_b64 = s_b64 + 0.01;
  // ...
  mpfr_set(S, s_mpfr, MPFR_RNDN);
  mpfr_add(s_mpfr, S, C, MPFR_RNDN);
}
printf("s_b64 = %.20lf", s_b64);

// |s_b64 - s_mpfr|/|s_b64| < 1e-6 ?
check_reverse_rel_error(s_b64, s_mpfr,
                                  1e-6);
mpfr_clears(s_mpfr, C, S, NULL);
```

Evaluate the impact of transformations

- Objective: check if the constraint is still satisfied
- Rely on fp2mp: LLVM instrumentation tool
  - duplicate floating-point instructions into their MPFR equivalent instructions
  - and allow to compute the result using a desired precision

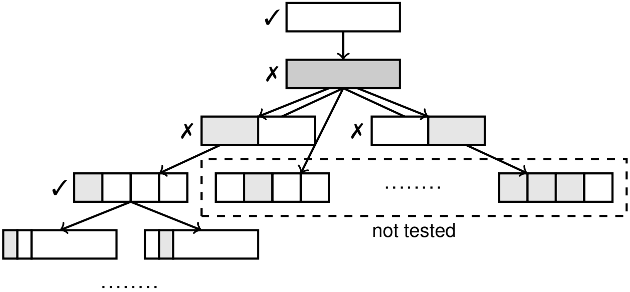
```
double s_b64 = 0.;

for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;
printf("s_b64 = %.20lf", s_b64);

// |s_b64 - s_mpfr|/|s_b64| < 1e-6 ?
check_reverse_rel_error(s_b64, 1e-6);
```

```
double s_b64 = 0.;
// ...
mpfr_t s_mpfr, C, S;
mpfr_init2(s_mpfr, 53);
mpfr_init2(C, 53);
mpfr_init2(S, 53);
mpfr_set_d(C, 0.01, MPFR_RNDN);

for (int i=1; i<=1000; i++) {
  s_b64 = s_b64 + 0.01;
  // ...
  mpfr_set(S, s_mpfr, MPFR_RNDN);
  mpfr_add(s_mpfr, S, C, MPFR_RNDN);
}
printf("s_b64 = %.20lf", s_b64);

// |s_b64 - s_mpfr|/|s_b64| < 1e-6 ?
check_reverse_rel_error(s_b64, s_mpfr,
                                   1e-6);
mpfr_clears(s_mpfr, C, S, NULL);
```

- Interest
  1. Apply transformations = modify MPFR initialisation precision
  2. Provide means to estimate errors due to transformations

Delta-Debugging algorithm

- Objective: isolate most relevant transformations
  - widely used in auto-tuning tools
  - `ddmax`: find a locally maximal set of changes → the contraint remains satisfied



- For each instruction → a list of possible precision (e.g. [`b32`, `b16`])
  - apply delta-debugging several times
  - find the lowest precision for each instruction

Back to motivating example

```
double s_b64 = 0.;

for (int i=1; i<=1000; i++)
  s_b64 = s_b64 + 0.01;

check_reverse_rel_error(s_b64, 1e-6);
```

- Example of our approach:
  - split iteration space $[1,1000]$ into 20 subspaces of 50 iterations

```
double s_optim = 0.;

for (int i=1; i<=50; i++)
  s_optim = s_optim + 0.01;

for (int i=51; i<=100; i++)
  s_optim = s_optim + 0.01;

for (int i=101; i<=150; i++)
  s_optim = s_optim + 0.01;

// ...
for (int i=951; i<=1000; i++)
  s_optim = s_optim + 0.01;

check_reverse_rel_error(s_optim, 1e-6);
```

- Now 20 instructions instead of 1 → 20 possible transformations
  - $2^{20} = 1048576$ possible configurations
- Finally which format per iteration to have

$$\frac{|\mathtt{s\_b64} - \mathtt{s\_optim}|}{\mathtt{s\_b64}} < 10^{-6}?$$

☺ Our tool's output: 9 instructions out of these 20 → 45% in binary32

Outline of the talk

1. Auto-tuning of iterative routines

2. Experimental results

3. Conclusion and perspectives

Impact of different strategies

	threshold	delta-debugging			split / factor = 10			split / factor = 50		
		b64 / b32	%	h:m:s	b64 / b32	%	h:m:s	b64 / b32	%	h:m:s
sum	1e-5	1 / 0	0.0	0:00:01	5 / 5	50.0	0:00:06	22 / 28	56.0	0:00:43
	1e-6	1 / 0	0.0	0:00:01	6 / 4	40.0	0:00:07	27 / 23	46.0	0:00:48
	1e-8	1 / 0	0.0	0:00:01	10 / 0	0.0	0:00:09	48 / 2	4.0	0:01:08
riemann	1e-8	3 / 2	40.0	0:00:04	15 / 35	70.0	0:00:41	10 / 240	96.0	0:01:44
	1e-9	3 / 2	40.0	0:00:04	22 / 28	56.0	0:00:53	14 / 236	94.4	0:02:04
	1e-11	5 / 0	0.0	0:00:04	44 / 6	12.0	0:01:05	19 / 231	92.4	0:02:31
arclength	1e-8	6 / 5	45.5	0:00:10	31 / 79	71.8	0:02:32	71 / 479	87.1	0:19:05
	1e-9	8 / 3	27.3	0:00:10	64 / 46	41.8	0:02:48	131 / 419	76.2	0:24:57
	1e-11	11 / 0	0.0	0:00:09	105 / 5	4.5	0:02:28	491 / 59	10.7	0:48:38
simpson	1e-7	4 / 6	60.0	0:00:06	18 / 82	82.0	0:01:42	14 / 486	97.2	0:02:13
	1e-9	6 / 4	40.0	0:00:09	58 / 42	42.0	0:02:28	182 / 318	63.6	0:27:20
	1e-11	10 / 0	0.0	0:00:09	91 / 9	9.0	0:03:21	439 / 61	12.2	0:41:46
nbody	1e-7	9 / 15	62.5	0:00:19	7 / 233	97.1	0:03:05	2 / 1198	99.8	0:02:16
	1e-10	21 / 3	12.5	0:00:27	212 / 28	11.7	0:08:17	742 / 458	38.2	5:12:40
	1e-11	24 / 0	0.0	0:00:24	228 / 12	5.0	0:07:26	1062 / 138	11.5	2:34:49
	1e-12	24 / 0	0.0	0:00:25	237 / 3	1.2	0:08:09	1160 / 40	3.3	3:03:27

Auto-tuning results for splitting strategy.

Impact of different strategies

	threshold	delta-debugging			unroll / factor = 10			unroll / factor = 50		
		b64 / b32	%	h:m:s	b64 / b32	%	h:m:s	b64 / b32	%	h:m:s
sum	1e-5	1 / 0	0.0	0:00:01	6 / 4	40.0	0:00:08	25 / 25	50.0	0:00:40
	1e-6	1 / 0	0.0	0:00:01	10 / 0	0.0	0:00:08	45 / 5	10.0	0:00:54
	1e-8	1 / 0	0.0	0:00:01	10 / 0	0.0	0:00:08	48 / 2	4.0	0:00:54
riemann	1e-8	3 / 2	40.0	0:00:04	19 / 31	62.0	0:00:29	2 / 248	99.2	0:02:00
	1e-9	3 / 2	40.0	0:00:04	25 / 25	50.0	0:00:46	43 / 207	82.8	0:04:39
	1e-11	5 / 0	0.0	0:00:04	44 / 6	12.0	0:01:02	213 / 37	14.8	0:07:36
arclength	1e-8	6 / 5	45.5	0:00:10	32 / 78	70.9	0:01:40	67 / 483	87.8	0:12:50
	1e-9	8 / 3	27.3	0:00:10	31 / 79	71.8	0:02:39	80 / 470	85.5	0:22:54
	1e-11	11 / 0	0.0	0:00:09	110 / 0	0.0	0:01:57	509 / 41	7.5	0:27:58
simpson	1e-7	4 / 6	60.0	0:00:06	10 / 90	90.0	0:00:39	23 / 477	95.4	0:03:09
	1e-9	6 / 4	40.0	0:00:09	56 / 44	44.0	0:01:24	179 / 321	64.2	0:18:18
	1e-11	10 / 0	0.0	0:00:09	95 / 5	5.0	0:02:23	413 / 87	17.4	0:39:05
nbody	1e-7	9 / 15	62.5	0:00:19	36 / 204	85.0	0:02:33	4 / 1196	99.7	0:07:55
	1e-10	21 / 3	12.5	0:00:27	206 / 34	14.2	0:09:28	900 / 300	25.0	2:33:01
	1e-11	24 / 0	0.0	0:00:24	228 / 12	5.0	0:08:29	1046 / 154	12.8	3:51:45
	1e-12	24 / 0	0.0	0:00:25	238 / 2	0.8	0:06:16	1176 / 24	2.0	1:35:16

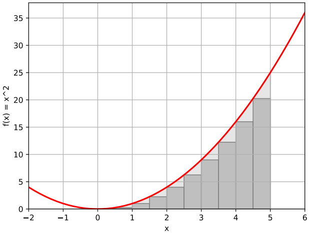
Auto-tuning results for unrolling strategy.

Focus on Riemann integral (1/3)

- Objective: compute the following integral for $f(x) = x^2$

$$\int_a^b f(x) \cdot dx \approx \frac{b-a}{n} \cdot \sum_{k=0}^{n-1} f\left(a + k \cdot \frac{b-a}{n}\right)$$

  - $[a,b] = [0,5]$
  - $n = 400$



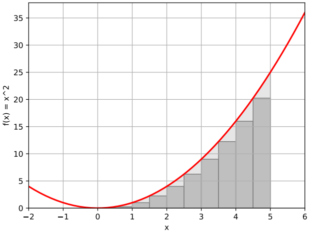
- For performance purpose → same format for the whole loop body

Focus on Riemann integral (1/3)

- Objective: compute the following integral for $f(x) = x^2$

$$\int_a^b f(x) \cdot dx \approx \frac{b-a}{n} \cdot \sum_{k=0}^{n-1} f\left(a + k \cdot \frac{b-a}{n}\right)$$

  - $[a,b] = [0,5]$
  - $n = 400$

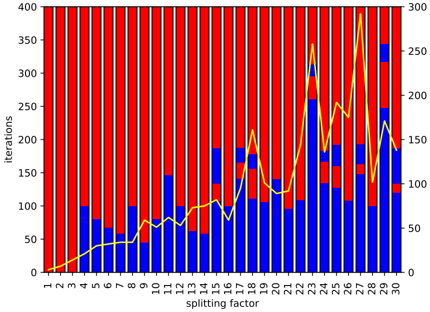


- For performance purpose → same format for the whole loop body

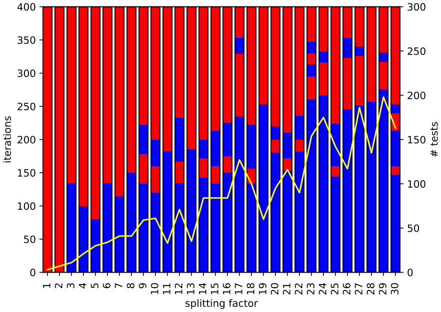
Which iterations can be done in lower precision such that the result remains at a given threshold of the binary64 result?

Focus on Riemann integral (2/3)

- Number of splittings = 1 → 30
- Available formats: binary64, binary32



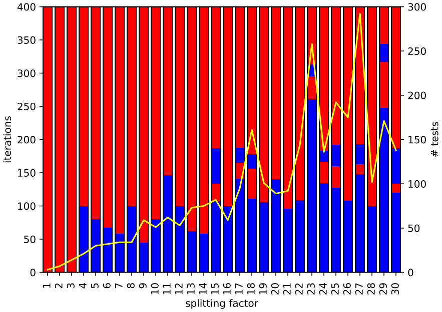
threshold = 10^{-9}



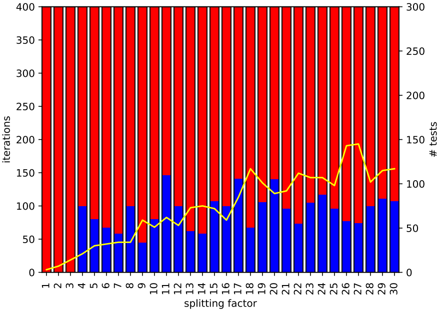
threshold = 10^{-8}

Focus on Riemann integral (3/3)

- Number of splittings = 1 → 30
- Available formats: binary64, binary32
- Only one precision change



threshold = 10^{-9}



threshold = 10^{-9}, max change = 1

Auto-tuning for unbounded loops (1/2)

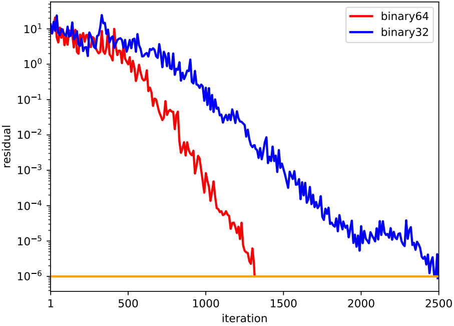
- Conjugate Gradient: method to solve the linear system $Ax = b$‘

```
$r_0 := p_0 := b - Ax_0$, and $k = 0$
while $\|r_k\| \geq \epsilon$ and $k <$ maxiter do
    $\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} := x_k + \alpha_k p_k$
    $r_{k+1} := r_k - \alpha_k A p_k$
    $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} := r_{k+1} + \beta_k p_k$
    $k = k + 1$
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)
    - $\epsilon = 10^{-6}$

Auto-tuning for unbounded loops (1/2)

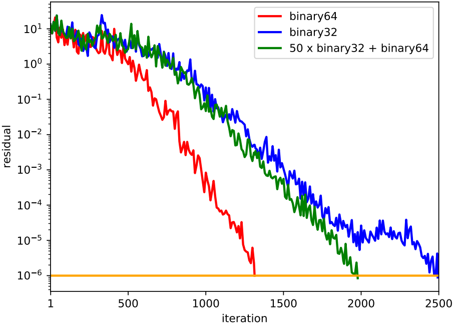
- Conjugate Gradient: method to solve the linear system $Ax = b$‘



- 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)
  - $\epsilon = 10^{-6}$
  - binary64 = 1315 iterations
  - binary32 = 2494 iterations

Auto-tuning for unbounded loops (1/2)

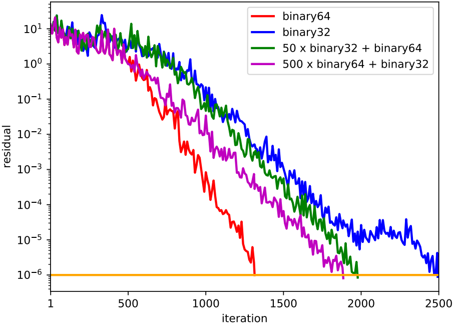
- Conjugate Gradient: method to solve the linear system $Ax = b$‘



- 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)
  - $\epsilon = 10^{-6}$
  - binary64 = 1315 iterations
  - binary32 = 2494 iterations

Auto-tuning for unbounded loops (1/2)

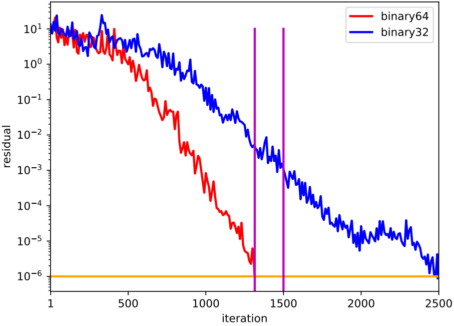
- Conjugate Gradient: method to solve the linear system $Ax = b$‘



- 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)
  - $\epsilon = 10^{-6}$
  - binary64 = 1315 iterations
  - binary32 = 2494 iterations

Auto-tuning for unbounded loops (1/2)

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

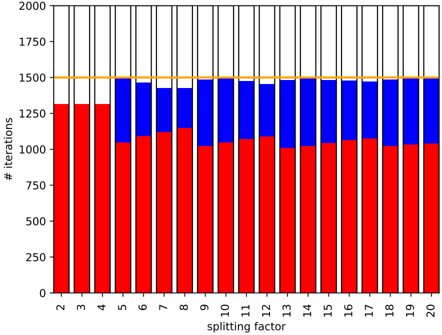


- In exact arithmetic, it converges in *n* iterations
- But in floating-point arithmetic, the number of iterations is linked to the precision of the computations
- Example: 494_bus matrix (Suite Sparse Matrix Collection)
  - $\epsilon = 10^{-6}$
  - binary64 = 1315 iterations
  - binary32 = 2494 iterations

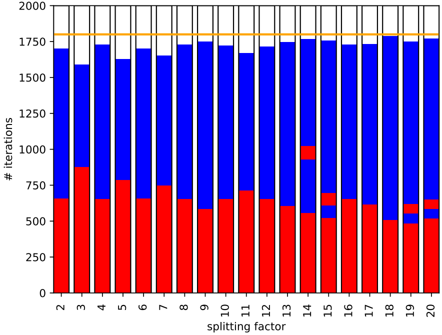
Among these 1315 iterations, which ones can be transformed into binary32, at the risk of increasing the total number of iterations?

Auto-tuning for unbounded loops (2/2)

- Number of splittings = 2 → 20
- Available formats: binary64, binary32



max iteration = 1500



max iteration = 1800

Outline of the talk

1. Auto-tuning of iterative routines

2. Experimental results

3. Conclusion and perspectives

Conclusion and perspectives

Contribution

- Dynamic tool to tune the precision of certain instructions in iterative routines
    - target instructions of loop bodies
    - based on loop transformation + fp2mp + delta-debugging

Future works

- Evaluate the speedup delivered by our tool
- Investigate other loop transformations
- Study how this approach scales → loop size, nested loops