Porting COSMO to Hybrid Architectures

T. Gysi¹, O. Fuhrer², C. Osuna³, X. Lapillonne³, T. Diamanti³, B. Cumming⁴, T. Schroeder⁵, P. Messmer⁵, T. Schulthess⁴,⁶,⁷

[¹] Supercomputing Systems AG, [²] Swiss Federal Office of Meteorology and Climatology, [³] Center for Climate Systems Modeling, ETH Zurich, [⁴] Swiss National Supercomputing Center (CSCS), [⁵] NVIDIA Corp., [⁶] Institute for Theoretical Physics, ETH Zurich, [⁷] Computer Science and Mathematics Division, ORNL

Programming weather, climate and earth-system models on heterogeneous multi-core platforms
Sept 12 - 13, 2012, NCAR, Boulder, CO
Why Improving COSMO?

- COSMO: Consortium for Small-Scale Modeling
- Used by 7 weather services and O(50) universities and research institutes

- High CPU usage @ CSCS (Swiss National Supercomputing Center)
  - 30 Mio CPU hours in total
  - ~ 50% on a dedicated machine
- Strong desire for improved simulation quality
  - Higher resolution
  - Larger ensemble simulations
  - Increasing model complexity

→ Performance improvements are critical!
Need for Higher Resolution in Switzerland

Resolution is of key importance to increase simulation quality.

- 2x resolution ≈ 10x computational cost

dx = 2km  
Reality

dx = 1km
COSMO port to hybrid architectures is part of HP2C Project

- 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/](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)
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
Requirements for a Portable Stencil Library

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

- **Loop-logic:** Defines stencil application domain
  - Platform dependent
- **Update-function:** Expression evaluated at each location
  - Platform independent

=> Treat two components separately
Loop-Logic expressed in Domain Specific Language (DSL)

- Define embedded domain specific language in C++ using type system/template metaprogramming
  - Code is written as type
  - Type is translated into sequence of operations (DSL compilation) at compile time
  - Operation objects (“code fragments”) are inserted at compile time (code generation)
- Pre-packaged loop objects for CPU and GPU
- We use this approach to generate the platform dependent loop-logic
Putting it all together..

```
IJKRealField laplacian, pressure;
Stencil stencil;
StencilCompiler::Build(
  stencil,
  "Example",
  calculationDomainSize,
  StencilConfiguration<Real, BlockSize<32,4>>(),
  ...
);

define_sweep<KLoopFullDomain>(
  define_stages(
    StencilStage<LapStage, IJBoundary<cComplete,0,0,0,0>>()
  )
);

stencil.Apply();

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
```

```
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(jplus1)] +
      ctx[data::At(jminus1)] +
      ctx[data::At(iplus1)] +
    ctx[data::At(iminus1)];
  }
};
```
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

~ CUDA programming model
(should be a good match for other platforms as well)
GPU Backend Overview

- **Storage**
  - IJK storage order
  - Coalesced reads in I direction

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

- **Data field indexing**
  - Store data field pointers and strides in shared memory
  - Store indexes in registers
HP2C Dycore Performance

- **CPU / OpenMP Backend**
  - Factor 1.6x - 1.7x faster than the standard COSMO implementation
  - Here: no SSE support (expect another 10% ~30% improvement)

- **GPU / CUDA backend**
  - Tesla M2090 (150 GB/s with ECC enabled) is roughly a factor 2.6x faster than Interlagos (16-Core Opteron CPU with 52 GB/s)
  - Ongoing performance optimization

---

**Speedup Diagram**

- GPU HP2C (Tesla C2090)
- CPU HP2C (Interlagos)
- CPU Fortran (Interlagos)
Acceleration of Physical Parametrizations: Current State

- Parametrizations: processes not described by the dynamics, such as radiation or turbulence. Account for about 20 to 25% of total runtime

- GPU versions of the parametrizations have been implemented in COSMO

- Currently implemented and tested physics:
  - Microphysics (Reinhardt and Seifert, 2006)
  - Radiation (Ritter and Geleyn, 1992)
  - Turbulence (Raschendorfer, 2001)
  - Soil (Heise, 1991)

- Account for 90-95% of physics in typical COSMO-2 run

- Only options for operational runs are supported
  - Unsupported features have been documented
Directives/Compiler choices for OPCODE

- **OpenAcc**: Open standard, supported by 3 compiler vendors PGI, Cray, Caps
  - Directives of choice for final OPCODE version
  - **CAPS**: Future approach

- **PGI proprietary**:
  - Enabled port of all kernels (some workarounds required)
  - **First implementation of the physics**
  - Translation to OpenAcc relatively straight forward

=> Testing code with different compilers can be very helpful!

```fortran
$acc parallel loop vector_length(N)
do i=1,N
  a(i)=b(i)+c(i)
end do
$acc end parallel loop
```
Implementation in COSMO

- Change to block data structure inside the physics
  - \( f(i,j,k) \rightarrow f_b(nproma,ke) \), with \( nproma = istartpar \times iendpar / nblock \).
  - \( nblock=1 \) for GPU run

- Physics loop restructured to iterate over blocks

```
transfer from CPU to GPU (ijk data f(i,j,k))
!start block loop
  do ib=1,nblock
    call copy_to_block
    call organize_gscp
    call organize_radiation
    call organize_turbulence
    call organize_soil
    call copy_back
  end do
transfer back GPU to CPU (ijk data f(i,j,k))
```

Required data on the GPU
- All operations on grid data computed on the GPU
- Physics timing region

inside physics scheme data is in block form
\( f_b(nproma,ke) \)
Porting Strategy for Parametrizations

Pencil Parallelization: horizontal direction, 1 thread per vertical column

- Most loop structures unchanged, one only adds directives
- In some parts: loop restructuring to reduce kernel call overheads, and profit from cache reuse.
- Remove NEC vector-optimization.

Remove Fortran automatic arrays in subroutines which are often called (to avoid call to cudamalloc)

Data regions to avoid CPU-GPU transfer

Use profiler to target specific parts which need further optimization: reduce memory usage, replace intermediate arrays with scalars ...
GPU/CPU comparison

CPU - original physics
16 cores (interlagos) - using MPI

- Time physics: 42.4 s
  (average time, without communication)

- Benchmark subdomain 128x112x60, 1h simulation with microphysics, radiation, turbulence and soil

- CPU-GPU results agree within roundoff error

GPU - block physics
1 core + 1 GPU (X2090)

- Time physics: 12.5 s
Test subdomain 128x112x60, 1h simulation
Currently running the block physics code on CPU (i.e. ignoring directives) is slower (total physics = 53 s). This is due to the GPU-loop reordering optimizations, not to the block structure. Having a single source code that runs efficiently on x86-CPU (i.e. excluding NEC) and GPU will require further work.

The GPU code runs 7% faster on CASTOR (C2090)
Summary and next steps

- Dycore ported using portable stencil library and DESL
- Physics ported using directives

- Dycore speedup of ~4x vs original code
- Physics speedup of ~3.4x vs original code

- Dycore speedup for relevant domain sizes retained for K20/SandyBridge

- Ongoing: Combining Dycore, Physics and Messaging Layer
Thank you!

pmessmer@nvidia.com
Need for Higher Resolution in Switzerland

Resolution is of key importance to increase simulation quality

*2x resolution ≈ 10x computational cost*