ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

# GPU Aware MPI and Applications:

# Stencil Computations


Mauro Bianco

Swiss National Supercomputing Centre

ETH Zurich


MUG, August 26th, 2013

# Motivation

- Metereological code COSMO needed a flexible interface to perform halo-update
  - 3D regular grid (array) PDE solver
- Other applications can take advantage of it
  - Portability across accelerated architectures and not
  - Memory layout customization
  - Handling of custom padding
  - Independency on arrays value types
  - Avoiding “reinventing the wheel”

# Generic Communication Layer

- GCL: High-level C++ library of communication patterns
- Layered architecture
  - L3: Communication of buffers from proc *i* to proc *j*
  - L2: Exploit data structures to perform data exchange (DSSL)
  - L1: Use high order functions to handle general data structures
- At L2 the following are available (others are possible)
  - Halo-update for regular grids
    - Dynamic: sizes known at "instantiation" time, pointers to data are not
    - Generic: sizes and pointers are not known at instantiation
  - Generic All-to-All

# Halo-Exchange (L3)

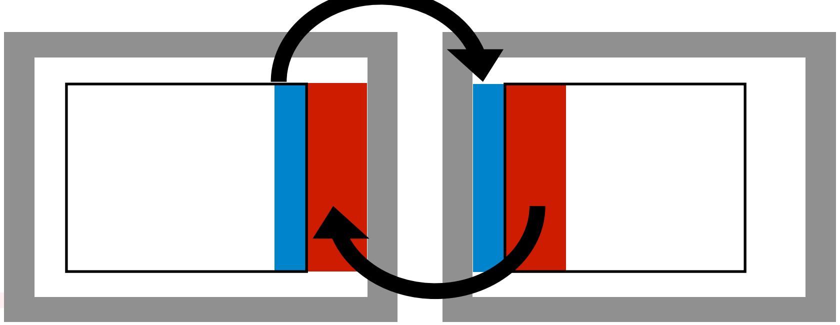
```
Halo_Exchange_3D<   _3D_process_grid_t<…>      > he(pg);
                  MPI_3D_process_grid_t<…>
```

```
he.register_send_to_buffer<-1,-1, 0>(pointera, sizea);
he.register_send_to_buffer<-1, 1,-1>(pointerb, sizeb);
…
```

```
he.register_receive_from_buffer<-1,-1,-1>(pointerc, sizec);
he.register_receive_from_buffer<-1, 1,-1>(pointerd, sized);
…
```

```
he.start_exchange();
he.wait()
```

# Example



```
typedef GCL::halo_exchange_dynamic_ut<GCL::layout_map<2,1,0>,
                                      GCL::layout_map<0,1,2>,
                                      double, 3,
                                      GCL::gcl_gpu,
                                      GCL::version_manual >
        pattern_type;

pattern_type he(period_type(true, false, false), CartComm);

he.add_halo<0>(H, H, H, DIM1+H-1, DIM1+2*H);
he.add_halo<1>(H, H, H, DIM2+H-1, DIM2+2*H);
he.add_halo<2>(H, H, H, DIM3+H-1, DIM3+2*H);

he.setup(3); // Maximum # of fields

he.pack(A_ptr_on_gpu, B_ptr_on_gpu, C_ptr_on_gpu);

he.exchange();

he.unpack(A_ptr_on_gpu, B_ptr_on_gpu, C_ptr_on_gpu);
```

# Halo-Update Generic: Example

```
typedef halo_exchange_generic
              <layout_map<0,1,2>, 3, arch_type, packing_type >
              pattern_type;

pattern_type he(period_type(per0, per1, per2), CartComm);

he.setup(...);

field_on_the_fly<type1, lout1, pattern_type::traits> field1(ptr1,
halo_dsc1);

field_on_the_fly<type2, lout2, pattern_type::traits> field2(ptr2,
halo_dsc2);

field_on_the_fly<type3, lout3, pattern_type::traits> field3(ptr3,
halo_dsc3);

he.pack(field1, field2, field3);

he.exchange();

he.unpack(field1, field2, field3);
```

# Algorithms – MPI_Pack Packing

- Both CPU and GPU
  - For each neighbor and each data-field build MPI_Datatype
  - Use MPI_Pack to pack data into linear buffer
  - Use MPI_Unpack to unpack data back to data-field
  - OpenMP is optional on CPU – not always beneficial
  - GPU suffer of poor MPI_Datatype handling
    - Implemented with subarray datatype
- On CPU performance is similar to Manual
- On GPU is orders of magnitude slower

# Algorithms – Datatype Packing

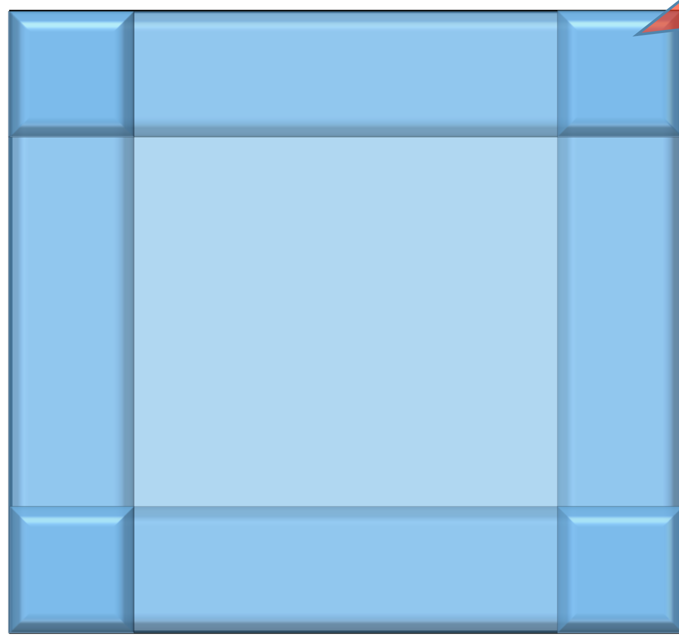
- Both CPU and GPU
  - For each data-field build subarray MPI_Datatype
  - For each neighbor build MPI_Datatype with all data-fields
    - Pointers are diffs are computed: datatype build at packing time
- Use a specialized L3 pattern that sends a single element
  - Maintainability problem (Could be fixed using MVAPICH)
- Performance on CPU similar to manual
  - But pack/unpack times are much smaller
- On GPU is orders of magnitude slower

# Algorithms – Manual Packing

- Exchange is done with isends, irecvs, and waits
  - It is possible to overlap communication and computation
- Packing on CPU
  - Loop on neighbors to (cache-friendly) collect data
  - OpenMP can be used to pack neighbors in parallel
    - Not always beneficial
- Packing on GPU
  - (Up to) 6 CUDA kernels per data-field
  - Each kernel collects data for multiple neighbors
  - One data-field packed at the time

# Packing Kernels - Front

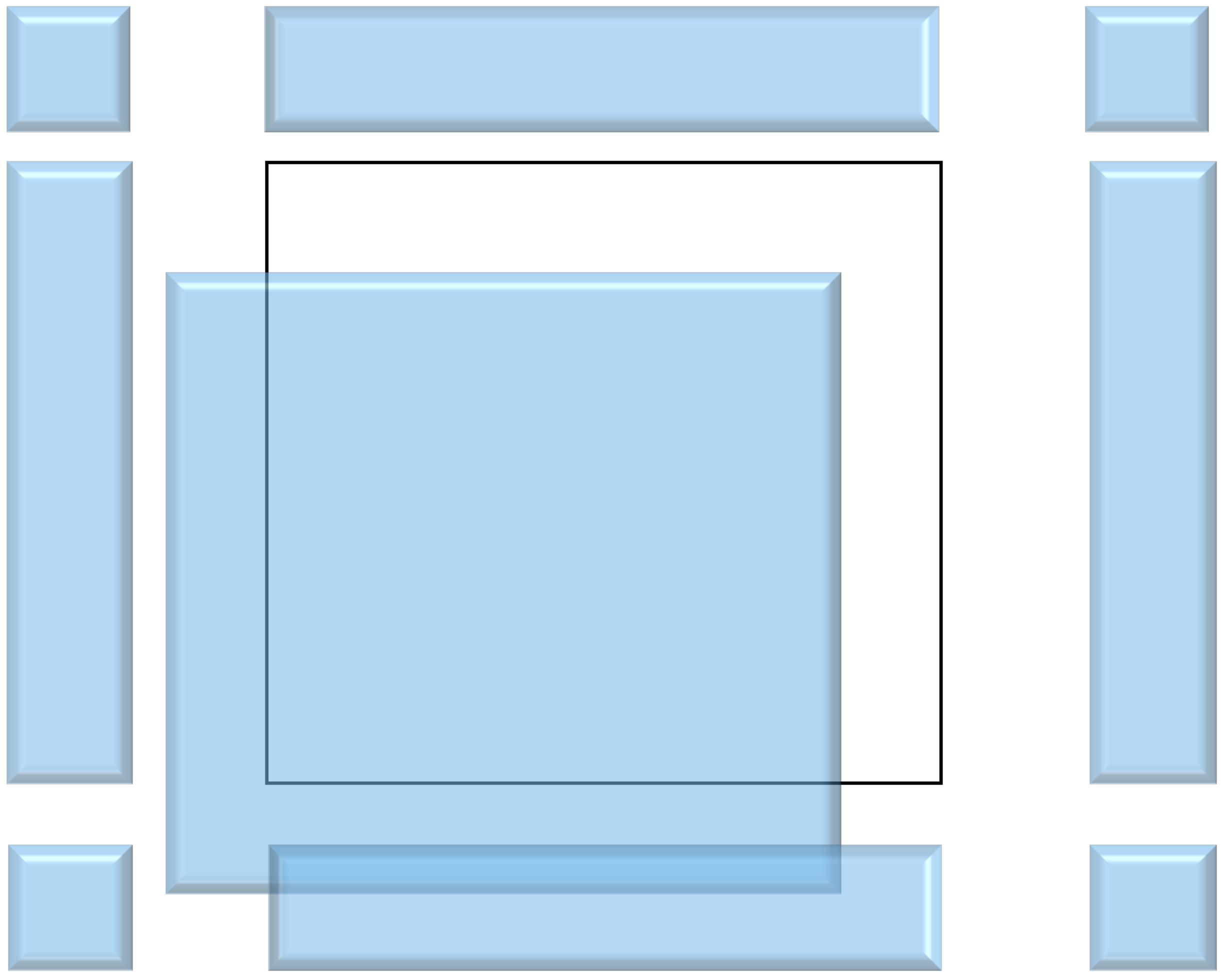
Single Kernel!
Can
MPI_Datatypes
beat this?



# Packing Kernels - Side

# Packing Kernels – Up

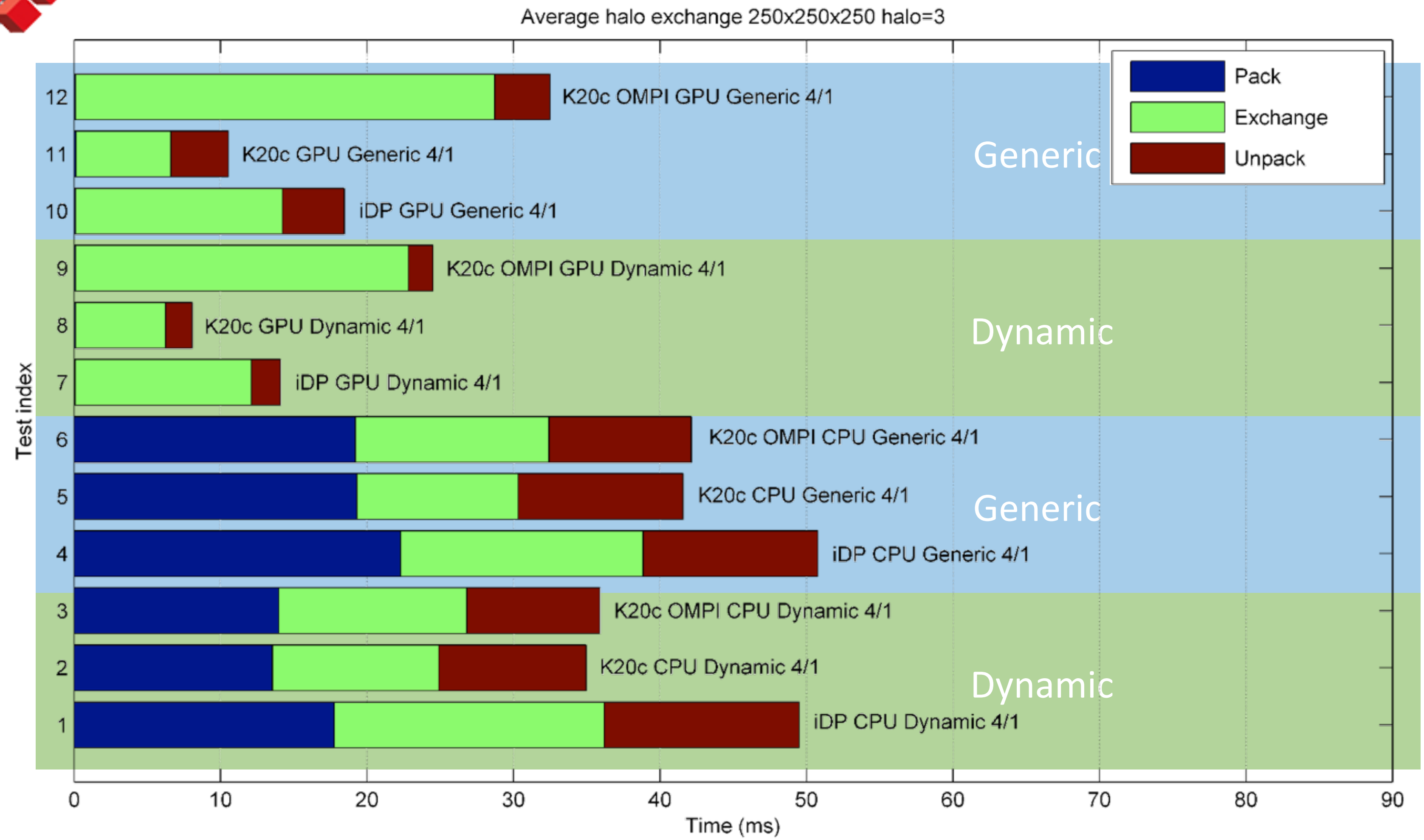
# Unpacking Kernels - Front



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National
Supercomputing Centre

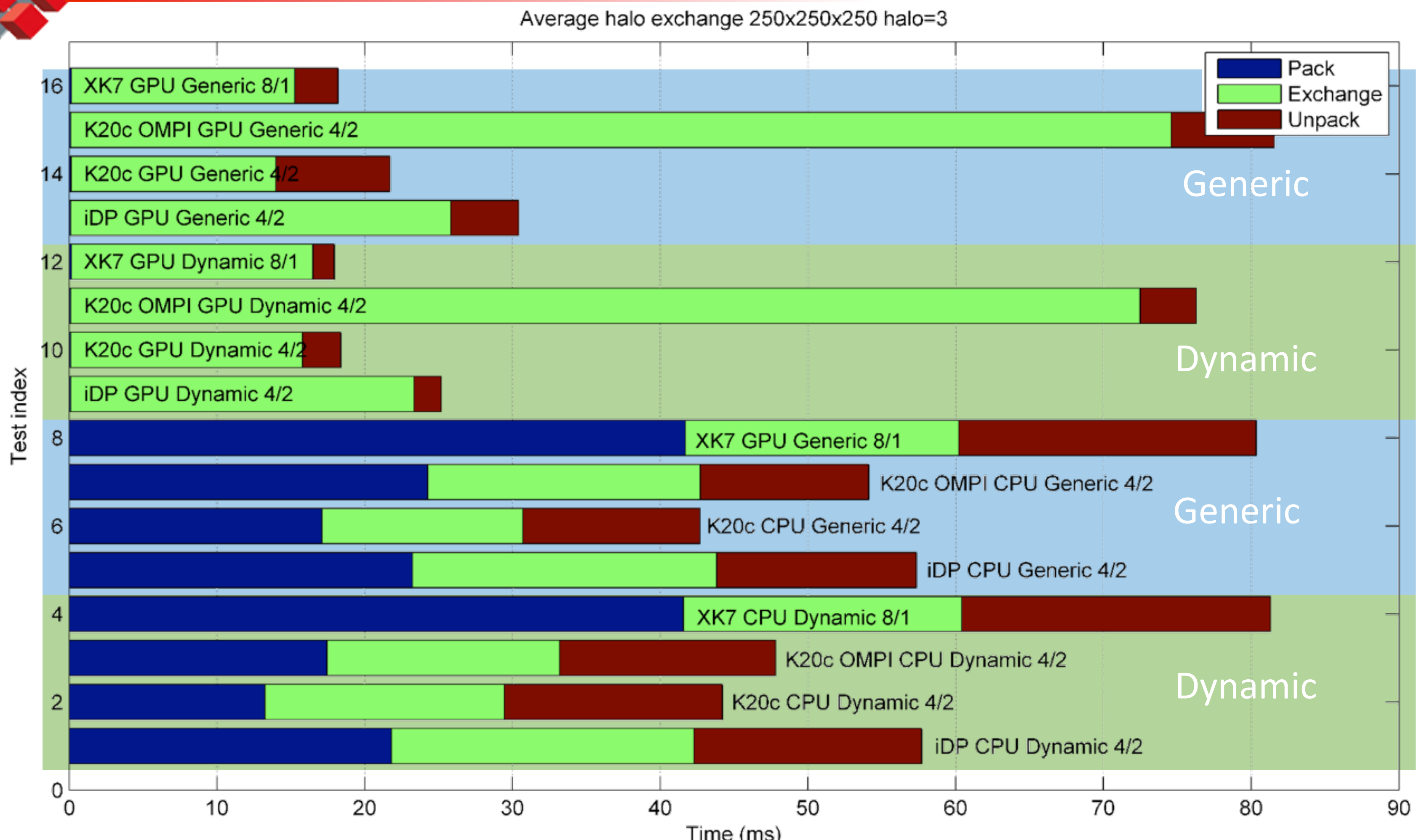
# CPU vs. GPU – Low contention




ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National
Supercomputing Centre

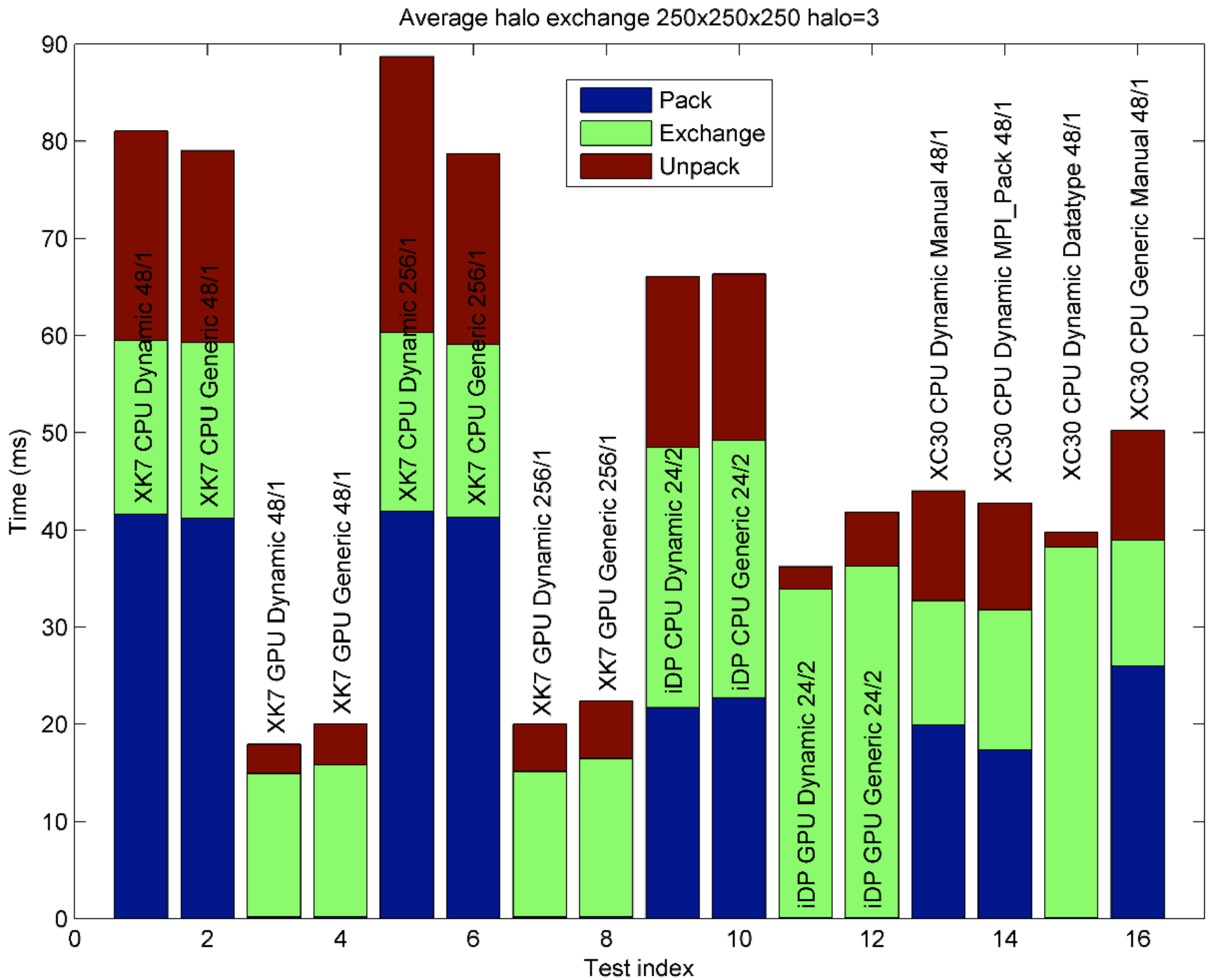
# CPU vs. GPU – High contention




ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

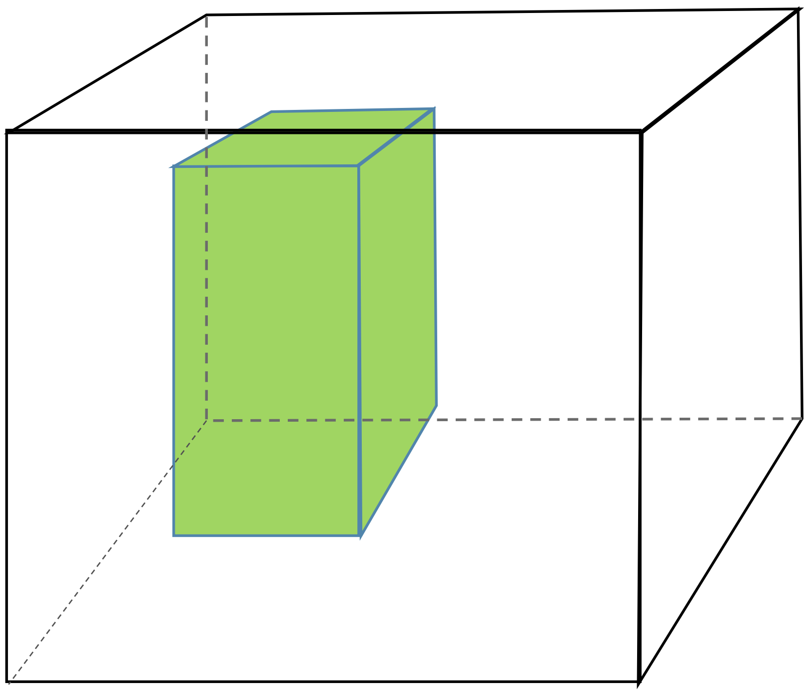
CSCS
Swiss National
Supercomputing Centre

# Scaling of GCL

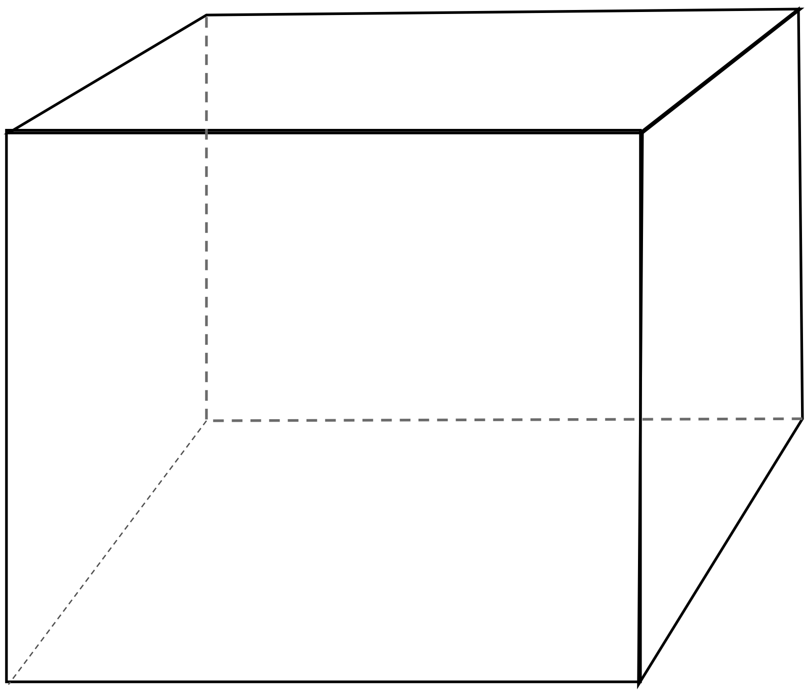


# MPI_Datatypes experiments

Send a 3D "contiguous" subarray from process 0 to process 1



Proc 0



Proc 1

# Three datatypes against kernels

## Subarray

```
MPI_Type_create_subarray(dims, lens, subsizes, offsets, MPI_ORDER_C,
                         arrayElementType, &newtype );
```

## Vector 2 steps

```
MPI_Type_vector (subm, subl, l, arrayElementType, &tmpt_0);
MPI_Type_hvector(subn, 1, m*l*sizeof(value_type), tmpt_0, &newtype);
```

## Vector 3 steps

```
MPI_Type_vector (subl, 1, 1, arrayElementType, &tmpt_0);
MPI_Type_hvector(subm, 1, l*sizeof(value_type), tmpt_0, &tmpt_1);
MPI_Type_hvector(subn, 1, m*l*sizeof(value_type), tmpt_1, &newtype);
```

## Manual (Pack kernel body)

```
 if ( (idx < subl) && (idy < subm) && (idz < subn) )
       buff[idz*subm*subl + idy*subl + idx] =
                        p.g_elem(starti+idz,startj+idy,startk+idx);
```

# 1 Process sending to itself

| Benchmark Size | Kernel | Subarray | Vector 2 | Vector 3 |
| --- | --- | --- | --- | --- |
| Array: 512x512x512<br>Block: 512x64x64 (pick (0,0,0)) | 155 | 441 | 447 | 447 |
| Array: 512x512x512<br>Block: 64x64x512 (pick (0,0,0)) | 151 | 151 | 151 | 170 |
| Array: 512x512x512<br>Block: 512x512x512 (pick (0,0,0)) | 311 | 292 | 292 | 292 |
| Array: 513x513x513<br>Block: 512x512x512 (pick (1,1,1)) | 313 | 2890 | 2722 | 3358 |

# 2 Processes / 1 per node

| Benchmark Size | Kernel | Subarray | Vector 2 | Vector 3 |
|---|---|---|---|---|
| Array: 512x512x512<br>Block: 512x64x64 (pick (0,0,0)) | 156 | 444 | 443 | 450 |
| Array: 512x512x512<br>Block: 64x64x512 (pick (0,0,0)) | 152 | 153 | 153 | 172 |
| Array: 512x512x512<br>Block: 512x512x512 (pick (0,0,0)) | 588 | 392 | 391 | 391 |
| Array: 513x513x513<br>Block: 512x512x512 (pick (1,1,1)) | 583 | 2819 | 2816 | 3479 |

# Conclusions

- MPI on GPUs can lead to good performance
- Engineering a library for portable communication is not trivial
  - Large portions of code need to be specialized
  - 2D and 3D codes diverge for GPUs
    - Using OpenACC?
  - Having MPI_Datatypes would be beneficial if they work WELL on all platforms
    - A library allows to write the code only once so the overhead of writing complex code can be amortized
- Advanced features may require some standardization

# That’s all

Thank you!