Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Intel CPUs and GPUs on structured-mesh stencil workloads with oneAPI

Istvan Reguly (PPCU ITK), Mark Lubin (Intel), Xiao Zhu (Intel)

Structured-mesh stencil codes

- 1D-3D (or more) applications
  - i,j,k cartesian indexing, implicit connectivity
  - „Trivial” parallelization:
    - no loop carried dependencies
    - possible reductions
    - No implicit solvers (e.g. Gauss-Seidel)

```
B[i][j] = 0.2 * (A[i][j]
               + A[i][j-1]
               + A[i][j+1]
               + A[i-1][j]
               + A[i+1][j]);
```

- Oxford Parallel library for Structured meshes (OPS) – a DSL in C++

```
void poisson_kernel_stencil(ACC<double> &A, ACC<double> &Anew) {
    Anew(0,0) = 0.25f *   ( A(1,0) + A(-1,0) + A(0,-1) + A(0,1));}

ops_par_loop(poisson_kernel_stencil, "poisson_kernel_stencil", blocks[i+ngrid_x*j], 2, iter_range,
         ops_arg_dat(u[i+ngrid_x*j], 1, S2D_00_P10_M10_0P1_0M1, "double", OPS_READ),
         ops_arg_dat(u2[i+ngrid_x*j], 1, S2D_00, "double", OPS_WRITE));
```

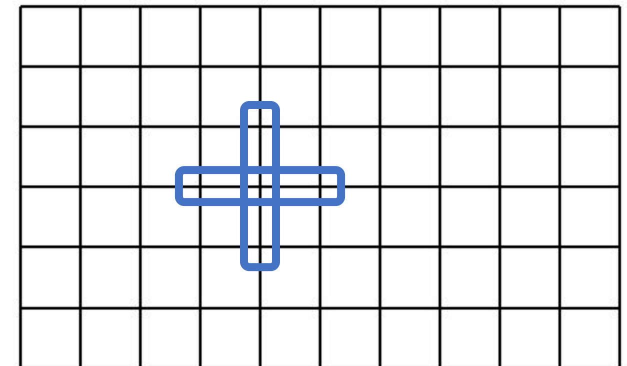
Look of a stencil loop - flat

```
Queue.submit([&](cl::sycl::handler &cgh) {
...//Accessors
  cgh.parallel_for<class apply_stencil>(
    cl::sycl::range<2>(jmax, imax),
    [=](cl::sycl::item<2> idx) {
...//Views, bounds checking

      Anew(0,0) = 0.25f *
        ( A(1,0) + A(-1,0)
        + A(0,-1) + A(0,1));
```

"**Flat**" parallel loop: just specify how many work items per dimension are required

Expressed from viewpoint of "current" gridpoint, with relative offsets

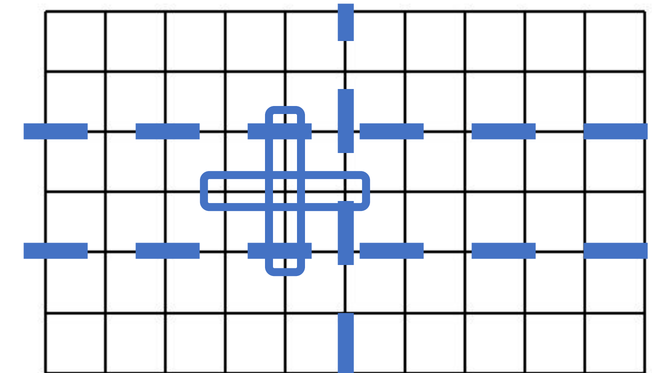


- Simplest way to do things – does not prescribe anything about how work items should be grouped or mapped to hardware

Look of a stencil loop – nd_range

```
Queue.submit([&](cl::sycl::handler &cgh) {
...//Accessors
  cgh.parallel_for<class apply_stencil>(
    cl::sycl::nd_range<2>(
        cl::sycl::range<2>(jmax, imax),
        cl::sycl::range<2>(4,    32)),
    [=](cl::sycl::nd_item<2> idx) {
...//Views, bounds checking
```

"**nd_range**" parallel loop: just specify how many work items per dimension and split them into workgroups



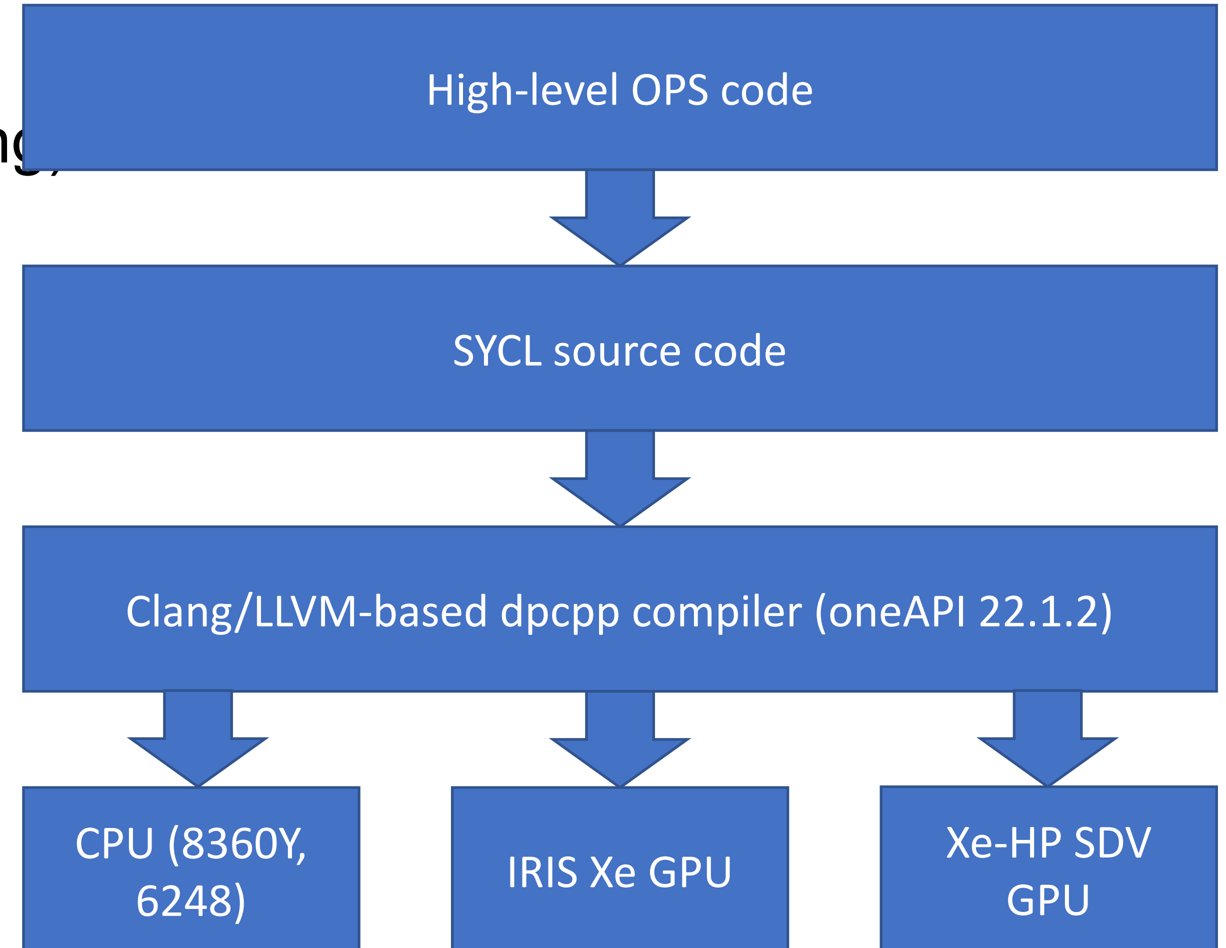
```
    Anew(0,0) = 0.25f *
       ( A(1,0) + A(-1,0)
       + A(0,-1) + A(0,1));
```

Expressed from viewpoint of "current" gridpoint, with relative offsets

- Explicit way of grouping work items and mapping them to hardware – can control/impact cache locality. But entirely up to the programmer!
- Could automate?

Test setup

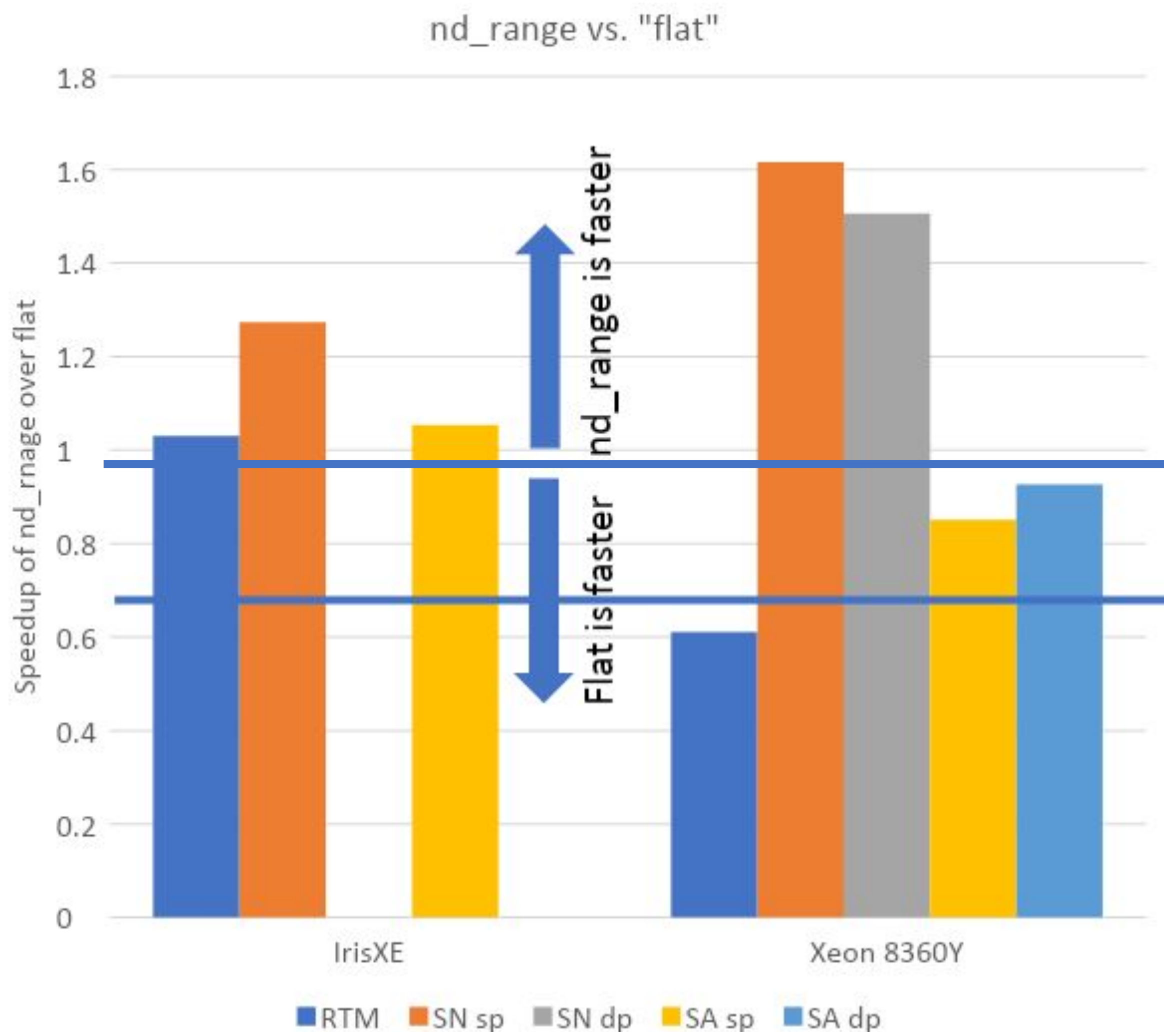
- Intel(r) Xeon(r) Platinum 8360Y
  - 36 cores @ 2.4 GHz (HyperThreading)
  - BabelStream: 153 GB/s
- IRIS Xe MAX GPU
  - 96 EU @ 1650 MHz
  - BabelStream: 60.2 GB/s
- Xe-HP SDV GPU
- Scaling:
  - Intel(R) Xeon(R) Gold 6248
  - 20 cores @ 2.5 GHz
  - BabelStream: 46 GB/s



Test applications

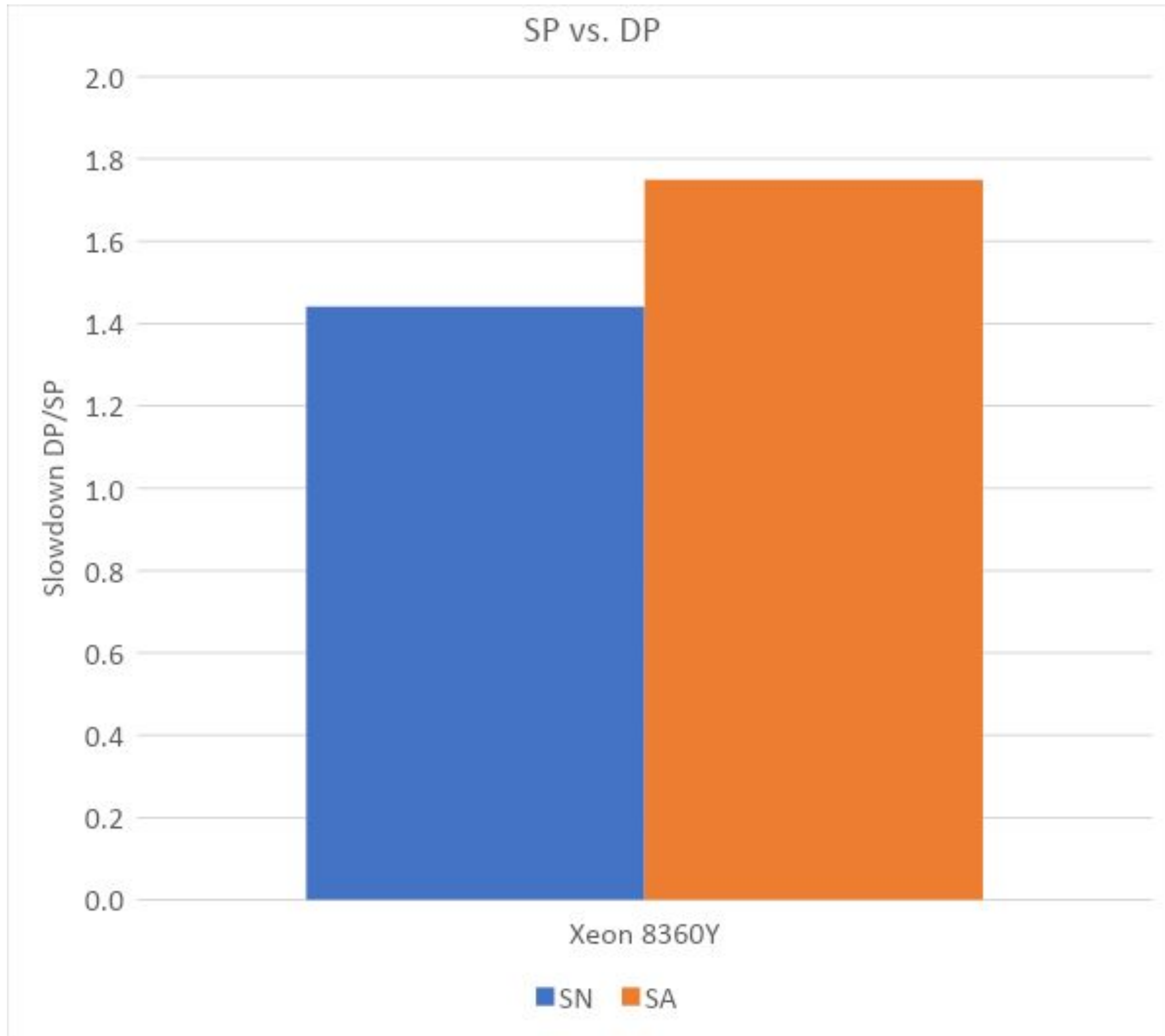
- CloverLeaf – 3D hydrodynamics (Mantevo suite) – 256^3
  - Eulerian-Lagrangian
  - Low-order stencils, bandwidth bound
  - 50+ kernels – many on the boundary only
- RTM – 3D Acoustic wave propagation – 320^3
  - Forward part of a seismic imaging application
  - 8th order stencil
  - 6 variables per point – AoS vs. SoA
- OpenSBLI – 3D Navier-Stokes solver
  - Store All (SA) version – 80 loops, some 8th order stencils, mostly low-order, bandwidth bound – 240^3
  - Stone None (SN) version – 18 loops, highly complex 4th order stencils, latency bound – 420^3

Loop formulation: nd_range or flat



- On GPUs, nd_range is almost always better
- RTM app workgroup size tuning:
  - Default: 32x4x2
  - Iris XE: +2.8%
  - Xeon 8360Y: +11.7%
- Xeon 8360Y: RTM flat 1.64x faster than nd_range
  - no size WG found that matches

Performance highlights



- Switching from single to double precision
  - 2x the FLOP instructions (higher latency too)
  - 2x the data movement
- OpenSBLI SN version mainly latency bound
- OpenSBLI SA version more bandwidth bound: closer to 2x

AoS vs. SoA on RTM

AoS vs. SoA data structures

Runtime (seconds)

18
16
14
12
10
8
6
4
2
0

1.15x

16.74

14.53

1.33x

6.61

4.96

nd_range IrisXE

flat Xeon 8360Y

AoS SoA

Level 0 and OpenCL backends

Level Zero vs. OpenCL

Speedup of L0 over OpenCL

1.16
1.14
1.12
1.1
1.08
1.06
1.04
1.02
1
0.98
0.96

SN sp
RTM
SA sp

Level Zero is faster

OpenCL is faster

Iris Xe

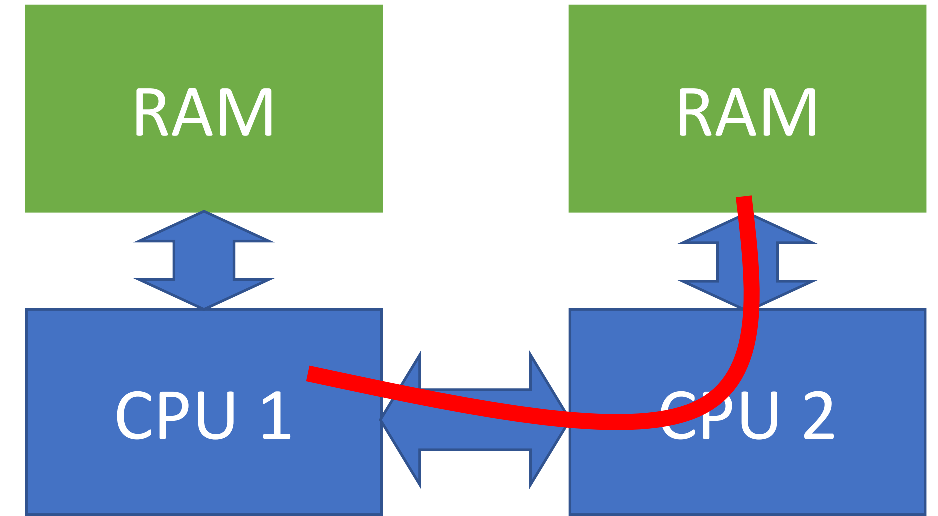
- Slight edge with L0
- Try OpenCL too…

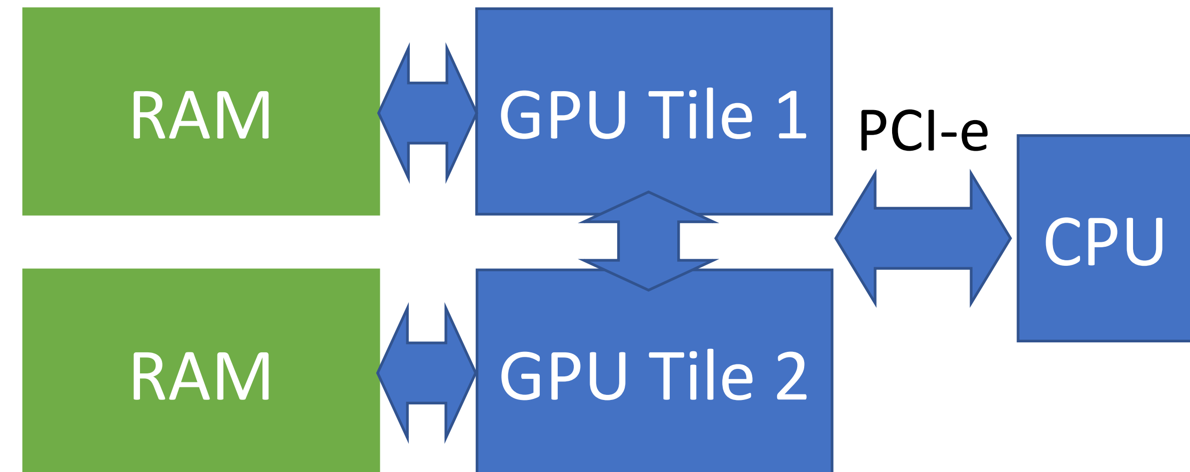
Single device conclusions

- Excellent performance, high device utilization (especially bandwidth in these cases)
- 2 key options:
  - Flat vs. nd_range
  - Level Zero vs. OpenCL
  - Largely independent of each other
- Currently often large differences, some reasons unclear
  - You should definitely try both, differences even on simple kernels
- Go with SoA…

Multi-CPU, multi-GPU scaling

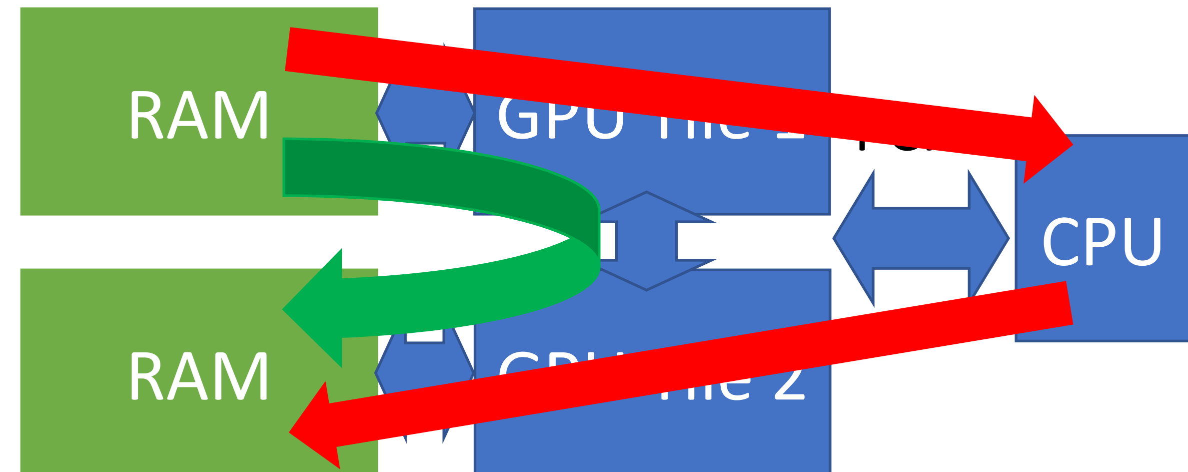
- All of the above from a single NUMA domain
- Good understanding of performance issues with OpenMP on NUMA CPUs
  - When available, use 1 MPI per socket
  - Otherwise accesses to UPI/QPI slower
- Multi-tile GPUs - NUMA
  - PCI-e link to host
  - Implicit scaling: use as one device
  - Explicit scaling: can be treated as 2 devices





Multi-GPU scaling

- Data communication between tiles happens either through tile-to-tile interconnect or the CPU, via PCI-e (a lot slower…)
- Options for explicit scaling:
  - Manage multiple devices and queues from the same process
    - Fast but not scalable…
  - Manage 1 tile per MPI process
    - Explicit copies through host memory slow…
    - GPU-Aware MPI (IMPI, MVAPICH2) GPU buffers passed to MPI functions
      - Performance depends on implementation
      - Currently goes through the host
      - But soon RDMA
      - Long-term scalable solution!



MPI Communications scheme

```
double *halo_buffer = cl::sycl::malloc_device(size, Queue);
...
Queue.submit([&](cl::sycl::handler &cgh) {...
 cgh.parallel_for<class pack_halo_data>(...)});
double *send_buffer;
if (GPU_aware_MPI)
  send_buffer = halo_buffer;
else
  send_buffer = ...;
  Queue.copy(halo_buffer,send_buffer, size);
Queue.wait();
MPI_Send(send_buffer, ...);
```

USM allocation of buffer to hold halo data

Kernel to pack buffer

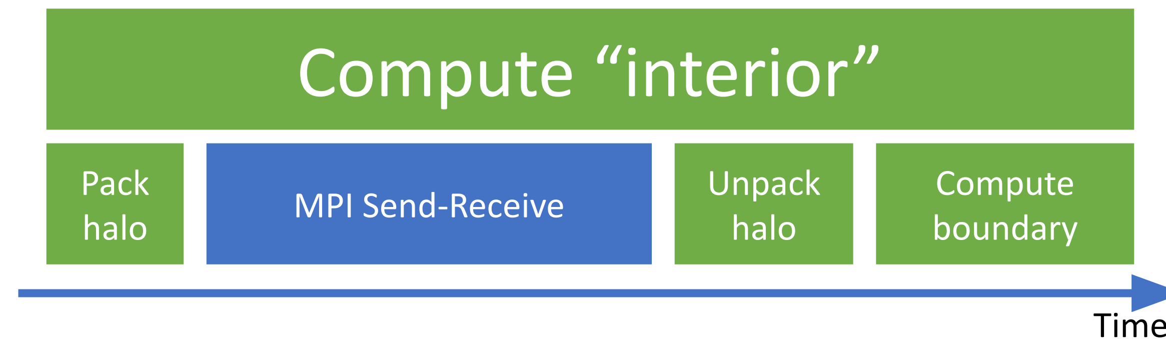
When using GPU-aware MPI, we can use the buffer directly

Otherwise copy to CPU explicitly

Send MPI message

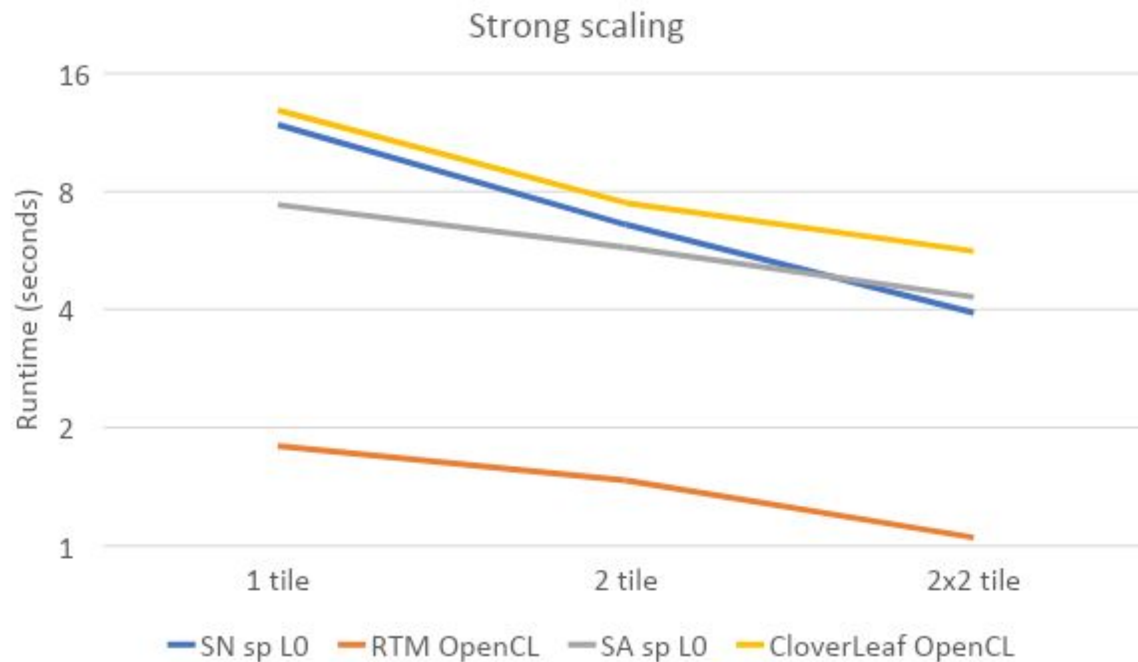
- USM is recommended for easy access to raw data and explicit copies
- Simple code path for with/without GPU-aware MPI
- Works on the CPU too – use “GPU-aware MPI” to avoid extra copy

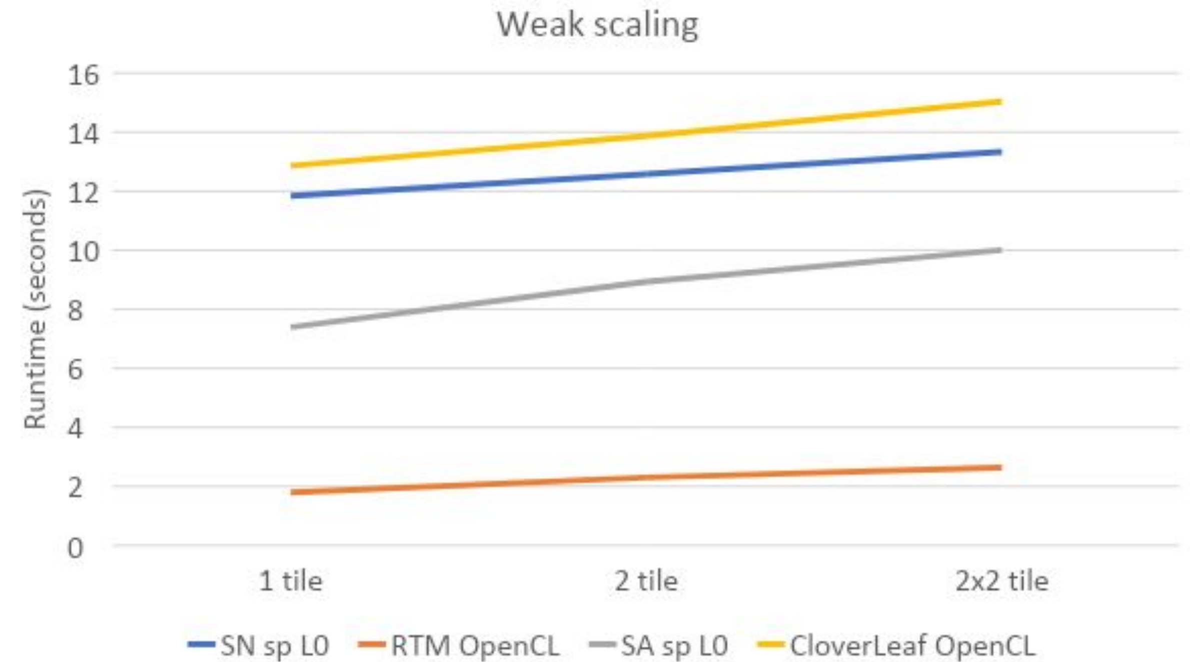
MPI Communications scheme



- Latency hiding scheme
- Concurrent streams of kernels and operations that are independent
- CPU free to process MPI comms while GPU is actively computing

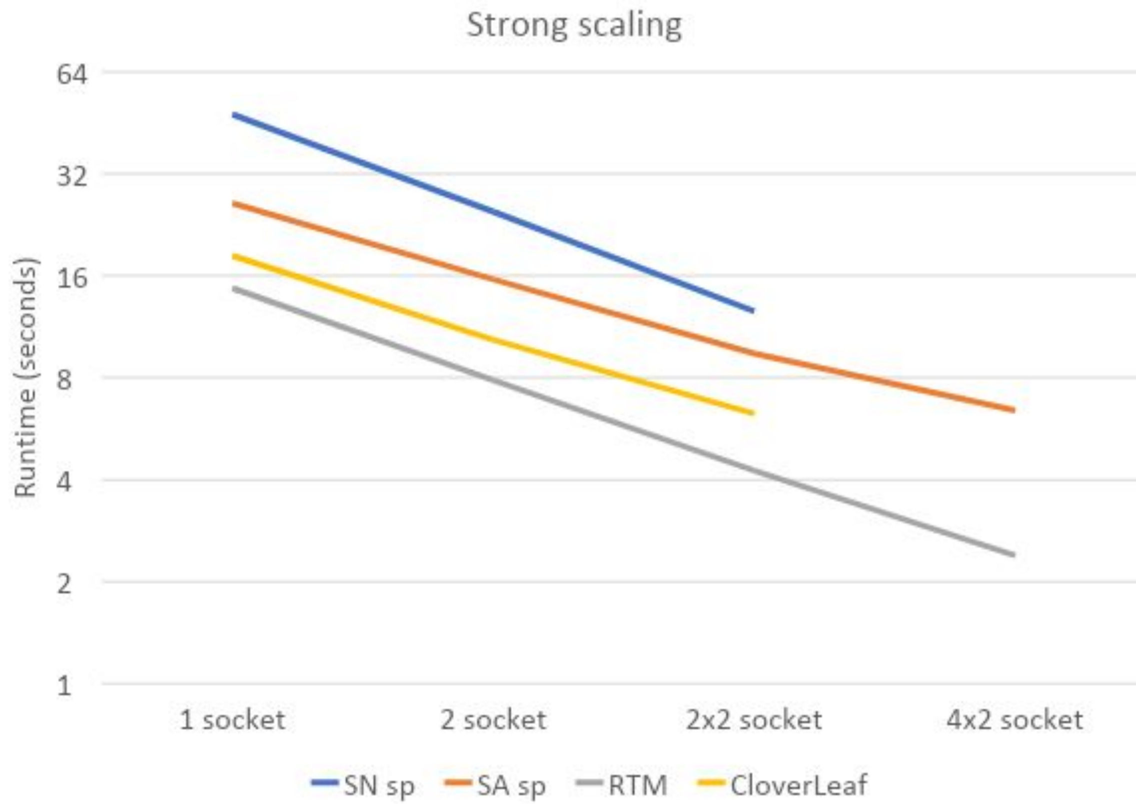
Strong & weak scaling on Xe-HP SDV





- Frequent MPI comms, all through CPU, without GPU-aware MPI (OpenMPI 4.1.3)
- Strong scaling: 45-75% parallel efficiency, weak scaling: 68-88%

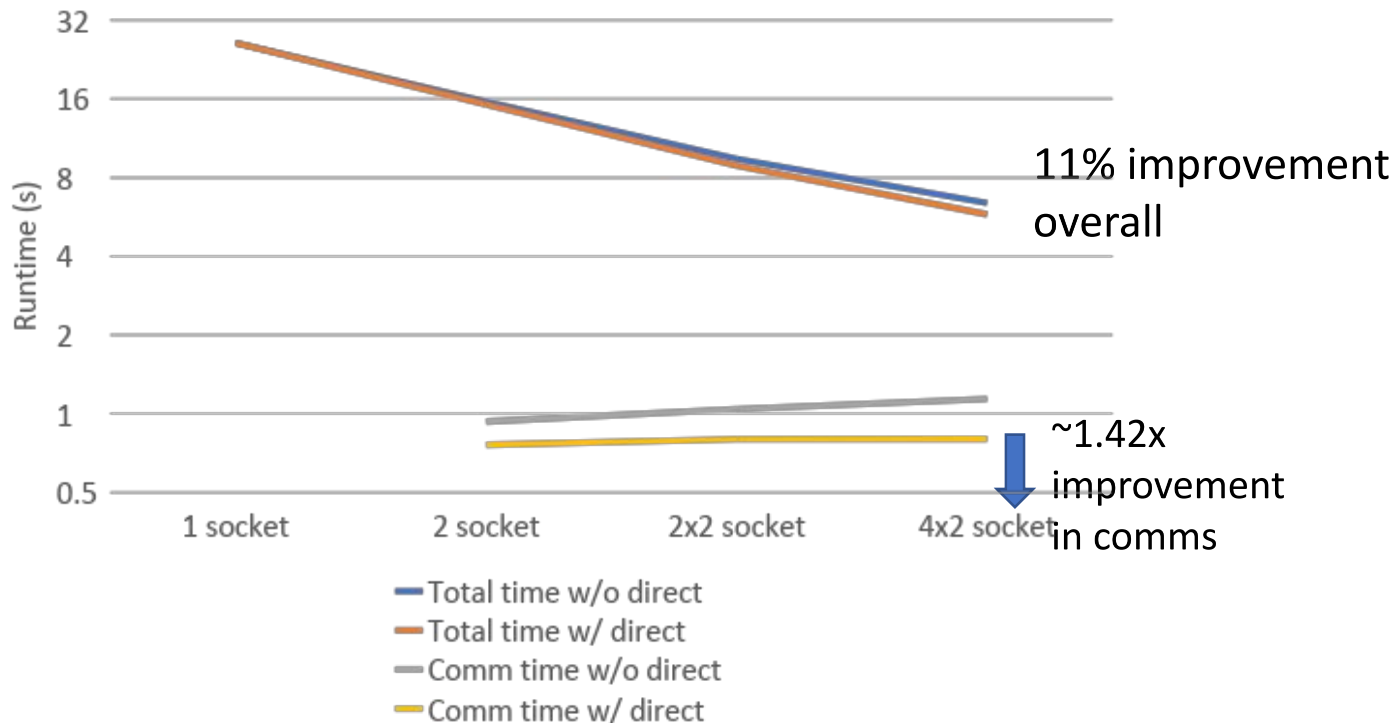
Strong & weak scaling on Gold 6248



- Strong scaling: 51-95% parallel efficiency, weak scaling: 91-98%

Copy-less transfers & GPU buffers

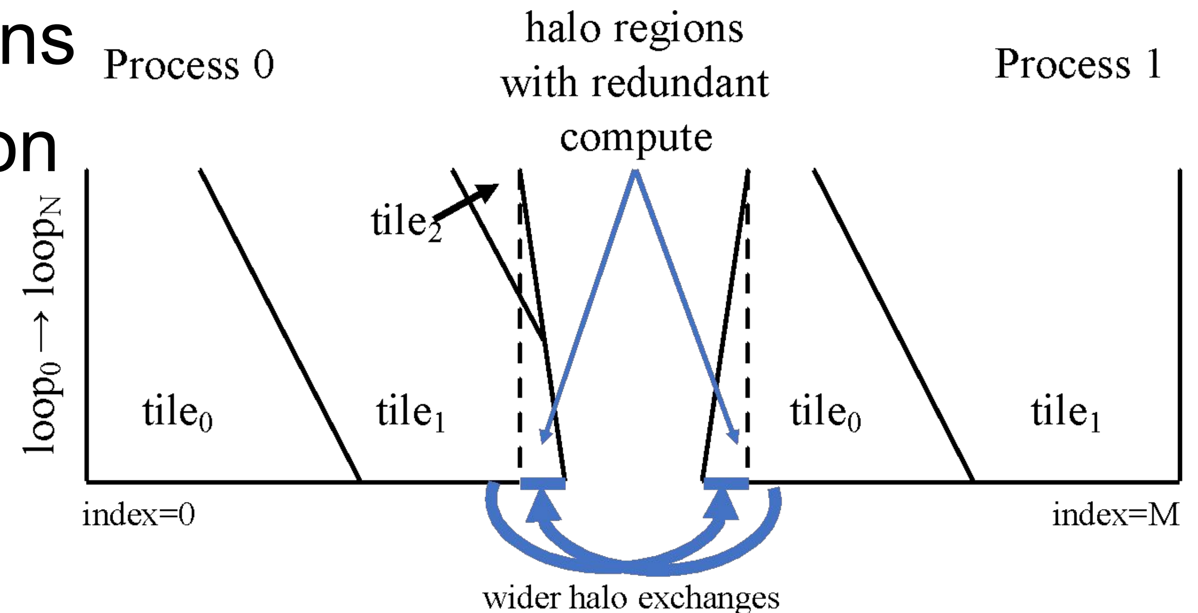
- IMPI has support for GPU buffers `export I_MPI_OFFLOAD=2`
- Results shown on CPU



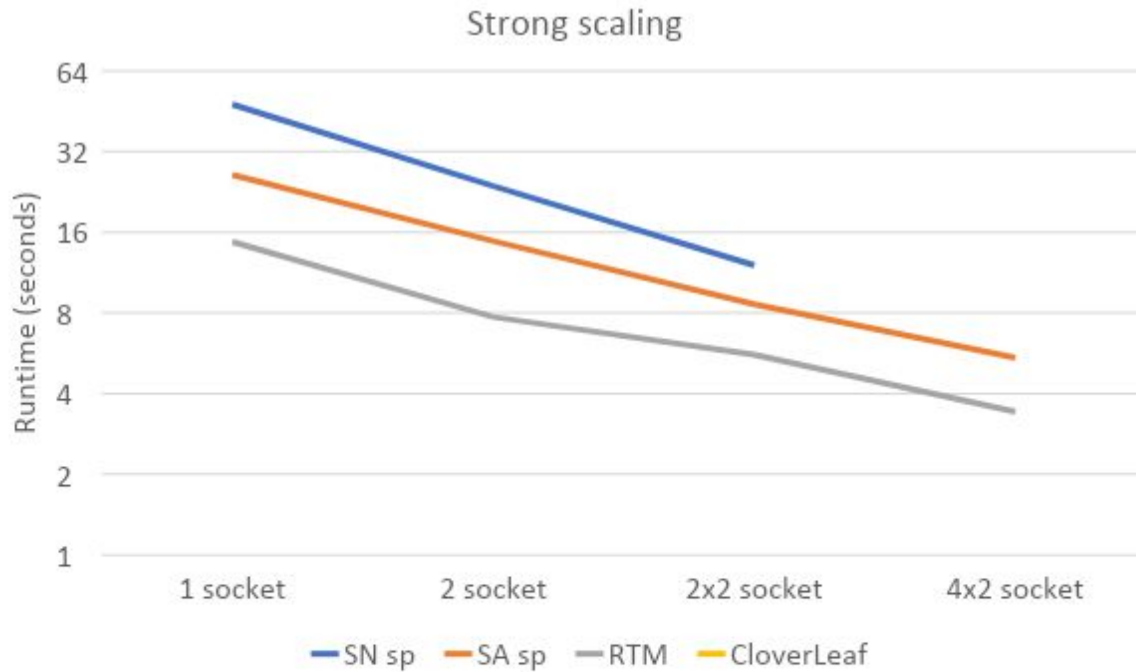
- OpenSBLI SA sp
  - 240^3, 100 iterations
- Xeon Gold 6248
- MPT 22.2

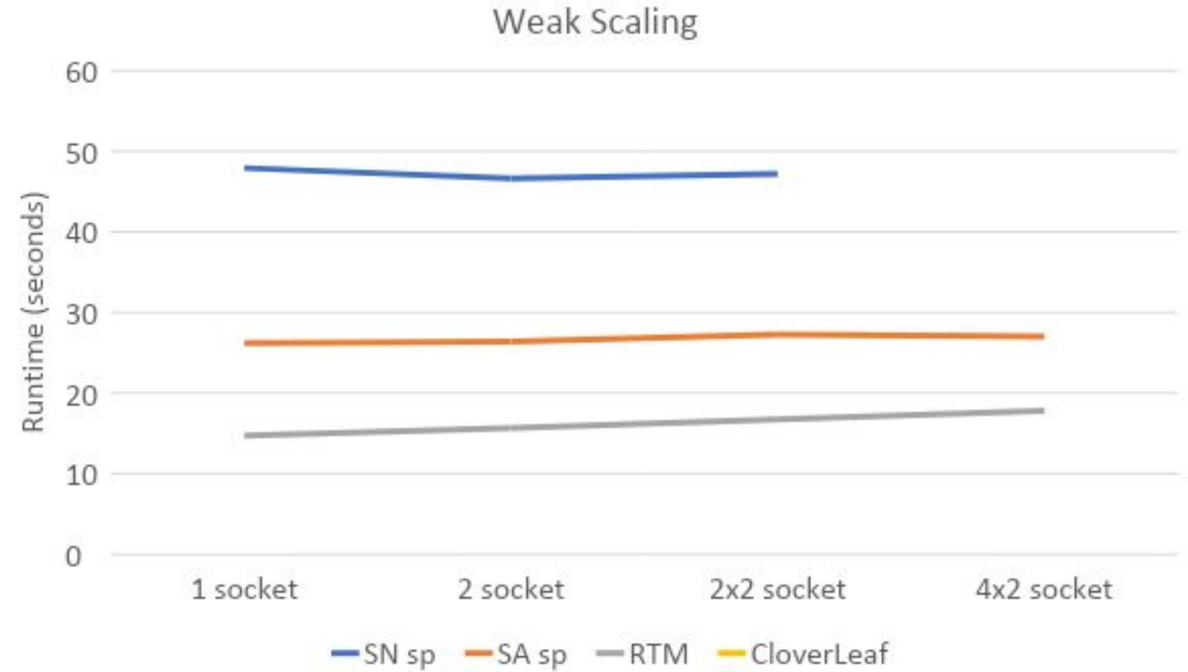
Improving scalability – communications avoidance

- GPU-GPU communications still suffer from much higher latency
  - GPU throughput vs. latency – small kernels/operations
  - Comms initiated from CPU
- Applies to CPUs too to a lesser extent
- OPS support for reducing frequency of comms in exchange for larger messages & redundant computations
- Cross-loop analysis and optimization
- Removes need for exchange on some work arrays: reduced overall volume



Improving scalability – communications avoidance





- SA sp improved efficiency: strong 51%->60% weak 91->97%
- RTM degraded efficiency: strong 77%->52% weak 97%->82%

Scalability conclusions

- NUMA-based GPUs – several ways to program
    - Implicit vs. explicit scaling – explicit likely to win out
    - Prepare for GPU-enabled MPI – use USM & 2 code paths: works with CPU too
- Communications more expensive with GPUs due to worse latency
    - Extra signaling, under-utilization
    - Try and exploit concurrency
    - Try and reduce number of messages, increase size