RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030 PROGRAMME DE RECHERCHE NUMÉRIQUE POUR L'EXASCALE

Automatic Multi-Versioning of Computation Kernels

ExaSoft General Assembly - Toulouse

Raphaël Colin
PhD student under the supervision of
Philippe Clauss & Thierry Gautier

November 7th 2024

Table of Contents

Context of this work

General goal

Optimizing scientific codes for exascale architectures

Context of this work

General goal

Optimizing scientific codes for exascale architectures

- PhD started in October 2023
- WP2: Just-In-Time code optimization with continuous feedback loop

Multi-Versioning

Definition

Have multiple versions of the same program (or part of a program), and use the best version for a given execution context.

Multi-Versioning

Definition

Have **multiple versions** of the same program (or part of a program), and use the **best** version for a given **execution context**.

- Execution context $\rightarrow$ what is it made of?
- Multiple versions $\rightarrow$ how are versions generated?
- Best version? $\rightarrow$ different metrics (performance, energy efficiency)

Execution context

Software

- Function parameters
- Value of global variables
- System parameters
- ...

Hardware

- Cache characteristics
- Accelerators
- CPU Architecture
- ...

Execution context - Impact on optimization I

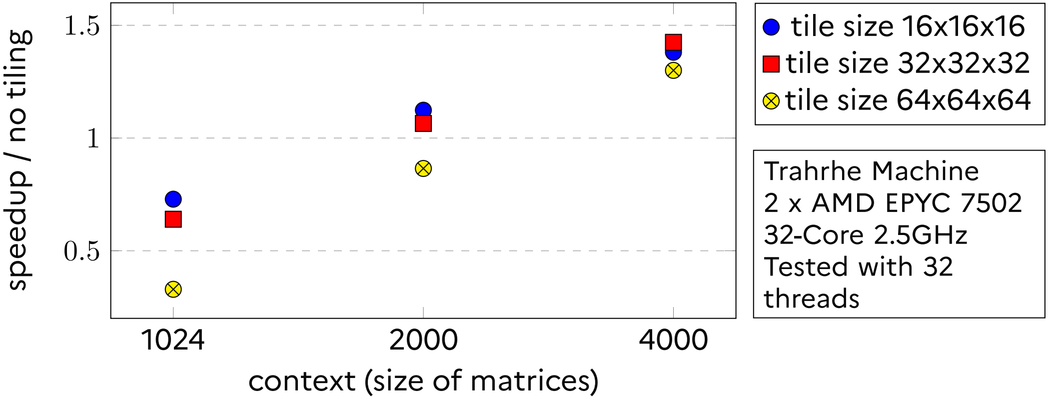

Figure 1: Speedup for the `3mm` Polybench benchmark by context

Execution context - Impact on optimization II

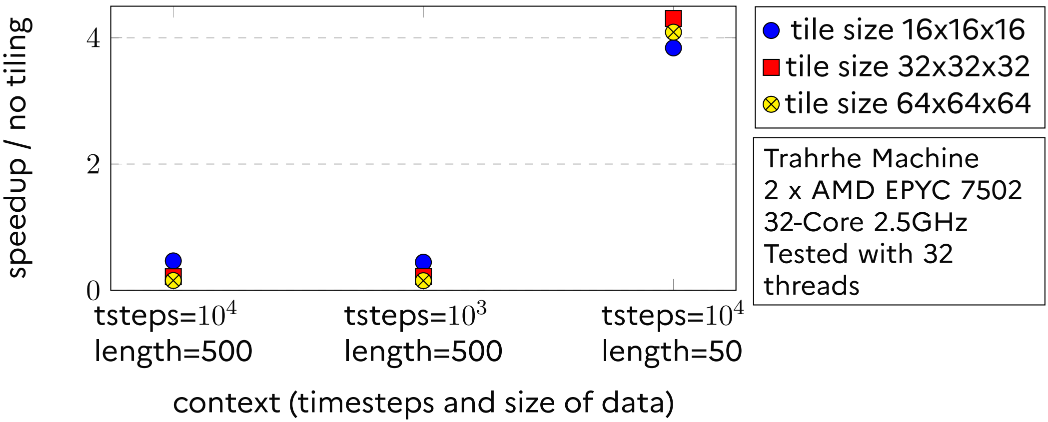

Figure 2: Speedups for the `dynprog` Polybench benchmark by context

Version generation

We can obtain multiple versions:

- by writing them by hand $\rightarrow$ different algorithms for the same problem
- by generating them automatically $\rightarrow$ optimize differently

Version generation

We can obtain multiple versions:

- by writing them by hand → different algorithms for the same problem
- **by generating them automatically** → optimize differently

Tools for optimization

- Classical compiler optimizations → function specialization
- Polyhedral optimizations → loop transformations

Polyhedral optimizations

The polyhedral model

- A mathematical model to represent loop nests
- Allows loop transformations respecting data dependencies
- Limited to affine loop nests

Tools

- Pluto
- Polly (clang)
- Graphite (gcc)

Apollo

Polyhedral model limitations

The polyhedral model can only be used with affine loop nests

Apollo [1, 2]

- Project of the Inria CAMUS team
- Using polyhedral optimizations on statically non-affine loops
- Dynamic optimization and JIT compilation

Apollo

Polyhedral model limitations

The polyhedral model can only be used with affine loop nests

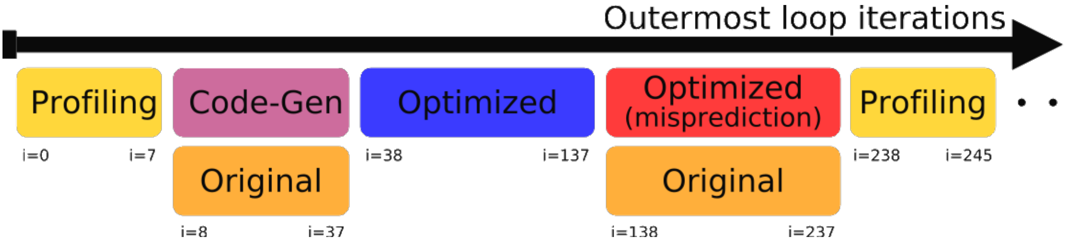

Figure 3: Apollo execution model

Apollo

We can leverage Apollo for automatic multi-versioning:

- JIT compilation
- Polyhedral optimization
- Prediction model (for memory accesses, scalar values, etc.)

Motivation for (dynamic) multi-versioning

Code transformations:

- Performance is hard to predict for a specific code [3]
- Harder to tailor for a specific execution context

Multi-versioning in Apollo

Existing implementation [4]

Each time a loop nest is encountered:

- A new loop transformation is computed
- It is executed and timed

Once all transformations are tested:

- Directly use the best found version

Multi-versioning in Apollo

Existing implementation [4]

Each time a loop nest is encountered:

- A new loop transformation is computed
- It is executed and timed

Once all transformations are tested:

- Directly use the best found version

Issues

- No context analysis
- The context isn't used to computed optimization parameters
- Multi-versioning is only partly dynamic (optimization parameters are pre-determined)

Designing a new system

Multiple phases

- Start by collecting execution contexts
- Analyze collected contexts
- Generate versions with the help of context parameters

Current state

- Collection of software context
- Ability to run the kernel separately for evaluation
- Leverage the existing Apollo features for loop transformation (loop tiling)

Example

```
#pragma apollo kernel
void computation_kernel(/* ... */) {
  #pragma apollo dcop
  for (/* ... */) {
    /* ... */
  }
}
```

Multi-versioning implementation

- More advanced context analysis
  - We can leverage Apollo's prediction models
- Target parametric and non-parametric optimizations:
  - Function specialization
  - Software pre-fetching
- Use more realistic benchmarks

In general

- Looking at multi-versioning for accelerators (GPUs)
- Considering energy efficiency

Thank you

Questions?

References I

[1] Aravind Sukumaran Rajam. "Beyond the realm of the polyhedral model : combining speculative program parallelization with polyhedral compilation". Theses. Université de Strasbourg, Nov. 2015. url: https://theses.hal.science/tel-01342007.

[2] Juan Manuel Martinez Caamaño et al. "Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones". en. In: *Concurr. Comput.* 29.15 (Aug. 2017), e4192.

[3] Spyridon Triantafyllis et al. "Compiler optimization-space exploration". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '03. USA: IEEE Computer Society, 2003, pp. 204–215. isbn: 076951913X.

References II

[4] Raquel Lazcano et al. "Runtime multi-versioning and specialization inside a memoized speculative loop optimizer". In: *Proceedings of the 29th International Conference on Compiler Construction*. CC 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 96–107. isbn: 9781450371209. doi: `10.1145/3377555.3377886`. url: `https://doi.org/10.1145/3377555.3377886`.