RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

cea cnrs Inria

FRANCE 2030
PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

Compilation and Automatic Code Optimization

Work Package 2

Philippe Clauss

September 24, 2025

University of Strasbourg & Inria Camus

WP2 Goals

- Make advanced code optimizations available to the HPC community
  automatic parallelization, *loop transformations*, *polyhedral optimizations*, ...
- Using static or runtime optimization techniques
  additional compilation passes, *just-in-time analysis & optimization*, ...
- Implement automatic code optimizations in mainstream HPC programming tools
  Clang-LLVM, *Kokkos*, ...
- User-transparent code optimizations yielding:
  - better computing resource usage
  - better time performance
  - less energy consumption

WP2 Members

- WP2 leaders: Philippe Clauss (Inria Camus) & Thierry Gautier (Grenoble)
- Other permanent researcher: Christophe Alias (Inria Cash)
- PhD Students:
  - Ugo Battiston (WP1+WP2, since October 2023)
  - Raphaël Colin (since October 2023)
  - Alec Sadler (since October 2023)
- Engineers:
  - Erwan Auer (since January 2024)
  - Pierre-Etienne Polet (since March 2024)

Running PhDs

- Ugo Battiston (WP1+WP2, since October 2023)
  Connecting Kokkos with the polyhedral model
  Advisors: Philippe Clauss & Marc Pérache (CEA, WP1)

- Raphaël Colin (since October 2023)
  Context-adapted multi-versioning of computation kernels
  Advisors: Philippe Clauss & Thierry Gautier

- Alec Sadler (since October 2023)
  Automatic Specialization of Polyhedral Programs on Sparse Structures
  Advisor: Christophe Alias

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030

PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

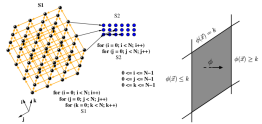
cea

cnrs

Inria

Connecting Kokkos with the Polyhedral Model

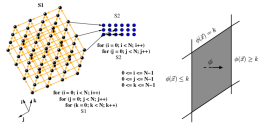
Connecting Kokkos with the polyhedral model



kokkos

- Kokkos:
  - a production level solution for writing modern C++ applications **in a hardware agnostic way** (CPU, GPU)
  - increasingly used by the HPC community (and by people involved in NumPEx)

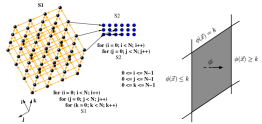
Connecting Kokkos with the polyhedral model



kokkos

- Kokkos:
  - a production level solution for writing modern C++ applications **in a hardware agnostic way** (CPU, GPU)
  - increasingly used by the HPC community (and by people involved in NumPEx)
- *hardware agnostic way* ⇒ **what is the price to pay?**

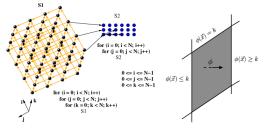
Connecting Kokkos with the polyhedral model



kokkos

- Kokkos:
  - a production level solution for writing modern C++ applications **in a hardware agnostic way** (CPU, GPU)
  - increasingly used by the HPC community (and by people involved in NumPEx)
- *hardware agnostic way* ⇒ **what is the price to pay?**
  - Many default configuration parameter values are hidden to the user (e.g. loop tile sizes, scheduling strategy, GPU block sizes, ...)
    ⇒ *Many performance improvement opportunities are unavailable to the user*

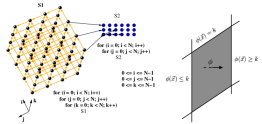
Connecting Kokkos with the polyhedral model



kokkos

- Kokkos:
  - a production level solution for writing modern C++ applications **in a hardware agnostic way** (CPU, GPU)
  - increasingly used by the HPC community (and by people involved in NumPEx)
- *hardware agnostic way* ⇒ **what is the price to pay?**
  - Many default configuration parameter values are hidden to the user (e.g. loop tile sizes, scheduling strategy, GPU block sizes, ...)
    ⇒ *Many performance improvement opportunities are unavailable to the user*
  - The C++ template structure make the source code much more convoluted for compilers
    ⇒ *Many performance improvement opportunities are unavailable to the compiler*

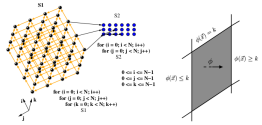
Connecting Kokkos with the polyhedral model



kokkos

- Kokkos:
  - a production level solution for writing modern C++ applications **in a hardware agnostic way** (CPU, GPU)
  - increasingly used by the HPC community (and by people involved in NumPEx)
- *hardware agnostic way* ⇒ **what is the price to pay?**
  - Many default configuration parameter values are hidden to the user (e.g. loop tile sizes, scheduling strategy, GPU block sizes, ...)
    ⇒ *Many performance improvement opportunities are unavailable to the user*
  - The C++ template structure make the source code much more convoluted for compilers
    ⇒ *Many performance improvement opportunities are unavailable to the compiler*
  - The kind of parallel code kernels that may be written with Kokkos are too restricted
    ⇒ *Only rectangular loop iteration domains, e.g. no triangular domains*
    ⇒ *Only one loop nest per kernel, no kernel made of several loop nests*
    ⇒ *missed optimization opportunities (e.g. loop fusion)*

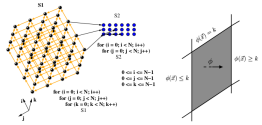
Connecting Kokkos with the polyhedral model



kokkos

- Our solution
  - Make advanced fully-automatic code optimizations available to Kokkos
  - Enable users to write a wider range of kernels
- Opportunities
  - Parallel kernels in Kokkos are *perfect candidates* for the *polyhedral model*
  - Existing code optimizers may be connected/adapted to Kokkos: LLVM-Polly, Pluto, ...
- First WP2 deliverable at the end of October

Connecting Kokkos with the polyhedral model



kokkos

- Our solution
  - Make advanced fully-automatic code optimizations available to Kokkos
  - Enable users to write a wider range of kernels
- Opportunities
  - Parallel kernels in Kokkos are *perfect candidates* for the *polyhedral model*
  - Existing code optimizers may be connected/adapted to Kokkos: LLVM-Polly, Pluto, ...
- First WP2 deliverable at the end of October
- *Attend Ugo Battiston's presentation in next Session!*

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030

PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

cea cnrs Inria

Context-Adapted Multi-Versioning of Computation Kernels

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**
    - **depending on the number of threads/cores, the best performing versions have different tile sizes!**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**
    - **depending on the number of threads/cores, the best performing versions have different tile sizes!**
  - Intel 20-Core Ultra 7 265: 0.95 sec. with 20 threads/cores without tiling

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**
    - **depending on the number of threads/cores, the best performing versions have different tile sizes!**
  - Intel 20-Core Ultra 7 265: 0.95 sec. with 20 threads/cores without tiling
    - best performing version with 16 threads/cores: 0.09 sec. with **tile size 32 x 256 x 128**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**
    - **depending on the number of threads/cores, the best performing versions have different tile sizes!**
  - Intel 20-Core Ultra 7 265: 0.95 sec. with 20 threads/cores without tiling
    - best performing version with 16 threads/cores: 0.09 sec. with **tile size 32 x 256 x 128**
    - best performing version with 8 threads/cores: 0.13 sec. with **tile size 16 x 32 x 32**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**
    - **depending on the number of threads/cores, the best performing versions have different tile sizes!**
  - Intel 20-Core Ultra 7 265: 0.95 sec. with 20 threads/cores without tiling
    - best performing version with 16 threads/cores: 0.09 sec. with **tile size 32 x 256 x 128**
    - best performing version with 8 threads/cores: 0.13 sec. with **tile size 16 x 32 x 32**
    - **for a given number of threads/cores on different processors, the best performing versions have different tile sizes!**

Context-adapted multi-versioning of computation kernels

- Example: 2mm-code (2 matrix multiplications in succession)
  - AMD EPYC 7502 32-Core Processor: 1.52 sec. with 32 threads/cores without tiling
    - best performing version with 32 threads/cores: 0.09 sec. **with tile size 64 x 32 x 128**
    - best performing version with 25 threads/cores: **0.09 sec. with tile size 64 x 64 x 256**
    - **same execution time with less threads/cores!**
    - best performing version with 16 threads/cores: 0.15 sec. with **tile size 8 x 128 x 256**
    - best performing version with 8 threads/cores: 0.29 sec. with **tile size 16 x 128 x 256**
    - **depending on the number of threads/cores, the best performing versions have different tile sizes!**
  - Intel 20-Core Ultra 7 265: 0.95 sec. with 20 threads/cores without tiling
    - best performing version with 16 threads/cores: 0.09 sec. with **tile size 32 x 256 x 128**
    - best performing version with 8 threads/cores: 0.13 sec. with **tile size 16 x 32 x 32**
    - **for a given number of threads/cores on different processors, the best performing versions have different tile sizes!**
  - **Performance portability is a fake!**

Context-adapted multi-versioning of computation kernels

- Code optimizations are sensitive to variations in the HW/SW execution context
  - HW context: #nodes, #cores, vector capability, cache sizes, memory hierarchy, ...
  - SW context: problem size, data input, function parameters, ...

Context-adapted multi-versioning of computation kernels

- Code optimizations are sensitive to variations in the HW/SW execution context
  - HW context: #nodes, #cores, vector capability, cache sizes, memory hierarchy, ...
  - SW context: problem size, data input, function parameters, ...
- Huge distortions of performance when running a given code in various contexts

Context-adapted multi-versioning of computation kernels

- Code optimizations are sensitive to variations in the HW/SW execution context
  - HW context: #nodes, #cores, vector capability, cache sizes, memory hierarchy, ...
  - SW context: problem size, data input, function parameters, ...
- Huge distortions of performance when running a given code in various contexts
  - The best performing code in a given context may be the worse in another context

Context-adapted multi-versioning of computation kernels

- Code optimizations are sensitive to variations in the HW/SW execution context
  - HW context: #nodes, #cores, vector capability, cache sizes, memory hierarchy, ...
  - SW context: problem size, data input, function parameters, ...
- Huge distortions of performance when running a given code in various contexts
  - The best performing code in a given context may be the worse in another context
  - Different optimizing code transformations per context are required

Context-adapted multi-versioning of computation kernels

- Our solution
  - Seeking the best performing code versions for observed actual contexts, on-the-fly
    - By testing different optimizing transformations
    - By simulating real execution samples
    - On separated nodes/cores
  - While the main code is running
  - Branching the main code on a better performing version
    - when ready
    - when the same context is met again

Context-adapted multi-versioning of computation kernels

- Our solution
  - Seeking the best performing code versions for observed actual contexts, on-the-fly
    - By testing different optimizing transformations
    - By simulating real execution samples
    - On separated nodes/cores
  - While the main code is running
  - Branching the main code on a better performing version
    - when ready
    - when the same context is met again
  - *Attend Raphaël Colin's presentation tomorrow morning!*

Context-adapted multi-versioning of computation kernels

- Our solution
  - Seeking the best performing code versions for observed actual contexts, on-the-fly
    - By testing different optimizing transformations
    - By simulating real execution samples
    - On separated nodes/cores
  - While the main code is running
  - Branching the main code on a better performing version
    - when ready
    - when the same context is met again
  - *Attend Raphaël Colin's presentation tomorrow morning!*

- Opportunities
  - Apollo:
    - A speculative parallelization platform based on Clang-LLVM and developed in the last years by team Camus
    - Currently extended to support runtime multi-versioning

Context-adapted multi-versioning of computation kernels

- Our solution
  - Seeking the best performing code versions for observed actual contexts, on-the-fly
    - By testing different optimizing transformations
    - By simulating real execution samples
    - On separated nodes/cores
  - While the main code is running
  - Branching the main code on a better performing version
    - when ready
    - when the same context is met again
  - *Attend Raphaël Colin's presentation tomorrow morning!*

- Opportunities
  - Apollo:
    - A speculative parallelization platform based on Clang-LLVM and developed in the last years by team Camus
    - Currently extended to support runtime multi-versioning
  - *Attend Erwan Auer's presentation in next Session!*

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030

PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

Conclusion

Conclusion

- Automatic Specialization of Polyhedral Programs on Sparse Structures:
  Attend Alec Sadler's presentation in next Session!
- Next main steps:
  - Consolidate and generalize our results
  - Make our software implementations the most robust and reliable
  - Target GPU codes
- Longer term:
  - Many more useful automatic code optimizations should be made available to the HPC community
- WP2 staff
  - 1 PhD to be hired in 2025/2026 (task 2.1)
  - Erwan Auer's engineer contract extended by 2 years (starting Jan. 2026)

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030

PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

cea cnrs Inria

Thank You!