OpenACC programming for GPGPUs: Rotor wake simulation

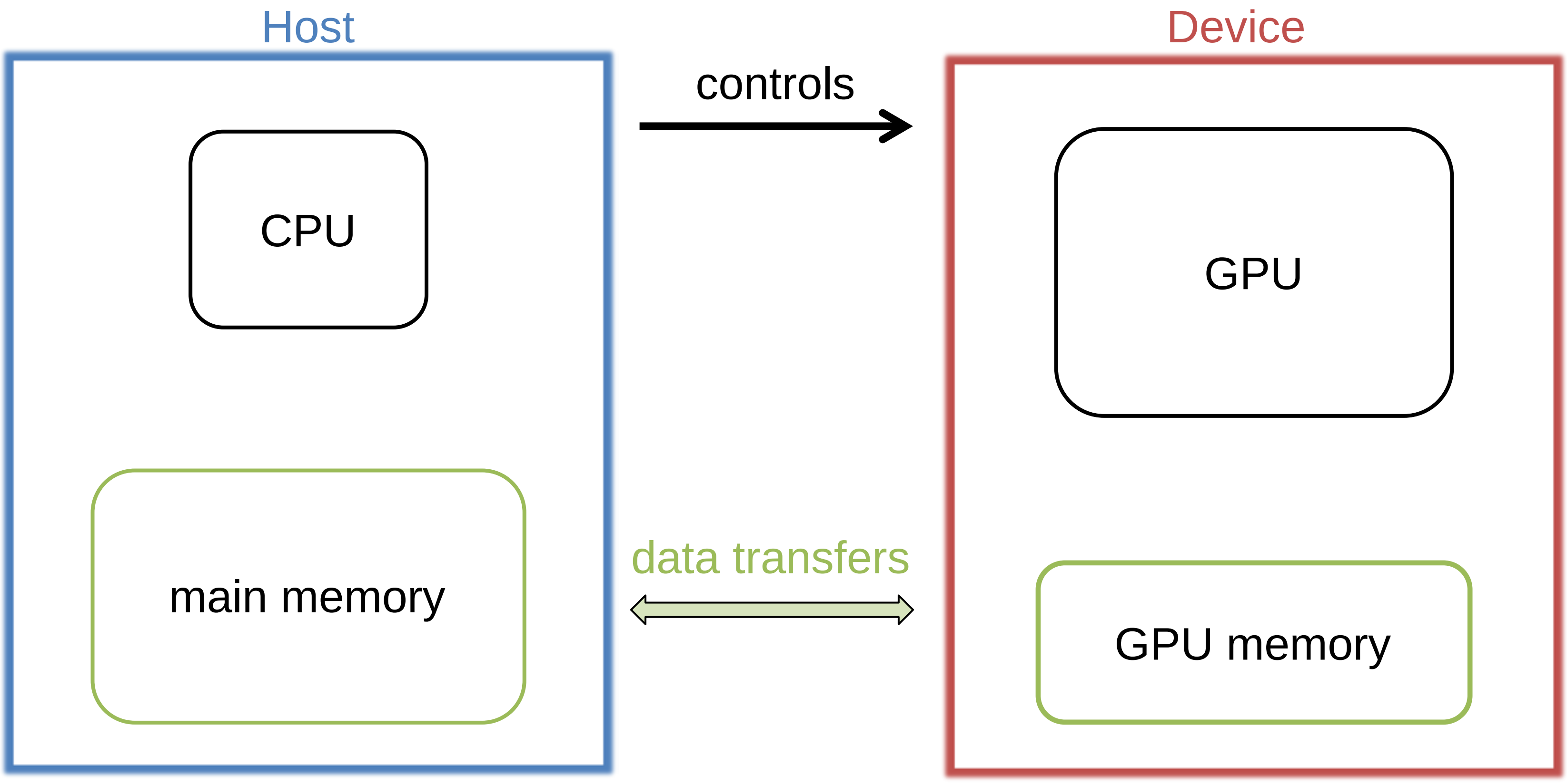
Melven Röhrig-Zöllner, Achim Basermann
Simulations- und Softwaretechnik

DLR

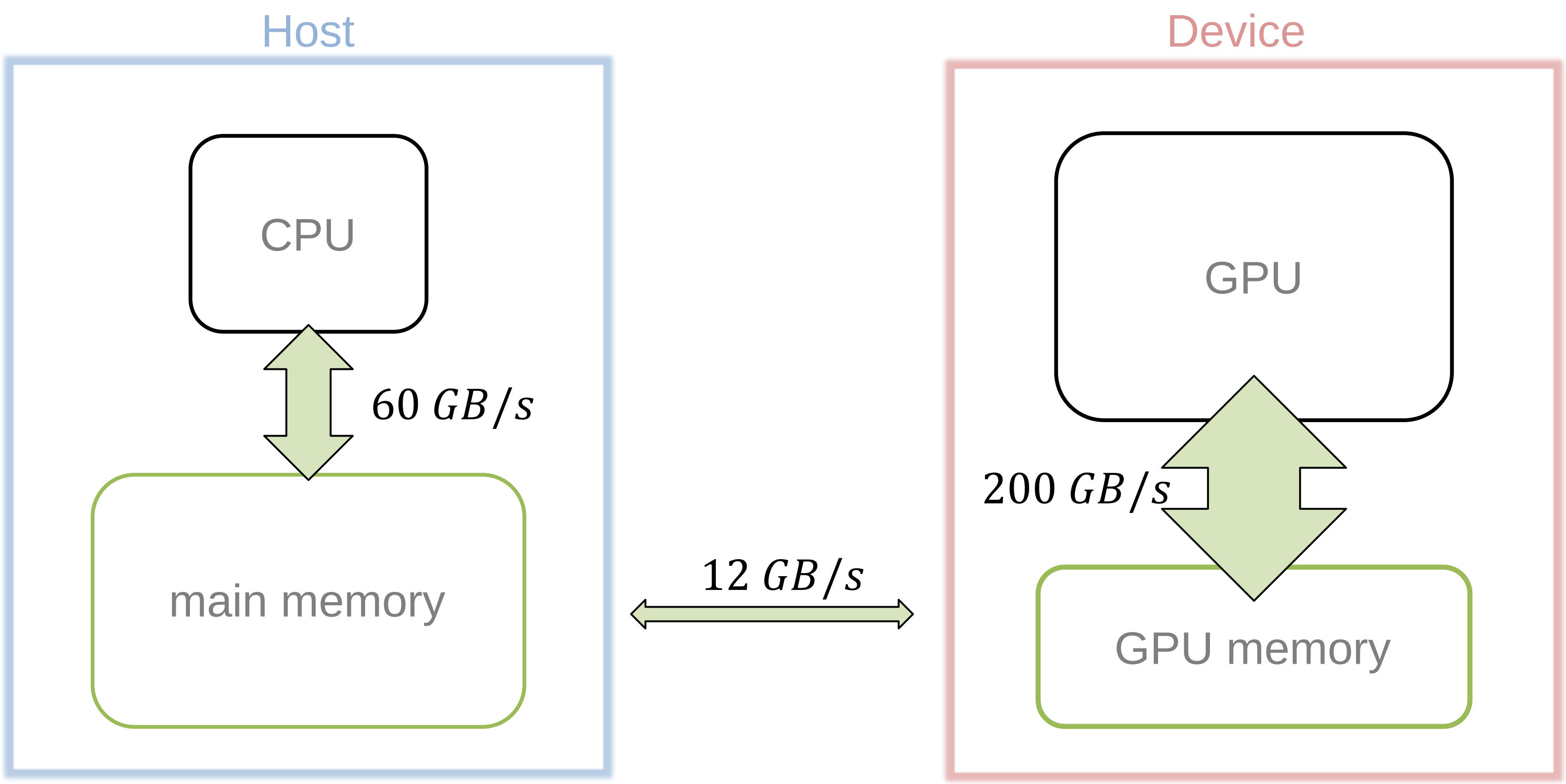
Outline

- **Hardware-Architecture (CPU+GPU)**
- GPU computing with OpenACC
- Rotor simulation code Freewake
- OpenACC port of Freewake
- Conclusion

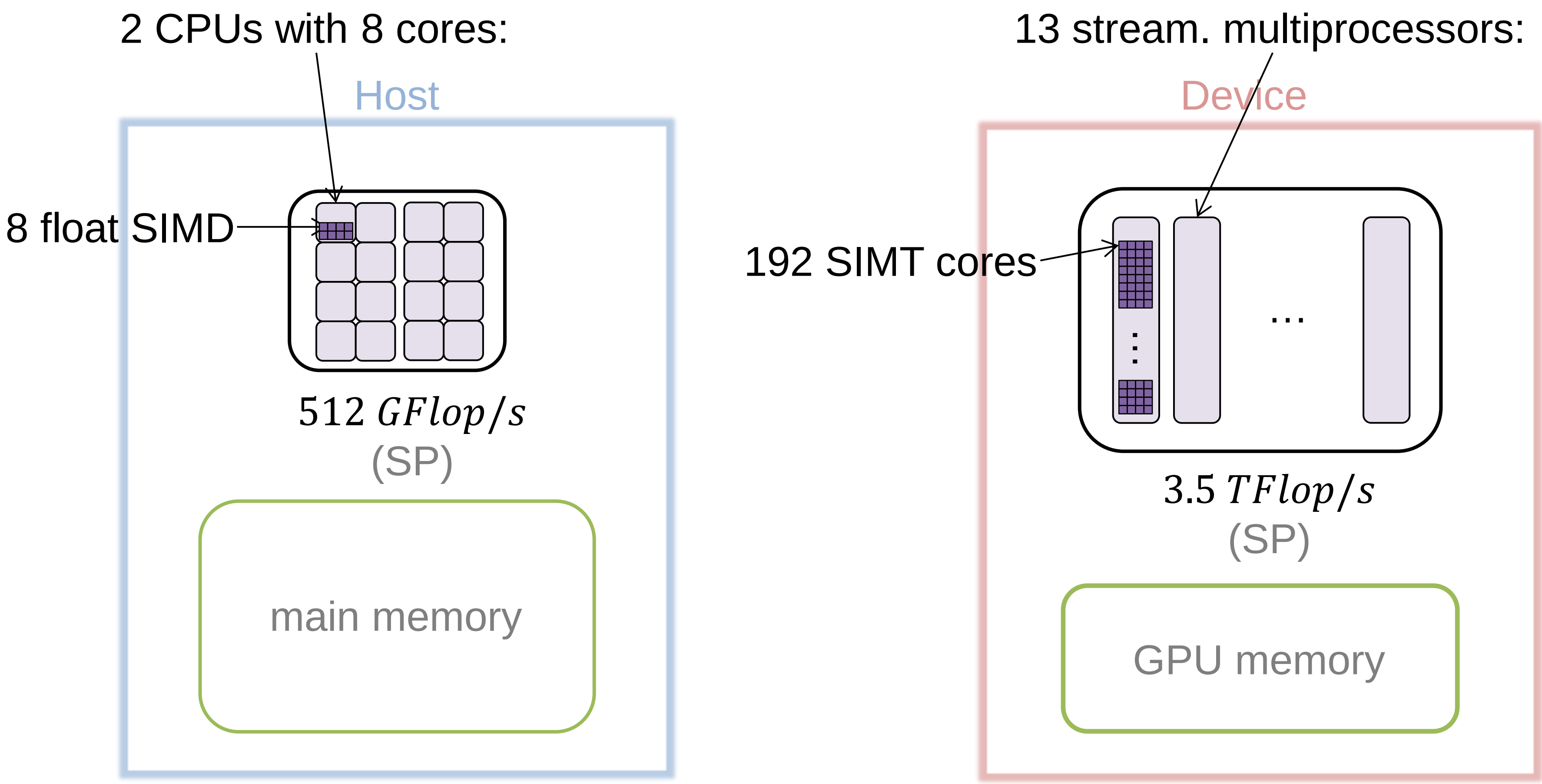
Hardware-Architecture: Overview



Hardware-Architecture: Data transfers



Hardware-Architecture: Calculations



Hardware-Architecture: Comparison

CPU

- SIMD parallelism:
  - SSE / AVX extensions (8 float)

- MIMD parallelism:
  - several CPUs (2)
  - multiple cores (8)
  - (possibly CPU threads)

- **Caches to avoid memory latency**

GPU

- SIMT parallelism:
  - 32 scalar threads form a warp
  - 1-32 warps form a thread block
    (32-1024 threads per block)

- MIMD parallelism:
  - thread blocks within a grid

- **Switch threads to hide latency**

→ Requires 100,000+ threads!

Outline

- Hardware-Architecture (CPU+GPU)
- **GPU computing with OpenACC**
- Rotor simulation code Freewake
- OpenACC port of Freewake
- Conclusion

OpenACC: Overview

- Language extensions similar to OpenMP
- Directive based
- Supported languages:
  - C
  - C++
  - **Fortran**
- Supported compilers:
  - CAPS
  - CRAY
  - **PGI**
  - (unofficial patches for GCC)

OpenACC: Fortran example

```fortran
program main
    real :: a(N)
    …
!$acc data copyout(a)
    ! Computation in several loops on the GPU:
    …
    !$acc parallel loop
    do i = 1, N
        a(i) = 2.5 * i
    end do

    …
!$acc end data
    ! Use results on the CPU
    …
end program main
```

OpenACC: C++ example

```
int main()
{
    float *a = new float[N];
    …
#pragma acc data copyout(a[0:N-1])
    // Computation in several loops on the GPU:
    …
    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        a[i] = 2.5 * i
    …
#pragma acc end data
    // Use results on the CPU
    …
    return 0;
}
```

OpenACC: Important features

- Explicit data movement between host and device (bottleneck!)
- Loop reductions:
  - calculate sum, minimum, etc over all iterations
  - (currently only for scalar variables)
- Explicit mapping to GPU hardware:
  - `gang` ↔ thread blocks
  - (`worker` ↔ warps)
  - `vector` ↔ threads within a warp

  → performance tuning
- Interoperable with CUDA

DLR

OpenACC: Reduction example

```
program main
    real :: a(N)
    real :: norm_a

    ! initialize a
    …
    norm_a = 0.
!$acc parallel loop reduction(+:norm_a)
    do i = 1, N
        norm_a = norm_a + a(i)*a(i)
    end do
!$acc end parallel loop
    norm_a = sqrt(norm_a)

    …
end program main
```

OpenACC: „acc kernels“ example

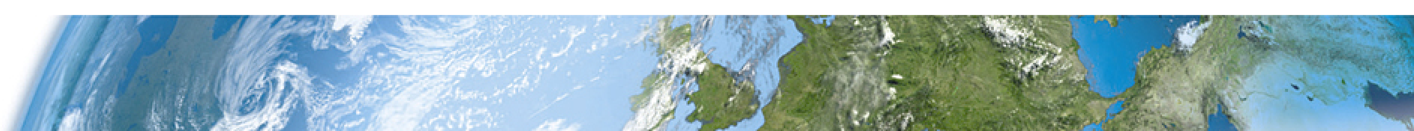
```
program main
    real :: a(N)
    real :: norm_a

    ! initialize a
    …
!$acc kernels
    norm_a = 0.
    do i = 1, N
        norm_a = norm_a + a(i)*a(i)
    end do
    norm_a = sqrt(norm_a)
!$acc end kernels

    …
end program main
```

OpenACC: „acc kernels“ compiler output

```
> pgf90 -Minfo -fast -acc -ta=nvidia kernels_test.f90

main:
  7, Generating present_or_copyin(a(:))
     Generating NVIDIA code
  9, Loop is parallelizable
     Accelerator kernel generated
      9, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     10, Sum reduction generated for norm_a
```

OpenACC: Fine-tuning example

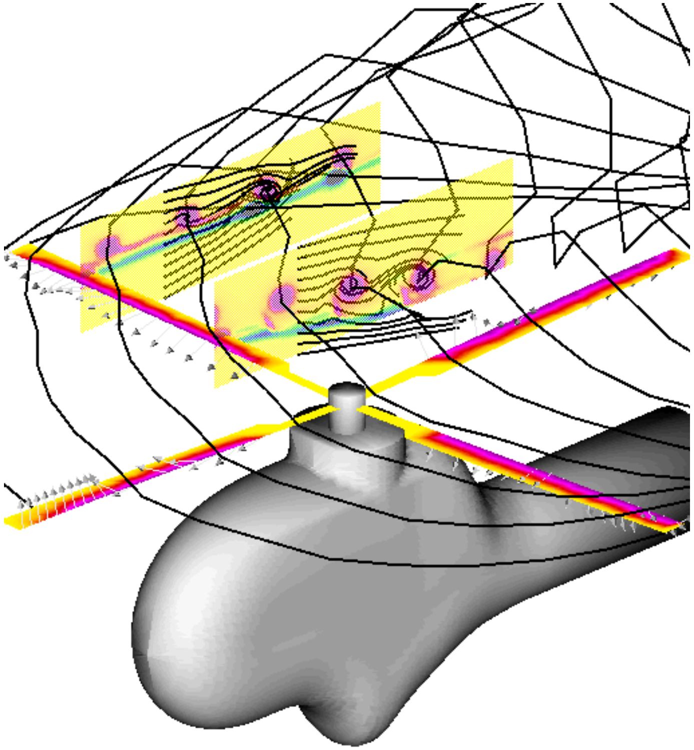
```
    …

!$acc kernels

    norm_a = 0.
    !$acc loop gang(1024) vector(128) reduction(+:norm_a)
    do i = 1, N
        norm_a = norm_a + a(i)*a(i)
    end do
    norm_a = sqrt(norm_a)

!$acc end kernels

    …
```



Outline

- Hardware-Architecture (CPU+GPU)
- GPU computing with OpenACC
- **Rotor simulation code Freewake**
- OpenACC port of Freewake
- Conclusion

Freewake: Overview



- Developed 1994-1996 by FT-HS
  - implemented in Fortran
  - MPI-parallel
- Used by the FT-HS rotor simulation code S4
- Simulates the flow around a helicopter's rotor
- Vortex-Lattice method
  - Discretizes complex wake structures with a set of vortex elements
- Based on experimental data (from the international HART program 1995)

Freewake: Comparison with „classical CFD“

„Classical“ CFD:

- Navier-Stokes-equations

Spatial discretization:

- velocity
- mesh in whole 3D domain

Discretization in time:

- Update velocity using small stencils

- numerical diffusion

Vortex methods:

- Vorticity equation (curl of velocity)

Spatial discretization:

- vorticity
- points/grid in interesting regions

Discretization in time:

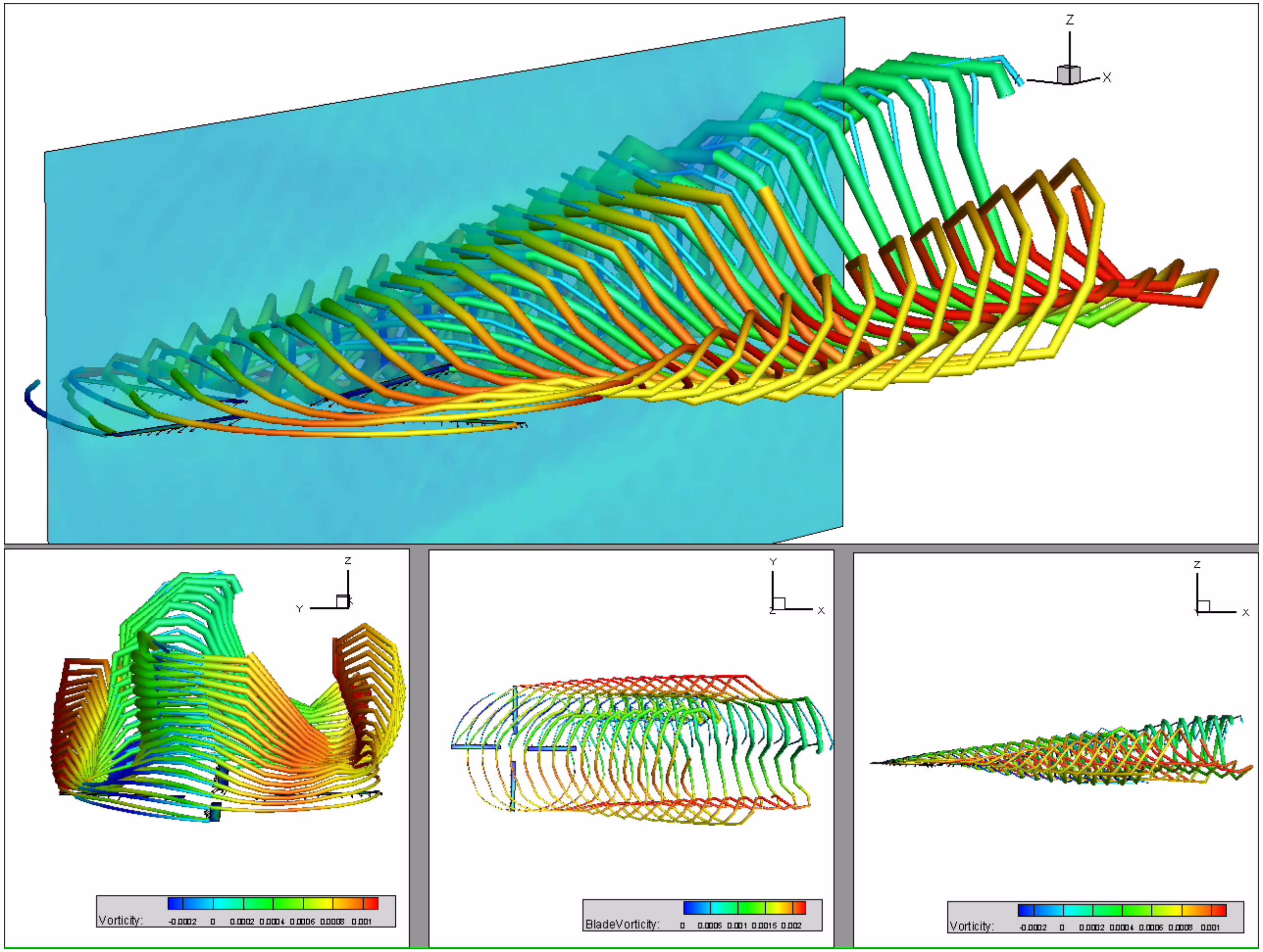
- Move points using induced velocity

- complex induced velocity calculation
  (similar: N-body problem)

Freewake: Velocity calculation

- moving 2D grid in space with cells of precalculated vorticity
- good initial wake geometry from analytical model
- Velocity at a point:
  - sum over induced velocities of all grid cells
- different formulas depending on the distance point ↔ cell
- interpolated cells to allow smaller time steps than grid resolution

Freewake: Vortex visualization



Outline

- Hardware-Architecture (CPU+GPU)
- GPU computing with OpenACC
- Rotor simulation code Freewake
- **OpenACC port of Freewake**
- Conclusion

Freewake OpenACC port: Simple benchmark

- Idea: only use formula for medium range
- Performance is compute bound:
  - Working set: $n = \sim 6300$ grid points → $\sim 250\ kB$
    (4 blades with 11x144 points)
  - Operations: $\sim 50n^2\ Flop$ per iteration
- Fastest CPU implementation (GCC):
  - uses vector-reductions from OpenMP 3.0
  - $\sim 280\ GFlop/s$ (SP)
  - ca. 40x faster than Free-Wake
- OpenACC GPU implementation (PGI):
  - only scalar reductions possible → not optimal & more complex
  - $\sim 360\ GFlop/s$ (SP)

Freewake OpenACC port: Avoiding branches in loops

if-statements in loops are problematic:

- GPU warp needs several passes for different branches (SIMT)
- may also prevent efficient vectorization on CPUs (SIMD)

Nested if-statements in Free-Wake:

- Grid boundaries:
  → partial loop unrolling by hand

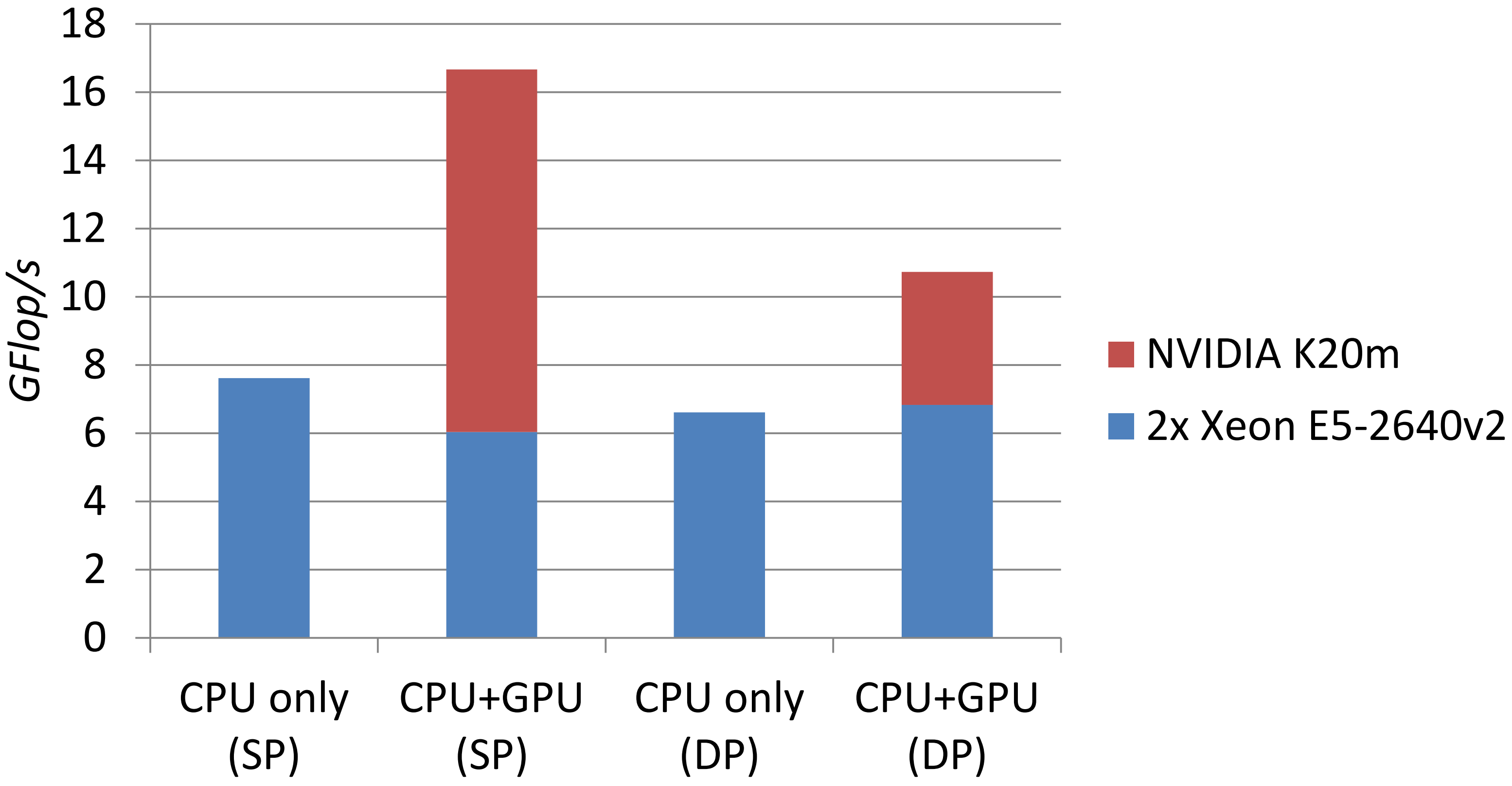
- “Flags”:
  → $result = flag * a + (1 - flag) * b$ for $flag \in \{0,1\}$

- Different formulas depending on distance point ↔ cell:
  - difficult!
  - currently best results with dedicated loops for individual cases

Freewake OpenACC port: Hybrid CPU/GPU calculations

- MPI-parallel:
  - Grid stored redundantly on all processes
  - Each process calculates the velocity of a set of points
- Hybrid calculation:
  - First MPI-process uses the GPU
  - All others stay on the CPU
  - uses `acc_set_device_type( acc_device_... )`
- Dynamic Load-Balancing:
  - Measure time in each iteration
  - Redistribute work appropriately

Freewake OpenACC port: performance results



Outline

- Hardware-Architecture (CPU+GPU)
- GPU computing with OpenACC
- Rotor simulation code Freewake
- OpenACC port of Freewake
- **Conclusion**

Conclusion

- Successfully ported the Freewake simulation to GPUs using OpenACC
  - original numerical method not modified
  - refactored & restructured a lot of code

→ Porting complex algorithms to GPUs is difficult
  - branches in loops hurt (much more than for CPUs)

- Loop restructuring may also improve the CPU performance
  (SIMD vectorization on modern CPUs)

- Stumbled upon several OpenACC PGI-compiler bugs (all fixed very fast)

- Future work:
  - extension to wind turbines
  - reduce complexity from $O(n^2)$ to $O(n \log n)$

Summary about OpenACC

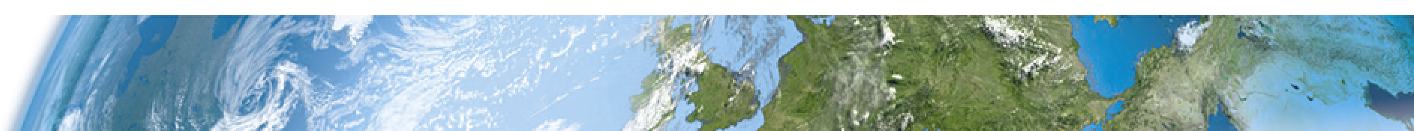
- Directive based:
    - Annotate loops
    - Specify data movement

- Portable:
    - same code basis for host (CPU) and device (GPU)
    - hybrid calculations
    - different current and future accelerators

→ **Code remains short & understandable**

To make it fast:

- still a lot of work (restructuring)
- background knowledge required



Questions?