Towards operational implementation of COSMO on accelerators at MeteoSwiss

Oliver Fuhrer[1], Tobias Gysi[2], Carlos Osuna[3], Xavier Lapillonne[3], Mauro Bianco[4], Thomas Schulthess[4], et al.

[1] Federal Office of Meteorology and Climatology MeteoSwiss, Switzerland
[2] Supercomputing Systems AG, Switzerland
[3] Center for Climate Systems Modeling / ETH, Switzerland
[4] Swiss National Supercomputing Centre CSCS / ETH, Switzerland

COSMO Model

- Regional weather and climate prediction model
- Community model
- O(70) universities and research institutes
- Operational at 7 national weather services

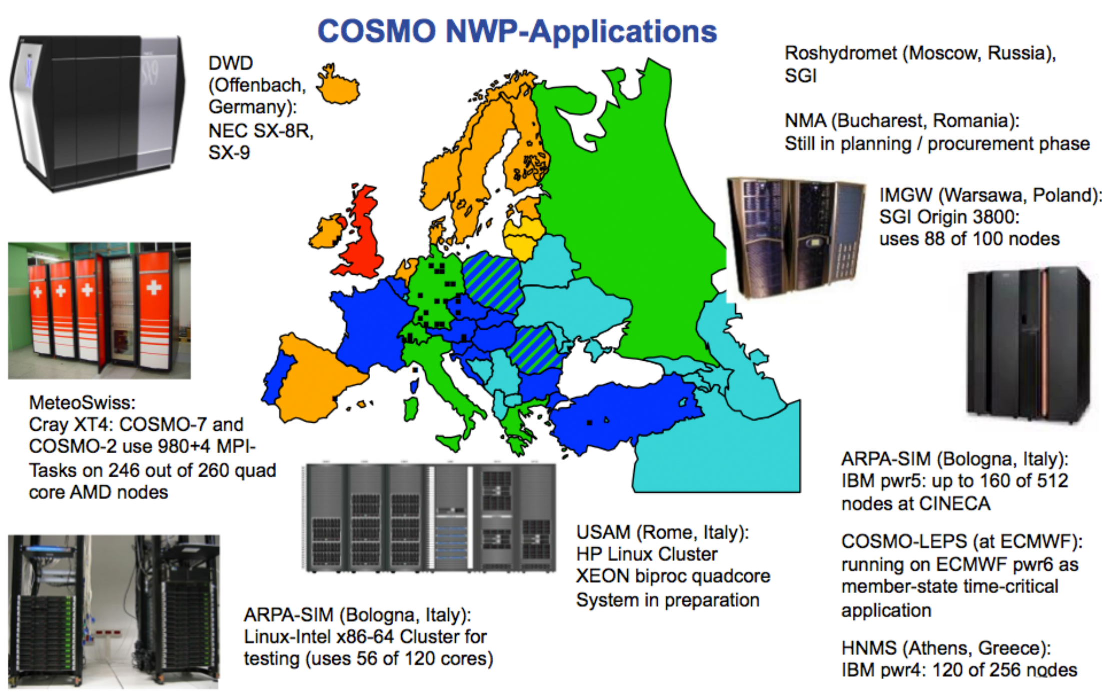
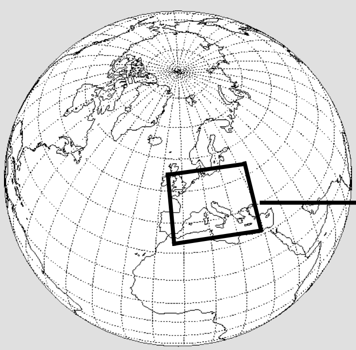


Figure: Ulrich Schättler

Model applications

ECMWF-Modell

16 km gridspacing
2 x per day
10 day forecast

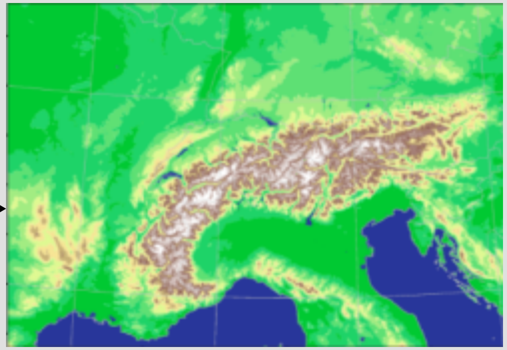


COSMO-7

6.6 km gridspacing
3 x per day
72 h forecast

COSMO-2

2.2 km gridspacing
7 x per day 33 h forecast
1 x per day 45 h forecast



Production with COSMO @ CSCS

Cray XE6 (Albis/Lema)

MeteoSwiss operational system

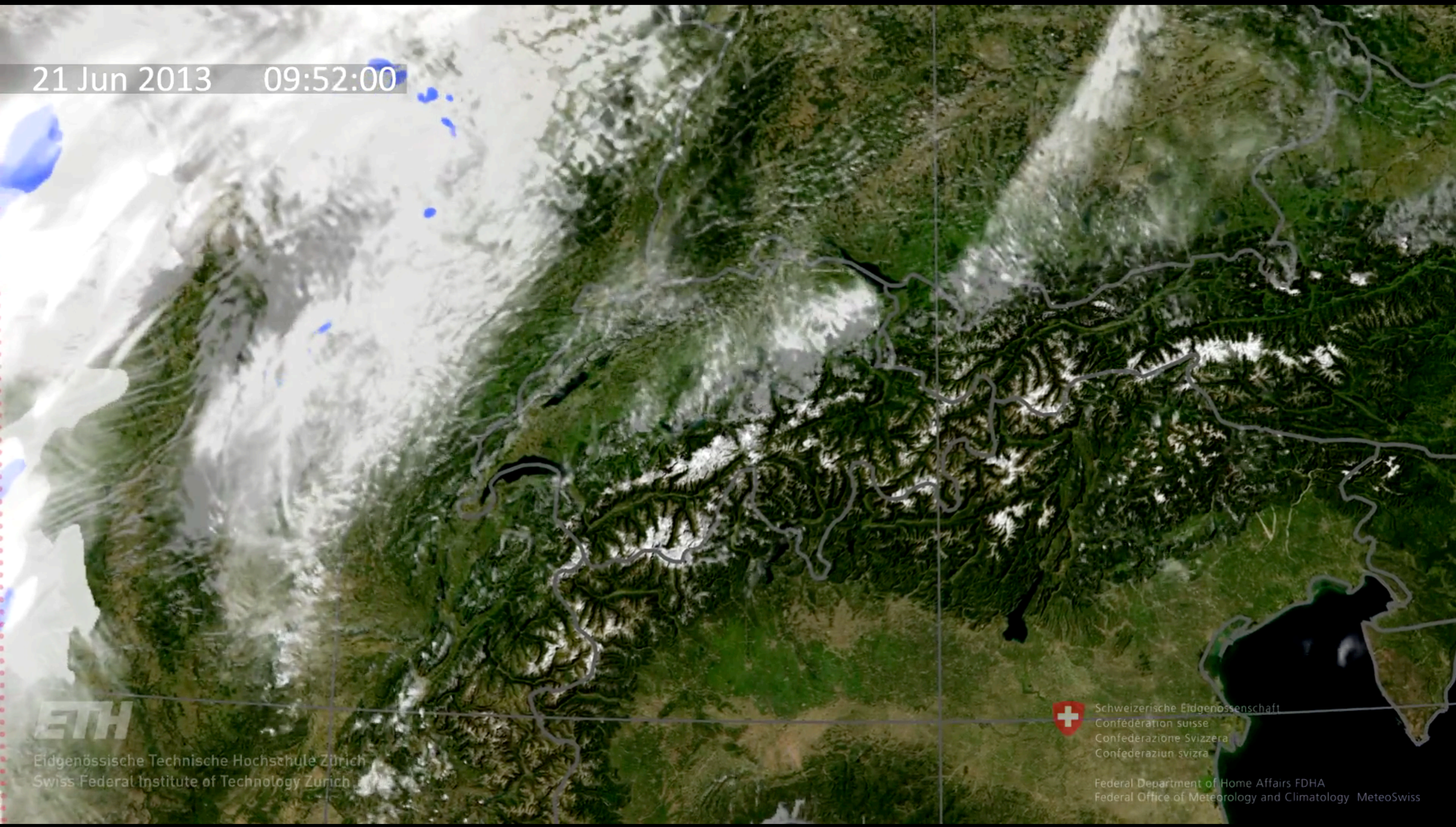
~15 Mio core hours / year

Cray XE6 (Rosa)

Research system

~15-20 Mio core hours / year

Future applications



COSMO Workflow

Initialization

Δt

Boundary conditions

Physics

Dynamics

Data assimilation

Relaxation

Diagnostics

Input / Output

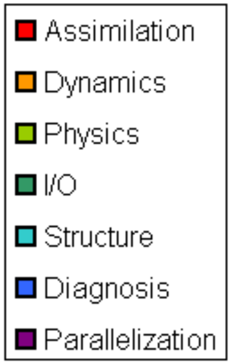
Properties

- PDEs
- Finite diferences
- Structured grid
- Sequential workflow

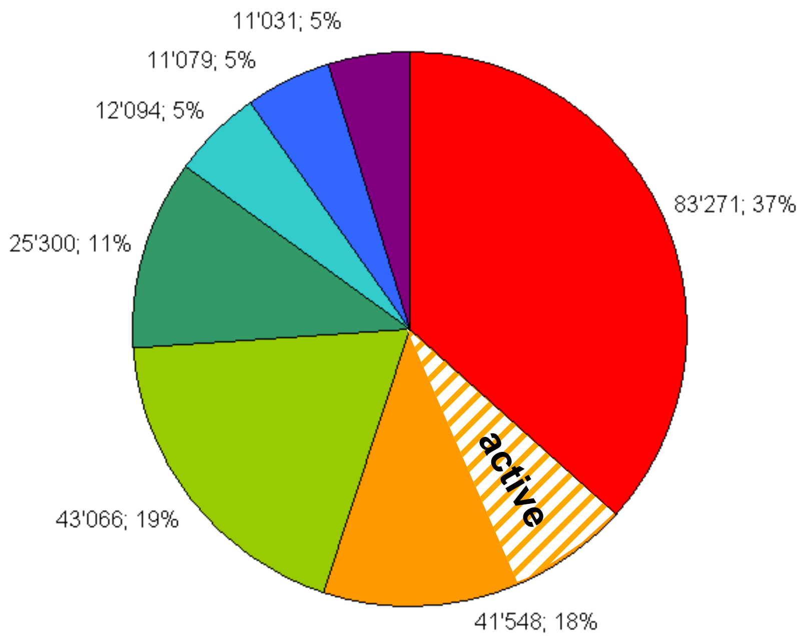
Cleanup

Code lines and runtime

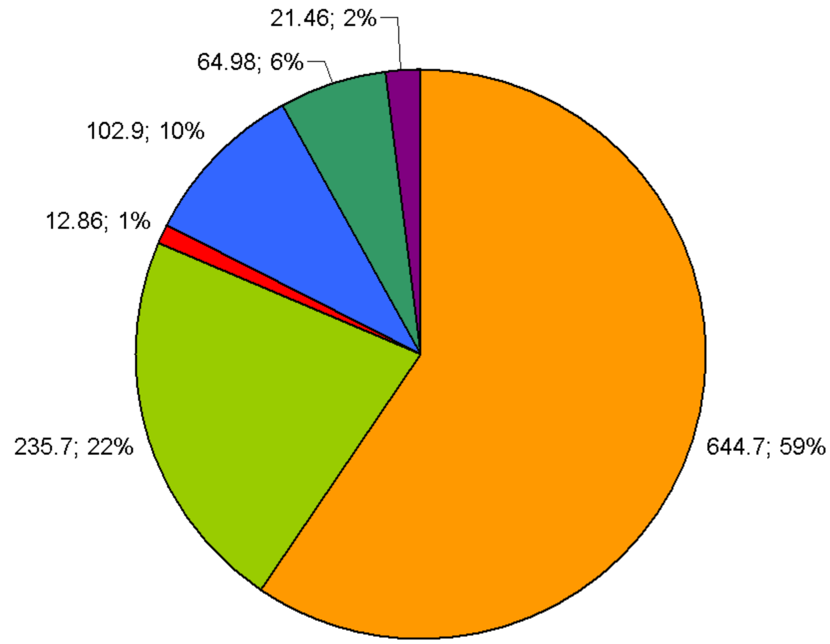
- 300’000 lines of Fortran 90 code



% lines of code



% runtime



Approach

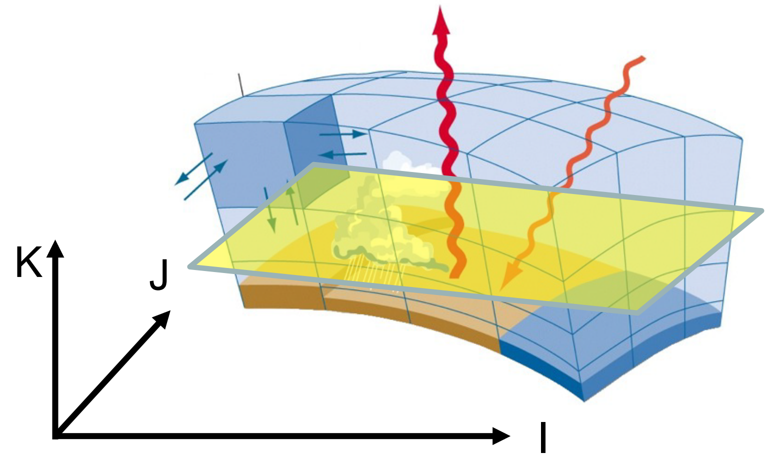
- **Dynamics**
  - 40k lines, 60% runtime
  - Few developers
  - Strongly memory bandwidth bound

Aggressive rewrite

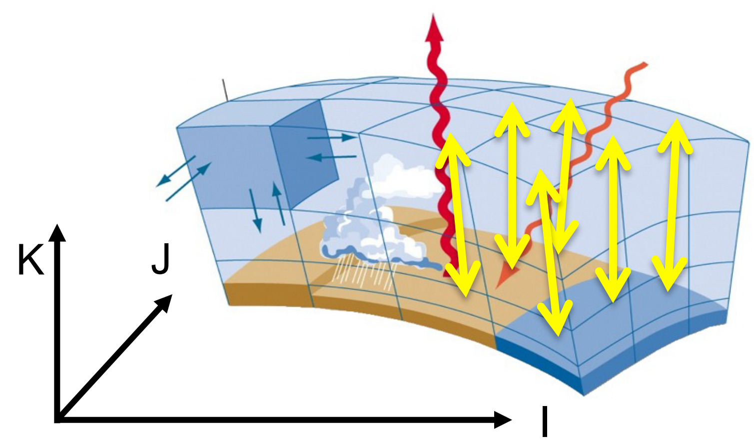
- Data structures
- C++
- DSEL

Key algorithmic motifs

1. **Finite difference stencil computations**
    - Focus on horizontal IJ-plane accesses
    - No loop carried dependencies



2. **Tri-diagonal solves**
    - vertical K-direction pencils
    - Loop carried dependencies in K



Code example

- Solution of tridiagonal linear system

$$
\begin{bmatrix}
b_1 & c_1 & & & 0 \\
a_2 & b_2 & c_2 & & \\
& a_3 & b_3 & \ddots & \\
& & \ddots & \ddots & c_{n-1} \\
0 & & & a_n & b_n
\end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}
=
\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}
$$

- Typical for implicit schemes (advection, diffusion, radiation, …)
- Abundant and performance critical in many dynamical cores

COSMO Version

- Algorithm: TDMA
- Language: Fortran
- Grid: Structured
  Data layout: (i,j,k)
- Parallelization: MPI in (i,j)
- Loop order: (jki)
- Blocking: (j)
- Vectorization: (i)
- Directives: NEC
- …

```fortran
! solve tridiag(a,b,c) * x = d

! pre-computation
...

do j = jstart, jend


  ! forward elimination
  do k = nk, 2, -1
    do i = istart, iend
!CDIR ON ADB(d)
      d(i,j,k) = ( d(i,j,k) - d(i,j,k+1) * c(i,j,k) ) * b(i,j,k)
    end do
  end do

  ! back substitution
  do k = 1, nk-1
    do i = istart, iend
!CDIR ON ADB(x)
      x(i,j,k+1) = a(i,j,k+1) * x(i,j,k) + d(i,j,k+1)
    end do
  end do

end do
```

Optimized CPU Versi

```fortran
! solve tridiag(a,b,c) * x = d

!$OMP PARALLEL DO SHARED(x) PRIVATE(a,b,c,d) COLLAP
do ib = 1, nblock_i
do jb = 1, nblock_j

  ! pre-computation
  ...

  do i = istart_block, iend_block
  do j = jstart_block, jend_block

    ! forward elimination
    do k = nk, 2, -1
      d(k,j,i) = ( d(k,j,i) - d(k+1,j,i) * c(k,j,i)
    end do

    ! back substitution
    do k = 1, nk-1
      x(k+1,j,i) = a(k+1,j,i) * x(k,j,i) + d(k+1,j,i)
    end do

  end do
  end do

end do
end do
!$OMP END PARALLEL DO
```

- Algorithm: TDMA
- Language: Fortran
- Grid: Structured
- Data layout: (k,j,i)
- Parallelization: Node in (i,j) and Core in (i,j)
- Loop order: (ijijk)
- Blocking: (i,j)
- No vectorization
- Directives: OpenMP
- …

Optimized GPU Versi

```
! solve tridiag(a,b,c) * x = d

!$ACC DATA COPYIN(a,b,c,d) COPYOUT(x)

!$ACC KERNELS LOOP, GANG(32), WORKER(8)
do i = istart, iend
do j = jstart, jend

  ! pre-computation
  ...

  ! forward elimination
  do k = nk, 2, -1
    d(i,j,k) = ( d(i,j,k) - d(i,j,k+1) * c(i,j,
  end do

  ! back substitution
  do k = 1, nk-1
    x(i,j,k+1) = a(i,j,k+1) * x(i,j,k) + d(i,j,k+1)
  end do

end do
end do
!$OMP END KERNELS LOOPS

!$ACC END DATA
```

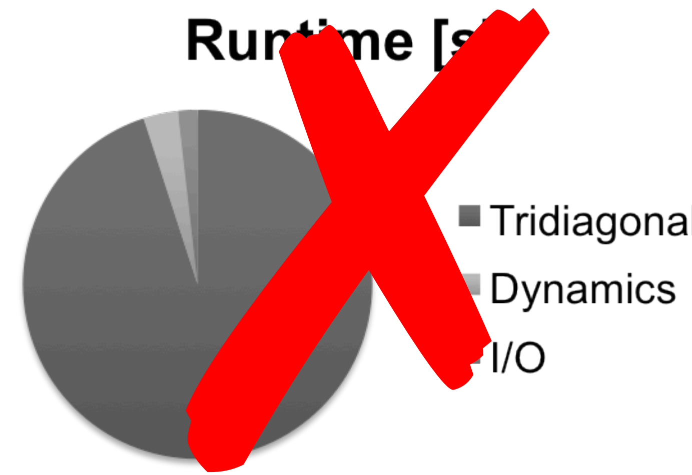
- Algorithm: TDMA
- Language: Fortran
- Grid: Structured
- Data layout: (i,j,k)
- Parallelization: Nodes (i,j) and Blocks (i,j)
- Loop order: (ijijk)
- No Blocking
- Vectorization: SIMD Threads (i,j)
- Directives: OpenACC
- …

Learnings

- **No separation of concerns** Code is a mix of mathematical model, numerical discretization, solution algorithm, and hardware dependent implementation details
- Optimizations are **hardware dependent** and **increase code complexity**
- Consequences
  - Hard to achieve performance portability with a single source code!
  - Hard to understand and modify
  - Hard to validate and debug
  - Hard to re-use

Easy way out?

- Can we replace the tridiagonal solve with a efficient, hardware specific implementation or library call?



- **Not really!**
  - Cost of moving the data excessive
  - No single hotspot (flat profile)
  - Amdahl's law

- Basic entities are the prognostic variables (ρ, u, v, w, θ, q_x, …) and we perform a series of expensive operations (advection, diffusion, physics, …) on them every timestep

Acceleration with GPUs?

- Stencils = low FLOP count per load/store
- Transfer of data on each timestep too expensive

*

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

vs

§

Transfer of ten prognostic variables

118 ms

All code which touches the prognostic variables on every timestep has to be ported

Solutions?

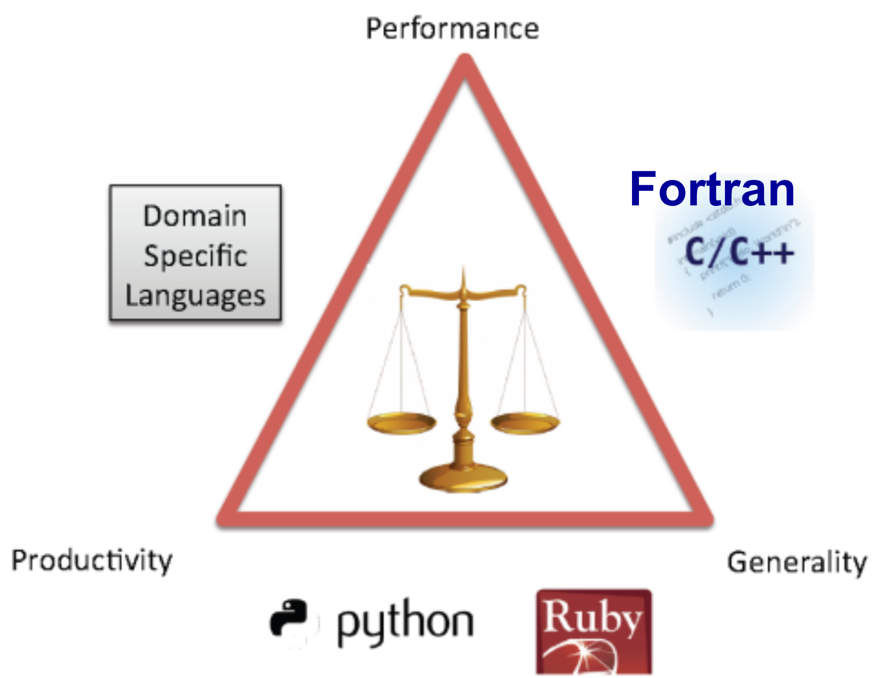
How can we achieve performance portability with COSMO?

- Good compromise (if it exists!)
- Several efficient source codes
- Separate model and algorithm from hardware specific implementation and optimization

Challenging computer science problem!

STELLA Library

- Domain specific (embedded) language (DSEL)
- C++ host language
- Implemented using template meta-programming



STELLA usage

```
DO k = 1, ke
  DO   j  = jstart, jend
    DO i  = istart, iend
      data_out(i,j,k) = -4.0_wp * data_in(i,j,k)                &
                      + data_in(i+1,j,k) + data_in(i-1,j,k)     &
                      + data_in(i,j+1,k) + data_in(i,j-1,k)
    ENDDO
  ENDDO
ENDDO
```

- Remove loops and data structures from user code

STELLA usage

```
// Laplacian stencil
template<typename TEnv>
struct Laplacian
{
    static T Do(Context ctx)
    {
        ctx[data_out::Center()] =
            - (T)4.0 * ctx[data_in::Center()]
            + ctx[data_in::At(iplus1)] + ctx[data_in::At(iminus1)]
            + ctx[data_in::At(jplus1)] + ctx[data_in::At(jminus1)]
    }
};
```

```
// Apply the Laplacian stencil to domain
StencilCompiler::Build(
    stencil_,
    "Laplacian",
    calculationDomain,
    StencilConfiguration<Real, BlockSize<8,8>>(),
    define_loops(
        define_sweep<cKIncrement>(
            define_stages(
                StencilStage<Laplacian,
                             IJRange<cComplete,0,0,0,0>,
                             KRange<FullDomain,0,0> >(),
            )
        )
    )
);
```

STELLA backends

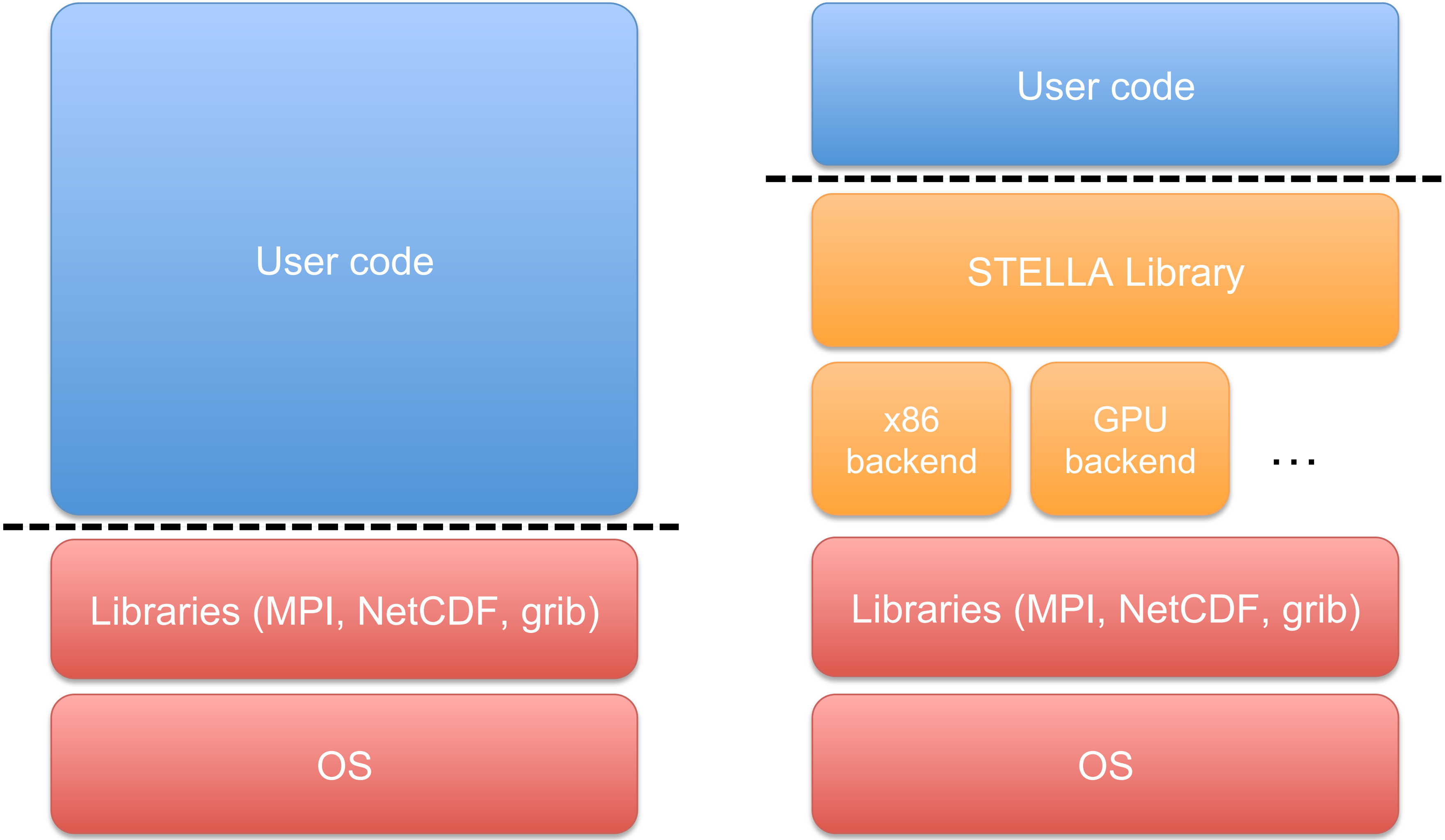
- **x86 CPU** (OpenMP, kji-storage)
  - Factor 1.5x – 1.8x faster than original code (on SB)
  - No explicit use of vector instructions (up to 30% improvement)

- **NVIDIA GPU** (CUDA, ijk-storage)
  - CPU vs. GPU is a factor 3.4x faster (SB vs. K20x)
  - Ongoing performance optimization

- …

- Possible to switch backend by modifying a single line

Separation of concerns



Approach

- **Dynamics**
  - 40k lines, 60% runtime
  - Few developers
  - Strongly memory bandwidth bound

Aggressive rewrite
- Data structures
- C++
- DSEL

- **Physics & Assimilation**
  - 130k lines, 25% runtime
  - Several developers
  - “Plug-in” from other models
  - Less memory bandwidth bound

Port to GPU
- keep source
- directives

Implementation

Setup

Copy to GPU

Input

Physics

OpenACC

Interface

Dynamics src_runge_kutta.f90

Relaxation src_relaxation.f90

STELLA Library

Interface

Δt

Assimilation

Halo-update → Interface

Diagnostics

OpenACC

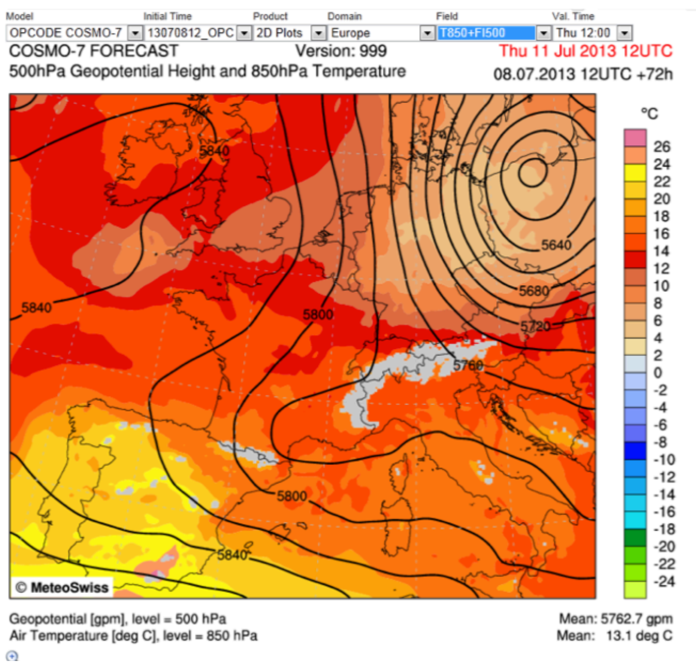
Copy from GPU

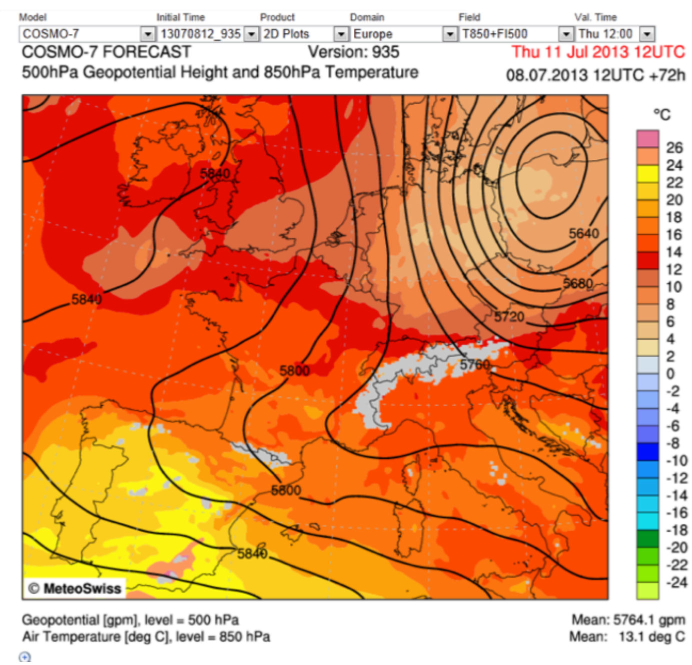
Output

Cleanup

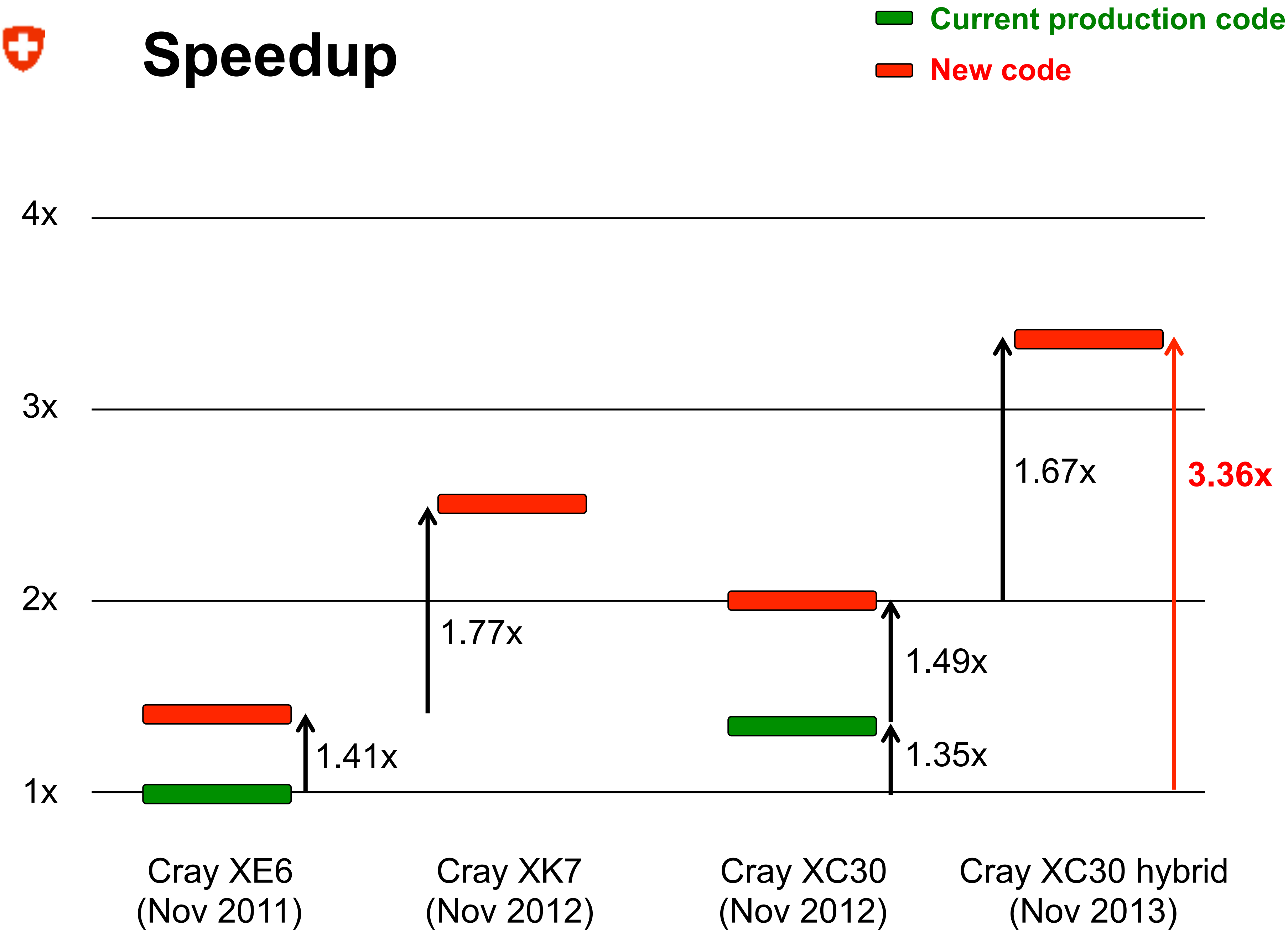
Current status

- Branch of COSMO running on GPU-hardware
- Regular runs (00 UTC and 12 UTC)
- Full operational chain
  (plots delivered into visualization software)
- Almost full featured, missing features in progress



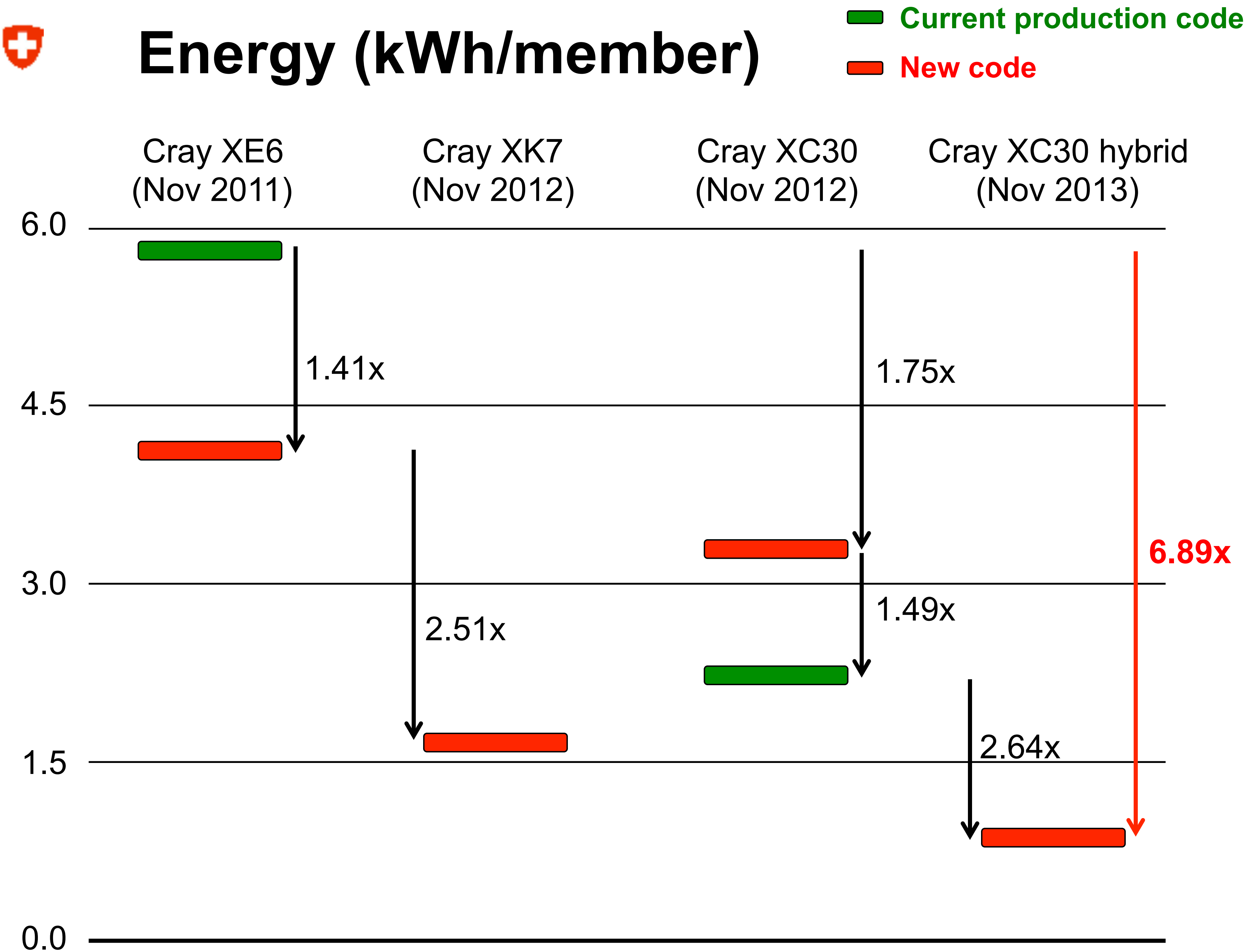


Speedup



Current production code
New code
4x
3x
2x
1x
1.41x
1.77x
1.49x
1.35x
1.67x
3.36x
Cray XE6 (Nov 2011)
Cray XK7 (Nov 2012)
Cray XC30 (Nov 2012)
Cray XC30 hybrid (Nov 2013)

Energy (kWh/member)



Learnings

- **Underestimated effort to integrate technologies**
  (C++/CUDA with Fortran/OpenACC, GPU and CPU)
- **Many technologies were/are not ready**
  (e.g. robust CUDA/OpenACC compilers, efficient G2G, **…**)
- **Asynchronous communication** not (yet) leveraged
- **Underestimated** complexity of heterogeneous code and the many use cases
- **GPU Porting** is accessible to domain scientists
  (both with STELLA and OpenACC)

Next steps

- Upgrade to latest model version
- Bring developments back to trunk
- Improve feature completeness
- Next version of STELLA

Conclusions

- Changing hardware architectures require (continually) adapting our codes
- Model codes are growing in length and complexity
- No consensus has emerged to deliver both high performance with high programmer productivity
- DSLs can help by…
  - freeing model developer from implementation details
  - retaining efficiency with single source code
  - making our codes more reusable and adaptable
  - joining efforts
- The implementation of COSMO dynamics demonstrates that this can work!

FAQ

“Climate change is so important, that our compute center will not buy a machine which does not work for our codes!”

“A master / PhD student will not be able to work with this code!”

“But we all know Fortran and don’t know C++!”

“A new compiler will be able to do this!”