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

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
CPU vs. GPU – Low contention

Average halo exchange 250x250x250 halo=3

Test index

0 10 20 30 40 50 60 70 80 90 Time (ms)

CPU  vs.  GPU  –  Low  contention

Pack  Exchange  Unpack

K20c OMPI GPU Generic 4/1
K20c GPU Generic 4/1
IDP GPU Generic 4/1
K20c OMPI GPU Dynamic 4/1
K20c GPU Dynamic 4/1
IDP GPU Dynamic 4/1
K20c OMPI CPU Generic 4/1
K20c CPU Generic 4/1
IDP CPU Generic 4/1
K20c OMPI CPU Dynamic 4/1
K20c CPU Dynamic 4/1
IDP CPU Dynamic 4/1
CPU vs. GPU – High contention

Average halo exchange 250x250x250 halo=3

Test index

0 10 20 30 40 50 60 70 80 90

Time (ms)

K20c GPU Dynamic 4/2
iDP GPU Dynamic 4/2
XK7 GPU Dynamic 8/1
K20c OMPI GPU Dynamic 4/2
K20c CPU Dynamic 4/2
XK7 CPU Dynamic 8/1
K20c OMPI CPU Dynamic 4/2
K20c CPU Dynamic 4/2
iDP CPU Dynamic 4/2

Pack
Exchange
Unpack

Generic
Dynamic
Generic
Dynamic
Scaling of GCL

Average halo exchange 250x250x250 halo=3

Time (ms)

Test index

Pack
Exchange
Unpack

XK7 CPU Dynamic 48/1
XK7 CPU Generic 48/1
XK7 GPU Dynamic 48/1
XK7 GPU Generic 48/1
XK7 GPU Dynamic 256/1
XK7 GPU Generic 256/1
XC30 CPU Dynamic Manual 48/1
XC30 CPU Dynamic Manual 256/1
XC30 CPU Dynamic MPI Pack 48/1
XC30 CPU Dynamic Datatype 48/1
XC30 CPU Dynamic Datatype 256/1

Swiss National Supercomputing Centre
MPI_Datatypes experiments

Send a 3D “contiguous” subarray from process 0 to process 1
Three datatypes against kernels

Subarray

```c
MPI_Type_create_subarray(dims, lens, subsizes, offsets, MPI_ORDER_C, arrayElementTyp
```e, &newtype);

Vector 2 steps

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

Vector 3 steps

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

Manual (Pack kernel body)

```c
if ( (idx < subl) && (idy < subm) && (idz < subn) )
    buff[idz*subm*subl + idy*subl + idx] =
        p.g_elem(starti+idz,startj+idy,startk+idx);
```e
1 Process sending to itself

<table>
<thead>
<tr>
<th>Benchmark Size</th>
<th>Kernel</th>
<th>Subarray</th>
<th>Vector 2</th>
<th>Vector 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array: 512x512x512 Block: 512x64x64 (pick (0,0,0))</td>
<td>155</td>
<td>441</td>
<td>447</td>
<td>447</td>
</tr>
<tr>
<td>Array: 512x512x512 Block: 64x64x512 (pick (0,0,0))</td>
<td>151</td>
<td>151</td>
<td>151</td>
<td>170</td>
</tr>
<tr>
<td>Array: 512x512x512 Block: 512x512x512 (pick (0,0,0))</td>
<td>311</td>
<td>292</td>
<td>292</td>
<td>292</td>
</tr>
<tr>
<td>Array: 513x513x513 Block: 512x512x512 (pick (1,1,1))</td>
<td>313</td>
<td>2890</td>
<td>2722</td>
<td>3358</td>
</tr>
</tbody>
</table>
## 2 Processes / 1 per node

<table>
<thead>
<tr>
<th>Benchmark Size</th>
<th>Kernel</th>
<th>Subarray</th>
<th>Vector 2</th>
<th>Vector 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array: 512x512x512 Block: 512x64x64 (pick (0,0,0))</td>
<td>156</td>
<td>444</td>
<td>443</td>
<td>450</td>
</tr>
<tr>
<td>Array: 512x512x512 Block: 64x64x512 (pick (0,0,0))</td>
<td>152</td>
<td>153</td>
<td>153</td>
<td>172</td>
</tr>
<tr>
<td>Array: 512x512x512 Block: 512x512x512 (pick (0,0,0))</td>
<td>588</td>
<td>392</td>
<td>391</td>
<td>391</td>
</tr>
<tr>
<td>Array: 513x513x513 Block: 512x512x512 (pick (1,1,1))</td>
<td>583</td>
<td>2819</td>
<td>2816</td>
<td>3479</td>
</tr>
</tbody>
</table>
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!