Transformation from DSL to Source Code for Multi-Physics Simulations with Flash-X

Johann Rudi

Argonne Scholar at Mathematics and Computer Science Division
Argonne National Laboratory

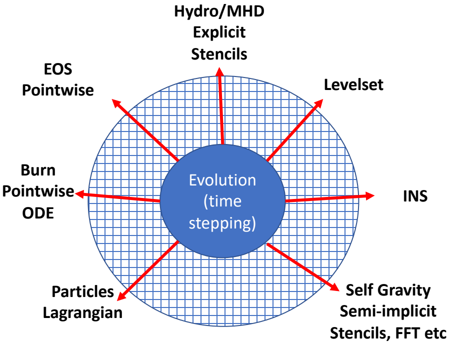
Fellow at Northwestern–Argonne Institute of Science and Engineering
Northwestern University

collaboration with
Mohamed Wahib (AIST/TokyoTech)
Jared O'Neal, Anshu Dubey (Argonne National Laboratory)

Pronouns: he/him/his/himself

Outline

Background: Multi-physics simulations with Flash-X[1]



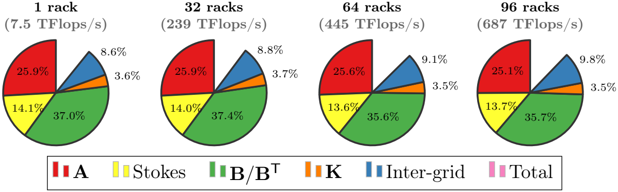
(Credit: A. Dubey)

- Flash-X's source code (FORTRAN & C++) is configured before compilation such that only desired physics units are included in the binary
- Physics units can be further decomposed into implementations for specific hardware platforms

[1] Flash-X is a new application code derived from FLASH

Performance of simulation relies on apply routines

Relative time spend in apply routines of a PDE solver[2]



Measured on IBM BlueGene/Q architecture (1 rack = 16,384 cores)

$\mathbf{A}$, Stokes, $\mathbf{B}/\mathbf{B}^\top$, $\mathbf{K}$ represent PDE operators.

Observe

- Highly optimized matrix-free apply routines dominate with $\sim$80 % of time
- Optimization of apply routines and its kernels is (highly) platform dependent
- Transition to new heterogeneous architectures, such as, single- or multi-GPU nodes from different vendors (Nvidia, AMD, Intel), involves substantial transformations and optimizations of code at many levels of abstraction

[2] Rudi et al. (2015), In: Proceedings of SC'15

Motivation and overview

Time stepping in Flash-X (and linear & nonlinear solvers in most other applications)

- Every iteration requires applying an operator of the underlying multi-physics PDE
- The operators are matrix-free apply routines
- Optimized kernels carry out computations on each grid cell

[3] presented by Tom Klosterman (SIAM PP22 MS79)

Motivation and overview

Time stepping in Flash-X (and linear & nonlinear solvers in most other applications)

- Every iteration requires applying an operator of the underlying multi-physics PDE
- The operators are matrix-free apply routines
- Optimized kernels carry out computations on each grid cell

Challenges arising due to heterogeneous platforms

- Kernels must be optimized for each platform → for now, leave this to skilled developers taking advantage of the Macro processor[3]
- Apply routines (loops over kernels) must be written for each platform → opportunity for developer-guided automation

[3] presented by Tom Klosterman (SIAM PP22 MS79)

Motivation and overview

Time stepping in Flash-X (and linear & nonlinear solvers in most other applications)

- Every iteration requires applying an operator of the underlying multi-physics PDE
- The operators are matrix-free apply routines
- Optimized kernels carry out computations on each grid cell

Challenges arising due to heterogeneous platforms

- Kernels must be optimized for each platform → for now, leave this to skilled developers taking advantage of the Macro processor[3]
- Apply routines (loops over kernels) must be written for each platform → opportunity for developer-guided automation

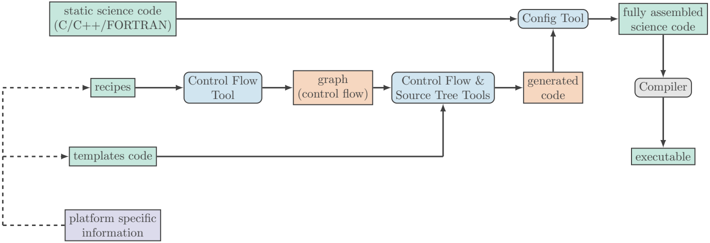
Propose: Automate generation of apply routines / driver code

- Recipes: Create a concise domain specific language (DSL)
- Orthogonalize: Separate domain knowledge and platform knowledge
- Code generation tool kit: Transform recipes to human-readable source code
- Hints: Users provide platform-dependent code optimizations hints, thus, tools remain simple and avoid exploring an intractable search space

[3] presented by Tom Klosterman (SIAM PP22 MS79)

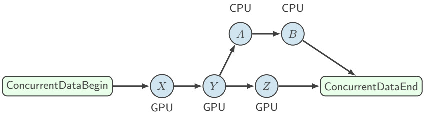
Outline

Generating code from recipes and code templates



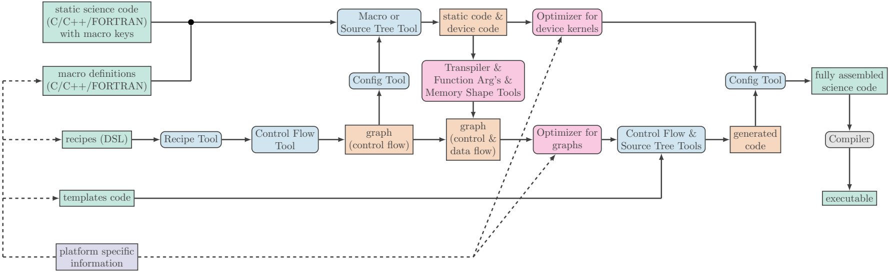
Chain of code generation tools (example above).

Example recipe (right) and resulting control flow graph (bottom).




```
# create new, empty graph
g    = ControlFlowGraph()

# add nodes to graph
dIn  = g.linkNode(ConcurrentDataBegin())(g.root)

wX   = g.linkNode(Work(name='X'))(dIn)
wY   = g.linkNode(Work(name='Y'))(wX)
wZ   = g.linkNode(Work(name='Z'))(wY)

wA   = g.linkNode(Work(name='A'))(wX)
wB   = g.linkNode(Work(name='B'))(wA)

dOut = g.linkNode(ConcurrentDataEnd())([wB,wZ])

# set node attributes
g.setNodeAttribute([wA, wB]    , 'device', 'CPU')
g.setNodeAttribute([wX, wY, wZ], 'device', 'GPU')
```

Overview of all components of code generation toolkit



- Input and output files shown as green boxes
- Intermediate outputs shown as orange boxes (can be inspected by humans)
- Code generation tools are blue boxes (currently in development)
- Optimization tools are pink boxes (future development)

Developers can select and combine tools

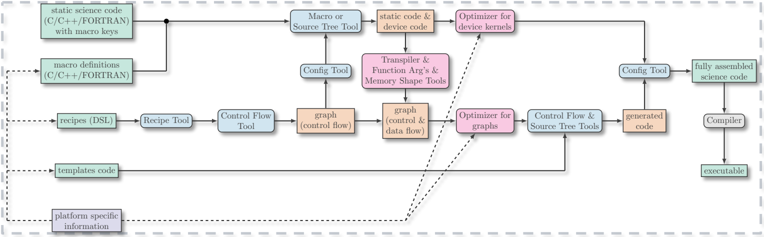


Fig: Example toolchain with full spectrum of tools

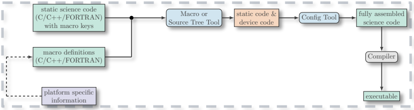


Fig: Example toolchain without recipe tools

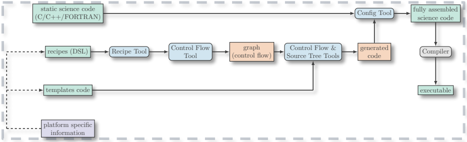


Fig: Example toolchain with recipe tools

Outline

Process control flow graphs into hierarchical graphs

Approach

1. Create a (flat) control flow graph where nodes (blue) represent computational work (i.e., kernels) and edges represent dependencies between kernels and data flow

Process control flow graphs into hierarchical graphs

Approach

1. Create a (flat) control flow graph where nodes (blue) represent computational work (i.e., kernels) and edges represent dependencies between kernels and data flow
2. Assign attributes to nodes representing which device it will execute on (e.g., CPU, GPU)

Process control flow graphs into hierarchical graphs

Approach

1. Create a (flat) control flow graph where nodes (blue) represent computational work (i.e., kernels) and edges represent dependencies between kernels and data flow
2. Assign attributes to nodes representing which device it will execute on (e.g., CPU, GPU)
3. Extract a hierarchical graph consisting of a quotient graph and subgraphs (orange) (which group kernels that will run on same device)

Definitions

Quotient graph: The nodes of a quotient graph Q of G form blocks of a partition of the nodes of G (Q contains orange circles, G contains blue circles).

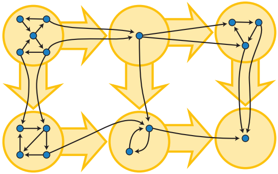


Illustration of a quotient graph
(Credit: wikipedia.org)

Subgraph: Nodes of G in the same block (orange circle) form a subgraph.

Process control flow graphs into hierarchical graphs

Approach

1. Create a (flat) control flow graph where nodes (blue) represent computational work (i.e., kernels) and edges represent dependencies between kernels and data flow
2. Assign attributes to nodes representing which device it will execute on (e.g., CPU, GPU)
3. Extract a hierarchical graph consisting of a quotient graph and subgraphs (orange) (which group kernels that will run on same device)
4. Generate device specific subroutines for aggregated device specific kernels (subgraphs)

Definitions

Quotient graph: The nodes of a quotient graph Q of G form blocks of a partition of the nodes of G (Q contains orange circles, G contains blue circles).

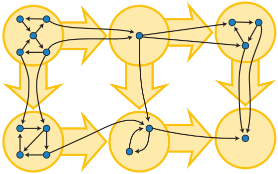


Illustration of a quotient graph
(Credit: wikipedia.org)

Subgraph: Nodes of G in the same block (orange circle) form a subgraph.

Process control flow graphs into hierarchical graphs

Approach

1. Create a (flat) control flow graph where nodes (blue) represent computational work (i.e., kernels) and edges represent dependencies between kernels and data flow
2. Assign attributes to nodes representing which device it will execute on (e.g., CPU, GPU)
3. Extract a hierarchical graph consisting of a quotient graph and subgraphs (orange) (which group kernels that will run on same device)
4. Generate device specific subroutines for aggregated device specific kernels (subgraphs)
5. Traversal of the coarse quotient graph yields the call sequence, thus the apply routine / driver code

Definitions

Quotient graph: The nodes of a quotient graph Q of G form blocks of a partition of the nodes of G (Q contains orange circles, G contains blue circles).

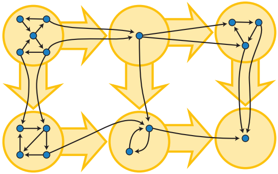
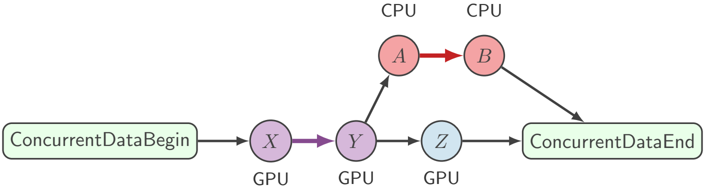


Illustration of a quotient graph (Credit: wikipedia.org)

Subgraph: Nodes of G in the same block (orange circle) form a subgraph.

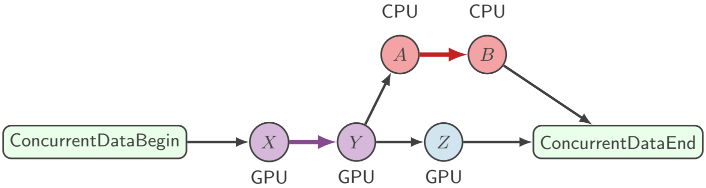
Previous example of a recipe and control flow graph

- Mark edges with a device change attribute (CPU-to-GPU or GPU-to-CPU) between any of the connected nodes
- Condensation of nodes that are connected by edges *without* device change



- Subgraph for CPU includes work/kernels A and B
- Subgraph for GPU includes work/kernels X and Y
- Z cannot be combined with X, Y because of concurrent edge $Y \to A$

Previous example: code generated from hierarchical control flow graph




```
// define task-function for GPU
void gpu_taskfn_00() {
  X_GPU();
  Y_GPU();
}

// define task-function for CPU
void cpu_taskfn_01() {
  A_CPU();
  B_CPU();
}
```

```
int main(int argc, char* argv[]) {
  { // begin concurrent data
    // execute task-function on GPU
    gpu_taskfn_00();
    { // begin concurrent work
      // execute work on GPU
      Y_GPU();
      // execute task-function on CPU
      cpu_taskfn_01();
    } // end concurrent work
  } // end concurrent data
  return 0;
}
```

Outline

Broader impact: Tools for performance portability

Tools are broadly applicable

- Do not assume a programming language (e.g., FORTRAN, C, ...) or parallelization framework (e.g., CUDA, HIP, OpenMP, OpenACC, ...)
- Do not try to infer optimizations, avoiding intractable search space and corner cases
- Ease burdens and increase productivity of developers working with scientific codes, in terms of code maintenance and platform migration
- Allow software communities to work together and separate concerns/tasks

Tools are flexible

- Each tool is simple and independent
- Multiple tools can be composed into toolchains or pipelines
- Developers can select tools they need and build their own portability framework (avoid one-solution-fits-all)

References I

Rudi, Johann, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W. J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas (2015). "An Extreme-Scale Implicit Solver for Complex PDEs: Highly Heterogeneous Flow in Earth's Mantle." In: *SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 5:1–5:12.

Rudi, Johann, Jared O'Neal, Mohamed Wahib, Anshu Dubey, and Klaus Weide (2021). *CodeFlow: Code Generation System for FLASH-X Orchestration Runtime*. Tech. rep. ANL-21/17. Argonne National Laboratory, Lemont, IL.