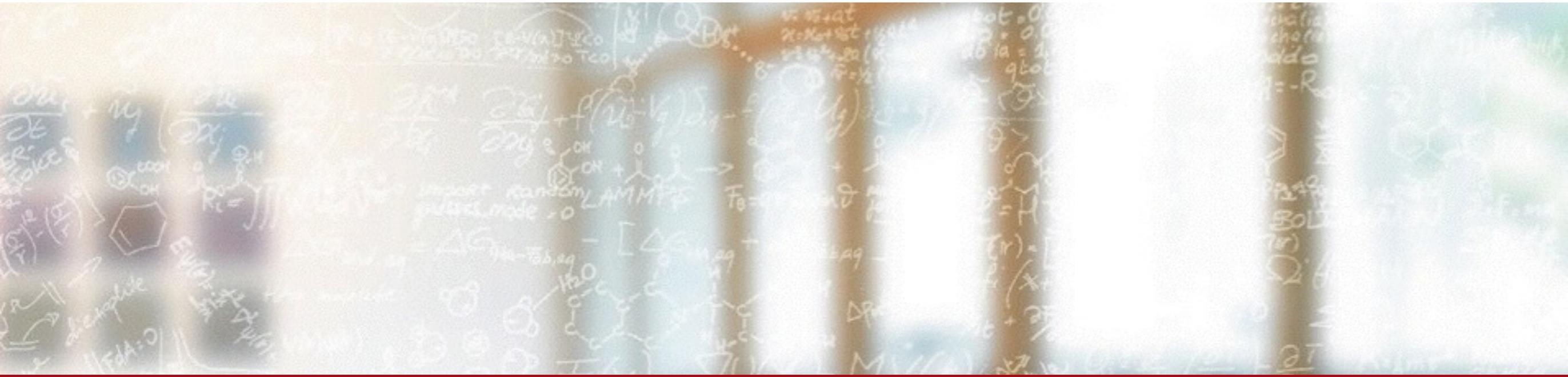




**CSCS**

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre

**ETH** zürich



# **The Road to Exascale: It's about the Journey, not the Flops**

**Will Sawyer, Swiss National Supercomputing Centre (CSCS)**

**EMiT 2016 Conference**

**June 2, 2016, Barcelona, Spain**

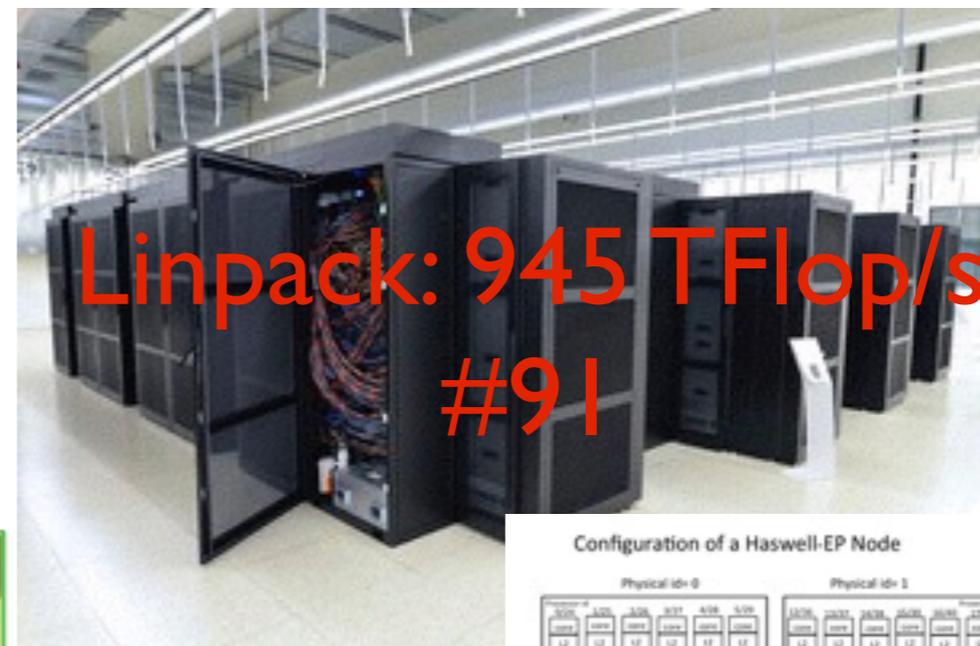
# CSCS in brief

CSCS has numerous customers from several scientific communities:

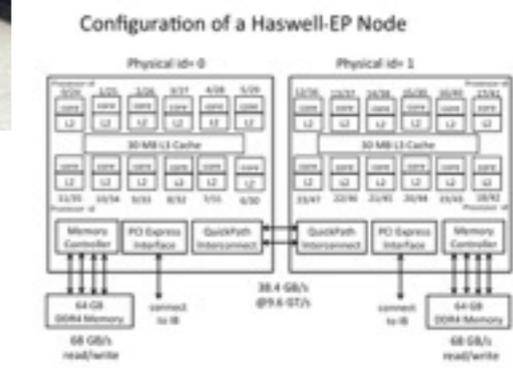
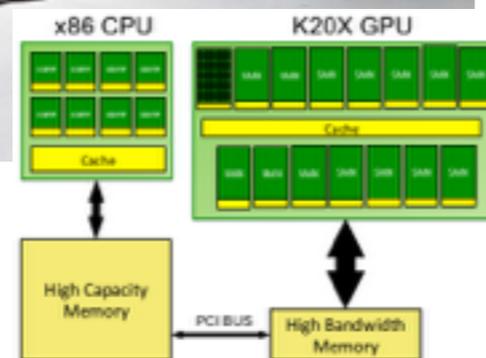
- Computational Chemistry and Material Science
- Climate / Numerical Weather Prediction
- Seismology, Solid-Earth dynamics
- Life sciences
- others...



Piz Daint

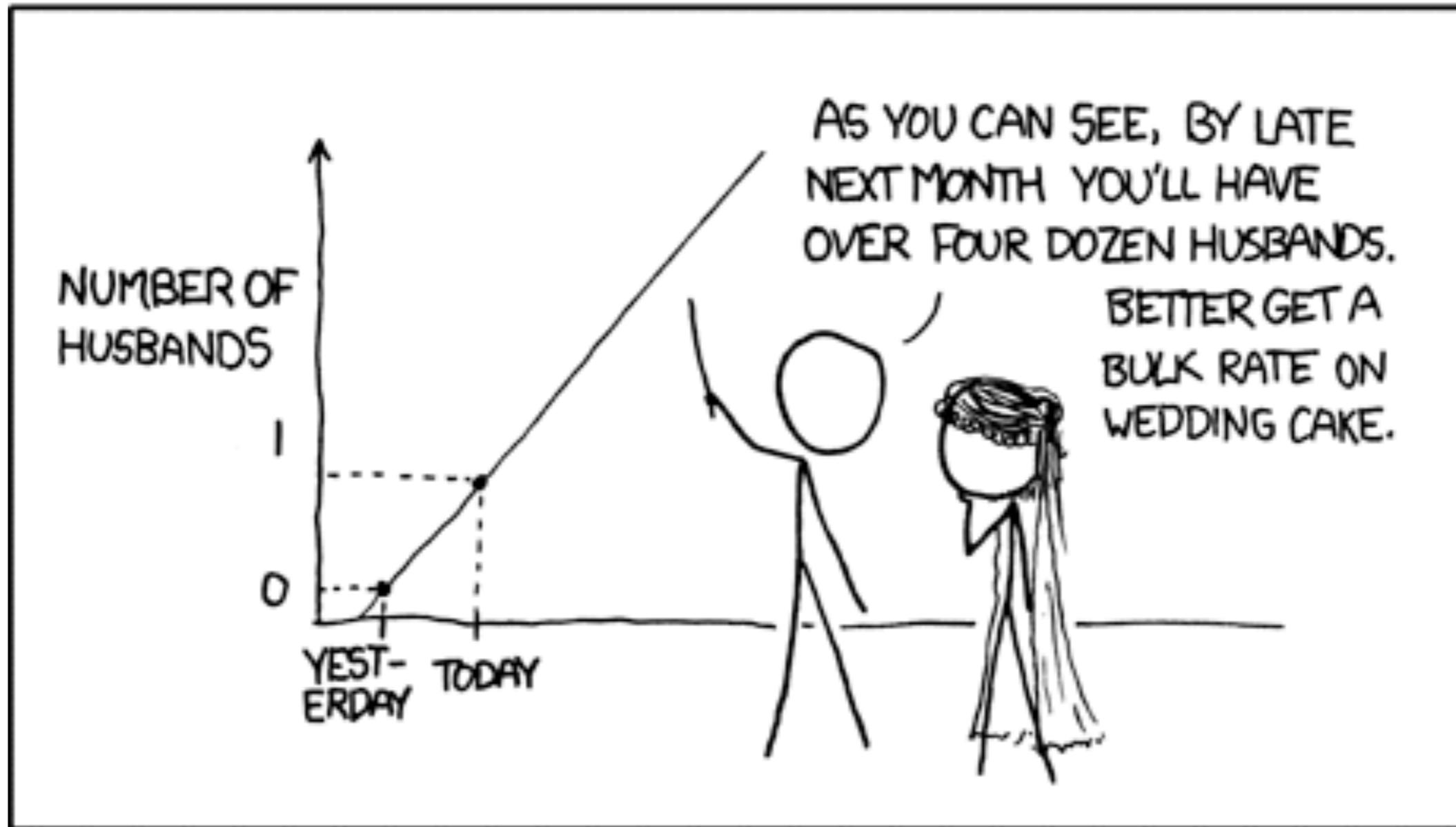


Piz Dora



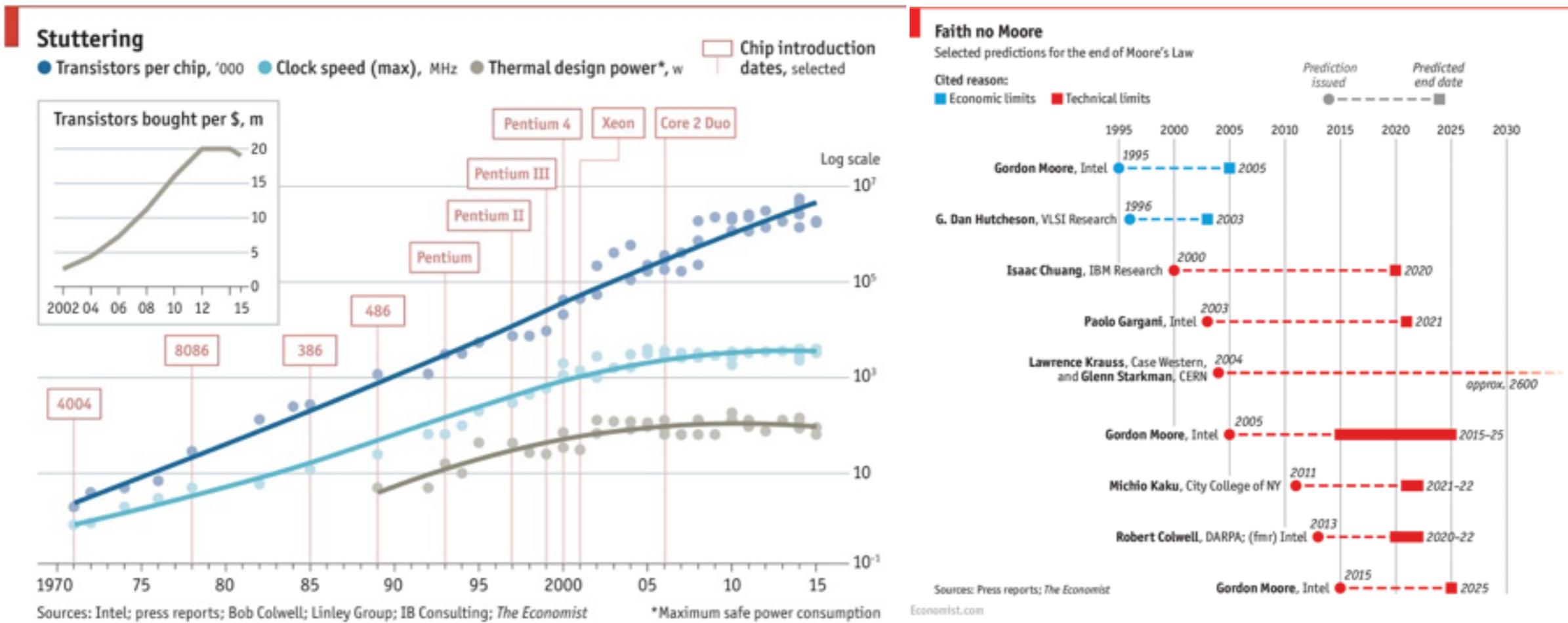
# Extrapolation is a fool's game

MY HOBBY: EXTRAPOLATING



# The End of Moore's Law

"Moore's law" is the *observation* that, over the history of computing hardware, the number of transistors in a dense integrated circuit has doubled approximately every two years. [https://en.wikipedia.org/wiki/Moore's\\_law](https://en.wikipedia.org/wiki/Moore's_law)



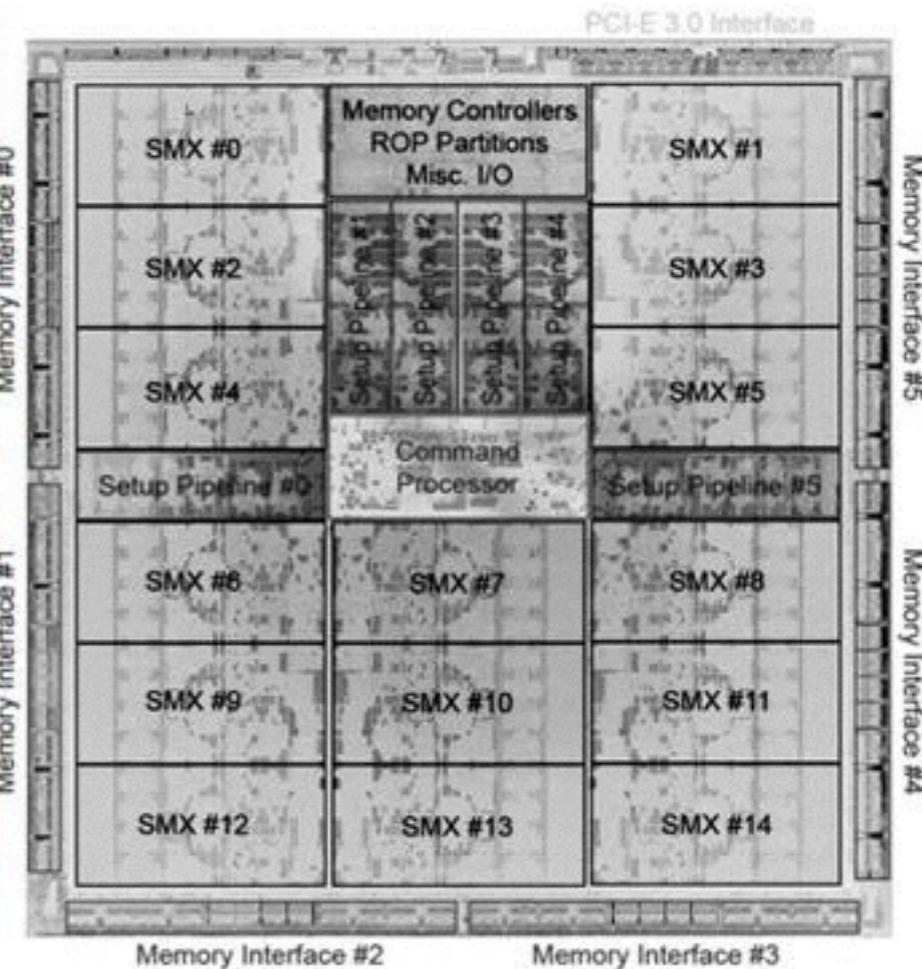
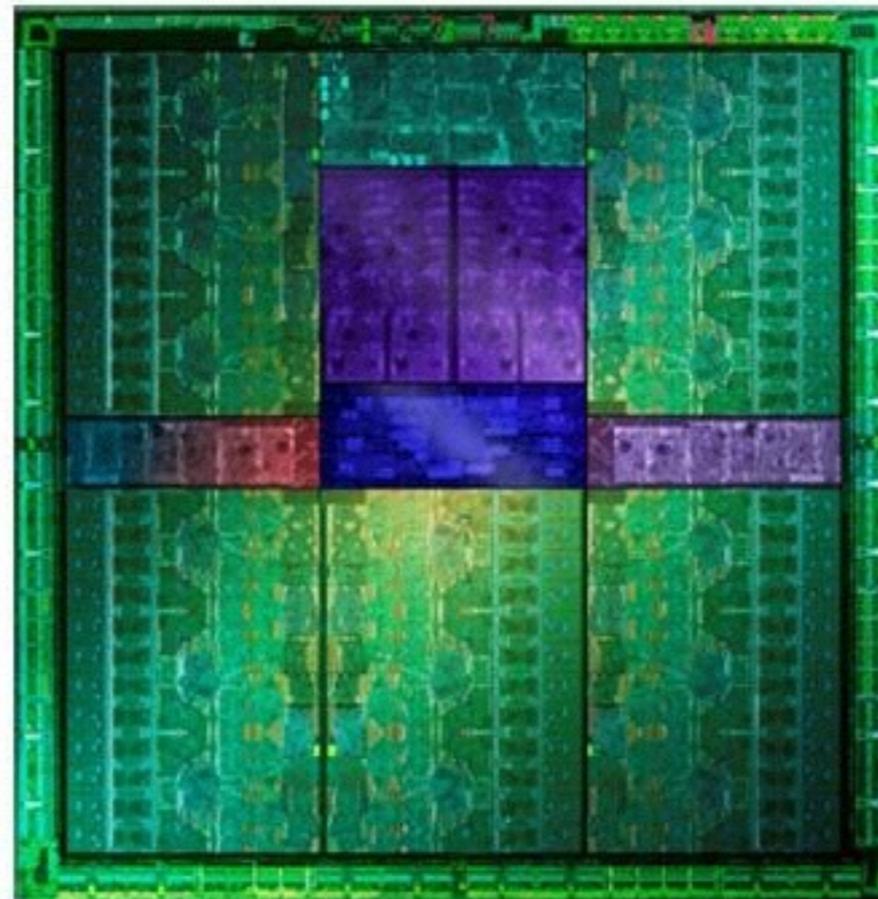
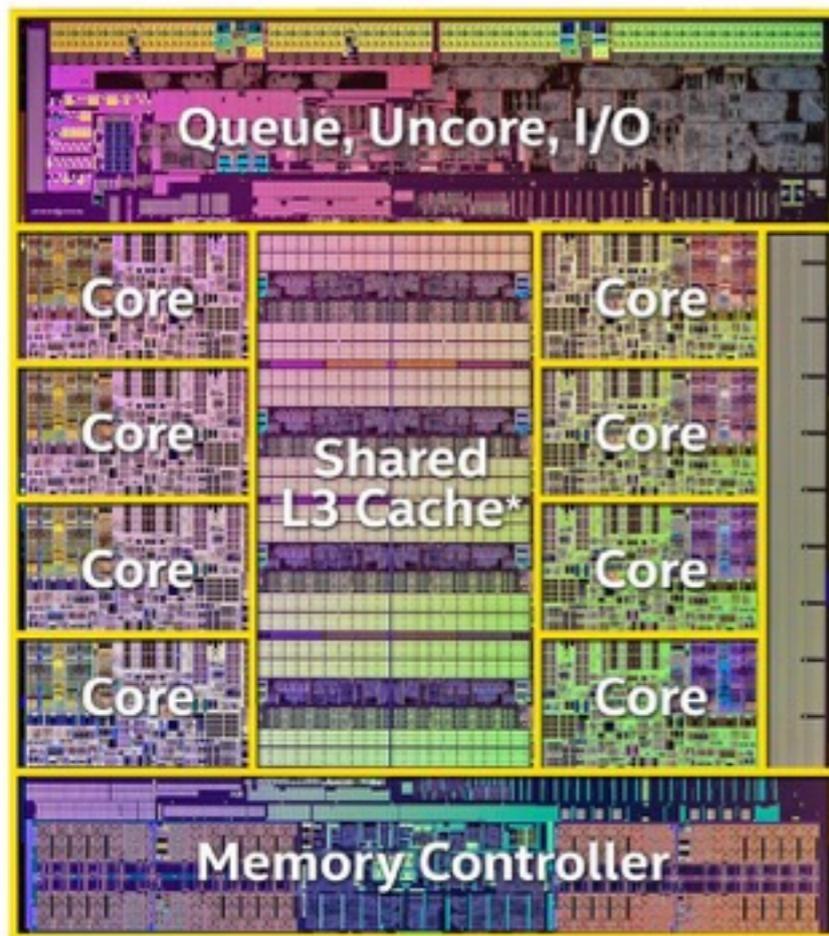
“The number of people predicting the death of Moore’s law doubles every two years.”

Peter Lee, VP Microsoft

# Human Ingenuity: simpler processors, but many more

Intel Haswell:  
8-core CPU

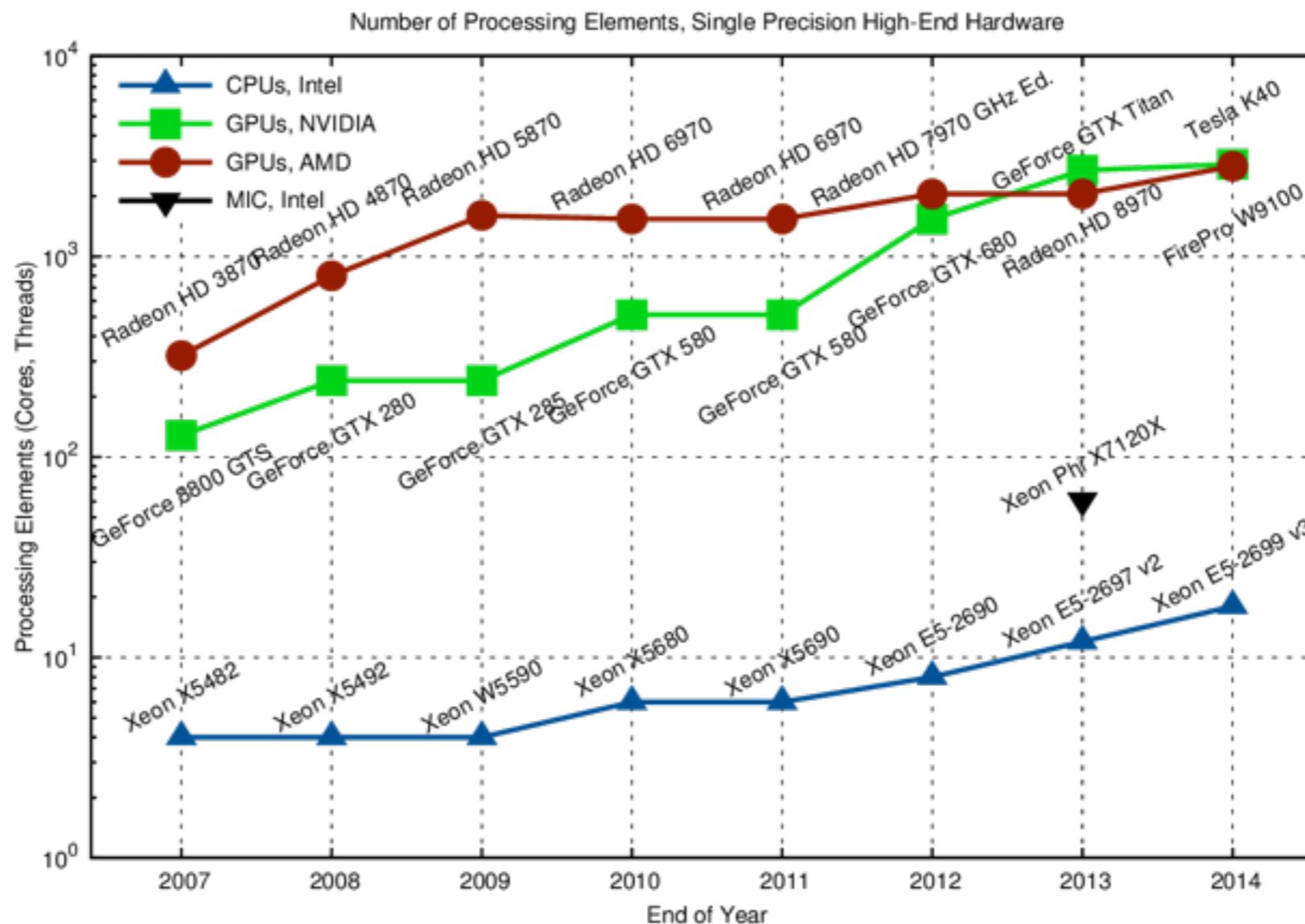
NVIDIA GK110 (old!): 15 SIMT Streaming multiprocessors each with 192 'cores'!



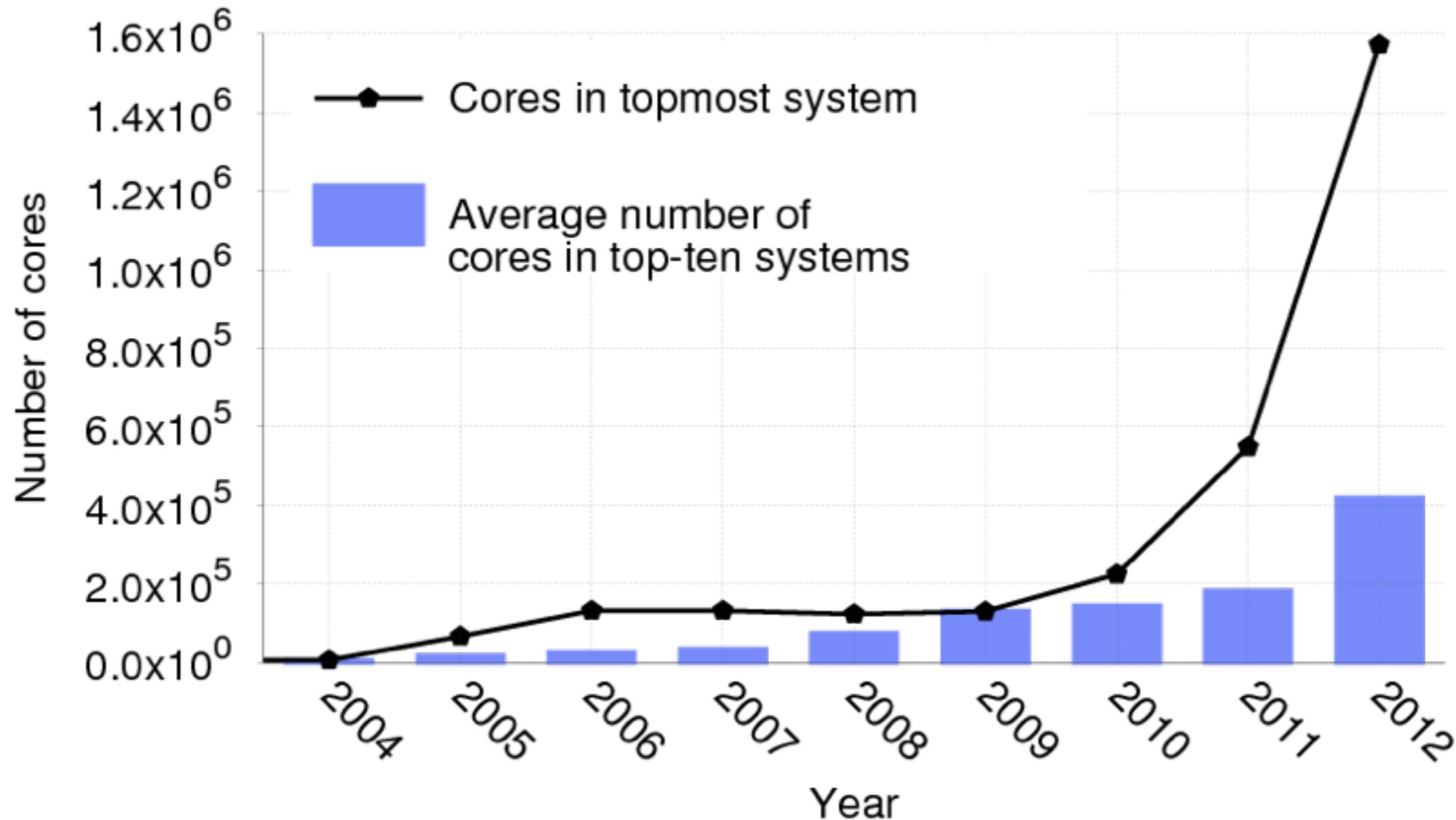
Keep in mind: GPU cores are made for pixel operations

# Evaluation of core counts

*The number of cores per socket is increasing for both CPUs, Xeon Phi, and GPUS. The latter two contain “lightweight” cores (e.g., for pixel calculations)*

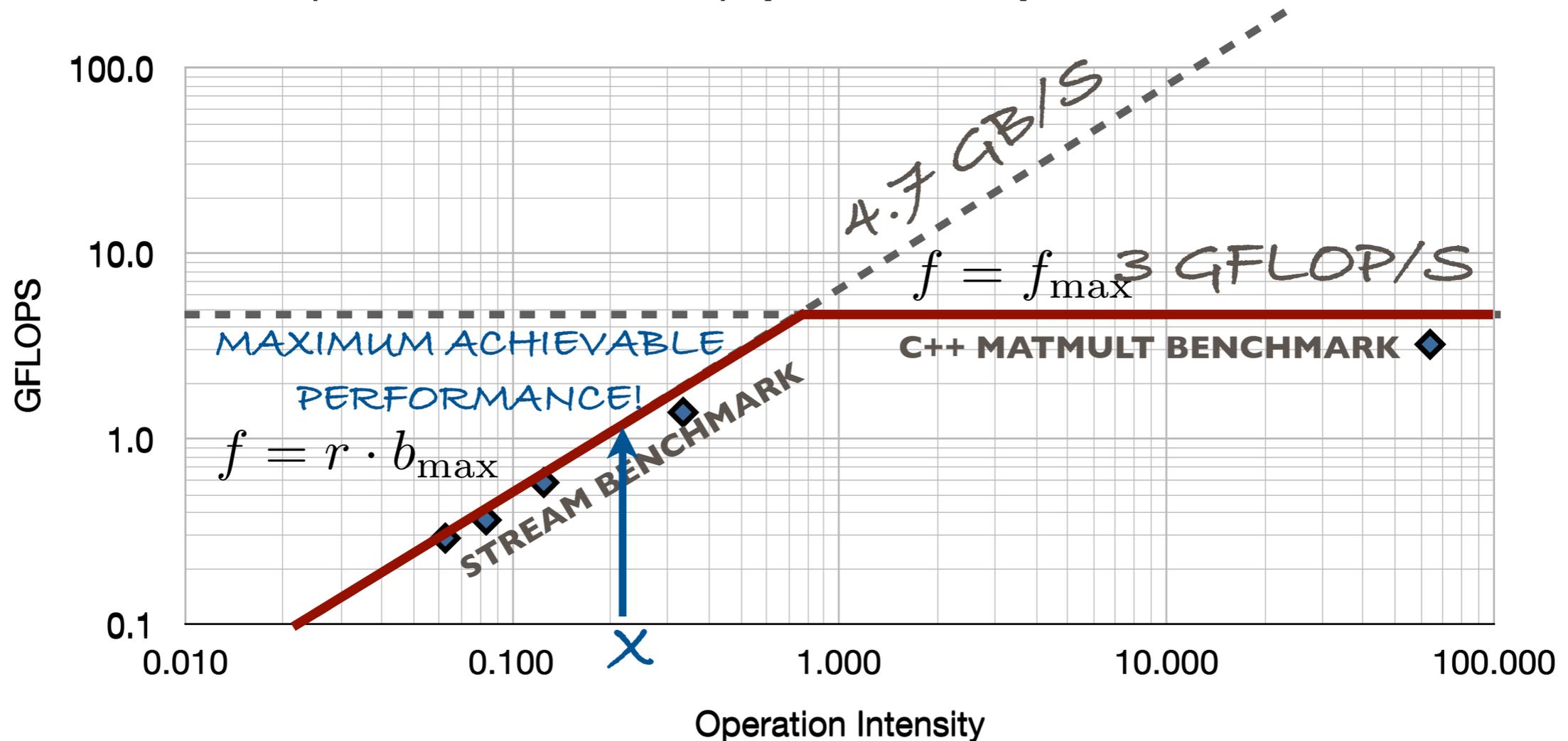


# Increase in number of cores in top 10 supercomputers



# But do we care so much about ExaFlop/s?

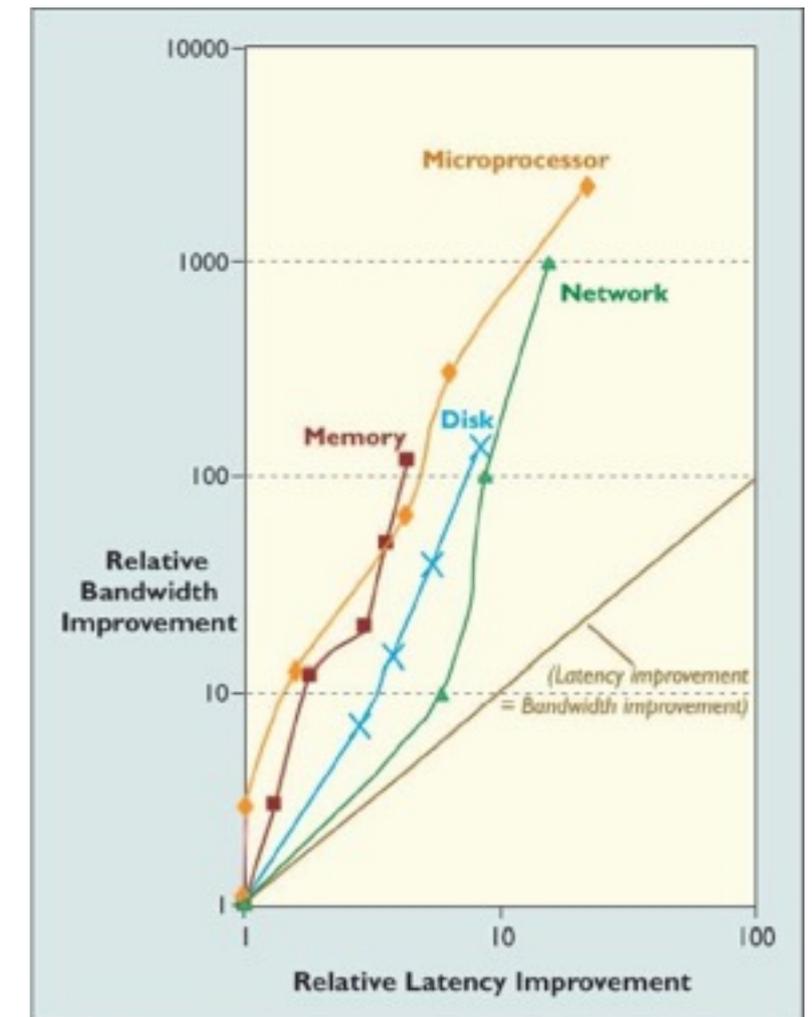
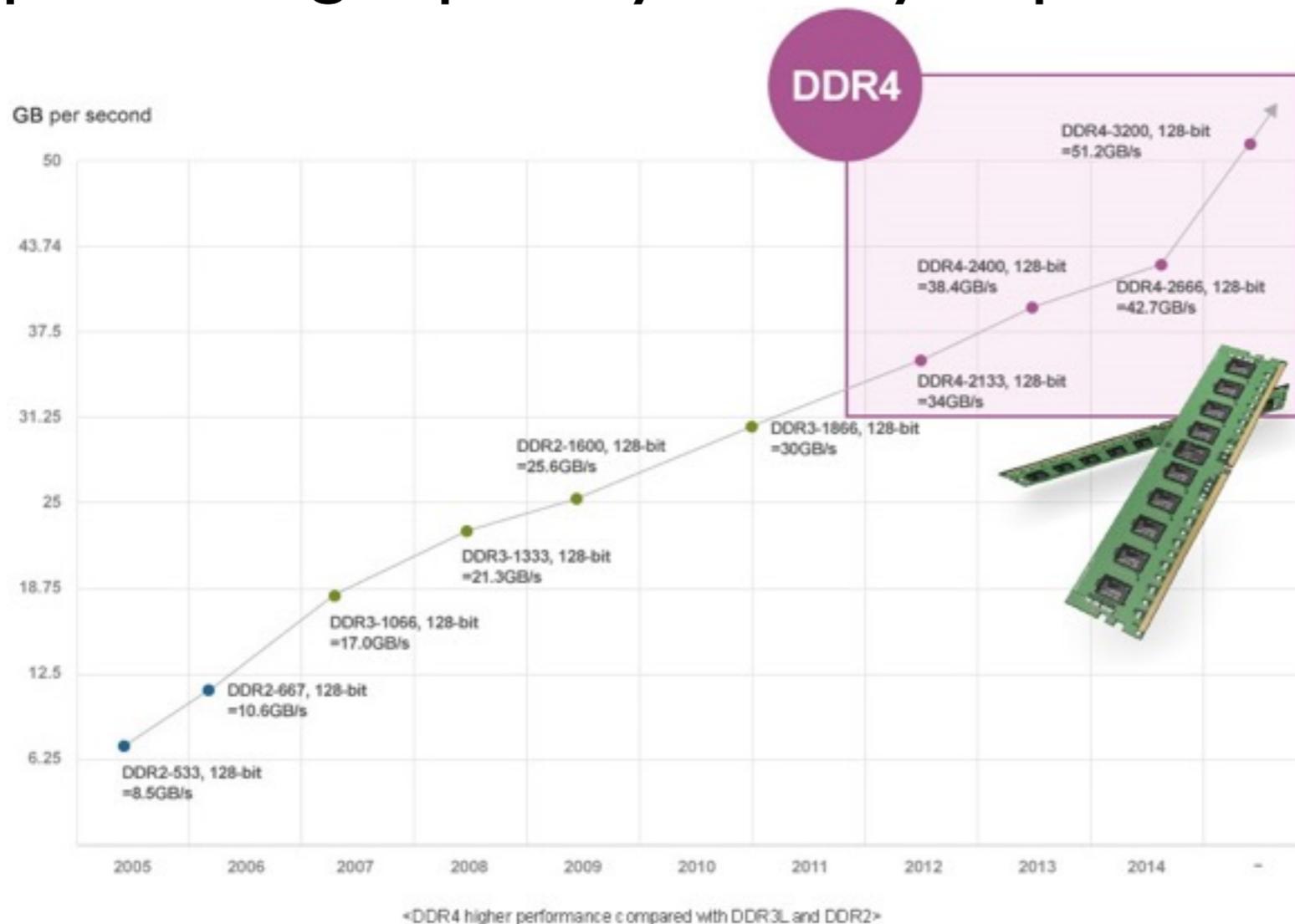
Majority of CSCS applications are memory-bandwidth limited, *roofline model* (Williams, et al.) provides performance model



◆ 4x Quad-Core AMD Opteron 8380 @ 2.5GHz - 1 Thread - C++

# Evolution of memory bandwidth

Memory bandwidth increases, albeit more slowly than processing capability; latency improves even more slowly

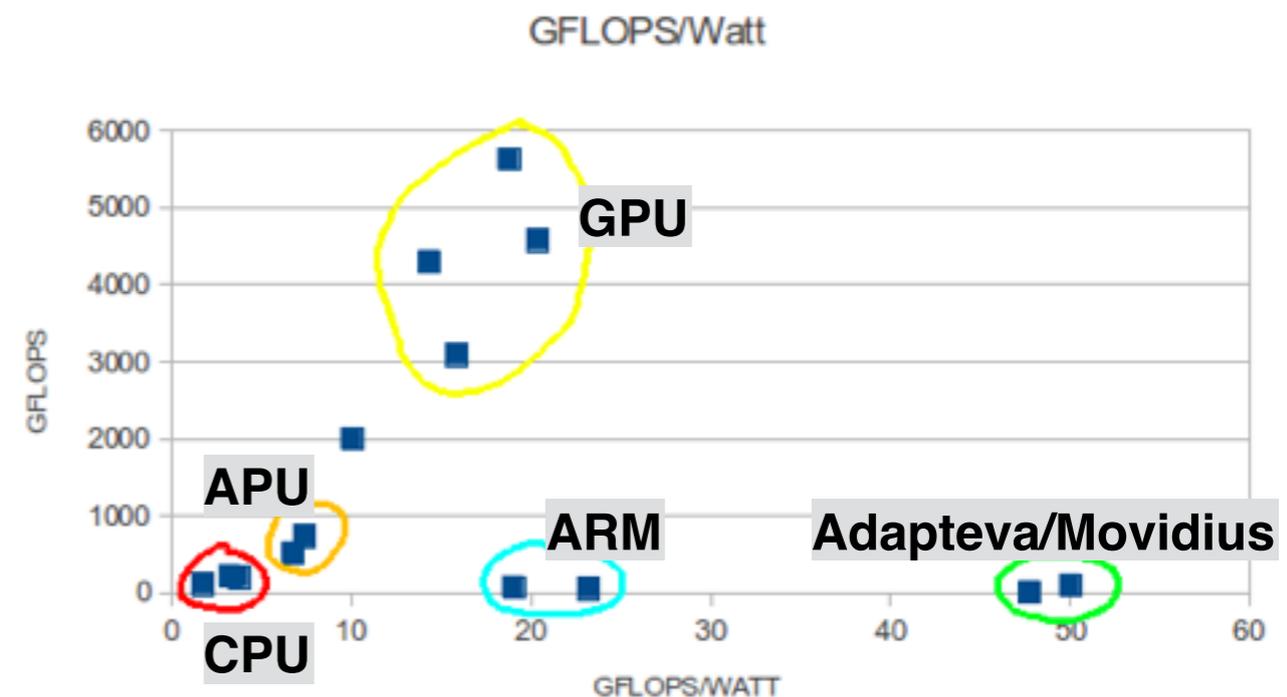
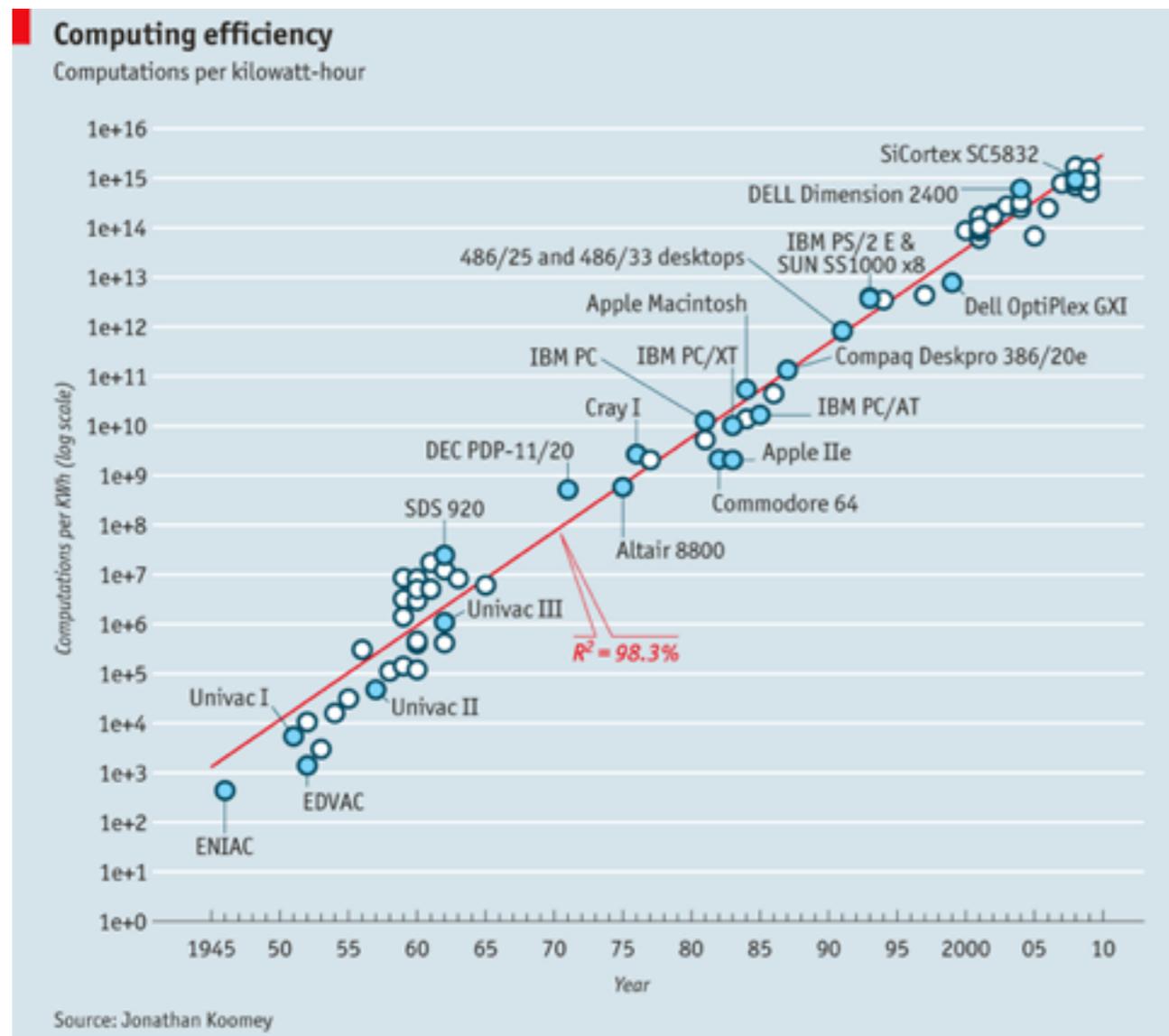


Note that latency improved about 10X while bandwidth improved about 100X to 1000X.

# Energy Consumption: heading for the “power wall”?

Koomey’s law is an observation that Flops/Watt doubles every 1.57 years

Architectures vary not only in Flop/s, but also Flops / Watt



Source (2012): <http://streamcomputing.eu>

# But there are physical boundaries to energy efficiency!

which takes more energy?

64-bit floating-point fused multiply add

or

moving three 64-bit operands 20 mm across the die

$$\begin{array}{r}
 934,569.299814557 \\
 \times \quad 52.827419489135904 \\
 \hline
 = 49,370,884.442971624253823 \\
 + \quad 4.20349729193958 \\
 \hline
 = 49,370,888.64646892
 \end{array}$$



← 20 mm →

Operation	Approx. Cost
DP FMADD floating point operation	100 pJ
DP DRAM read-to-register	4800 pJ
DP word transmit-to-neighbor	7500 pJ
DP word transmit-across system	9000 pJ

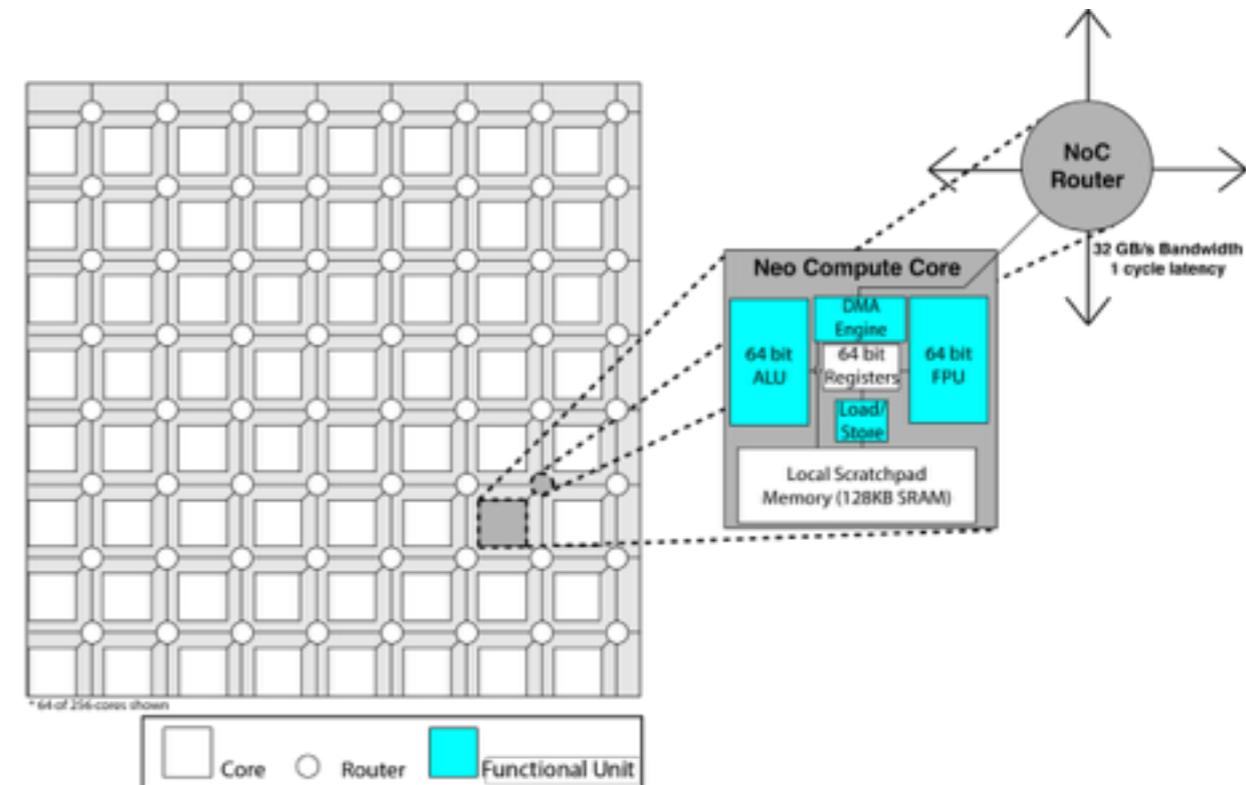
Projections by  
J. Shalf (LBNL)

moving data is expensive – exploiting data locality critical to energy efficiency

# REX Computing: restrict data movement to local core in grid

“Existing processors were designed when amount of energy to move data was roughly equal to energy required to do useful computation with that data.”

Chip	GFLOPs (FP32)	GFLOPs (FP64)	Watt	GFLOPs/W (64)
<b>REX Neo</b>	<b>512</b>	<b>256</b>	<b>4</b>	<b>64</b>
Intel Xeon E5 (2014)	1012	506	145	3.4
Xeon Phi 7120 (2014)	1900	1200	300	4
Xeon Phi Knights Landing (2016)	4500	3000	250	12
NVIDIA K40 (2013)	4290	1430	245	5.8
NVIDIA K80 (2015)	8740	2910	300	9.7
Adapteva Epiphany IV	102	N/A	2	N/A
Kalray MPPA-256	230	43	11	3.9



- Traditional caches out the window
- Tasks access data with high locality
- Inter-task data dependence through router
- *Programming trickier: development toolchains need to evolve*

# “Gordon Bell prize law” increase in sustained performance $\sim 10^3$ every decade



~1 Exaflop/s

100 million or billion processing cores (!)

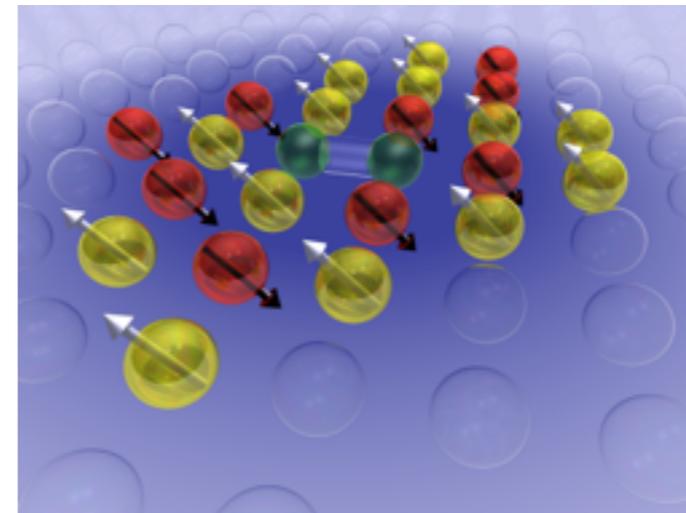


1 Gigaflop/s  
Cray YMP  
8 processors



1.35 Petaflop/s  
Cray XT5  
150'000 processors

1.02 Teraflop/s  
Cray T3E  
1'500 processors



1988

First sustained GFlop/s  
Gordon Bell Prize 1988

1998

First sustained TFlop/s  
Gordon Bell Prize 1998

2008

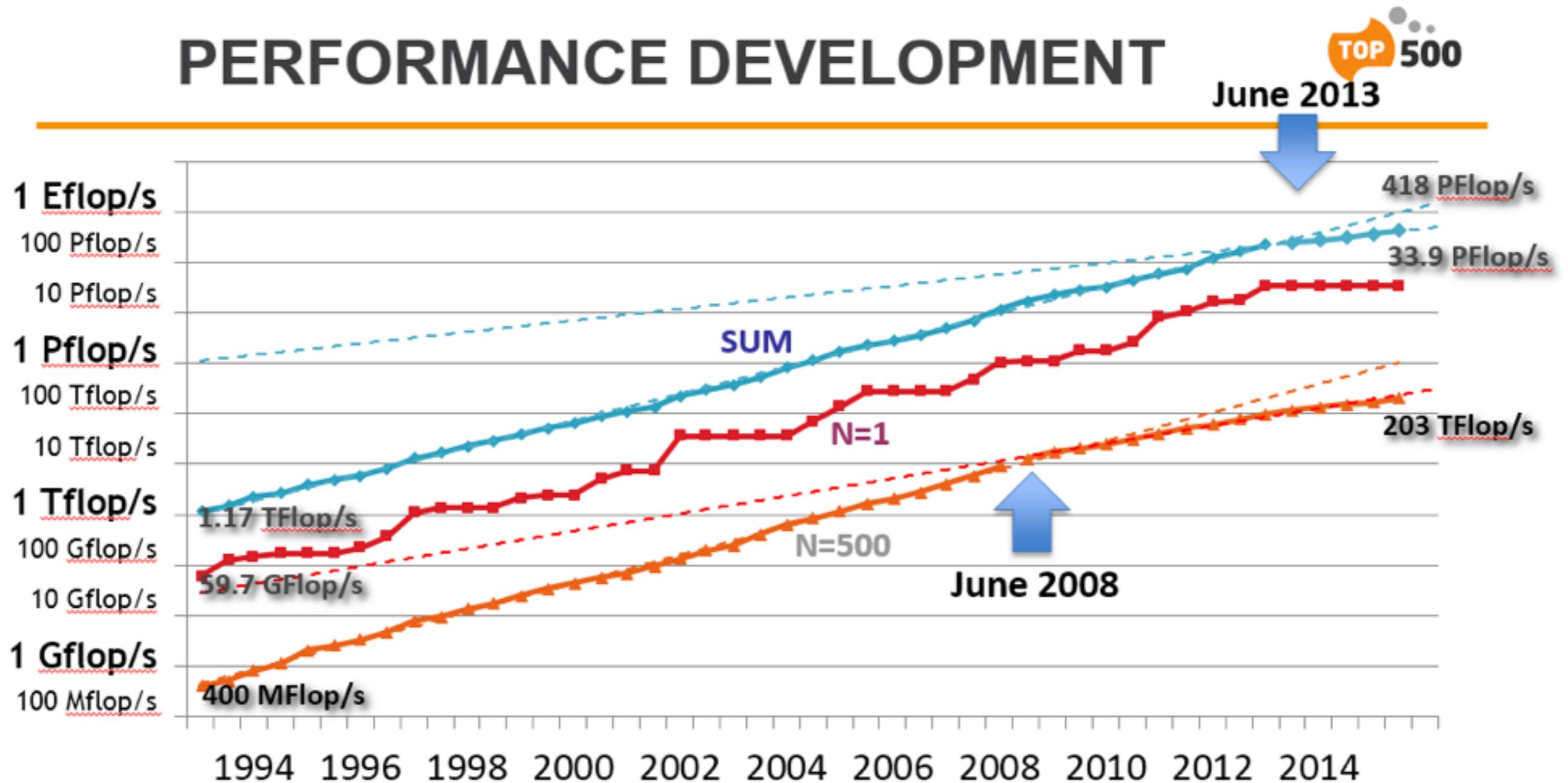
First sustained PFlop/s  
Gordon Bell Prize 2008

2018?

Another 1,000x increase in sustained performance

# But wait: is the increase in HPC performance decreasing?

## PERFORMANCE DEVELOPMENT



# Extrapolating Energy Efficiency

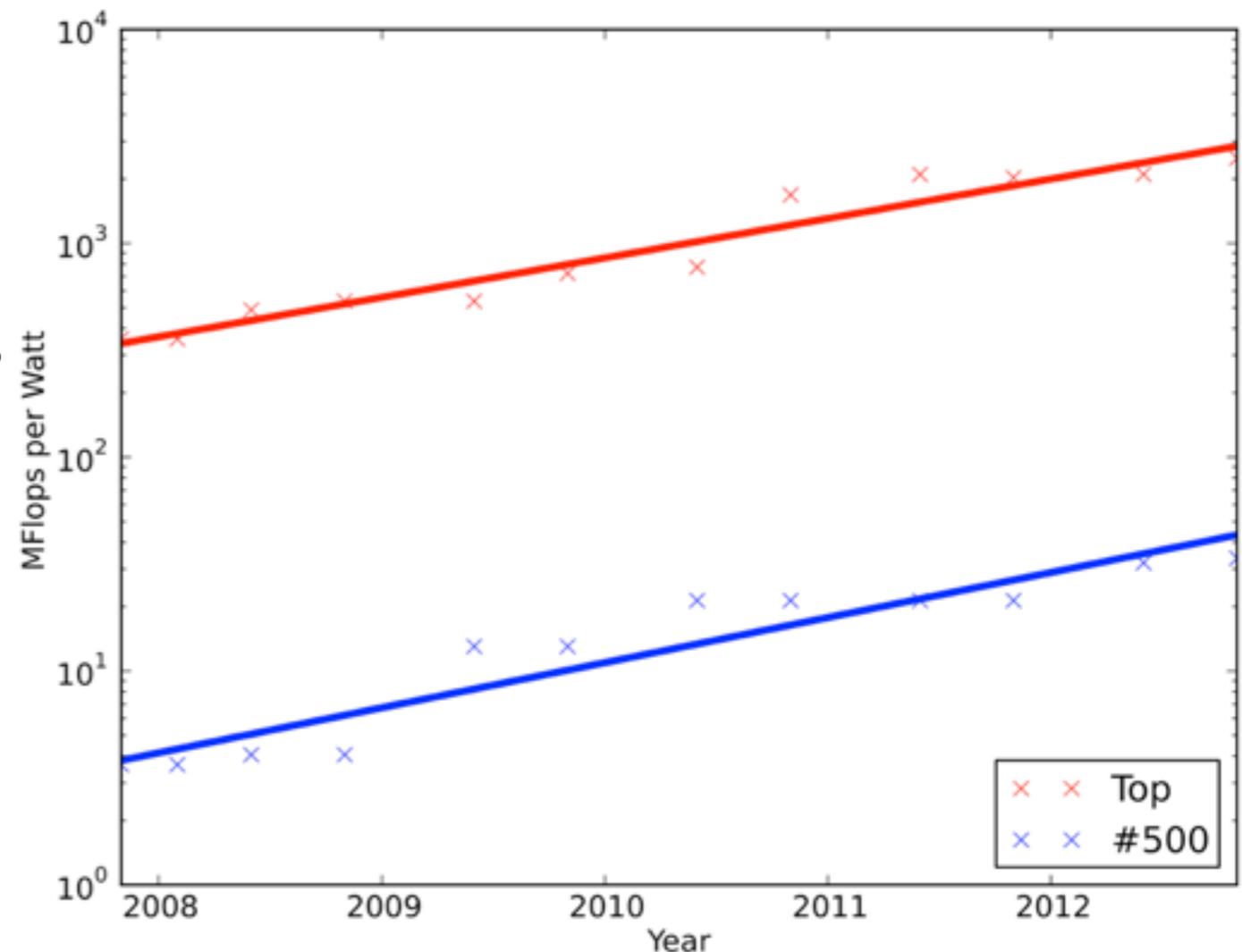
Let's say 20 MW is the socially acceptable limit for a leadership computing facility. Assume Koomey's Law continues

Confirmed by [green500.org](http://green500.org) list in last 8 years

“Greenest PetaFlop/s Machine”  
([green500.org](http://green500.org))

Piz Daint peak: 7.1 PFlops,  
2.3 MW (2013)

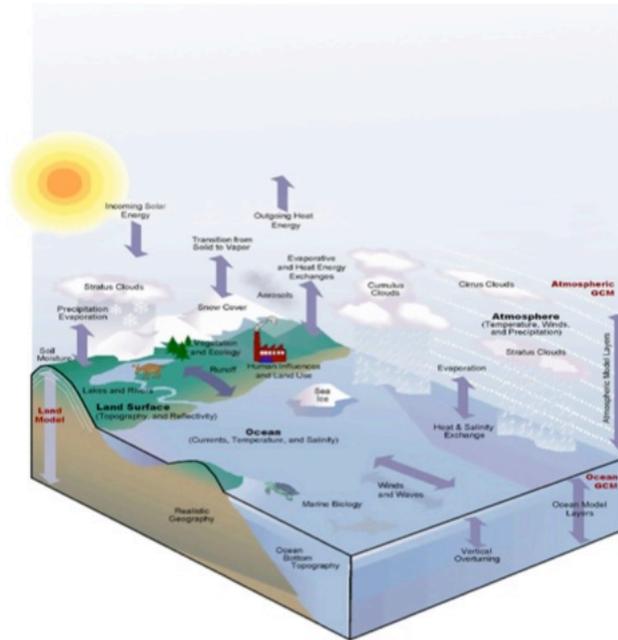
**=> ExaScale Daint:  
6.3 years (2020)**



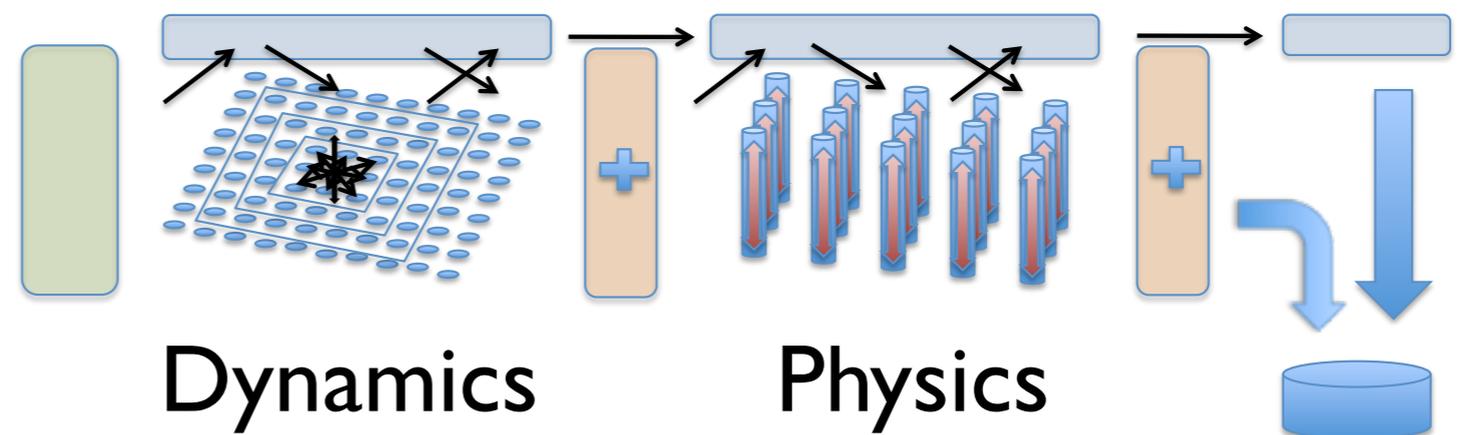
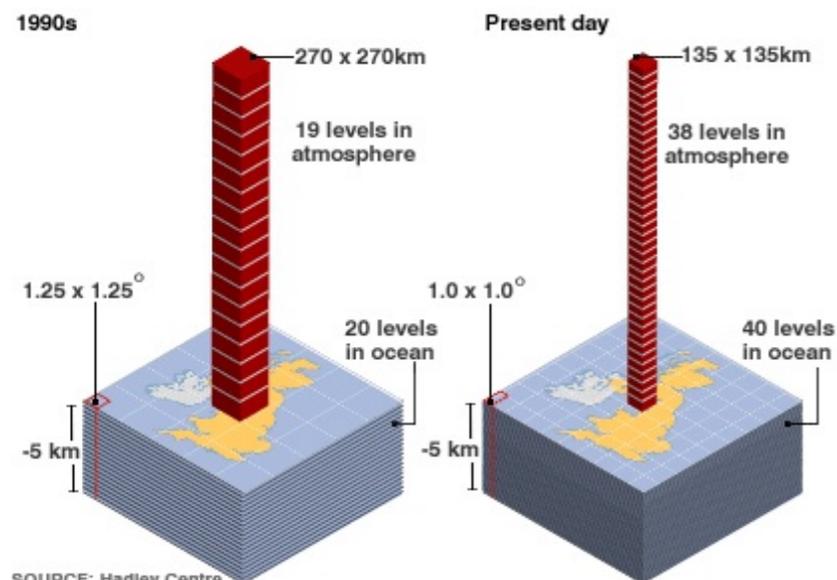
# Part I: Conclusions

- *Exascale will happen within the next 5-10 years*
- Human ingenuity has found ways to circumvent physical limitations. *But the price has been new complexity!*
- The drastic increase in core count means: developers must expose massive multi-level parallelism on the scale of billions of threads, this often requires new algorithms
- New architectures require new programming paradigms  
=> see Part 3
- *For Zettascale (with current concepts): all bets are off !*

# Use Cases: Atmospheric General Circulation Models



- Earth is a giant heat engine
- Movement of the atmosphere (“Dynamics”)
- Parameterization of sub-grid phenomena (“Physics”)
- Dynamics and Physics calculate “tendencies” which alter “state”

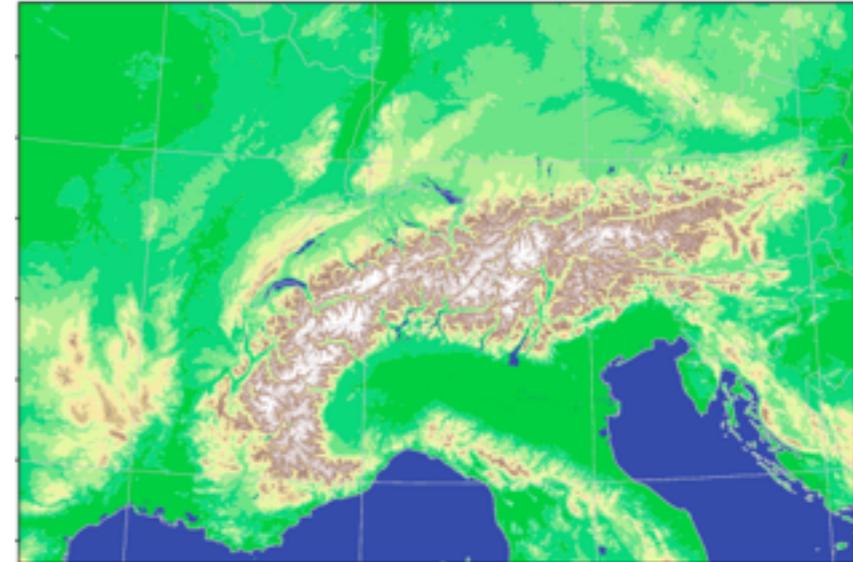


# Comparison approaches to port code to GPU architecture

- COSMO: port of entire Climate/NWP model to GPUs
  - ▶ 2009 - present, ~20 FTEs
  - ▶ Dynamical core rewritten with C++ / DSEL, CUDA backend
  - ▶ OpenACC directives for physical parameterizations
  - ▶ Production model running on dedicated GPU cluster
- ICON: port of parts of climate/NWP model to GPUs
  - ▶ 2011 - 2015, ~3 FTEs
  - ▶ PRACE 2IP WP8 funding (2011-2014)
  - ▶ Dynamical core, advection with OpenACC directives
  - ▶ OpenACC directives for some physical parameterizations

# COSMO HP2C Project (2009-2014)

- COSMO : Limited-area model, structured grid
  - Run operationally by 7 national weather services Germany, Switzerland, Italy, Poland, ...
  - Used for climate research in several academics institution
- Challenge: write code that
  - runs efficiently on different architectures
  - allow domain scientist to easily bring new developments
  - will still be a good fit in the future
- Target architectures : Multicore CPUs (x86) and GPUs, potentially Xeon Phi
- Approaches :
  - New Library **STELLA** : dynamical core. Improved performance and CPU, enable GPU
  - **OpenACC directives** : rest of the code, e.g. physics, assimilation ... . Allow to run on GPU, no improvement on CPU.



Credits: Lapillonne + COSMO team (C2SM/MCH/CSCS/SCS)

# STELLA Stencil library

A stencil definition consists of two parts:

- Loop logic: defines stencil application domain and execution order
- Update function: expression evaluated at every location

The loop logic is platform dependent by the update function is not: treat the two separately

```
DO k = 1, ke
  DO j = jstart, jend
    DO i = istart, iend
```

loop-logic

```
    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)
```

update-function / stencil

```
  ENDDO
ENDDO
ENDDO
```

loop-logic

Credits: Lapillonne + COSMO team (C2SM/MCH/CSCS/SCS)

# Programming the new dynamical core

```
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

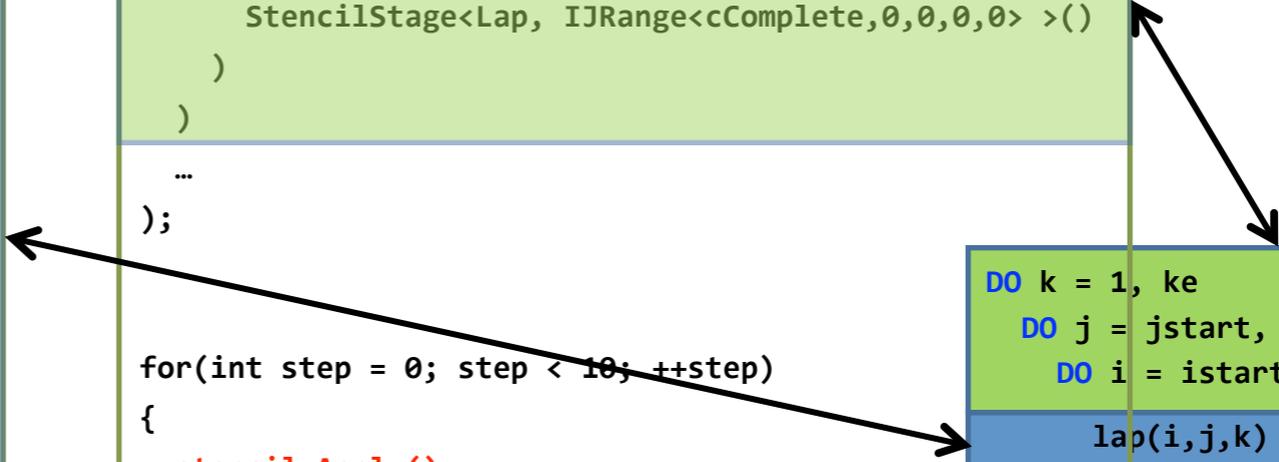
```
IJKRealField lapfield, datafield;
Stencil stencil;

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; step < 10; ++step)
{
    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 in COSMO

velocities	$\frac{\partial u}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \frac{\partial E_h}{\partial \lambda} - v V_a \right\} - \zeta \frac{\partial u}{\partial \zeta} - \frac{1}{\rho a \cos \varphi} \left( \frac{\partial p'}{\partial \lambda} - \frac{1}{\sqrt{\gamma}} \frac{\partial p_0}{\partial \lambda} \frac{\partial p'}{\partial \zeta} \right) + M_u$
	$\frac{\partial v}{\partial t} = - \left\{ \frac{1}{a} \frac{\partial E_h}{\partial \varphi} + u V_a \right\} - \zeta \frac{\partial v}{\partial \zeta} - \frac{1}{\rho a} \left( \frac{\partial p'}{\partial \varphi} - \frac{1}{\sqrt{\gamma}} \frac{\partial p_0}{\partial \varphi} \frac{\partial p'}{\partial \zeta} \right) + M_v$
	$\frac{\partial w}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \left( u \frac{\partial w}{\partial \lambda} + v \cos \varphi \frac{\partial w}{\partial \varphi} \right) \right\} - \zeta \frac{\partial w}{\partial \zeta} + \frac{g}{\sqrt{\gamma}} \frac{\rho_0}{\rho} \frac{\partial p'}{\partial \zeta} + M_w + g \frac{\rho_0}{\rho} \left\{ \frac{(T - T_0)}{T} - \frac{T_0 p'}{T p_0} + \left( \frac{R_v}{R_d} - 1 \right) q^v - q^l - q^f \right\}$
pressure	$\frac{\partial p'}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \left( u \frac{\partial p'}{\partial \lambda} + v \cos \varphi \frac{\partial p'}{\partial \varphi} \right) \right\} - \zeta \frac{\partial p'}{\partial \zeta} + g \rho_0 w - \frac{c_{pd}}{c_{vd}} p D$
temperature	$\frac{\partial T}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \left( u \frac{\partial T}{\partial \lambda} + v \cos \varphi \frac{\partial T}{\partial \varphi} \right) \right\} - \zeta \frac{\partial T}{\partial \zeta} - \frac{1}{\rho c_{vd}} p D + Q_T$
water	$\frac{\partial q^v}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \left( u \frac{\partial q^v}{\partial \lambda} + v \cos \varphi \frac{\partial q^v}{\partial \varphi} \right) \right\} - \zeta \frac{\partial q^v}{\partial \zeta} - (S^l + S^f) + M_{q^v}$
	$\frac{\partial q^{l,f}}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \left( u \frac{\partial q^{l,f}}{\partial \lambda} + v \cos \varphi \frac{\partial q^{l,f}}{\partial \varphi} \right) \right\} - \zeta \frac{\partial q^{l,f}}{\partial \zeta} - \frac{g}{\sqrt{\gamma}} \frac{\rho_0}{\rho} \frac{\partial P_{l,f}}{\partial \zeta} + S^{l,f} + M_{q^{l,f}}$
turbulence	$\frac{\partial e_t}{\partial t} = - \left\{ \frac{1}{a \cos \varphi} \left( u \frac{\partial e_t}{\partial \lambda} + v \cos \varphi \frac{\partial e_t}{\partial \varphi} \right) \right\} - \zeta \frac{\partial e_t}{\partial \zeta} + K_m^v \frac{g \rho_0}{\sqrt{\gamma}} \left\{ \left( \frac{\partial u}{\partial \zeta} \right)^2 + \left( \frac{\partial v}{\partial \zeta} \right)^2 \right\} + \frac{g}{\rho \theta_v} F^{\theta_v} - \frac{\sqrt{2} e_t^{3/2}}{\alpha_{Ml}} + M_{e_t}$

## Timestep

implicit (sparse)      explicit (RK3)      implicit (sparse solver)      explicit (leapfrog)



# Physical Parameterizations: OpenACC directives

- Parallelization: horizontal direction, 1 thread per vertical column
- Most loop structures unchanged
- For time critical parts additional optimizations are considered :  
loop restructuring (reduce kernel overhead), scalar replacements, on the fly computations [1]

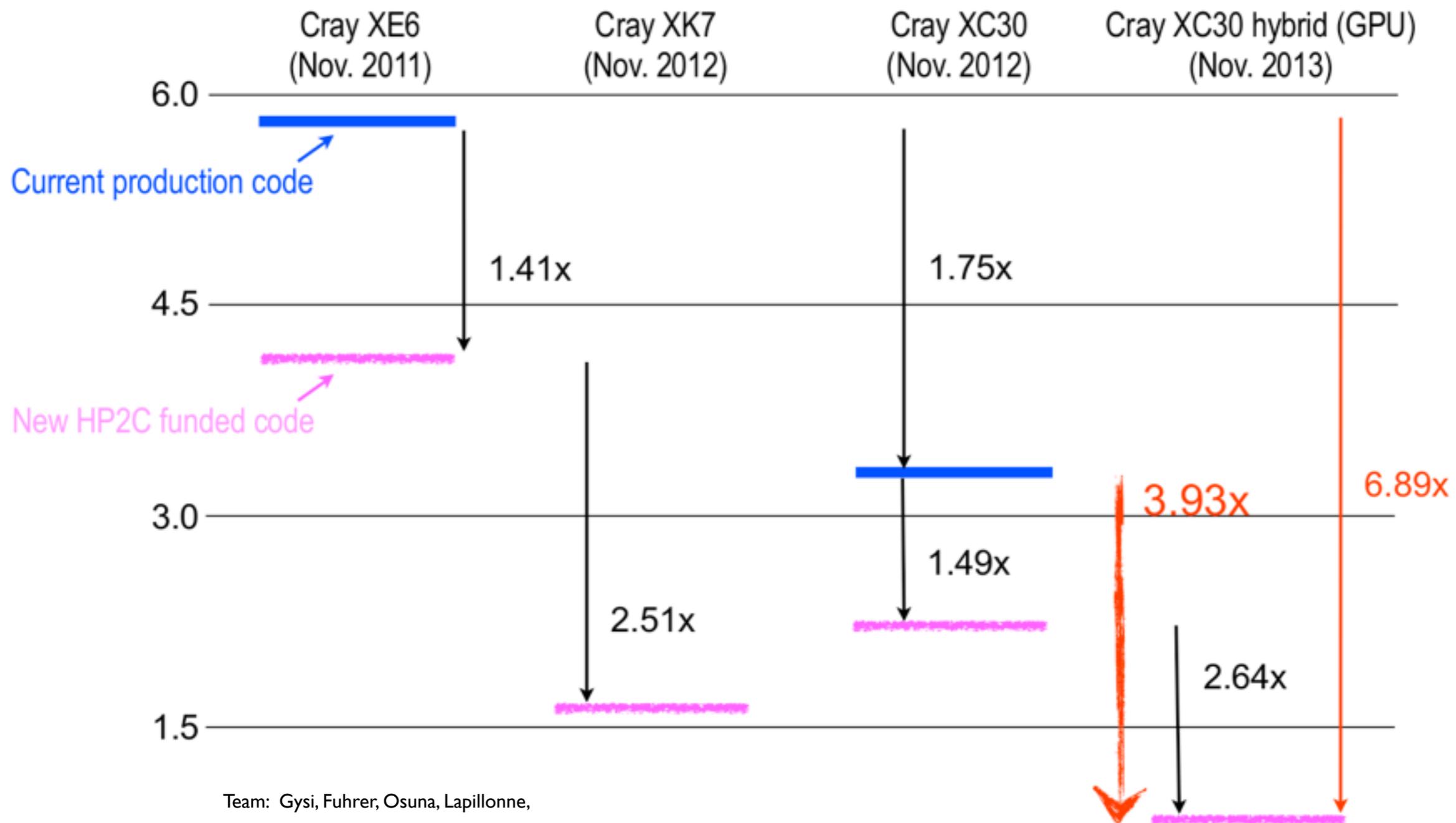
```
!$acc data present(a,c1,c2)
!vertical loop
do k=2,Nz
!work 1
!$acc parallel loop vector_length(N)
do ip=1,nproma
c2(ip)=c1(ip,k)*a(ip,k-1)
end do
!$acc end parallel loop
!work 2
!$acc parallel loop vector_length(N)
do ip=1,nproma
a(ip,k)=c2(ip)*a(ip,k-1)
end do
!$acc end parallel loop
end do
!$acc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



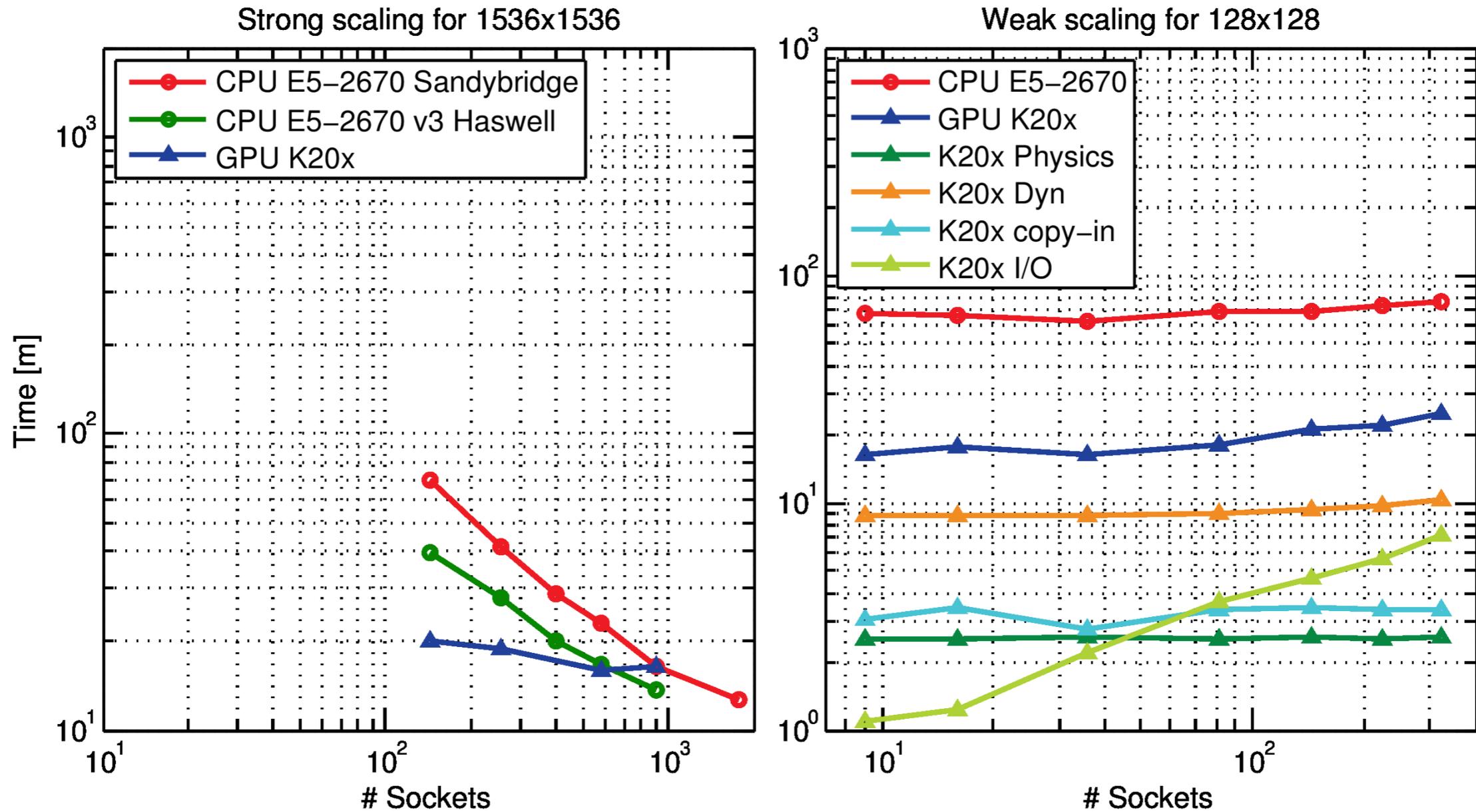
```
!$acc data present(a,c1)
!$acc parallel loop vector_length(N)
do ip=1,nproma
!vertical loop
do k=2,Nz
!work 1
c2=c1(ip,k)*a(ip,k-1)
!work 2
a(ip,k)=c2*a(ip,k-1)
end do
end do
!$acc end parallel loop
!$acc end data
```

- Some GPU optimizations degrade performance on CPU : keep separate routines

# COSMO Performance: Improvement Energy-to-solution



# COSMO scalability

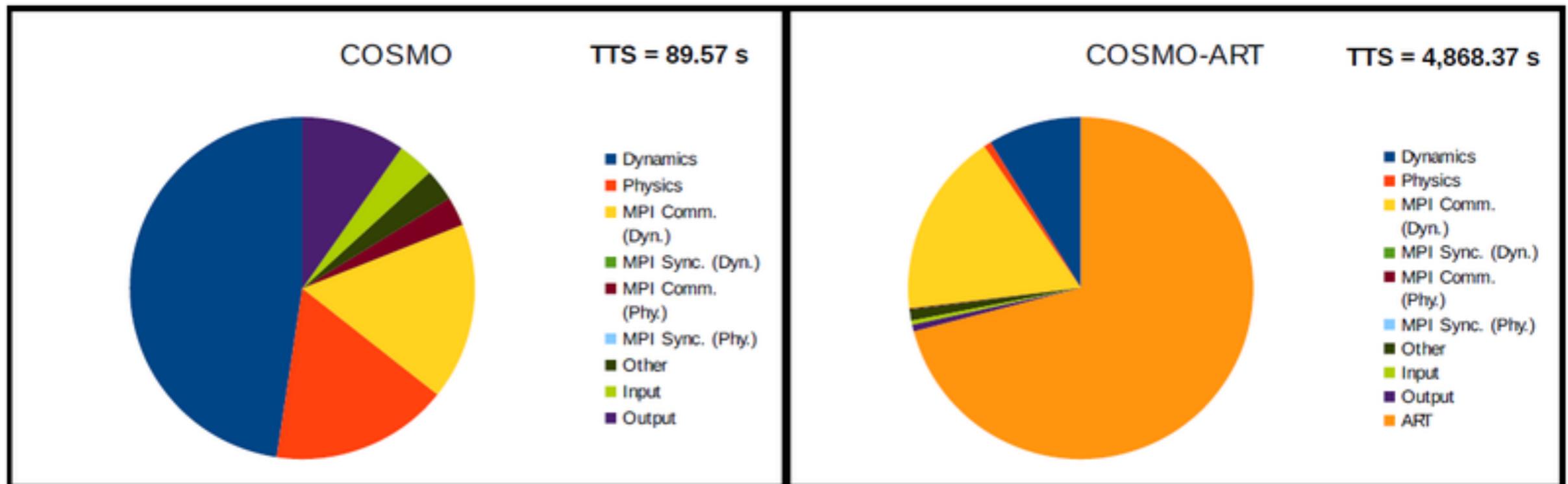


**Bottom Line: scaling to some 100's of GPU nodes**

Benchmarking: David Leutwyler, ETH

# COSMO-ART: Extension for atmospheric chemistry

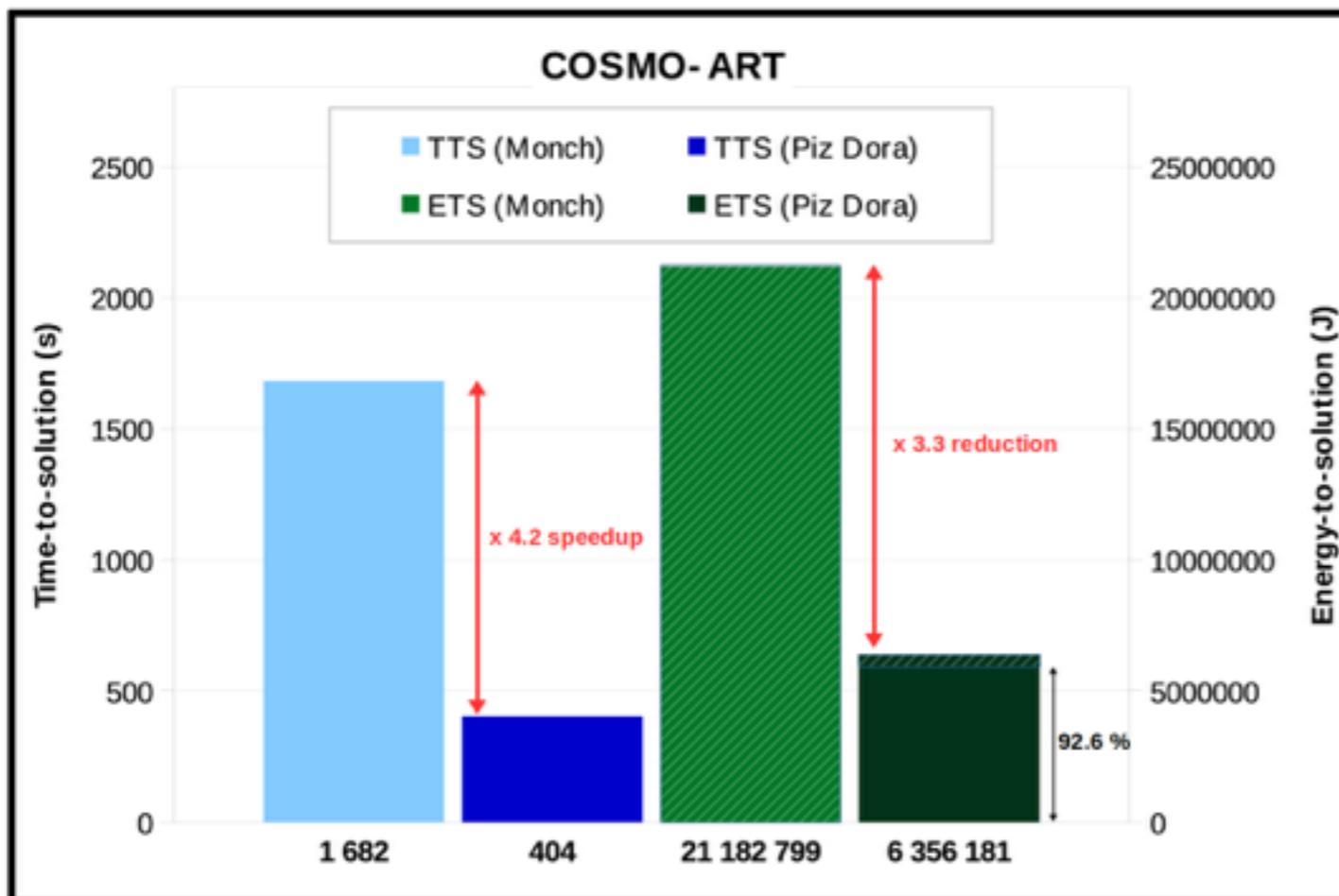
The Aerosol Reactive Transport (ART, Karlsruhe Inst. of Tech.) extension to COSMO calculates concentrations of aerosols, and advects (disperses) them according to the winds. Massive increase in computing time!



- *Exa2Green* EU-funded FP7 project on energy efficiency (2013-2015)
- *Goal: 5x improvement in energy-to-solution for COSMO-ART*
- *Tools: optimizations, new algorithms, emerging architectures,...*

# COSMO-ART: Extension for atmospheric chemistry

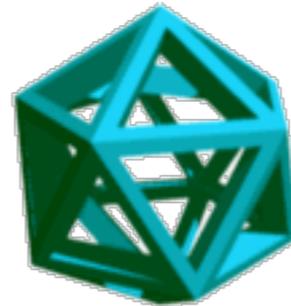
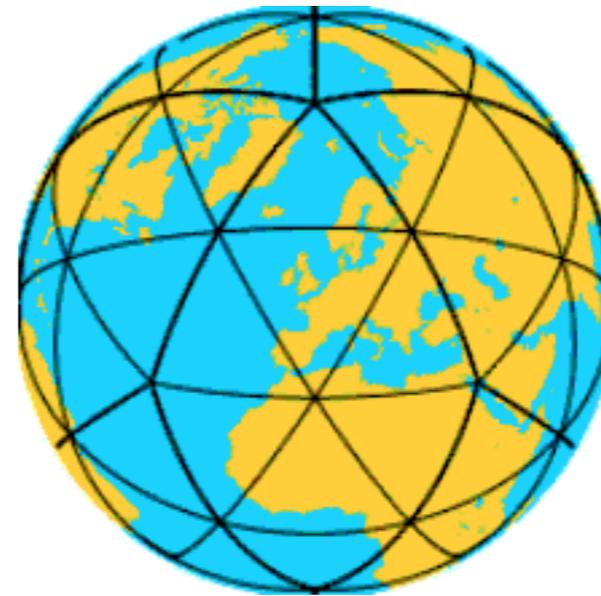
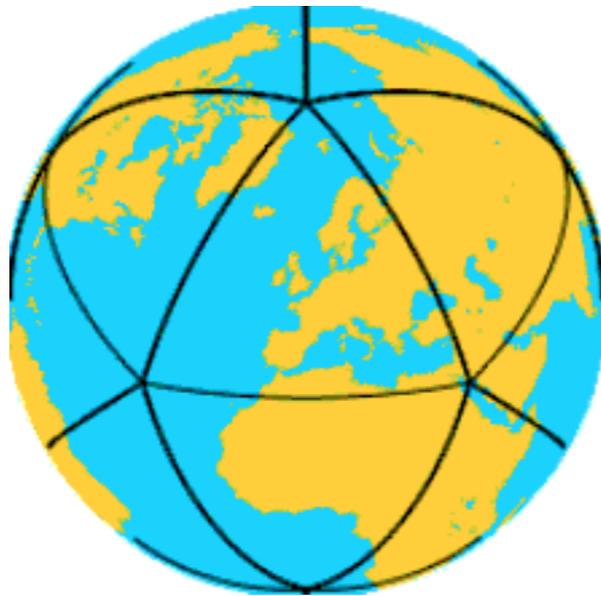
- Single-precision ART version
- New time integrator (Fanourgakis, Lelieveld, Taraborelli, developed in PRACE 2IP WP8)
- Replaced COSMO with “OPCODE” COSMO from HP2C project



## Remarks:

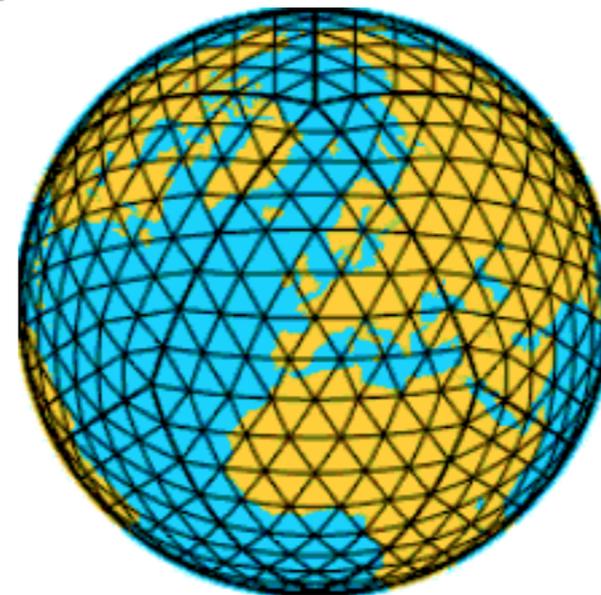
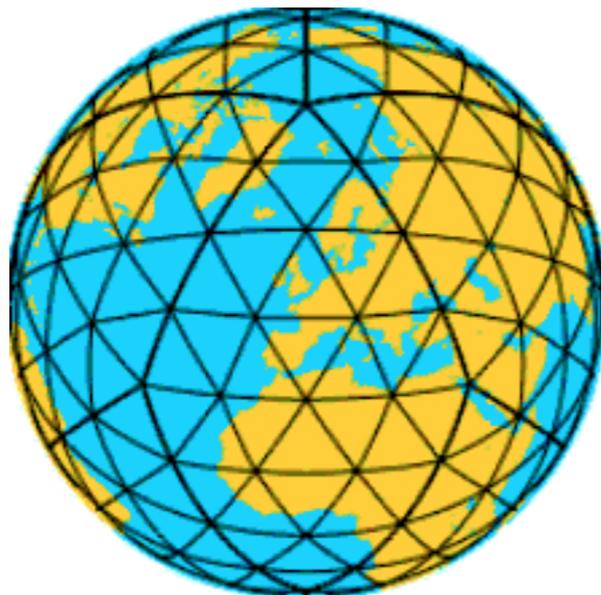
- GPU implementation not achieved (likely > 5x)
- Also optimized for time-to-solution; pure E2S optimization would yield more than 5x improvement

# ICON Horizontal Grid



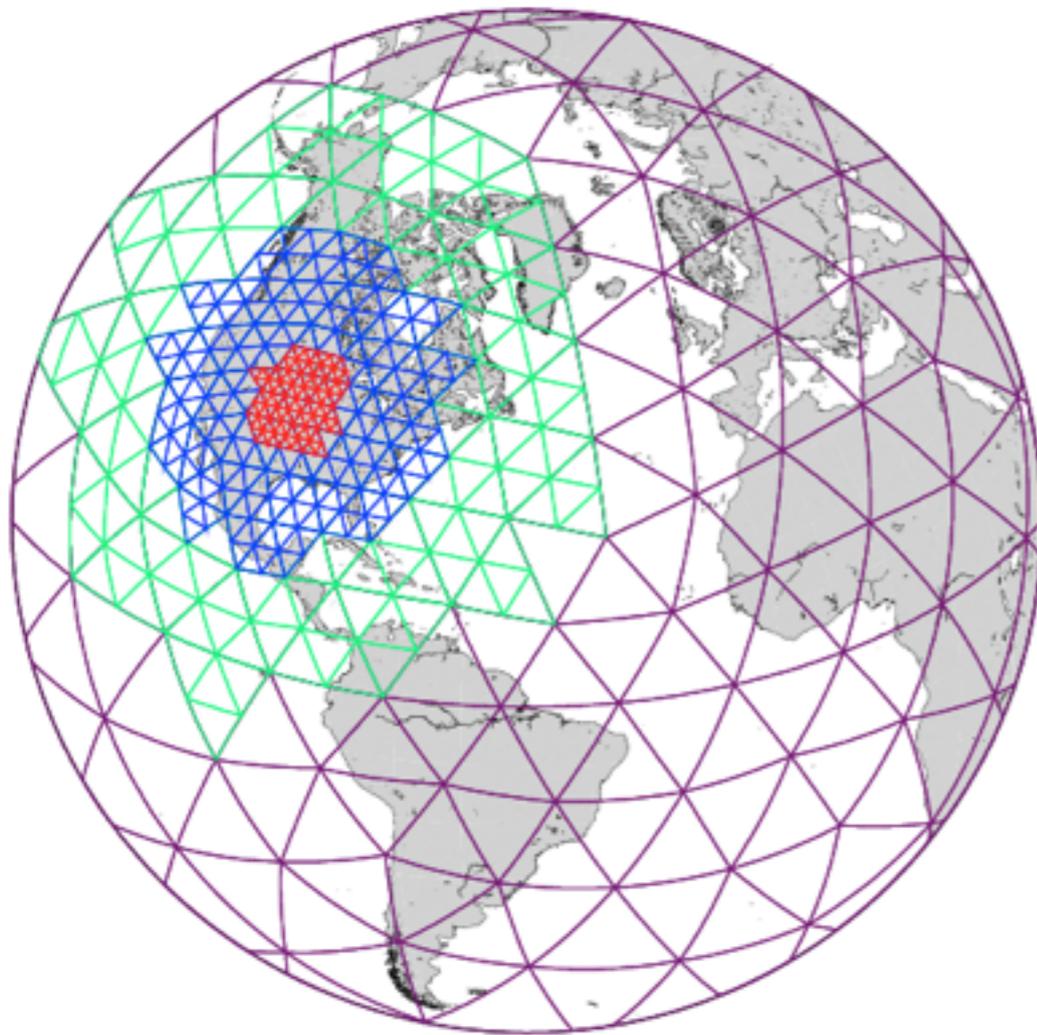
R2B0

R2B1

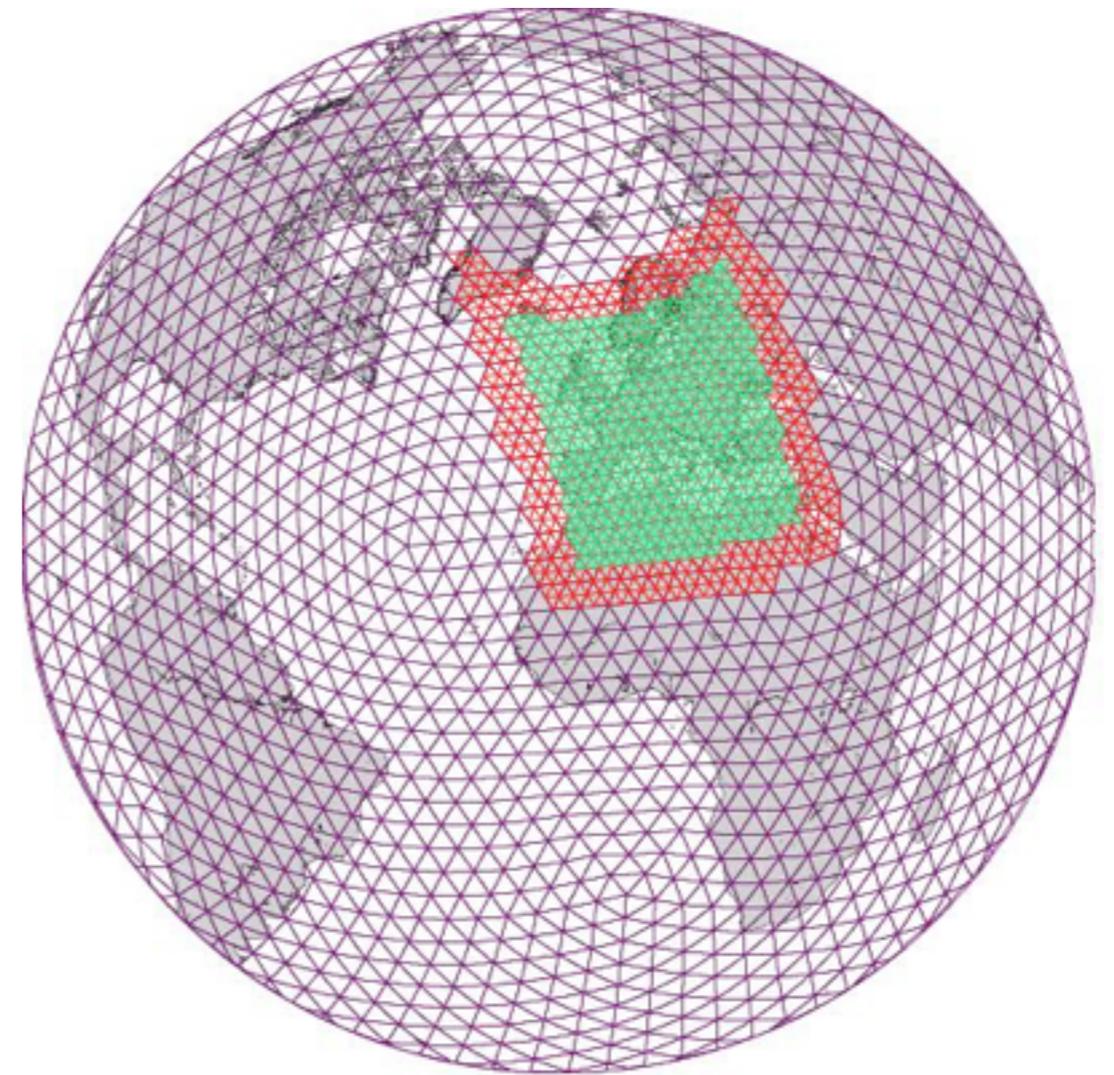


R2B2

# Nested Domains



circular nests



latitude-longitude nests

# OpenACC: Copying data to/ from GPU outside of time loop

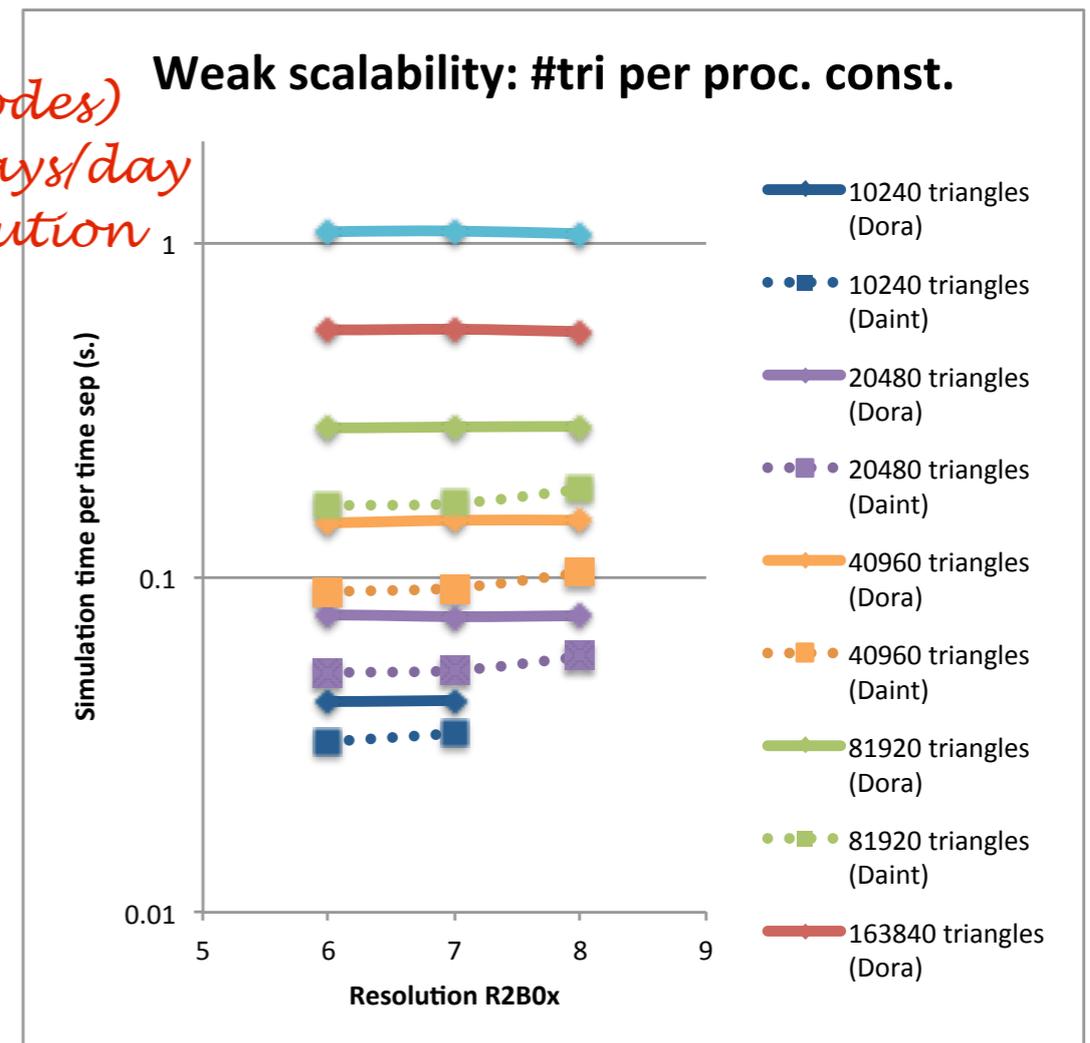
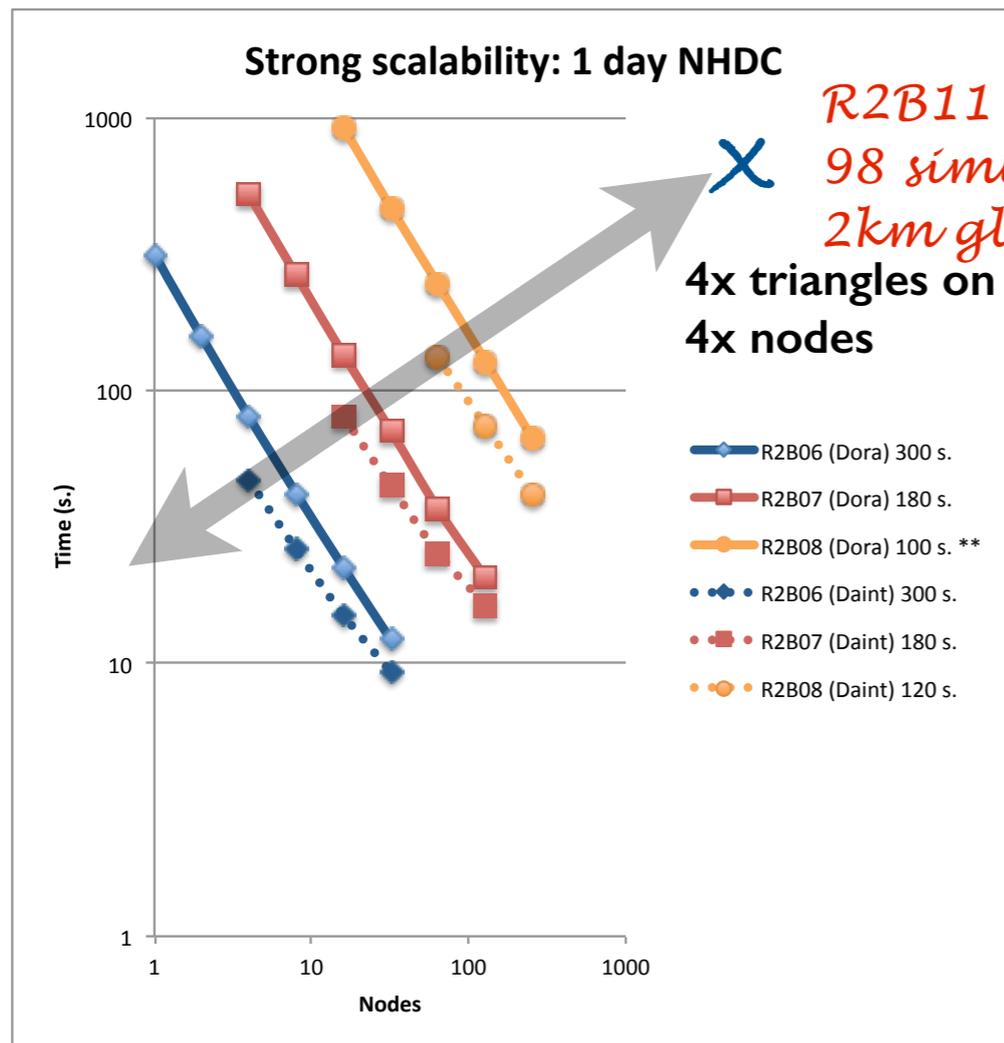
```
#if defined( _OPENACC )
    CALL init_gpu_variables( )
    CALL save_convenience_pointers( )
!$ACC DATA COPY( p_int_state, p_patch, p_nh_state, prep_adv ), IF ( i_am_accel_node )
    CALL refresh_convenience_pointers( )
#endif
    TIME_LOOP: DO jstep = (jstep0+jstep_shift+1), (jstep0+nsteps)
    :
    ENDDO TIME_LOOP
#if defined( _OPENACC )
    CALL save_convenience_pointers( )
!$ACC END DATA
    CALL refresh_convenience_pointers( )
    CALL finalize_gpu_variables( )
#endif
```

# Example: 3-D Divergence (mo\_math\_divrot.f90)

```
#ifdef OPENACC_MODE
!$ACC DATA PCOPYIN( ptr_patch, ptr_int, vec_e ), PCOPY( div_vec_c ), IF( i_am_accel_node )
!ACC_DEBUG UPDATE DEVICE ( vec_e, div_vec_c ) IF( i_am_accel_node )
!$ACC PARALLEL, PRESENT( ptr_patch, ptr_int, vec_e, iidx, iblk, div_vec_c ), &
!$ACC IF( i_am_accel_node )
!$ACC LOOP GANG
#else
!$OMP PARALLEL
!$OMP DO PRIVATE(jb,i_startidx,i_endidx,jc,jk) ICON_OMP_DEFAULT_SCHEDULE
#endif
  DO jb = i_startblk, i_endblk
    CALL get_indices_c(ptr_patch, jb, i_startblk, i_endblk, &
                     i_startidx, i_endidx, rl_start, rl_end)
!$ACC LOOP VECTOR
  DO jc = i_startidx, i_endidx
    DO jk = slev, elev
      div_vec_c(jc,jk,jb) = &
        vec_e(iidx(jc,jb,1),jk,iblk(jc,jb,1)) * ptr_int%geofac_div(jc,1,jb) + &
        vec_e(iidx(jc,jb,2),jk,iblk(jc,jb,2)) * ptr_int%geofac_div(jc,2,jb) + &
        vec_e(iidx(jc,jb,3),jk,iblk(jc,jb,3)) * ptr_int%geofac_div(jc,3,jb)
    END DO
  END DO
END DO
#endif OPENACC_MODE
!$ACC END PARALLEL
!ACC_DEBUG UPDATE HOST(div_vec_c) IF( i_am_accel_node )
!$ACC END DATA
#else
!$OMP END DO NOWAIT
!$OMP END PARALLEL
#endif
```

# ICON non-hydrostatic dynamical core scaling

Compare ICON Trunk Piz Dora (2x Haswell sockets) vs. Piz Daint nodes w/ K20x (both with Cray CCE)



$$\frac{[R2B07 Dora 64 (s)]}{[R2B07 Daint 64 (s)]} = 1.45$$

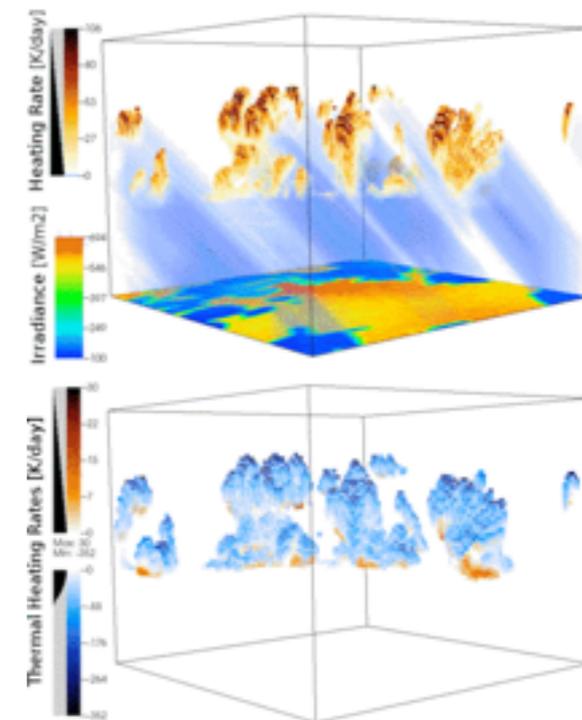
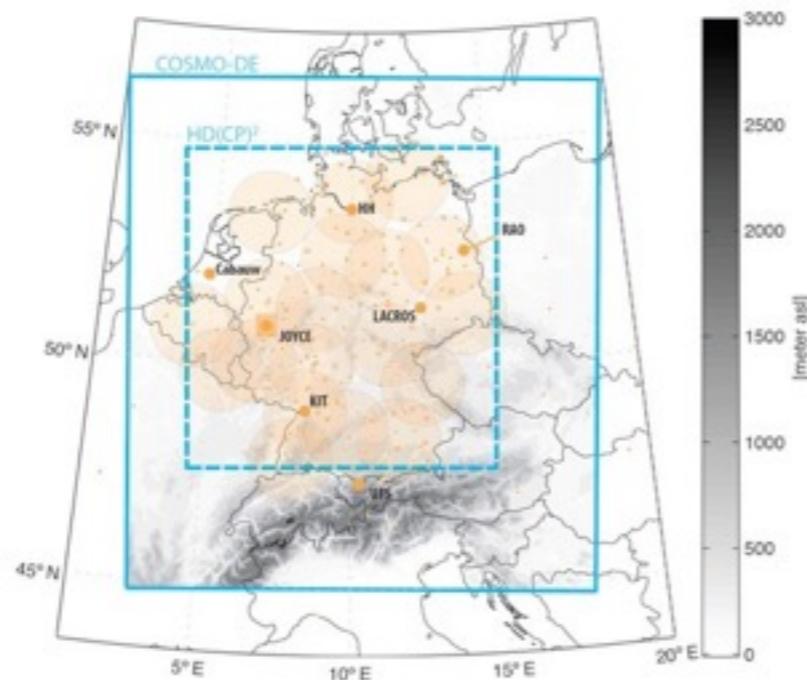
$$\frac{[BW Daint node (GB/s)]}{[BW Dora node (GB/s)]} = \frac{165}{110} = 1.5$$

# ICON: what climate simulation could we do with ExaScale?

Improvement in memory speed and size is key!

Factor of 10x in horizontal (200m resolution) should be possible (100-200x memory); time step now 1/10s., partially compensated by better memory bandwidth.

Good news: *ICON HDCP<sup>2</sup> project runs at 200m! subgrid processes resolved, mostly dynamics*



Bad news: at best back at 10 days/day. Typical simulation: 10y+  
1 year @20 MW = 175 GWh => 20M Euro => 35 kt CO<sub>2</sub>e

# Part 2: What we learned from programming Piz Daint

- *Piz Daint forces us to think about performance portability, not only to CPU and GPU systems (also Xeon Phi, and newer architectures)*
- Low-level programming (CUDA, CUDAFortran, OpenCL) also employed: challenging and do not ensure software longevity
- Directive-based approaches (OpenMP, OpenACC) seem easier at first sight, but have have numerous downsides
  - implicit synchronizations
  - lack of compiler support
- Full rewrites (e.g., COSMO dynamics) not necessarily harder
- Embedded domain-specific languages (STELLA) promising

# Part 3: How do we program Exascale Machines?

Fundamental problem: HPC machines are upgraded every few years, but models are expected to run for decades... How do we

- protect scientists' investment in rewriting software?
- ensure their software is performance-portable to new platforms?
- use paradigms which have a long-term future?
- and, if even possible, separate the concerns of the scientist and the HPC programmer?

*Simple reality: HPC is not a big business sector; tools that endure come from larger IT markets! Forget about Chapel, UPC, maybe even Fortran*

*“The war between Fortran and C++ is over, and C++ won.”*

— Michael Heroux (Sandia NL)



# Taking stock of HPC programming tools

- HPC Languages: Fortran, C, C++, possibly Python
- HPC Low-level programming systems
  - MPI (1994): interface (library) for distributed memory
  - OpenMP (1999) / OpenACC (2011): multi-threading interface with support for accelerators
  - CUDA (2007): proprietary extensions to C++ for accelerators
  - OpenCL (2009): open standard C99 extensions for heterogeneous platforms (CPU, GPU, FPGA, DSP,...)

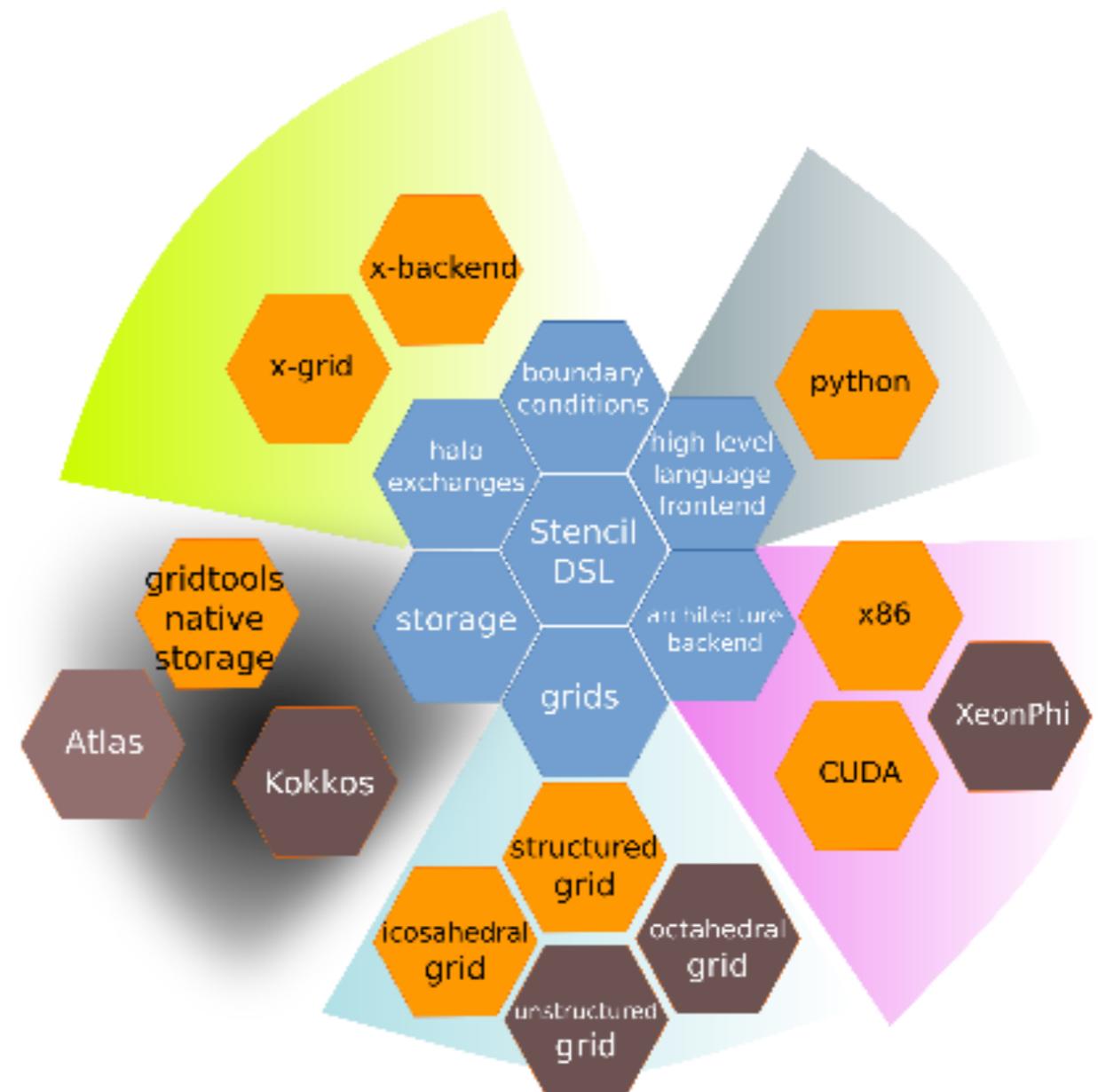
*Avoiding debate: none of these address all the issues from previous slide*

*Still: many experts believe MPI+XXX will endure into Exascale*

# Domain-Specific Frameworks

## e.g. GridTools Ecosystem

- Set of C++ APIs / libraries
- Larger class of *stencil-based* problems
  - From structured to “less” structured
- Performance Portability
- Intuitive interface
  - Get application field specific concepts
- Basic building blocks
  - Reminiscent to the application fields
  - Each application could be more precise
- Composability
  - Shared data structures, naming conventions, API structure
- Interoperability
  - Even usable from Python



Soon publicly available: <https://github.com/eth-cscs/gridtools>

# GridTools: brief example

- Use functional specification within the loop body  
e.g., `f_stencil<divergence, 1, 0, 0>(input)`
- Operator function: loop over all points, perform operations involving neighbors, update

```

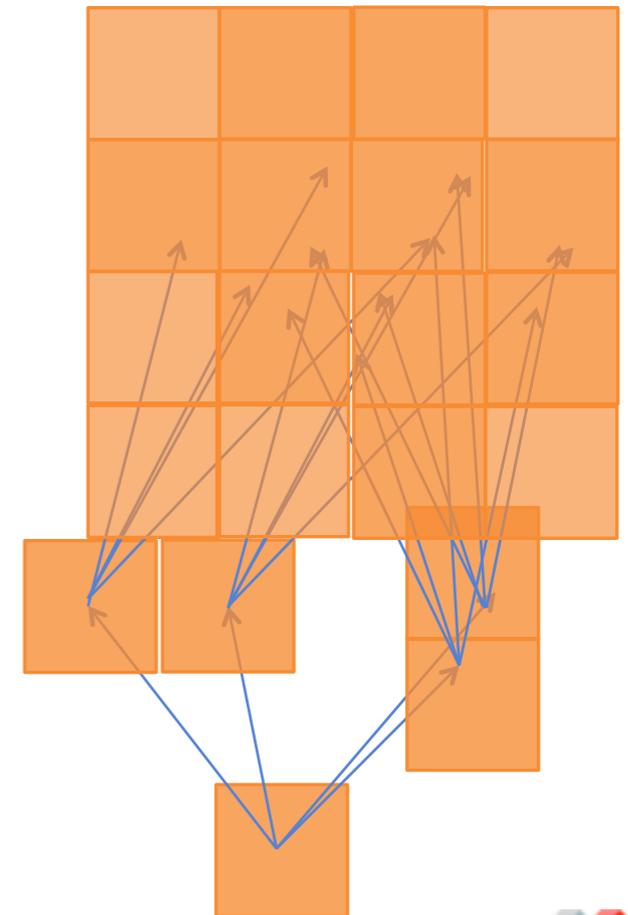
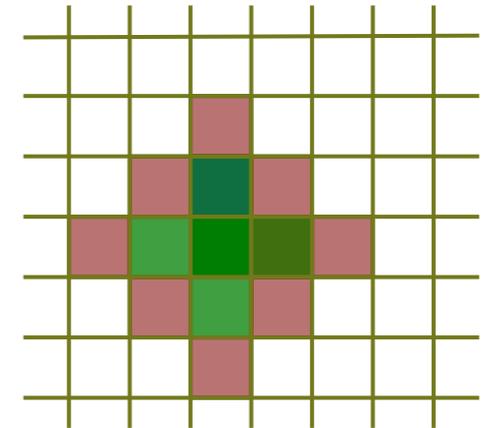
struct lap_function {
    using out = arg_type<0>;
    using in = const arg_type<1, range<-1, 1, -1, 1> >;
    using arg_list = logical_aggregator<out, in>;

    template <typename FieldAccessor>
    GT_FUNCTION
    static void Do(FieldAccessor const & eval, lap_region) {
        eval(out()) = 4*eval(in()) -
            (in( 1, 0, 0) + in( 0, 1, 0) +
             in(-1, 0, 0) + in( 0,-1, 0));
    };

    computation* horizontal_diffusion =
        make_multistage<BACKEND> (
            execute<parallel>(), field_pack, coords,
            bind_operator<lap>(p_lap(), p_in()),
            make_independent(
                bind_operator<flx>(p_flx(), p_in(), p_lap()),
                bind_operator<fly>(p_fly(), p_in(), p_lap()),
                bind_operator<out>
                    (p_out(), p_in(), p_flx(), p_fly(), p_coeff())
            );
    };

```

Multi-stage  
composition:



# Task-based / data-centric application architecture

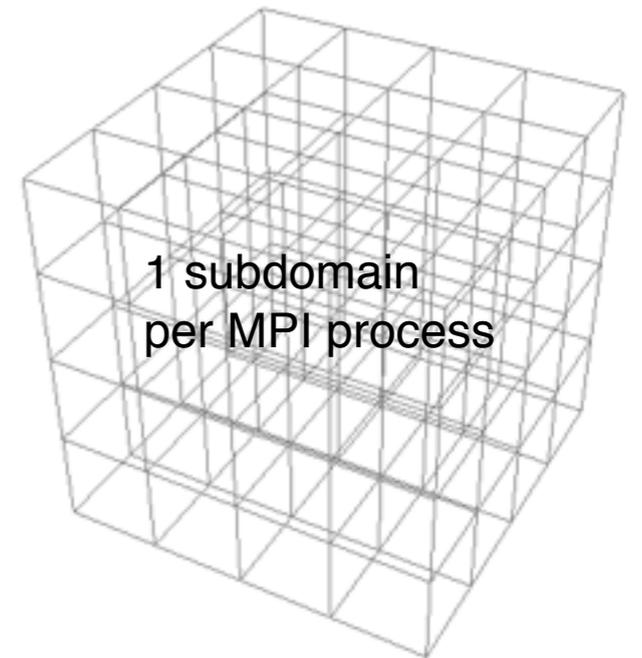
## Classical SPMD programming (1 subdomain per process)

Advantages: (numerous)

- Portable to many systems
- Clear separation of parallel model dependencies (e.g., through message passing, halo exchange, etc.)
- Domain scientists write sequential code on subdomain

Disadvantages:

- Not well suited to emerging many-core architectures
- sensitive to latencies, load imbalance, heterogeneity



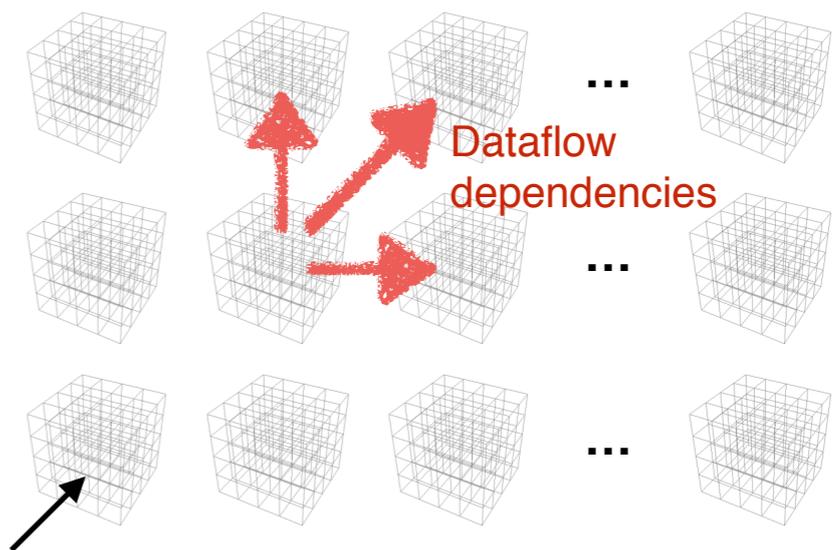
## Task-centric / dataflow execution model

Advantages: (numerous)

- Shares most strengths of SPMD model
- Better suited for emerging many-core architectures
- Tolerates latencies, heterogeneity

Disadvantages:

- Data dependencies must be defined carefully



Patch (subdomain, subgraph)  
Many per MPI process

# HPX: Create unified API for task-based parallelism

*Motivation: processors are now multi-core, race conditions and thread management are complicated. Billion-way asynchronous parallelism needed*

HPX (High Performance ParalleX) is a general purpose C++ runtime system for parallel and distributed applications of any scale. Some key concepts:

- Standards-based approach — extension to C++11/14
- latency hiding
- lightweight, fine-grain parallelism (as opposed to heavyweight threads)
- constraint-based synchronization (rather than global barriers)
- adaptive locality control (instead of static data distribution)

```
int universal_answer() { return 42; }
void wait_for_7_million_years(){
    hpx::this_thread::sleep(std::chrono::years(7000000)); }
// ...
hpx::future<hpx::tuple<hpx::future<int>, hpx::future<void>>> f =
    hpx::when_all(
        hpx::async(&universal_answer),
        hpx::async(&wait_for_7_million_years);
    );
```

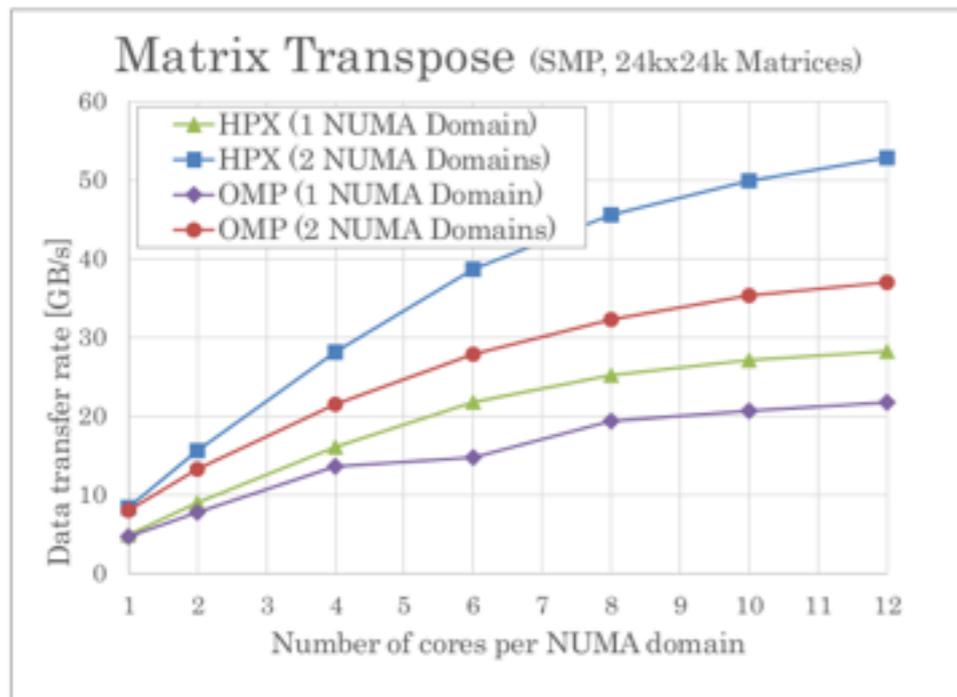
# HPX matrix transposition

```
void transpose(std::vector<double>& A,
std::vector<double>& B)
{
#pragma omp parallel for
  for (std::size_t i = 0; i != order; ++i)
    for (std::size_t j = 0; j != order; ++j)
      B[i + order * j] = A[j + order * i];
}
int main()
{
  std::vector<double> A(order * order);
  std::vector<double> B(order * order);
  transpose(A, B);
}
```

```
void transpose(hpx::future<sub_block> Af, hpx::future<sub_block> Bf,
boost::uint64_t block_order, boost::uint64_t tile_size) {
  const sub_block A(Af.get());
  sub_block B(Bf.get());

  if(tile_size < block_order) {
    for(boost::uint64_t i = 0; i < block_order; i += tile_size) {
      for(boost::uint64_t j = 0; j < block_order; j += tile_size) {
        boost::uint64_t max_i = (std::min)(block_order, i + tile_size);
        boost::uint64_t max_j = (std::min)(block_order, j + tile_size);
        for(boost::uint64_t it = i; it != max_i; ++it) {
          for(boost::uint64_t jt = j; jt != max_j; ++jt){
            B[it + block_order * jt] = A[jt + block_order * it];
          }
        }
      }
    }
  }
  else { ... }
}

phase_futures.push_back(
  hpx::dataflow(
    &transpose,
    A[from_block].get_sub_block(A_offset, block_size),
    B[b].get_sub_block(B_offset, block_size),
    block_order, tile_size
  )
);
```



# HPX — The API

As close as possible to C++1x standard library, where appropriate, for instance:

C++ Standard Library	HPX extension
<code>std::thread</code>	<code>hpx::thread</code>
<code>std::future</code>	<code>hpx::future</code> (including N4107, ‘Concurrency TS’)
<code>std::async</code>	<code>hpx::async</code> (including N3632)
<code>std::bind</code>	<code>hpx::bind</code>
<code>std::function</code>	<code>hpx::function</code>
<code>std::tuple</code>	<code>hpx::tuple</code>
<code>std::any</code>	<code>hpx::any</code> (P0220, ‘Library Fundamentals TS’)
<code>std::parallel::for_each</code>	<code>hpx::parallel::for_each</code> (N4105, ‘Parallelism TS’)
<code>std::parallel::task_block</code>	<code>hpx::parallel::task_block</code>
<code>std::vector</code>	<code>hpx::partitioned_vector</code>

- Extensions to the standard APIs, where necessary

<https://github.com/STELLAR-GROUP/hpx>

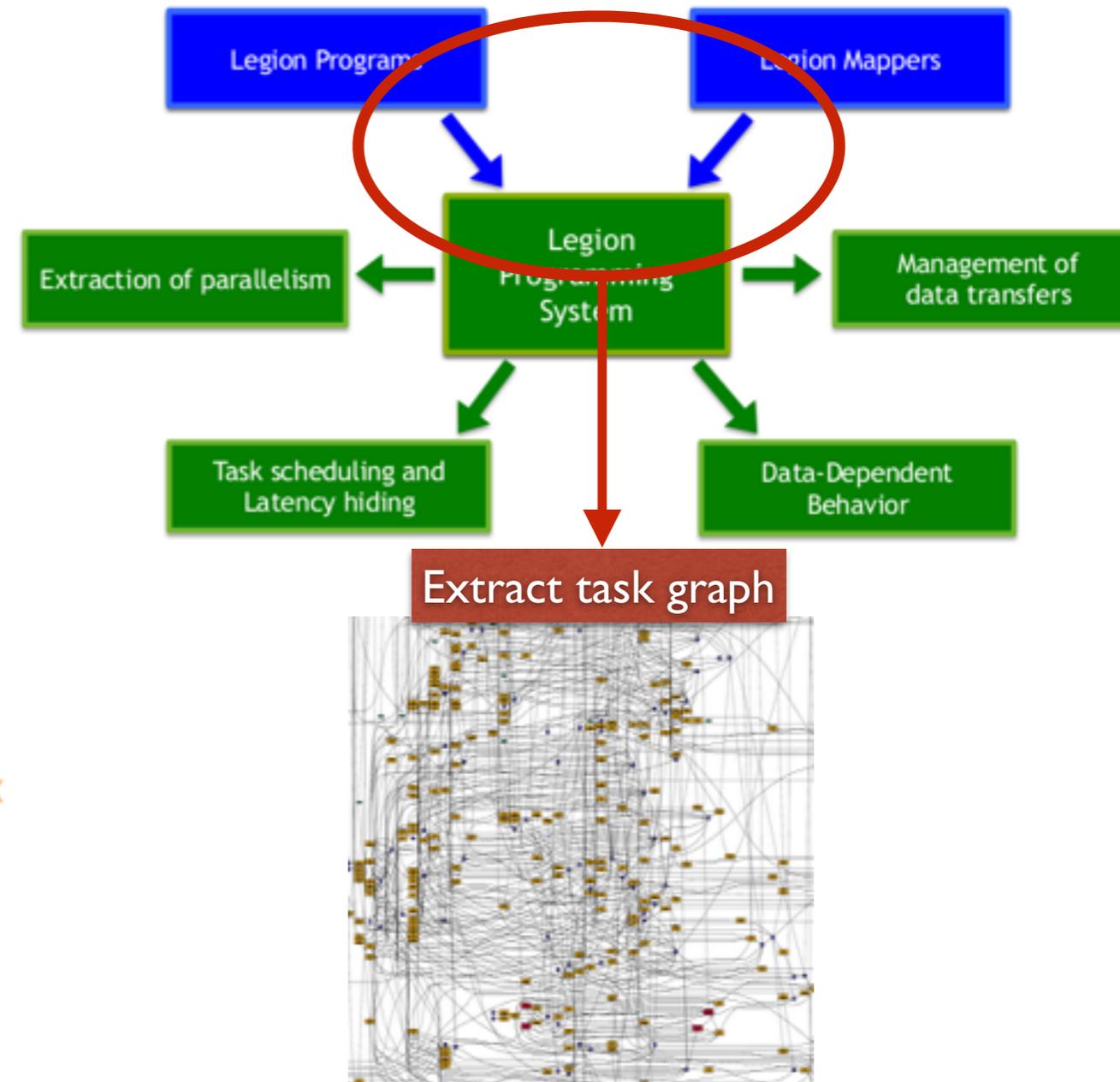
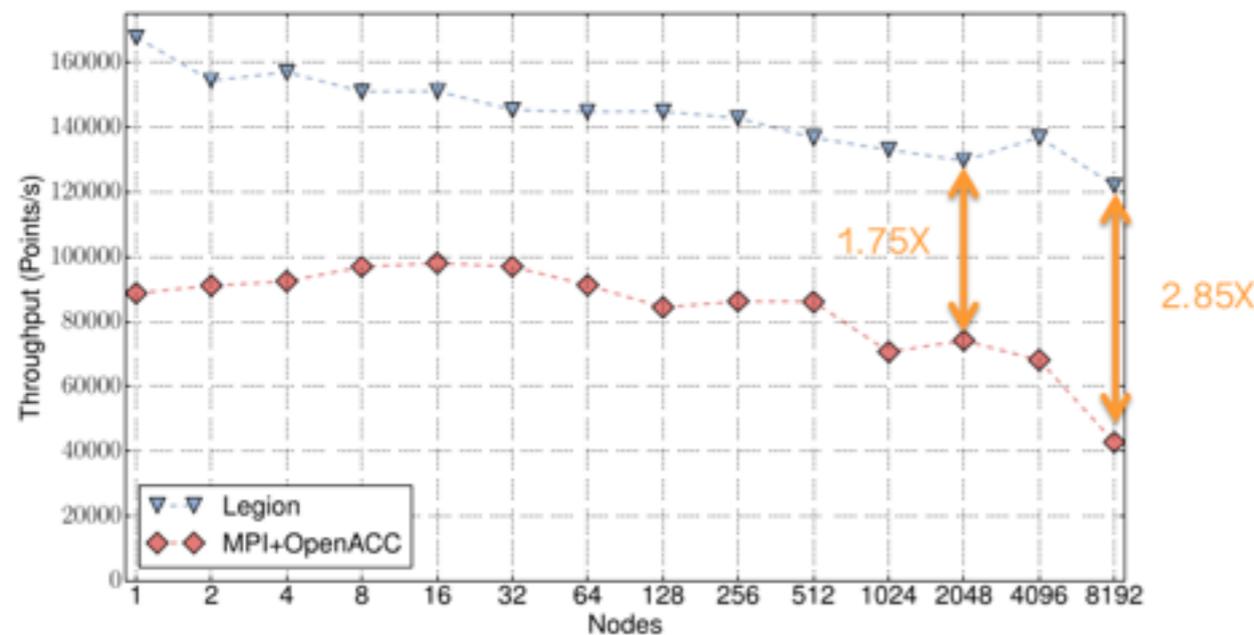
# Legion: yet another data-centric programming paradigm

Legion is a data-centric programming model for writing HPC applications for distributed heterogeneous systems. Key concepts

- user specification of data properties
- automation to 'extract' parallelism
- abstractions for tasks and data

S3D turbulent combustion

Weak scaling results on Titan out to 8K nodes



<http://legion.stanford.edu/>

Credits: Bauer, McCormick, Legion team

# Programming paradigms: parting thoughts

- *HPC is subject to forces from larger markets, these may determine the tools we use*
- *On the level of scientific fields, domain-specific frameworks can hide the industry de-facto standards underneath (separation of concerns...)*
- Synchronous programming models, e.g., OpenMP, OpenACC may give way to task-based models, e.g., HPX, Legion, ...
- But: the safest bet is to use paradigms embedded in living languages, such as C++, Python
- Even MPI could be subsumed if proper support in languages is available, as well as adequate distributed memory runtime systems

# Take home messages

- *Moore's Law is doomed, but we still reach Exascale in the next ten years*
- *Many fields have problems ready to exploit Exascale, albeit many simulations are memory bandwidth bound*
- *The **major challenge is programming these new machines** to ensure software longevity and performance portability*
- *Larger markets may determine the paradigms HPC will use*
- *Safest investment is parallel paradigms in language standards, such as C++; task-based paradigms will flow into C++17/20*
- *Domain-specific frameworks promising for particular fields*

*Thanks for your attention*