C2SM
Center for Climate Systems Modeling

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Adapting Numerical Weather Prediction codes to heterogeneous architectures: porting the COSMO model to GPUs

O. Fuhrer, T. Gysi, **X. Lapillonne**, C. Osuna, T. Dimanti, T. Schultess and the HP2C team

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

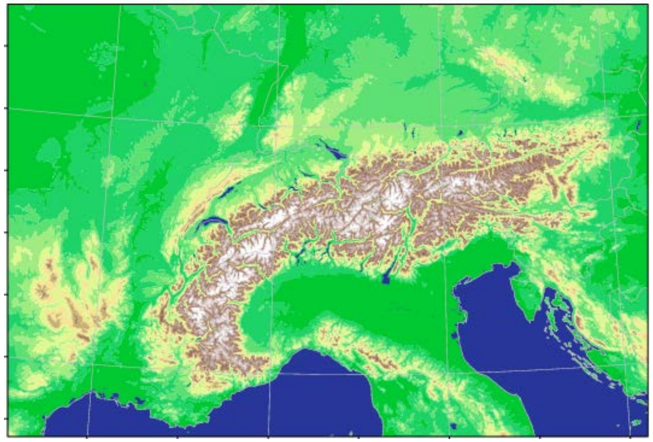
Eidgenössisches Departement des Innern EDI
Bundesamt für Meteorologie und Klimatologie MeteoSchweiz

HP2C

scs
super computing systems

CSCS
Swiss National Supercomputing Centre

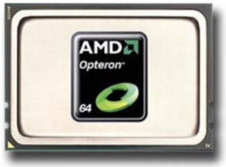
The COSMO model



- **Limited-area** model
- Run operationally by several National Weather Service within the Consortium for Small Scale Modelling: Germany, Switzerland, Italy, Poland, Greece, Rumania, Russia.
- Used for climate research in several academics institutions

Why using Graphical Processor Units ?

- Higher peak performance at lower cost / power consumption
- High memory bandwidth

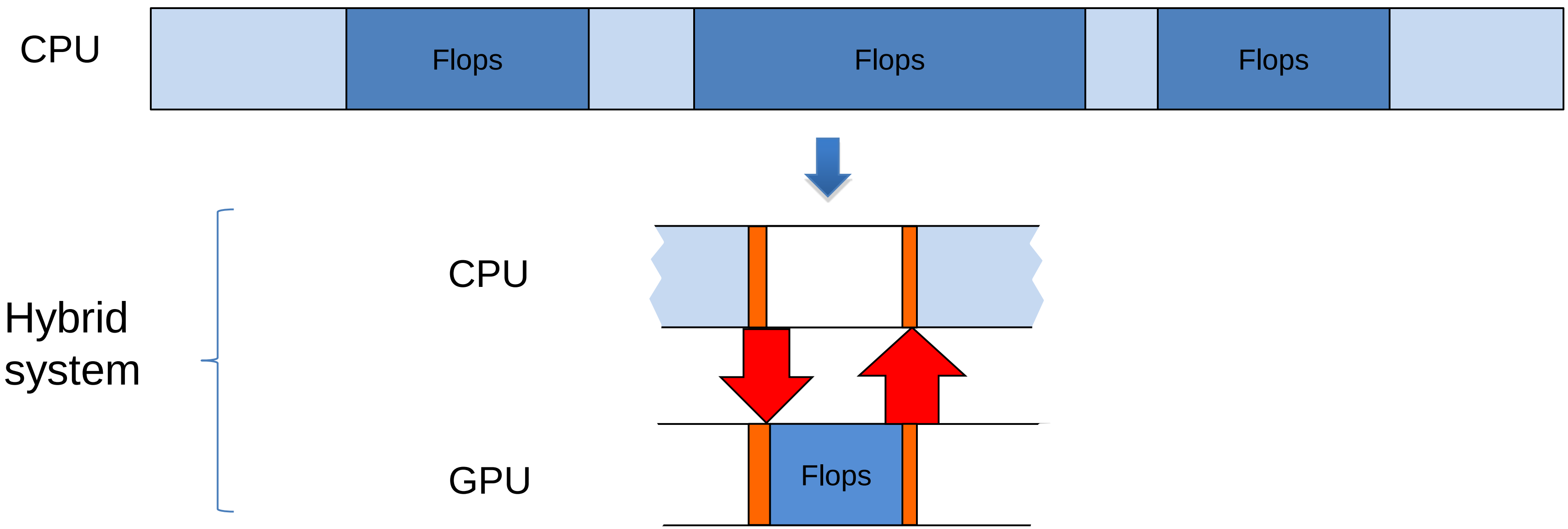


	Cores	Freq. (GHz)	Peak Perf. S.P. (GFLOPs)	Peak Perf. D.P. (GFLOPs)	Memory Bandwith (GB/sec)	Power Cons. (W)
CPU: AMD Opteron (Interlagos)	16	2.1	268	134	57	115
GPU: Fermi X2090	512	1.3	1330	665	155	225

X 5 X 3

Using GPUs : the accelerator approach

- CPU and GPU have different memories



- Most intensive parts are ported to GPU, data is copied back and forth between the GPU and the CPU between each accelerated part.

What does this mean for NWP application ?

- Low FLOP count per load/store (stencil computation)
- Example with COSMO-2 (operational configuration at MeteoSwiss) :

*

Part	Time/Δt
Dynamics	**172 ms**
Physics	**36 ms**
Total	**253 ms**

vs

§

Transfer of ten prognostic variables

118 ms

CPU-GPU data transfer time is large with respect to computation time:

Accelerator approach might not be optimal

* CPU measurements: Cray XT6, Magny-Cours, 45 nodes, COSMO-2

§ GPU measurements: PCIe 8x, HP SL390, 2 GB/s one way, 8 sub-domains

Our strategy : full GPU port

- **All code which uses grid data fields at every time step is ported to GPU**

Input/Setup → keep on CPU / copy to GPU

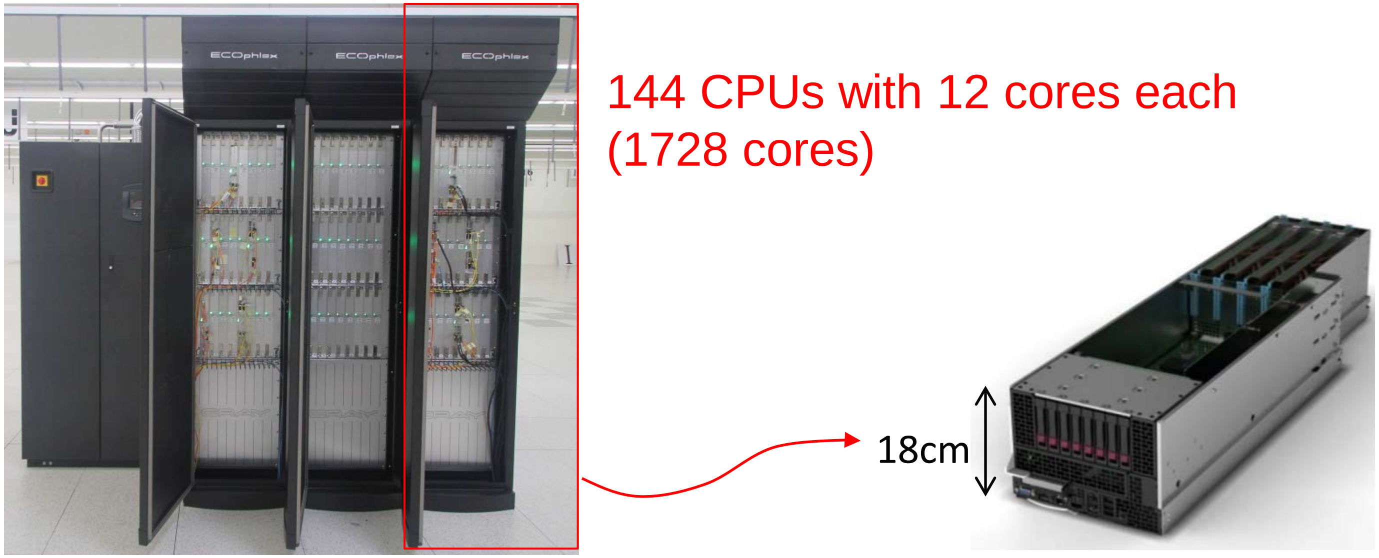
Δt

- **Initialize timestep** → OpenACC directives
- **Physics** → OpenACC directives
- **Dynamics** → C++ rewrite (uses CUDA)
- **Assimilation** → OpenACC directive (part on CPU)
- **Halo-update** → GPU-GPU communication library (GCL)
- **Diagnostics** → OpenACC directives
- **Output** → keep on CPU / copy from GPU

Finalize

The HPCC OPCODE project

- Part of the Swiss High Performance High Productivity initiative
- Prototype implementation of the COSMO production suite of MeteoSwiss making aggressive use of GPU technology
- Same time-to-solution on substantially cheaper hardware:



1 cabinet Cray XE5

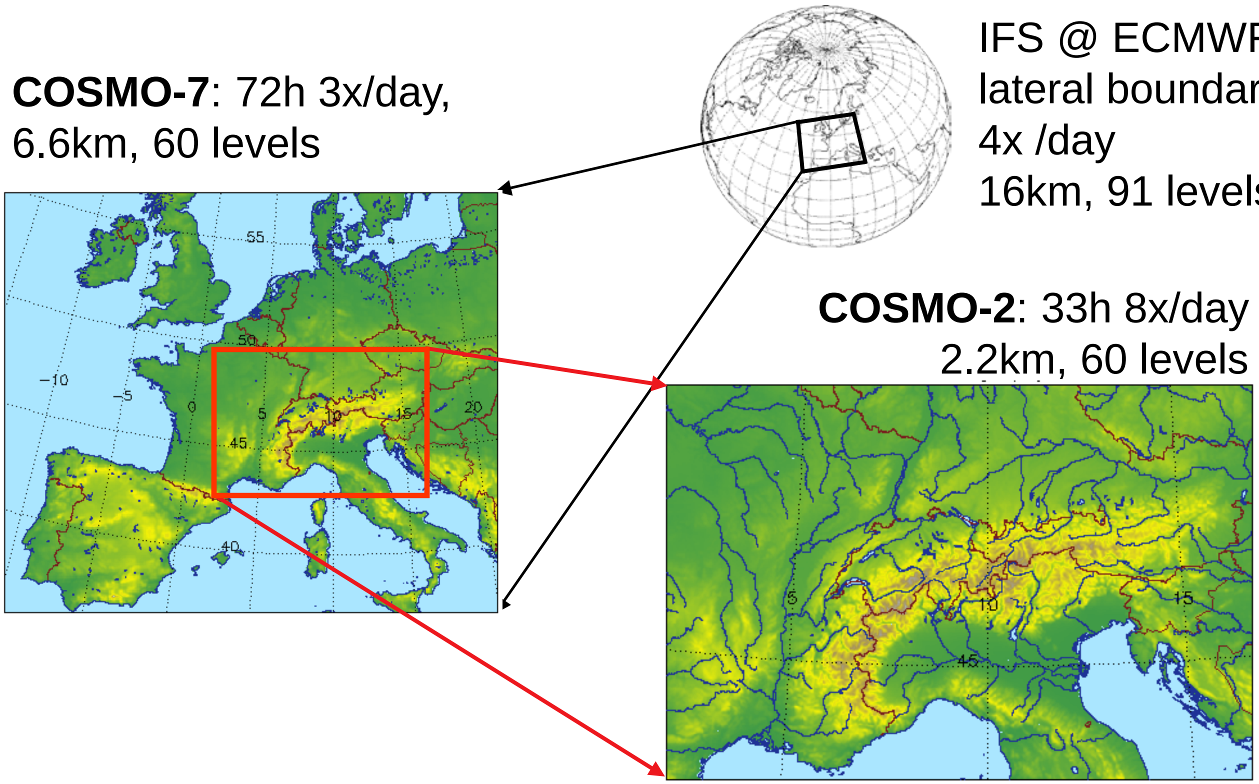
GPU based hardware

- OPCODE prototype code should be running by end 2012
- These developments are part of the priority project POMPA within the COSMO consortium : preoperational GPU-version of COSMO for 2014

MeteoSwiss COSMO-7 and COSMO-2

COSMO-7: 72h 3x/day,
6.6km, 60 levels

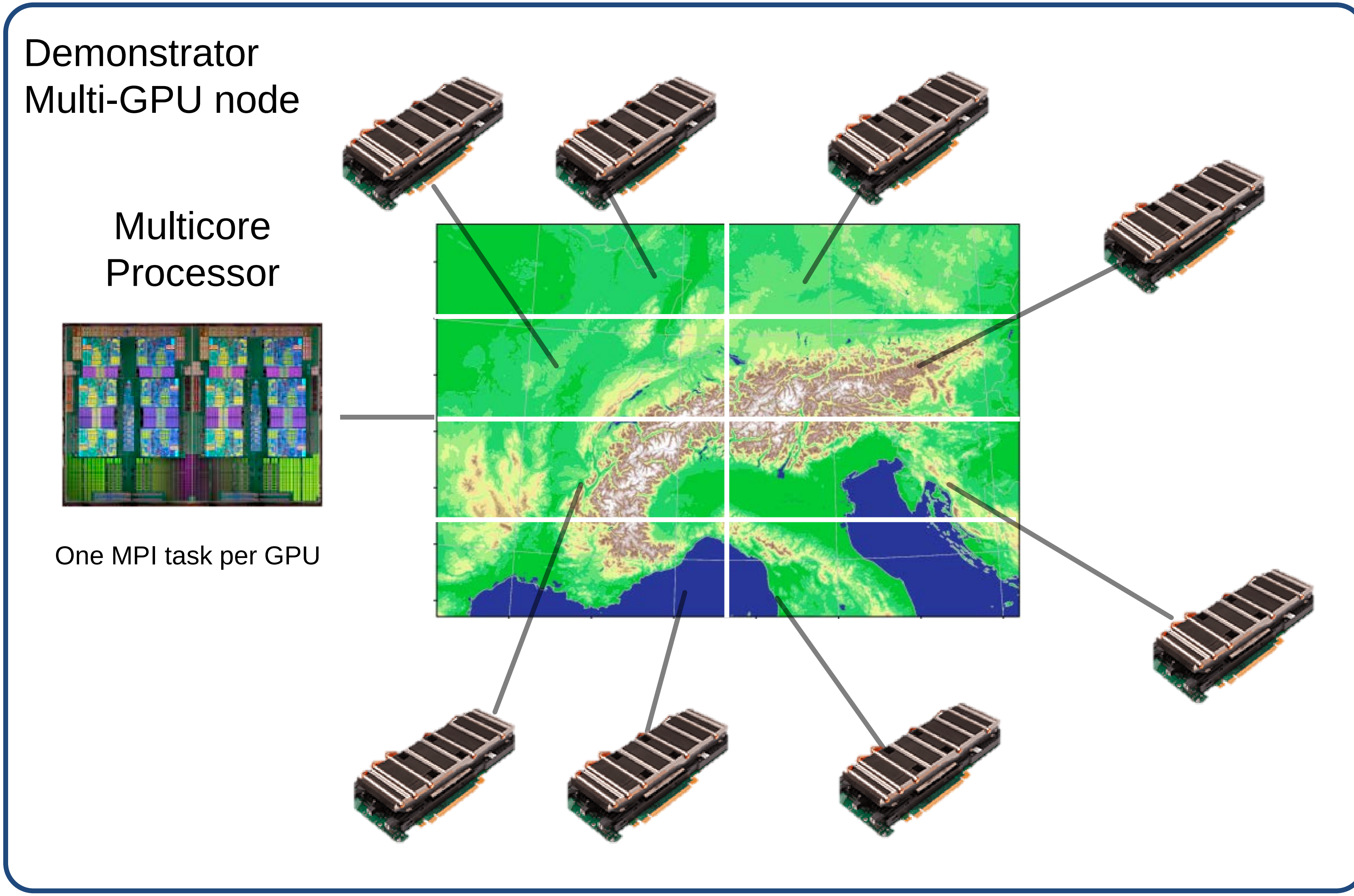
IFS @ ECMWF:
lateral boundaries
4x /day
16km, 91 levels



COSMO-2: 33h 8x/day
2.2km, 60 levels

520 x 350 x 60 grid points

COSMO on the demonstrator system



Dynamical core refactoring

Dynamics

- Small group of developers
- Memory bandwidth bound
- Complex stencils (IJK-dependencies)
- 60% of runtime
- 40 000 lines (18%)

→ Complete rewrite in C++

→ Development of a stencil library

→ Target architectures CPU (x86) and GPU

→ Could be extended to other architectures

→ Long term adaptation of the model

Communication library

- Requirement for multi-node communications that can be called from the new dynamical core.

→ New communication library (GCL)

→ Can be called from C++ and Fortran code

→ Can handle CPU-CPU and GPU-GPU communication

→ Allows overlap of communication and computation

Note : only single node results will be presented

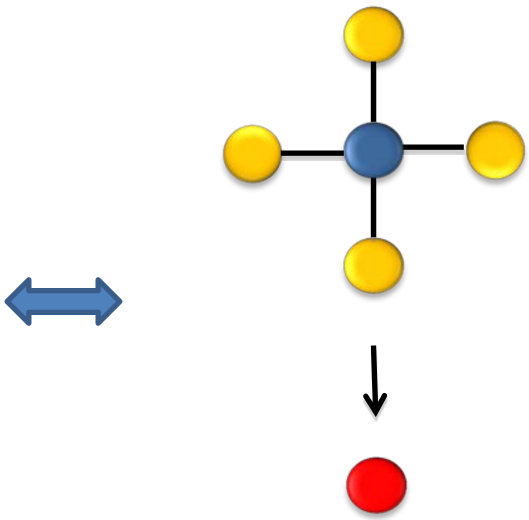
Stencil computations

- COSMO is using finite differences on a structured grid
- Stencil computation is the dominating algorithmic motif within the dycore

- Stencil Definition
- Kernel updating array elements according to a fixed access pattern

- Example 2D Laplacian

```
lap(i,j,k) = -4.0 * data(i,j,k) +
    data(i+1,j,k) + data(i-1,j,k) +
    data(i,j+1,k) + data(i,j-1,k);
```



Stencil library development

Motivation

- Provide a way to implement stencils in a platform independent way
- Hide complex/hardware dependent optimizations from the user

→ Single source code which is performance portable

Solution retained

- DSEL : Domain Specific Embedded Language
- C++ library using template meta programming
- Optimized back-ends for GPU and CPU

	CPU	**GPU**
Storage Order (Fortran)	KIJ	IJK
Parallelization	OpenMP	CUDA

Stencil code concepts

loop-logic

update-function / stencil

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

A stencil definition consists of 2 parts

- Loop-logic: Defines stencil application domain and execution order ➡ DSEL
- Update-function: Expression evaluated at every location ➡ USER

While the loop-logic is platform dependent the update-function is not
→ treat the two separately

Programming the new dycore

```
enum { data, lap };

template<typename TEnv>
struct Lap
{
  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)];
  }
};
```

Update-function

```
IJKRealField lapfield, datafield;
Stencil stencil;

StencilCompiler::Build(
  stencil,
  "Example",
  calculationDomainSize,
  StencilConfiguration<Real, BlockSize<32,4> >(),
  …
  define_sweep<KLoopFullDomain>(
    define_stages(
      StencilStage<Lap, IJRange<cComplete,0,0,0,0> >()
    )
  )
  …
);

for(int step = 0
{
  stencil.Apply(
}
```

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
```

Dynamics, single-node performance

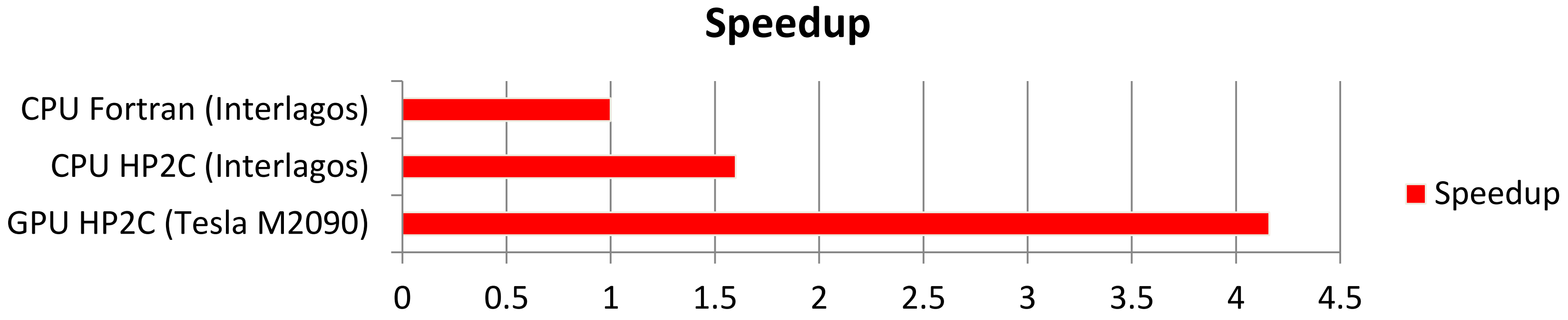
- **Test domain 128x128x60. CPU: 16 cores Interlagos CPU; GPU : X2090**

CPU / OpenMP Backend

- Factor 1.6x - 1.7x faster than the COSMO dycore
- No explicit use of vector instructions (10% up to 30% improvement)

GPU / CUDA backend

- Tesla M2090 (GPU with 150 GB/s memory bandwidth) is roughly a factor 2.6x faster than Interlagos (CPU with 52 GB/s memory bandwidth)
- Ongoing performance optimizations



Pros and Cons of the rewrite approach

Pros

•Performance and portability

•Better separation of implementation strategy and algorithm

•Single source code

•The library suggest / forces certain coding conventions and styles

•Flexibility to extend to other architecture

Cons/difficulties

•This is a big step with respect to the original Fortran code

- Can this be taken over by main developers of the dycore ? (workshop, knowledge transfer …)

•Adding support for new hardware platforms requires a deep understanding of the library implementation

Physics and data assimilation port to GPU

Physics
- Large group of developers
- Some code maybe shared with other model
- Less memory bandwidth bound
- Simpler stencils (K-dependencies)
- 20% of runtime
- 43 000 lines (19%)

→ GPU port with OpenACC directives
→ Optimization of the code to get optimal performance on GPU
→ Most routines have for the moment a GPU and CPU version

Data assimilation
- Very large code
- 83 000 lines (37%)
- 1 % of runtime

→ GPU port with OpenACC directives only for parts accessing multiple grid data field
→ No code optimization
→ Single source code
→ Some parts still computed on CPU

Directives / Compiler choices for OPCODE

```
!$acc parallel
 !$acc loop gang vector
do i=1,N
 a(i)=b(i)+c(i)
end do
!$acc end parallel
```

OpenAcc: Open standard, supported by 3 compiler vendors PGI, Cray, Caps

- Solution retained for OPCODE (for physics and assimilation)
- PGI : some remaining issues with the compiler
- Cray: The code can be compiled and run. Gives correct results
- CAPS: not investigated yet

- PGI proprietary:
  - First implementation of the physics
  - Translation to OpenAcc is not an issue

➡ Being able to test code with different compilers is essential

Implementation strategy with directives

- 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.

```
!$acc data present(a,c1,c2)
!vertical loop
do k=2,Nz
 !work 1
 !$acc parallel loop vector_length(N)
 do ip=1,nproma
  c2(ip)=c1(ip,k)*a(ip,k-1)
 end do
 !$acc end parallel loop
!work 2
!$acc parallel loop vector_length(N)
do ip=1,nproma
    a(ip,k)=c2(ip)*a(ip,k-1)
end do
!$acc end parallel loop
end do
!$acc end data
```

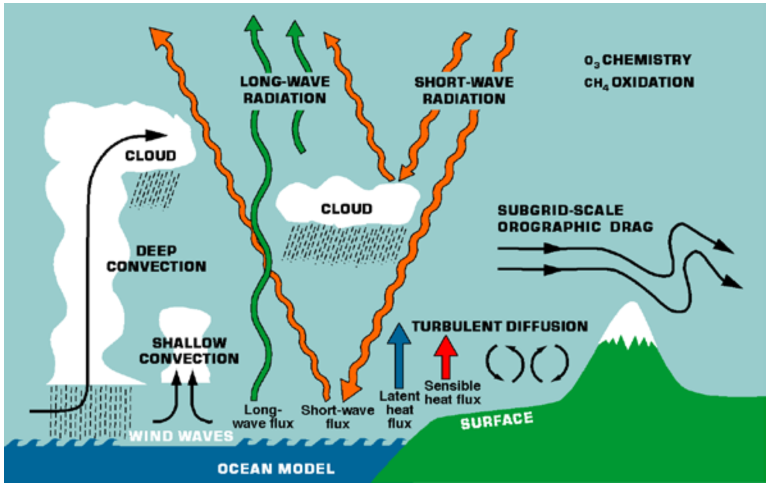
→

```
!$acc data present(a,c1)
!$acc parallel loop vector_length(N)
do ip=1,nproma
 !vertical loop
 do k=2,Nz
  !work 1
  c2=c1(ip,k)*a(ip,k-1)
  !work 2
   a(ip,k)=c2*a(ip,k-1)
 end do
end do
!$acc end parallel loop
!$acc end data
```

- 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 …

Lapillonne and Fuhrer submitted to Parallel Processing Letters

Ported Parametrizations

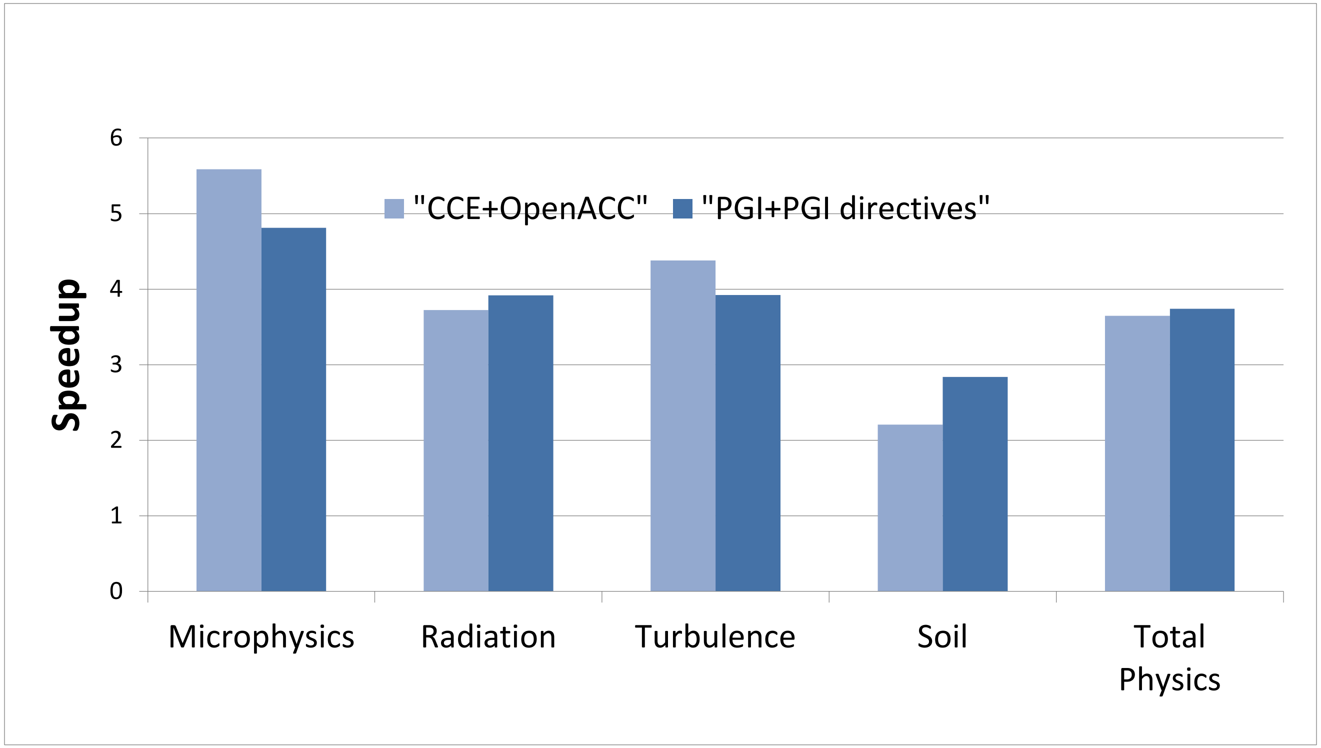


- Currently implemented and tested physics:
  - Microphysics (ice-scheme)
  - Radiation (Ritter-Geleyn)
  - Turbulence (Raschendorfer)
  - Soil (Terra)

- These 4 parametrizations account for 90-95% of the physics time of a typical COSMO-2 run. First meaningful real case simulations are possible with this reduced set.

Performance results for the physics

- Test domain 128x128x60 – 16 cores Interlagos CPU vs X2090 GPU



- Overall speed up x3.6
- Similar performance with Cray CCE and PGI
- Running the GPU-Optimized code on CPU is about 25% slower
  ➔ separate source for time critical routines

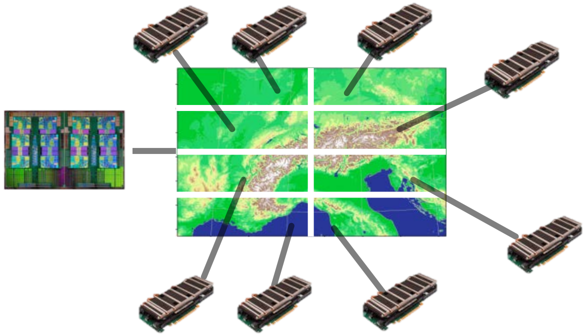
Our experience using directives

- Relatively easy to get the code running
- Useful to port large part of the code
- Requires some work to get performance: data placement, restructuring, additional optimization …
  - Ex: GPU part of assimilation is 20% to 50% slower on GPU than on CPU
- Having a single source code that run efficiently on GPU and CPU (x86) is still an issue

Putting all together: can we run this ?

- The dynamical core is compiled as a library:
  - gcc + nvcc
- Linked with the Fortran part + OpenACC
  - So far only working with Cray CCE
- GPU pointers are passed from fortran to C++ library using the host_data directive:
  - No data transfer required !

Conclusions



- Our strategy : full GPU port
- Dynamics : complete rewrite
- Physics and data assimilation : OpenACC directives
- Could achieve same time to solution than current operational with a Multi-GPU node having o(10) GPUs : demonstrator system for end 2012

Acknowledgments

- J. Pozanovick and all CSCS support
- R. Ansaloni, Cray
- P. Messmer, Nvidia
- M. Wolfe, M. Colgrove PGI