RAIM 2024

Perpignan, November 4-6

# Performance on SIMD architectures of auto-tuned programs for matrix multiplication


Youssef Fakhreddine and Guillaume Revy

Univ Perpignan Via Domitia, DALI, Perpignan, France

LIRMM, Univ Montpellier, CNRS (UMR 5506), Montpellier, France

# Project context and core goal

Context : ANR JCJC PADOC

- PADOC: Performances- and Accuracy-aware Data format Optimization in numerical Codes

Motivation : Development of tools to optimize data formats in numerical computation applications

- To improve their performance, by making better use of modern architectures,
- Without degrading the accuracy of their results.

Goal : 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.

# Motivation and key achievements

- ⊕ 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 ⚠

- Achievements:

  RAIM 2023

  - Build a dynamic auto-tuning tool that targets instructions in iterative routines based on loop transformation + fp2mp + delta-debugging

  RAIM 2024

  - Automate the transformations proposed by our tool DD-FP2MP
  - Evaluate the speedup delivered in matrix multiplication on SIMD architecture

# Outline of the talk

1. Auto-tuning approach for iterative routines

2. Analysis of performances in SIMD architectures

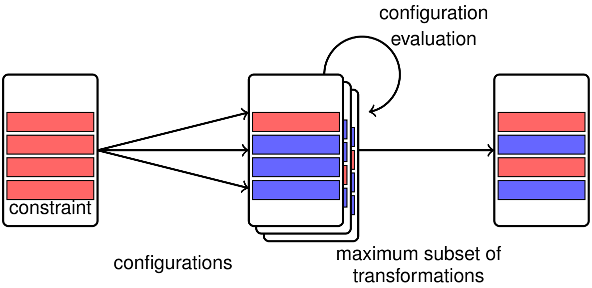
3. Experimental results

4. Conclusion and perspectives

# Outline of the talk

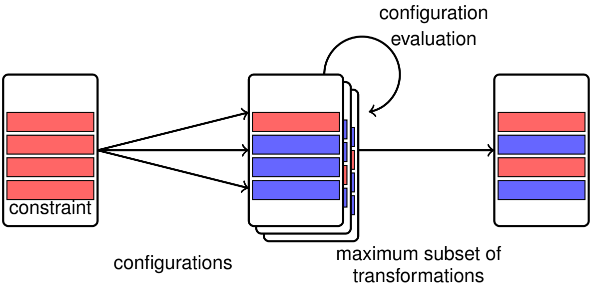
# Main flow of dynamic tools

- 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?

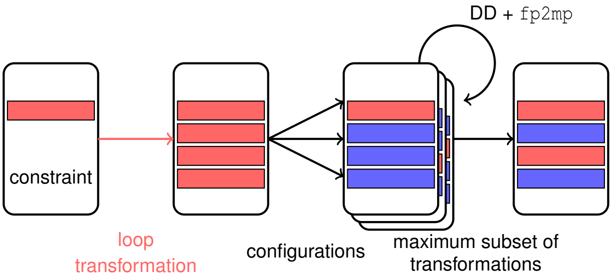
# Outline of our project

## Originality of the proposed approach

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

## Main steps

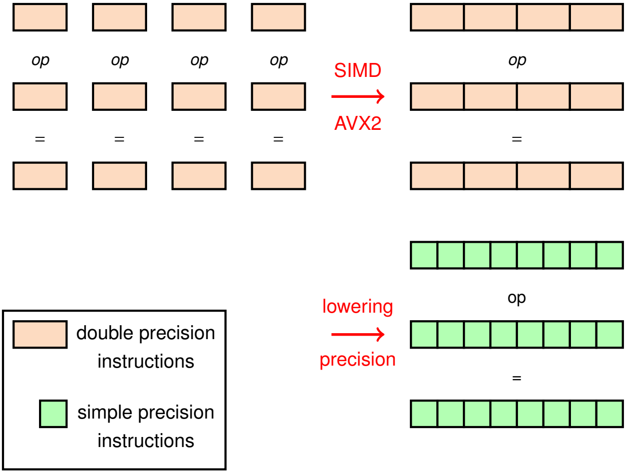
- loop transformation (splitting, unrolling)
- configuration evaluation → fp2mp
- building of maximum subset of transformations → delta-debugging



# Outline of the talk

1. Auto-tuning approach for iterative routines
2. Analysis of performances in SIMD architectures
3. Experimental results
4. Conclusion and perspectives

# SIMD paradigm




How can we make good use of this to improve our auto-tuning process?

# Matrix multiplication vectorisation

## Our matrix multiplication c code

```
for(int k = 0; k <= n-1; k += 1) {
  for(int i = 0; i <= n-1; i += 1) {
    for(int j = 0; j <= n-1; j += 1)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```

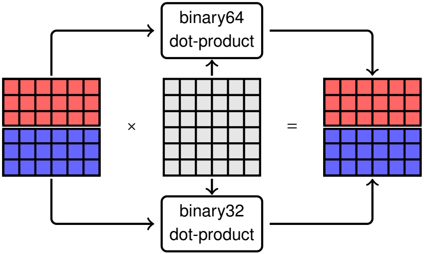
## Vectorised matrix multiplication pseudocode

```
for(int k = 0; k <= n-1; k += 1) {
  for(int i = 0; i <= n-1; i += 1) {
    for(int j = 0; j <= n-1; j += 4)
      C[i][j...j+3] += A[i][k] * B[k][j...j+3];
  }
}
```

Vectorisation

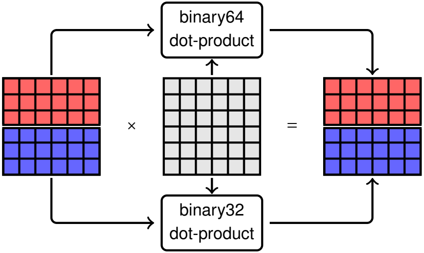
Which loop should we split?

# Vectorised matrix multiplication splitting

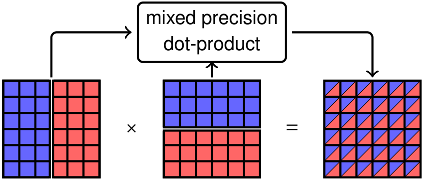



splitting over i

# Vectorised matrix multiplication splitting

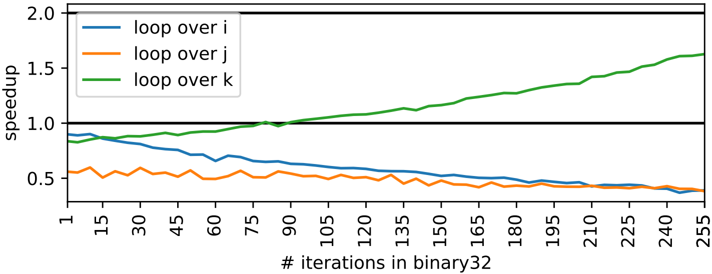



splitting over i



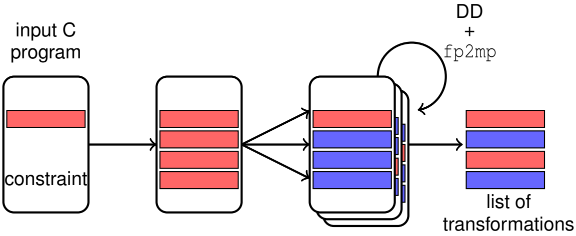

splitting over k

# Expected speedup on loop-splitting for size-256 matrix multiplication



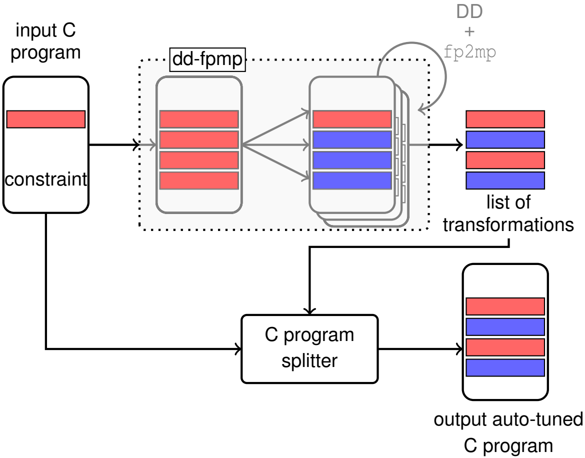

- Splitting Strategy
  - Split each loop (over i, j, and k) into two subloops
  - Apply binary64 to binary32 transformations on the first subloop
  - Vary the end index of the first subloop from 1 to 255 (step of 5)

# New workflow at C level




- LLVM IR level splitting
  - Dependent on the compiler being used
  - gives hints to be applicated by the user

# New workflow at C level




- LLVM IR level splitting
  - Dependent on the compiler being used
  - gives hints to be applicated by the user
- New approach
  - Introduced a new loop splitting tool at the C level
  - Based on Python, applicable to any iterative program
  - gives back an optimised C program
  - the output C program can be compiled with any compiler and executed

# C level splitting

```
 void matmul(double *A, double *B, double *C,
             int n) {
   int i, j, k;
#pragma clang loop split_optimization(enable)
   // SPLIT_FOR [indvar=k, start=0, end=n-1, step=1]
   // REPLACEMENT:
   //   A > A_b32 > heap (n*n)
   //   B > B_b32 > heap (n*n)
   //   C > C_b32 > heap (n*n)
   // INITIALISATION:
   //   for(i = 0; i <= n-1; i += 1) {
   //     for(j = 0; j <= n-1; j += 1) {
   //       A_b32[i*n+j] = A[i*n+j];
   //       B_b32[i*n+j] = B[i*n+j];
   //     }
   //   }
   // PREFIX:
   //   for(i = 0; i <= n-1; i += 1) {
   //     for(j = 0; j <= n-1; j += 1)
   //       C_b32[i*n+j] = C[i*n+j];
   //   }
   // SUFFIX:
   //   for(i = 0; i <= n-1; i += 1) {
   //     for(j = 0; j <= n-1; j += 1)
   //       C[i*n+j] = C_b32[i*n+j];
   //   }
   for (k = 0; k <= n-1; k += 1) {
     for (i = 0; i <= n-1; i += 1) {
       for (j = 0; j <= n-1; j += 1)
         C[i*n+j] += A[i*n+k]*B[k*n+j];
     }
   }
   // END SPLIT_FOR
 }
```

- SPLIT_FOR Surrounds loops to be split based on induction variable, start/end values, step.
- REPLACEMENT Manages binary64 to binary32 variable replacement.
- INITIALISATION Inserts initialization for lower precision variables before loops.
- PREFIX / SUFIX Handles cast moving before and after subloops.

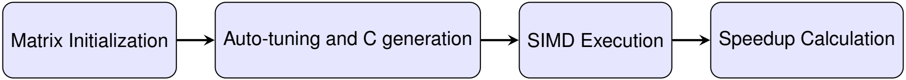
# Generated splitted C code

- Example Configuration:
  - Python list [[0, 63, True], [64, 255, False]]
  - Splits the loop into:
    - Subloop 1: Iteration 0 to 63 using binary32
    - Subloop 2: Iteration 64 to 255 using binary64
- Generated C Program
  - Includes declaration, allocation, and deallocation of lower precision variables
  - Cast moving code inserted only for subloops with reduced precision
  - Consecutive subloops of the same precision are collapsed

```
// DECLARATION
float *A_b32, *B_b32, *C_b32;
// ALLOCATION
// ... initialisation
for(i = 0; i <= n-1; i += 1) {
  for(j = 0; j <= n-1; j += 1) {
    A_b32[i*n+j] = A[i*n+j];
    B_b32[i*n+j] = B[i*n+j];
  }
}

// ... loop id = {0}
for(i = 0; i <= n-1; i += 1) {
  for(j = 0; j <= n-1; j += 1)
    C_b32[i*n+j] = C[i*n+j];
}
for (k = 0; k <= 63; k += 1) {
  for (i = 0; i <= n-1; i += 1) {
    for (j = 0; j <= n-1; j += 1)
      C_b32[i*n+j] += A_b32[i*n+k]*B_b32[k*n+j];
  }
}
for(i = 0; i <= n-1; i += 1) {
  for(j = 0; j <= n-1; j += 1)
    C[i*n+j] = C_b32[i*n+j];
}
// ... loop id = {1}
for (k = 64; k <= n-1; k += 1) {
  for (i = 0; i <= n-1; i += 1) {
    for (j = 0; j <= n-1; j += 1)
      C[i*n+j] += A[i*n+k]*B[k*n+j];
  }
}
// END AUTO-TUNED LOOP
// DEALLOCATION
```

# Outline of the talk

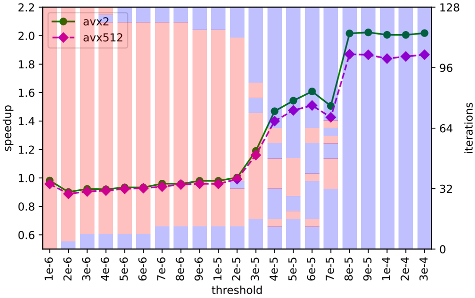
# Experimental Setup



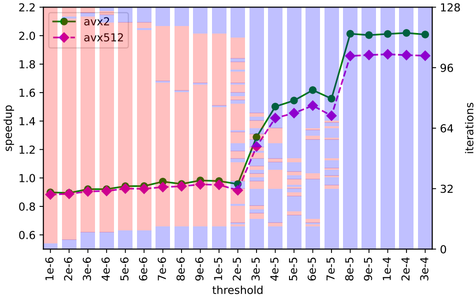

- Matrix generation factors:
  - size
  - condition number
- Available formats:
  - Binary64
  - Binary32
- Threshold
  - $\frac{||C\_Bmix - C\_B64||_\infty}{||C\_B64||_\infty}$
- Speedup
  - RDTSC
- Splitting factor
  - number of subloops created resulting of the splitting
- Number of changes
  - number of switches between data formats, adding performance casts

# Speedup and precision patterns 1/2

- matrix size = 128



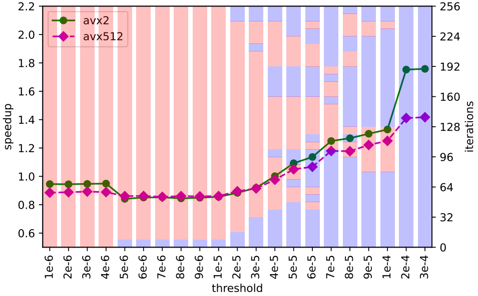

splitting factor = 32



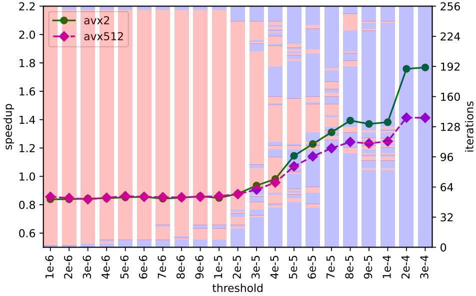

splitting factor = 128

# Speedup and precision patterns 2/2

- matrix size = 256




splitting factor = 32




splitting factor = 128

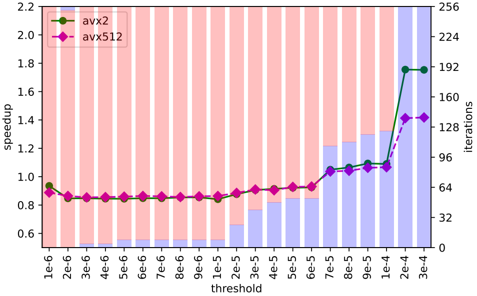
# Number of allowed precision changes impact

- matrix size $= 256$
- splitting factor $= 64$

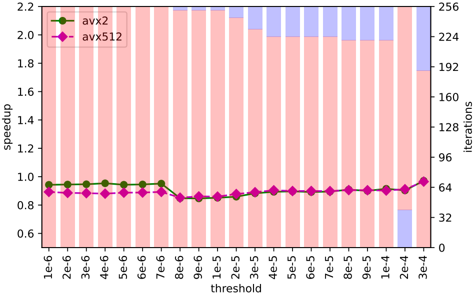
| allowed changes | threshold $\epsilon$ | 1e-6 | 4e-6 | 7e-6 | 1e-5 | 4e-5 | 7e-5 | 1e-4 | 4e-4 |
|---|---|---|---|---|---|---|---|---|---|
| $m = 1$ | # changes | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| | # iterations in b32 | 0 | 4 | 8 | 8 | 48 | 108 | 124 | 256 |
| $m = 2$ | # changes | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 0 |
| | # iterations in b32 | 0 | 8 | 12 | 12 | 80 | 172 | 208 | 256 |
| $m = \infty$ | # changes | 0 | 2 | 2 | 4 | 6 | 8 | 10 | 0 |
| | # iterations in b32 | 0 | 8 | 12 | 16 | 96 | 196 | 220 | 256 |

# Condition number impact

- matrix size = 256
- splitting factor = 64




$\kappa = 100$




$\kappa = 1000$

# Outline of the talk

# 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
- Automate the transformations proposed by the tool
- Demonstrated tool effectiveness in matrix multiplication, showing significant performance improvements.

## Future works

- Study how this approach scales → loop size, nested loops
- Gain prediction
- Investigate other loop transformations

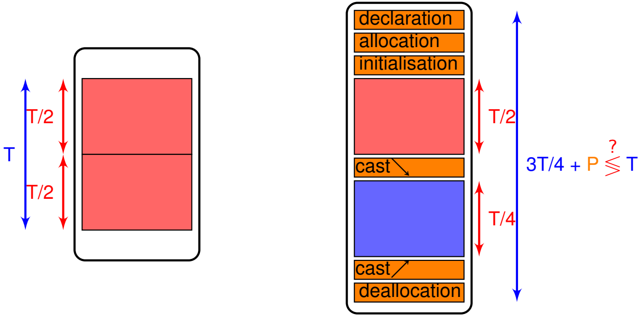
# Thank You for Your Attention!

☺

Do you have any questions?

# Gain prediction

## Ongoing




*Initial loop*                *optimised loop*

How can we predict the speedup in advance so that we can avoid executing configurations that are likely to yield no improvements?

# 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+=2) {
  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+=2) {
  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, 24);
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

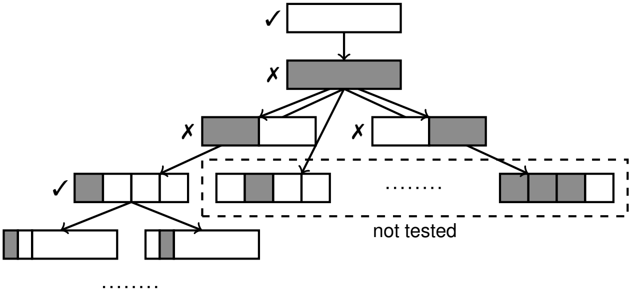
```
double s_b64 = 0.;

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, 24);
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