Rewrite of the COSMO Dynamical Core

Tobias Gysi¹, Peter Messmer², Tim Schröder², Oliver Fuhrer³, Carlos Osuna⁴, Xavier Lapillonne⁴, Will Sawyer⁵, Mauro Bianco⁵, Ugo Varetto⁵, David Müller¹ and Thomas Schulthess⁵,⁶,⁷

¹Supercomputing Systems AG; ²Nvidia; ³Swiss Federal Office of Meteorology and Climatology; ⁴Center for Climate Systems Modeling, ETH Zurich; ⁵Swiss National Supercomputing Center; ⁶Institute for Theoretical Physics, ETH Zurich; ⁷Computer Science and Mathematics Division, ORNL
Outline

- Project Overview
- C++ Stencil Library
- GPU Backend
- Performance
- Conclusions
Project Overview
• R&D services provider for the industry founded in 1993
• 75 highly skilled and motivated development engineers, architects and project managers, based in Zurich, Switzerland
• Our customer projects are our one and only focus & business

Industrial Controls & Sensors
**SCS**

super computing systems

- R&D services provider for the industry founded in 1993
- 75 highly skilled and motivated development engineers, architects and project managers, based in Zurich, Switzerland
- Our customer projects are our one and only focus & business
SCS
super computing systems

- R&D services provider for the industry founded in 1993
- 75 highly skilled and motivated development engineers, architects and project managers, based in Zurich, Switzerland
- Our customer projects are our one and only focus & business
super computing systems

- R&D services provider for the industry founded in 1993
- 75 highly skilled and motivated development engineers, architects and project managers, based in Zurich, Switzerland
- Our customer projects are our one and only focus & business
HP2C

• Part of the Swiss HPCN strategy (hardware / infrastructure / software)
• Strong focus on hybrid architectures for real world applications
• 10 Projects from different domains - http://www.hp2c.ch/
  • Cardiovascular simulation (EPFL)
  • Stellar explosions (University of Basel)
  • Quantum dynamics (University of Geneva)
  • …
• COSMO-CCLM
  1. Cloud resolving climate simulations (IPCC AR5)
  2. Adapt existing code (hybrid, I/O)
  3. Aggressive developments (different programming languages, GPUs)
What is COSMO?

- Consortium for Small-Scale Modeling
- Used by 7 weather services and O(50) universities and research institutes
Weather Prediction for Switzerland (MeteoSwiss)

**ECMWF**
- 2x per day
- 16 km lateral grid, 91 layers

**COSMO-7**
- 3x per day 72h forecast
- 6.6 km lateral grid, 60 layers

**COSMO-2**
- 8x per day 24h forecast
- 2.2 km lateral grid, 60 layers

Source: MeteoSwiss
Why Improving COSMO?

• High CPU usage @ CSCS (Swiss National Supercomputing Center)
  • 30 Mio CPU hours per year in total
  • About half of it on a dedicated machine for weather prediction
• There is a strong interest in improving the simulation quality
  • Higher resolution
  • Larger ensemble simulations
  • Increasing model complexity

→ Performance improvements are crucial!
Higher Resolution

Resolution is of key importance to increase simulation quality

2x resolution $\approx$ 10x computational cost

Source: Oliver Fuhrer
Ensemble Simulations

Run multiple simulations and gather statistics

Ensembles are of key importance for quantifying the simulation stability

N members ≈ Nx computational cost

Source: André Walser
COSMO Insights

- Partial differential equations on a structured grid (variables are velocity, temperature, pressure, humidity etc.)
- Explicit solves in the horizontal using finite difference stencils
- Implicit solves in the vertical doing one tri-diagonal solve per column

Due to implicit solves in the vertical we can work with longer time steps (2 km and not 60 m grid size is relevant)
COSMO Algorithmic Motifs

1. Tri-diagonal solves (Thomas algorithm)
   • Vertical K-direction pencils
   • Loop carried dependencies in K
   • Parallelization in horizontal IJ-plane

2. Finite difference stencil computations
   • Focus on horizontal IJ-plane accesses
   • No loop carried dependencies
Stencil Computations

- Stencils are kernels updating array elements according to a fixed pattern

\[
alap(i,j,k) = -4.0 \times \text{data}(i,j,k) + \text{data}(i+1,j,k) + \text{data}(i-1,j,k) + \text{data}(i,j+1,k) + \text{data}(i,j-1,k);
\]

- 5 flops per 6 memory accesses $\sim 0.1$ flops/Byte
- A Tesla 2090 delivers up to 665 Gflops / 150 GB/s $\sim 4.4$ flops/Byte

$\rightarrow$ Stencil codes are memory bandwidth bound rather than flop limited
What can be done?

- Adapt the code employing bandwidth saving strategies
  - Computation on-the-fly
  - Increase data locality
- Leverage the high memory bandwidth of GPUs

<table>
<thead>
<tr>
<th></th>
<th>Peak Performance</th>
<th>Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interlagos</td>
<td>294 Gflops</td>
<td>52 GB/s</td>
</tr>
<tr>
<td>Tesla 2090</td>
<td>665 Gflops</td>
<td>150 GB/s</td>
</tr>
</tbody>
</table>
COSMO Code Structure

This profile is representative for weather prediction as well as climate research (extensive chemical simulations can change the profiling)

→ Concentrate on Dynamics and Physics ("little" code, big portion of runtime)

Source: Oliver Fuhrer
COSMO Refactoring Approach

Physics
- Large group of developers
- Plug-in code from other models
- Less memory bandwidth bound
- Simpler stencils (K-dependencies)
- 20% of runtime

→ Keep source code (Fortran)
→ GPU port with directives (OpenACC)

Dynamics
- Small group of developers
- Memory bandwidth bound
- Complex stencils (IJK-dependencies)
- 60% of runtime

→ Aggressive rewrite in C++
→ Development of a stencil library
→ Still single source code multiple library back-ends for x86 / GPU

Rewriting the whole code was no short term option - rewriting critical parts was doable!
Project History

Feasibility - 2010
- 3 month feasibility
- Understand characteristics
- Hand tuning of 3 kernels

Rewrite - 2011
- CPU only stencil library
- Rewrite of the dynamics
- GPU backend

OPCODE - 2012
- Operational COSMO Demonstrator
- Integration of new dynamics
- Integration of new physics

>2x speedup but high code complexity → rewrite using a library hiding complexity

Rewrite is working → prove it is possible to glue everything together
C++ Stencil Library
Stencil Library Ideas

- Implement a stencil library using C++
  - 3D structured grid
  - Parallelization in horizontal IJ-plane (sequential loop in K due to Tri-diagonal solves)
  - Multi-node support using explicit halo exchanges (not part of the presentation)
- Abstract the hardware platform (CPU / GPU)
  - Single source
  - Adapt loop order and storage layout to the platform
- Hide / simplify complex and “ugly” optimizations
  - Leverage software caching
  - Blocking
Stencil Library Parallelization

- Shared memory parallelization
  - Support for 2 levels of parallelism
- Coarse grained parallelism
  - Split domain into blocks
  - Distribute blocks to CPU cores
  - No synchronization & consistency required
- Fine grained parallelism
  - Update block on a single core
  - Lightweight threads / vectors
  - Synchronization & consistency required

Similar to CUDA programming model (should be a good match for other platforms as well)
Stencil Code Concepts

- A stencil definition consists of 2 parts
  - **Loop-logic** defining the stencil application domain and order (green)
  - **Update-function** defining the update formula (blue)

```
DO k = 1, ke
  DO j = jstart, jend
    DO i = istart, iend
      lap(i,j,k) = data(i+1,j,k) + data(i-1,j,k) + data(i,j+1,k) + data(i,j-1,k) - 4.0 * data(i,j,k)
    ENDDO
  ENDDO
ENDDO
```

- Writing a stencil library is challenging (no straight-forward API as for matrix operations)
  - The loop logic is platform specific and needs to be inside the library
  - Different update-function for every stencil needs to be user code
Define Loop-Logic Using a Domain Specific Language (DSL)

- C++ allows to define embedded domain specific languages using the type system / template meta programming
  - Code is written as type
  - At compile-time this type is translated in a sequence of operations (compilation)
  - The operation objects are instantiated (code generation)
- We use this approach to generate the platform dependent loop-logic
enum { data, lap };

template<
typename TEnv>
struct LapStage
{
    STENCIL_STAGE(TEnv)

    STAGE_PARAMETER(FullDomain, data)
    STAGE_PARAMETER(FullDomain, lap)

static void Do(Context ctx, FullDomain)
{
    ctx[lap::Center()] =
    -4.0 * ctx[data::Center()] +
    ctx[data::At(iplus1)] +
    ctx[data::At(iminus1)] +
    ctx[data::At(jplus1)] +
    ctx[data::At(jminus1)];
}
};

IJKRealField lapfield, datafield;
Stencil stencil;

StencilCompiler::Build(
    stencil,
    "Example",
    calculationDomainSize,
    StencilConfiguration<Real, BlockSize<32, 4>>(),
    pack_parameters(
        Param<lap, cInOut>(lapfield),
        Param<data, cIn>(datafield)
    ),
    concatenate_sweeps(
        define_sweep<KLoopFullDomain>(
            define_stages(
                StencilStage<LapStage, IJRange<cComplete,0,0,0,0>>()
            )
        )
    )
);

stencil.Apply();

Update-function

Stencil Setup

DO k = 1, ke
    DO j = jstart, jend
        DO i = istart, iend
        
        lap(i,j,k) = data(i+1,j,k) + ...
    
ENDDO
ENDDO
ENDDO
Stencil Library Concepts - Parameter Handling

// define the parameter names
class { data, lap };

// define parameter fields
IJKRealField lapfield, datafield;

// pack parameters
pack_parameters(
    Param<lap, cInOut>(lapfield),
    Param<data, cIn>(datafield)
)

// define the parameter names
class { data, lap };

// access demonstration stage
template<typename TEnv>
struct AccessDemoStage
{
    STENCIL_STAGE(TEnv)
    STAGE_PARAMETER(FullDomain, lap)
    STAGE_PARAMETER(FullDomain, data)

    static void Do(Context ctx, FullDomain)
    {
        ctx[data::Center()] = 1.0;
        ctx[lap::At(jplus1)] = 2.0;
        ctx[lap::At(Offset<1,2,0>())] = 3.0;
    }
};

Context

lap ↔ lapfield
data ↔ datafield

Enum associates parameters

Create Context object linking enum and parameter field

Access parameters via Context operator[]()
Stencil Library Concepts - Loop Definition

// DSEL loop definition
concatenate_sweeps(
    define_sweep<KLoopFullDomain>(
        define_stages(
            StencilStage<StageA, IJRange<cComplete,-1,1,-1,1> >(),
            StencilStage<StageB, IJRange<cComplete,0,0,0,0> >()
        ),
    ),
    define_sweep<KLoopFullDomain>(
        define_stages(
            StencilStage<StageC, IJRange<cComplete,0,0,0,0> >()
        )
    )
)

Sweeps are loops over the full domain (sequential loop over K)

Stages are parallel “loops” over IJ (horizontal planes)

! Fortran equivalent, first sweep
DO k = 1, ke
    DO j = jstart-1, jend+1
        DO i = istart-1, iend+1
            ! StageA
            ENDDO
        ENDDO
        DO j = jstart, jend
            DO i = istart, iend
                ! StageB
                ENDDO
            ENDDO
    ENDDO
! second sweep
DO k = 1, ke
    DO j = jstart, jend
        DO i = istart, iend
            ! StageC
            ENDDO
        ENDDO
    ENDDO
Advanced Features – Vertical K-Domains

KMinimum (Center | KPlus1 | KMinus1 | …)

FlatCoordinates

KMaximum (Center | KPlus1 | KMinus1 | …)

Domains specify a subset of the stencil application domain

K

FullDomain

TerrainCoordinates
Advanced Features – Vertical K-Domain Specific Update

- The domain concept allows to define custom update functions at the K-boundary
- The second parameter of the Do() methods allows the library to call the right function using method overloading

```cpp
template<typename TEnv>
struct BoundaryStage
{
    STENCIL_STAGE(TEnv)
    STAGE_PARAMETER(FullDomain, data)

    static void Do(Context ctx, FullDomain)
    {
        ctx[data::At(kplus2)] = 1.0;
    }

    static void Do(Context ctx, KMaximumMinus1)
    {
        ctx[data::At(kplus1)] = 1.0; // k+1 instead of k+2
    }

    static void Do(Context ctx, KMaximumCenter)
    {
        ctx[data::Center()] = 1.0; // center instead of k+2
    }
};
```
Advanced Features - Functions

- Provide functionality used by multiple stencil stages
  - Finite difference operators
  - Tri-diagonal solver
- Increase code readability and maintainability
  - Less code duplication
  - Reduce stage complexity
- Many stencils differ only by a direction / offset
  - Implement them only once and pass direction / offset as parameter (e.g. upwind advection scheme works in I and J direction)
Advanced Features - Buffers

- Buffers are temporary data fields valid during one stencil execution
- Buffers are used to communicate results between stages / sweeps
- Buffers have an optimized storage format and should be preferred over manually defined intermediate fields
- Buffers could be placed in registers / shared memory on a GPU

```cpp
StencilCompiler::Build(
    stencil,
    /* name, domain size, config */
    pack_parameters(
        Param<flx, cInOut>(flux),
        Param<data, cIn>(pressure)
    ),
    define_buffers(
        StencilBuffer<lap, Real, IJKColumn<KRangeFullDomain> >()
    ),
    concatenate_sweeps(
        define_sweep<KLoopFullDomain>(
            define_stages(
                StencilStage<LapStage, IJRange<cIndented,0,1,0,1> >(),
                StencilStage<FluxStage, IJRange<cComplete,0,0,0,0> >()
            )
        )
    )
);```

- Buffers are temporary data fields valid during one stencil execution
- Buffers are used to communicate results between stages / sweeps
- Buffers have an optimized storage format and should be preferred over manually defined intermediate fields
- Buffers could be placed in registers / shared memory on a GPU
Stencil Library Summary

- The library manages to separate loop-logic and update function
- The loop logic is defined using a domain specific language
  - The language abstracts the parallelization / execution order of the update function
- A single source code compiles for multiple platforms
  - Currently there are efficient back-ends for GPU and CPU

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Order (Fortran)</td>
<td>KIJ</td>
<td>IJK</td>
</tr>
<tr>
<td>Parallelization</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
</tbody>
</table>

→ One could add any platform providing a standard compliant C++ compiler
Libraries and Tools

• Boost MPL library
  • Meta programming data structures and algorithms
  • C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond by David Abrahams and Aleksey Gurtovoy
• Googletest unit test library
• CMake build system
  • Platform independent (Windows / Unix)
  • CUDA support
GPU Backend
GPU Backend Overview

• Storage
  • IJK storage order
    (planes contiguous in memory)
  • Coalesced reads in I direction
    (first horizontal direction)

• Parallelization
  • Parallelize in IJ dimension
    (blocks are mapped to CUDA blocks)
  • Block boundary elements are updated using additional warps

• Data field indexing
  • Pointers and strides in shared memory
  • Indexes in registers

CUDA grid splitting IJ plane into blocks

Block with boundary (use additional boundary warps)
Software Managed Cache

- Software managed caching means that the library user defines manually which data fields shall be cached
- Experimental support for software managed cache
  - Support for caching of vertical K-direction accesses in registers
  - Caching of horizontal IJ-plane accesses in shared memory is planned

```c
define_caches(
    KCache<data, cFill, KWindow<0,1>, KRangeFullDomain>()
),
define_stages(
    StencilStage<CacheStage, IJRange<cComplete,0,0,0,0> >()
)
```
Software Managed Cache Performance

Software managed caching is crucial for the GPU performance (currently kernels with IJ / horizontal dependencies cannot benefit as a shared memory implementation is missing)

KCache Speedup

- Total: 1.83
- FastWaveWTPP
- VerticalAdvectionUVW
- AdvectionPDZ
- VerticalAdvectionTPP
- VerticalDiffusionTracers2
- VerticalDiffusionUVW
- VerticalDiffusionT
Performance
Hardware Setup

- Performance measurements were done on Todi
  - Cray XK6 running @ CSCS
  - AMD Interlagos (16-Cores)
  - Tesla X2090 (512-Cores)
Application Performance

- The CPU backend is 1.6x – 1.7x faster than the standard COSMO dynamics
  - Note that we use a storage layout different from COSMO
  - The CPU version will not be able to realize the full speedup due to transposes
- The GPU backend is 2.9x faster than the CPU backend
- A 4.7x speedup is achieved going from COSMO to HP2C dynamics running on GPU

Application Speedup - Interlagos 16 Core vs. Tesla X2090

<table>
<thead>
<tr>
<th></th>
<th>HP2C dynamics (GPU)</th>
<th>HP2C dynamics (CPU)</th>
<th>COSMO dynamics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speedup</td>
<td>4.73</td>
<td>2.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusions
Conclusions

• Initial rewrite complete
  • Regression test against Fortran code (same algorithms)
  • Concentrate on features used in production
• It is possible to develop a performance portable stencil library using C++
  • Libraries like Boost MPL help dealing with the complexity
  • Designing such a library is hard opt for an iterative development approach
• It is possible to work with CUDA using complex C++ constructs
  • Note that the tools are not yet on the level of GNU or MSVC
• It is worth considering a rewrite in spite of large code size / complexity
  • Rewriting parts and combination with OpenACC might be an option
Tobias Gysi - Supercomputing Systems AG
tobias.gysi@scs.ch
Backup Slides
Advanced Features - Functions

Functions are similar to stages except for the parameter definition

```cpp
template<typename TEnv>
struct SymmetricSum
{
    STENCIL_FUNCTION(TEnv)
    FUNCTION_PARAMETER(0, offset)
    FUNCTION_PARAMETER(1, data)

    static T Do(Context ctx, FullDomain)
    {
        typedef Offset<
            -offset::I::value,
            -offset::J::value,
            -offset::K::value>
            SymOffset;

        return
        ctx[data::At(offset())] +
        ctx[data::At(SymOffset())];
    }
};
```

```cpp
template<typename TEnv>
struct LapStage
{
    STENCIL_STAGE(TEnv)
    STAGE_PARAMETER(FullDomain, data)
    STAGE_PARAMETER(FullDomain, lap)
    USING(SymmetricSum)

    static void Do(Context ctx, FullDomain)
    {
        ctx[lap::Center()] =
        -4.0 * ctx[data::Center()] +
        ctx[SymmetricSum::With(iplus1, data::Center())] +
        ctx[SymmetricSum::With(jplus1, data::Center())];
    }
};
```

Functions are executed using the context operator[]( ). Note that SymmetricSum works for I and J direction (no code duplication)
GPU Buffers

- Buffers are data fields used to store intermediate results of a stencil computation
  - Every block shall work on a private memory area (no conflicts with unsynchronized neighbor blocks)
  - In order to increase cache efficiency we would like to store information needed by a block as local as possible

→ GPU buffers use blocked storage
  (slightly less over fetch as no data from neighbor block is cached)
How to Provide Multiple Back-ends?

- Switch between back-ends using preprocessor switch
  - Only one back-end active at once (compiler availability, no naming conflicts)
  - Use switch only for the library API definition
- Inside the library
  - Share code between back-ends if possible
  - Interfaces with backend specific implementations (compile-time polymorphism)

```c
#ifdef __CUDA_BACKEND__
typedef DataFieldCUDA<Real,
                     StorageFormat<IJBoundary, StorageOrder::KJI, CUDAAlignment>> > IJKRealField;
#else
  
typedef DataFieldOpenMP<Real,
                           StorageFormat<IJBoundary, StorageOrder::JIK, OpenMPAlignment>> > IJKRealField;
#endif
```
Update-Function Definition

- Define the update-function using a functor
  - Function object
  - Static methods only
  - Do() method implements update logic
- Pass parameters using a context object
  - Containing parameter tuple

Update-functions are called StencilStages

```cpp
// define the Laplace stencil stage
template<typename TEnv>
struct LapStage {
    STENCIL_STAGE(TEnv)
    STAGE_PARAMETER(FullDomain, data)
    STAGE_PARAMETER(FullDomain, lap)

    static void Do(Context ctx, FullDomain) {
        ctx[lap::Center()] =
            -4.0 * ctx[data::Center()] +
            ctx[data::At(iplus1)] +
            ctx[data::At(iminus1)] +
            ctx[data::At(jplus1)] +
            ctx[data::At(jminus1)];
    }
};
```
Stencil-Definition

- Data field definition
- Stencil object definition
  - Pass parameters
  - DSL loop-logic definition (details follow later on)
- Stencil application
  - Call stage Do() method

```cpp
// data field definition
IJKRealField lapfield, datafield;
// stencil object definition
Stencil stencil;
StencilCompiler::Build(
    stencil,
    ..., // additional parameters neglected for simplicity
    pack_parameters(
        Param<lap, cInOut>(lapfield),
        Param<data, cIn>(datafield)
    ),
    concatenate_sweeps(
        define_sweep<KLoopFullDomain>(
            define_stages(
                StencilStage<LapStage, IJRange<cComplete,0,0,0,0> >()
            )
        )
    )
);
// stencil application
for(int step=0; step<NUM_OF_STEPS; ++step) stencil.Apply();
```
CUDA and C++

- CUDA findings
  - C++ features used for meta programming work
  - Deeply nested function calls / data structures work
  - Boost MPL works

- CUDA pitfalls
  - `__device__ / __host__` are very tedious to use e.g. for classes which shall work on host and device (some classes are implemented separately for device and host using the preprocessor)
  - Debugging complex code can be very slow as you cannot debug the release build
  - There were stability problems when working with the visual profiler (the command line profiler worked fine)
HP2C Dycore Kernels

- Fast Wave solver
- Advection
  - 5th order advection
  - Bott 2 advection (cri implementation in z direction)
- Implicit vertical diffusion and advection (slow tendencies)
- Horizontal diffusion
- Support for pollen and COSMO art
- Initial integration into COSMO done (Fortran wrapper)