Performance Portability Without Relying on C++ Based Abstractions

PERMAVOST Workshop Orlando 2023
Anshu Dubey
Contributors: Tom Klosterman, Jared O’Neal, Johann Rudi, Mohamed Wahib, Klaus Weide
This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357
Starting Point in Extensible Software Architecture

- Building blocks of code
  - Hierarchy of granularity
  - Units, subunits, components

- Multiple alternative implementations
  - Null implementations of API
  - High degree of composability
  - High degree of customizability

- A tool that can arbitrate on what to include when
  - Self describing code components
# Config file for the gravity module. Available sub-modules:

# Constant  Spatially/temporally constant gravitational field
# PlanePar  1/r^2 field for a distant point source
# PointMass  1/r^2 field for an arbitrarily placed point source
# Poisson  Field for a self-gravitating matter distribution
# UserDefined  A user-defined field

**REQUIRES** Driver

**DEFAULT** Constant

**PPDEFINE** GRAVITY

**EXCLUSIVE** Constant PlanePar PointMass Poisson UserDefined

**PARAMETER** useGravity BOOLEAN TRUE
The Key to Composability

- Config files
- Setup
- Alternative Implementations
- Sub components

encoded metadata
Units I need
Units I don’t work with
State variables I need
AMR specific needs
Runtime parameters I want...
Platform Heterogeneity

Computation
- CPU
- GPU
- Other accelerators
- Other devices

Memory
- Cache hierarchy
- Device memory
- NVram
- Other types

Network
- Between nodes
- Within node
- With I/O
- Other types
Mechanisms Needed by the Code

Mechanisms to unify expression of computation
- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms to map work to computational targets
- Figuring out the map
  - Expression of dependencies
  - Cost models
  - Expressing the map

Mechanisms to move work and data to computational targets
- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data offnode

So what do we need?
- Abstractions layers
- Code transformation tools
- Data movement orchestrators
Philosophy of Design

- Let the code developer decide what should be done for optimization on a platform
  - Make it easy to have that happen without coding to metal

- Have a set of tools, each with limited functionality
  - Tools remain simple and easy to maintain by non-experts
  - Combination of tools provides a powerful solution

- Tools can permute and combine building blocks, do some code translation and compose a full application

- As far as possible tools also have building blocks
CGkit

- Generating Code from Recipes and code Templates

recipes (DSL) written by humans → Recipe Tool (DSL Parser) → Control Flow Tool → Source Tree Tool → source code for time stepping

templates (FORTRAN or C/C++)
CGkit

Generating Code from Recipes and code Templates

recipes (DSL) written by humans → Recipe Tool (DSL Parser) → Control Flow Tool → Source Tree Tool → source code for time stepping

templates (FORTRAN or C/C++)

Example recipe

```java
1. dIn = ConcurrentDataBegin();
2. aR = Action(routine='function_R')(dIn)
3. aS = Action(routine='function_S')(aR)
4. aT = Action(routine='function_T')(aS)
5. aX = Action(routine='function_X')(aS)
6. aY = Action(routine='function_Y')(aX)
7. aZ = Action(routine='function_Z'()){aT,aY})
8. dOut = ConcurrentDataEnd();
9. ConcurrentHardware(CPU={'actions': [aR,aS,aT,aZ]},
   GPU={'actions': [aX,aY]})
```

Resulting control flow graph

Orthogonal separation of concerns

- express dependencies
- express hardware mapping
Milhoja – domain specific runtime

- A Toolkit for Building Pipelines

**Distributors**
- Use block iterator
- Aggregate blocks if necessary
- Initiate asynchronous transfers if necessary
- Push blocks to other elements

**Thread Teams**
- Block
  - Task Fcn A (CPU)
  - threads = 3
  - Computation to apply to each data item

- Packet of Blocks
  - Task Fcn B (GPU)
  - threads = 7
  - Number of threads in team activated to apply action to data items

**Helpers**
- Initiate asynchronous transfers if necessary
- Translate data types

**Data Flow & Movement**

**Host-Side Thread Balancing**
Thread Team Configurations

- Expose Hierarchy of Parallelism

Task functions applied to each block:
- in order within a pipeline and
- concurrently across pipelines.

Allow for running concurrently independent actions from different physics units.
Macroprocessor – unify static code

- Mimic the functionality of template meta-programming
  - Single source code with specializations for variants

- Code in building blocks
  - That can be permuted and combined
  - Smaller building blocks can be fused into bigger ones for performance if needed
Modification in Configuration

Encoded metadata
Other components I need
Components I don’t work with
State variables I need
Runtime parameters I want ...

Express code with embedded macros
- Let macros have multiple alternative definitions
- Implement mechanism to select specific macro definition
- Implement mechanism to safely include more than one definition
- Allow inline, recursion and arguments in macros

Lowest granularity -- subroutine
Code Expressed with Keys

@M declare
@M directive1
@M loop_2d
   .... computation 1
   .... computation 2
@M endloop_2d_spl
@M directive2
@M loop_2d_spl
   .... computation 3
@M endloop_2d

Definitions for CPU
[declare]
   definition= cpu-specific declarations
[directive1]
   definition= !omp directive for cpu
[endloop_2d_spl]
   definition= @M endloop_2d
[directive2]
   definition= !omp directive2 for gpu
[loop_2d_spl]
   definition= @M loop_2d

Common definitions
[loop_2d]                   [endloop_2d]
   definition= do j=1,n      definition= end do
   do i=1,m                   end do

Definitions for GPU
[declare]
   definition= gpu-specific declarations
[directive1]
   definition= !omp directive1 for gpu
[endloop_2d_spl]
   definition= @M endloop_2d
[directive2]
   definition= !omp directive2 for gpu
[loop_2d_spl]
   definition= @M loop_2d

gpu-specific declarations
!!omp directive1 for GPU
do j=1,n
   do i=1,m
      .... computation 1
      .... computation 2
      .... computation 3
   end do
end do

!!omp directive2 for GPU
do j=1,n
   do i=1,m
      .... computation 3
   end do

Macros

Processor

CPU
specific definitions

Common definitions

GPU
specific definitions

Macro

Processor

Common definitions

CPU
specific declaration

!!omp directive for CPU
do j=1,n
   do i=1,m
   .... computation 1
   .... computation 2
   .... computation 3
   end do
end do

!!omp directive1 for GPU
do j=1,n
   do i=1,m
   .... computation 1
   .... computation 2
   end do
end do

!!omp directive2 for GPU
do j=1,n
   do i=1,m
      .... computation 3
   end do
end do
If(telescoping) then
call gcfill
  @M iter_begin
  @M hy_save_state_1blk
  @M hy_prepare_stages
do stage = 1,last_stage
    @M hy_set_limits
    @M hy_do_one_stage
    if(stage==last_stage)
      @M hy_update_state_1blk
      endif
  end do
  @M iter_end
else
  @M hy_save_global_state
  @M hy_prepare_stages
do stage = 1,last_stage
    Call Gcfill
    @M iter_begin
      @M hy_do_one_stage
      @M hy_update_global_state
    @M iter_end
  end do
end if

[hy_do_one_stage]
definition =
call hy_grav (@M hy_grav_args)
call hy_getFaceFlux (@M hy_ff_args)
call hy_addFluxes (@M hy_af_args)
call hy_updateSolution (@M hy_us_args)
call Eos

Subroutine hy_getFaceFlux (@M hy_ff_args)
  @M hy_ff_declare
do dir=1,NDIM
    @M hy_set_loop
    @M hy_start_loop
      @M hy_fill_tmp_blk
      @M hy_reconstruct
    @M hy_riemann
    @M hy_save_fluxes
    @M hy_end_loop
  end subroutine hy_getFaceFlux

Examples of CPU definitions
[hy_start_loop]
definition =
  @M loop_begin_2d(limits)
[hy_reconstruct]
definition =
call reconstruct (@M hy_rec_args)

Examples of GPU definitions
[hy_set_loop]
definition =
[hy_reconstruct]
definition =
  @M loop_begin_3d(limits)
call reconstruct (@M hy_rec_args)
  @M loop_end_3d
Library of templates for time-stepping
Setup tool (arbitrate)
Static physics code
• Componentized
• Encoded with macros
Platform specific information
Recipe for control flow in time stepping
Human in the loop
Library of runtime pipelines
Mechanism to map work to computational targets
• Figuring out the map
• Expressing the map
Unify expression of computation, setup tool and macroprocessor
• Alternative definitions/implementations
• Arbitration on which one to pick
CGKit
Source code for time stepping and Interface to Milhoja
Macroprocessor
Source code for physics operators
Setup tool (code assembly)
Fully assembled and configured source code
Compiler Linker
Executable
Milhoja (runtime library)
Mechanism to move work and data to targets
• Moving between devices
• Hiding latency of movement
The Toolchain

- Has been developed to minimize direct knowledge of Flash-X
- Some will be released as stand-alone tools
- Each one operates essentially independently
- Minimize the amount of recoding
  - In the code and in the tools
- A performance model to inform the optimizers
Porting to a new platform

- In an ideal world
  - Add to the library of runtime pipelines
  - Add to the library of recipes templates
  - Add to the knowledge base of the performance model

- In real world
  - Add variants for some solvers with alternative definitions of macros

- In the worst case
  - Develop new algorithms and add whole alternative implementation for some solvers