Parallelization of the FV3 dycore for GPU and MIC processors

Mark Govett, Jim Rosinski, Jacques Middlecoff, Yonggang Yu, Daniel Fiorino, Lynd Stringer

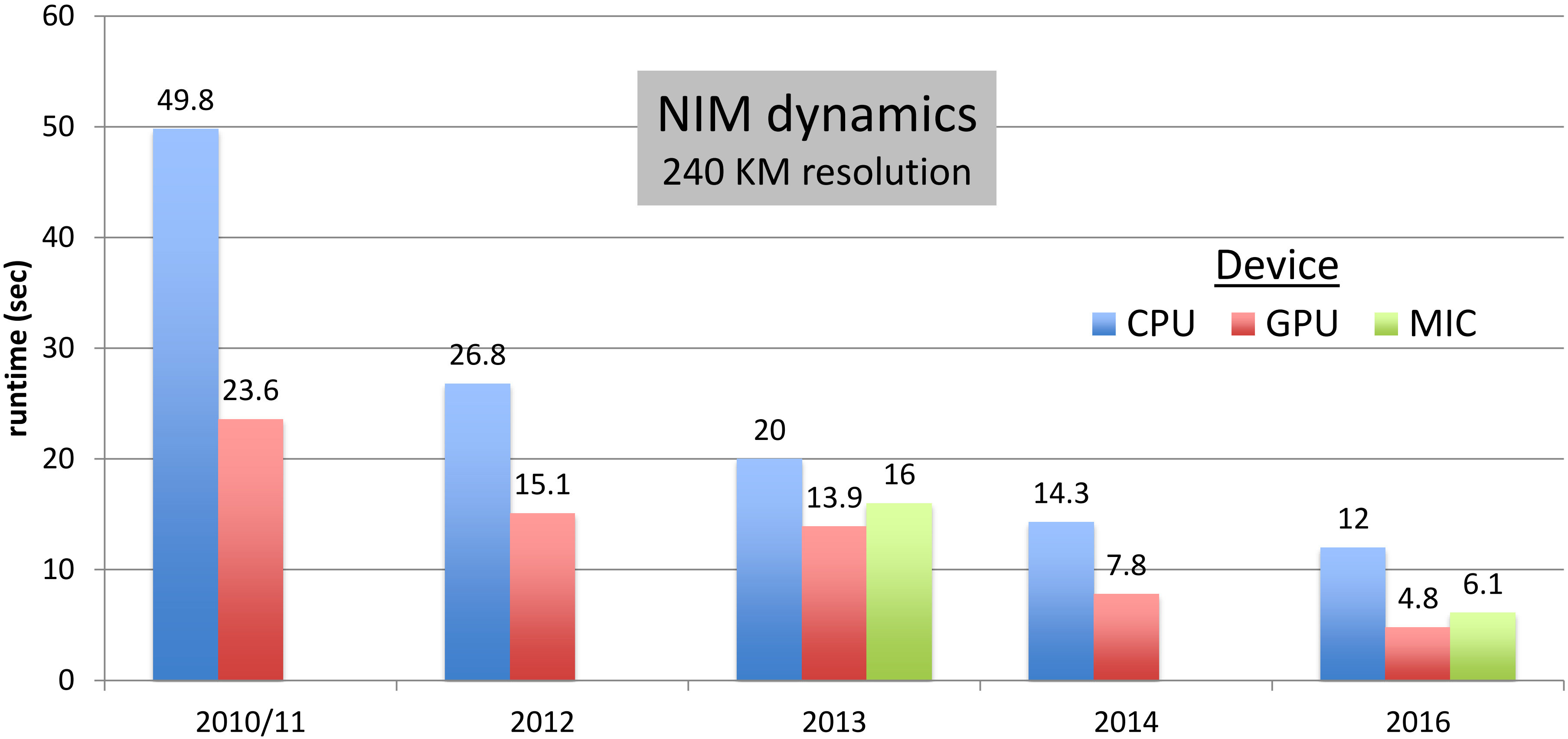
Outline

- Transition from NIM to FV3
- Performance Portability
  - NIM performance & scaling
  - FV3 parallelization

NIM Achievements

- Designed for fine-grain parallel
  - Icosahedral uniform grid
  - Lookup table to access neighbor cells
- Performance portable with a single source code
  - OpenACC for GPU
  - OpenMP for CPU, MIC
  - F2C-ACC compiler improved OpenACC compilers
- Enabled fair comparison between CPU, GPU & MIC
  - Single source code
  - Bitwise exact between CPU, GPU & MIC
  - Optimized for all architectures
  - Same generation hardware, standard chips
- Benchmark code for NOAA fine-grain procurement

Device Performance

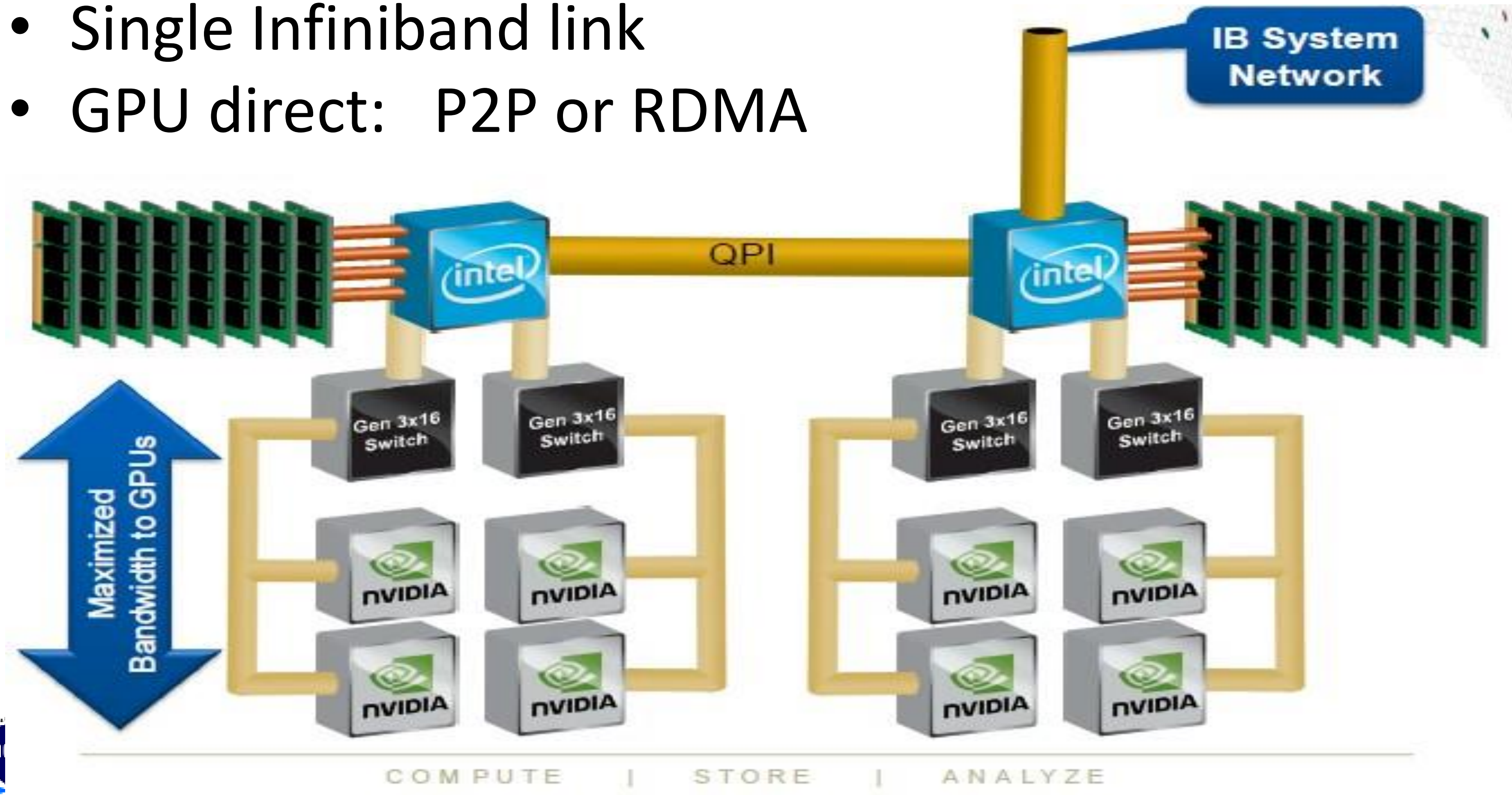


Year	CPU: 2 sockets	Cores	GPU	Cores	MIC	cores
2010/11	Westmere	12	Fermi	448		
2012	SandyBridge	16	Kepler K20x	2688		
2013	IvyBridge	20	Kepler K40	2880	Knights Corner	61
2014	Haswell	24	Kepler K80	4992		
2016	Broadwell	30	Pascal	3584	Knights Landing	68

Cray: CS-Storm Node

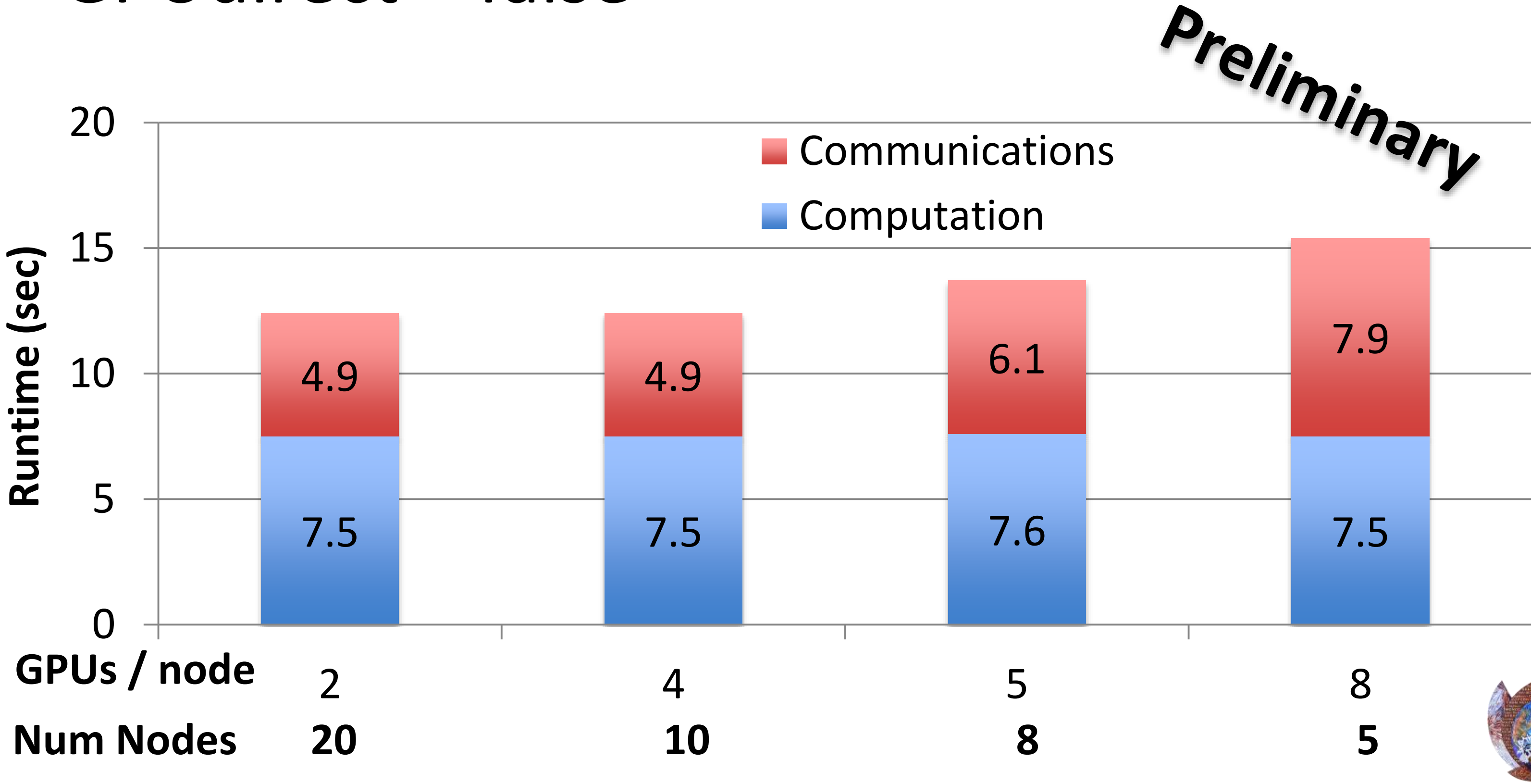
- Up to 8 GPUs / node
- QPI between CPU sockets
- Single Infiniband link
- GPU direct: P2P or RDMA

FDR: 40 Gb/sec
QDR: 54 Gb /sec
EDR: 100 Gb /sec



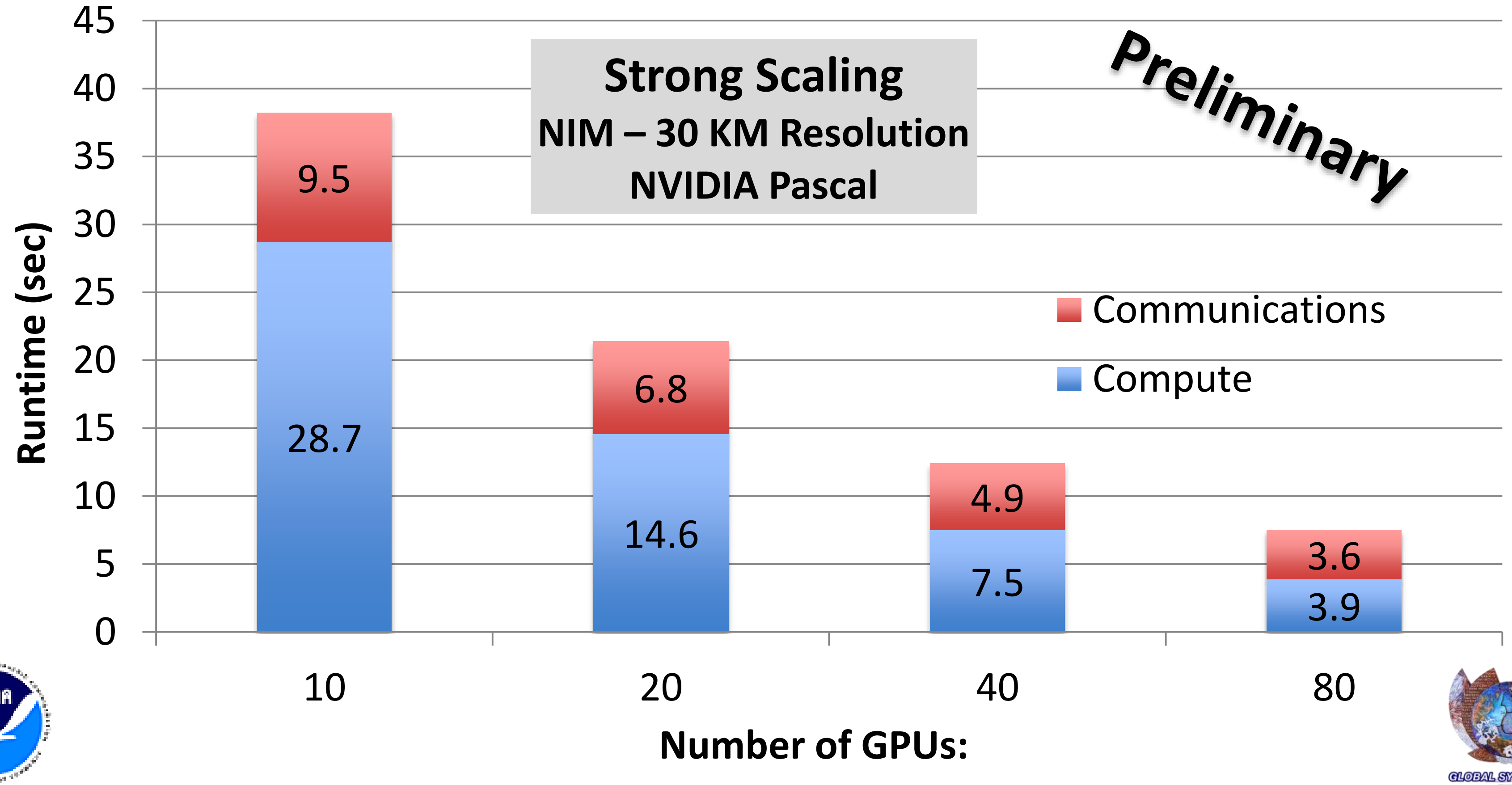
Weak Scaling, CS Storm

- NIM 30 KM resolution
- 40 Pascal P100s, 2 – 8 GPUs / node
- GPUdirect = false



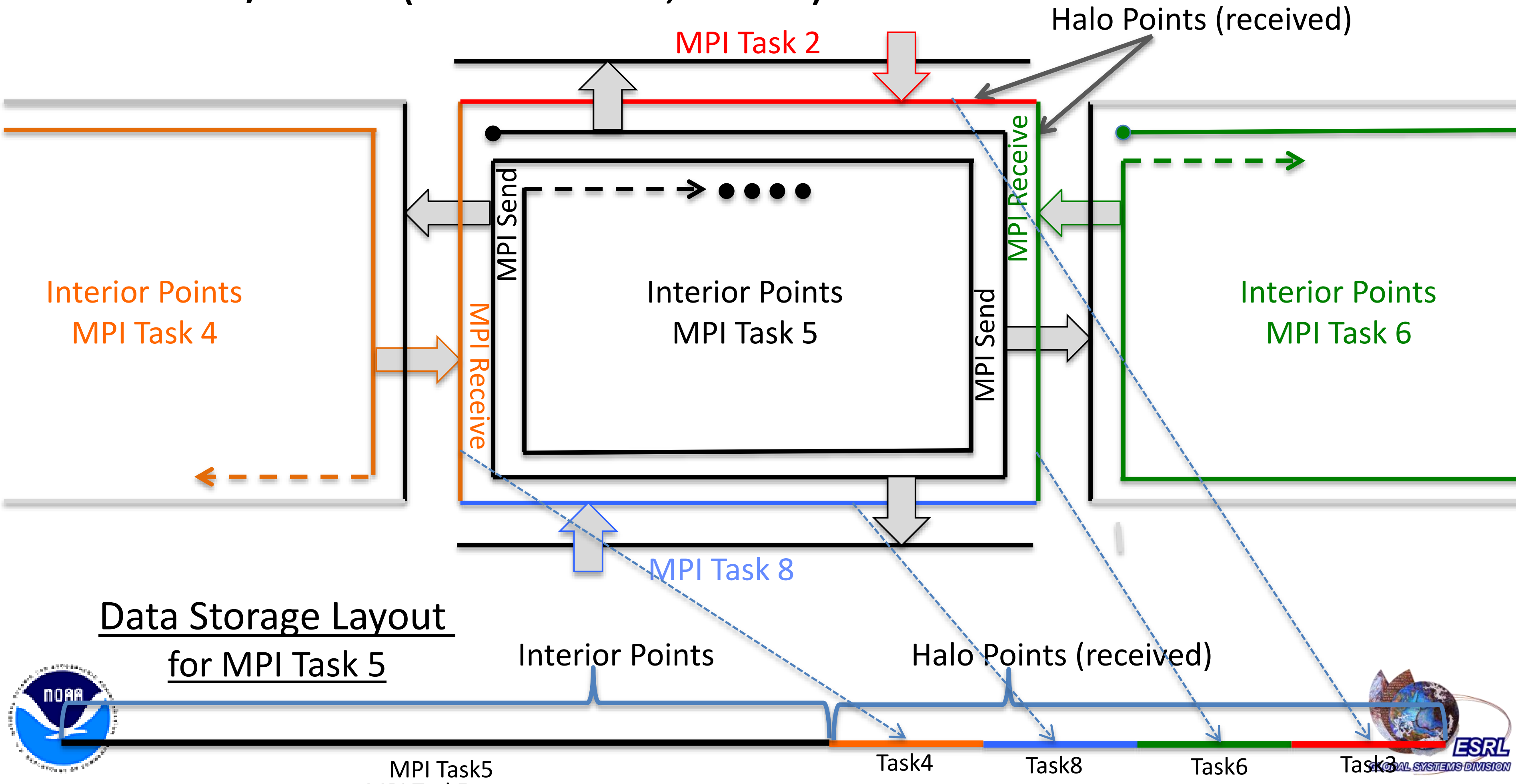
Strong Scaling, CS Storm

- 5 – 40 CPU nodes,
- 2 Pascal GPUs/ per node
- GPUdirect = false



Spiral Grid Optimization: NIM

- Eliminate MPI message packing / unpacking by ordering grid points
- Gave a 35% speedup in dynamics runtime using 16 MPI tasks / GPU (Middlecoff, 2015)



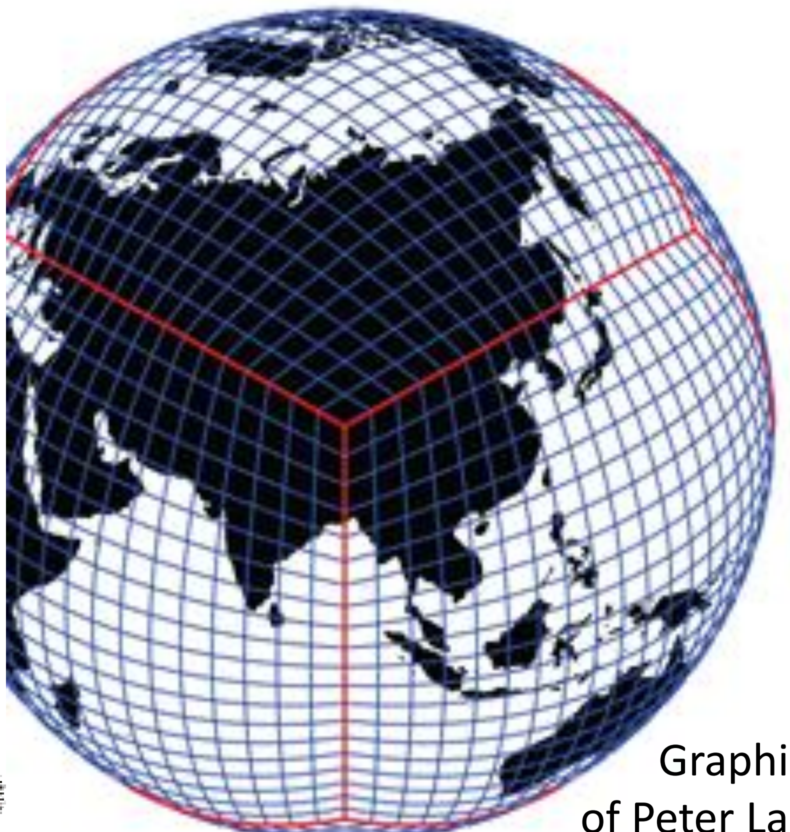
From NIM to FV3

Cube Sphere (FV3)

- 6 faces, 1 MPI rank per face
- Edge and corner points
- Direct access: i-j-k ordering

Icosahedral (NIM)

- Uniform Grid
- No special grid points
- Indirect access: k–I ordering



Graphic courtesy
of Peter Lauritzen (NCAR)

Fine-grain Parallelization of FV3

- Early work by NVIDIA demonstrated poor performance with original code
- Goal is to adapt the FV3 to run on GPU, MIC
  - Expose sufficient parallelism
  - Minimize code changes
  - Maintain single source code
- Achieve performance portability
  - OpenMP for CPU, MIC
  - OpenACC for GPU
- Bitwise exact results between CPU, GPU, MIC

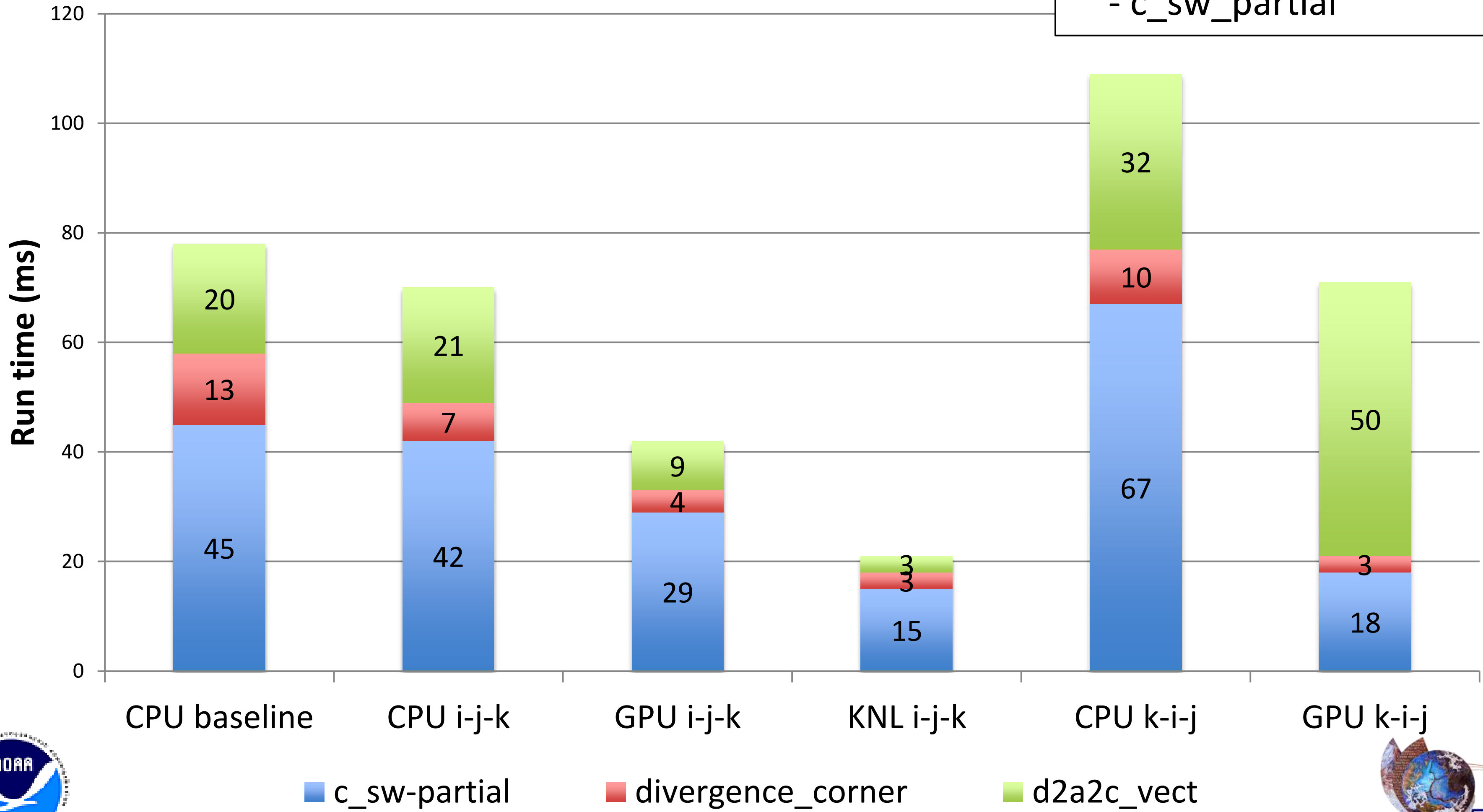
GPU Parallelization

- C_SW (shallow water)
  - Push “k” loop into routines to expose more parallelism for GPU
    - Little benefit for MIC, CPU,
  - Two test cases built
    - I – J – K ordering
    - K – I – J ordering
  - Optimizations for CPU, GPU, MIC
    - Evaluation of results
    - Performance benefit versus impact to code

C_SW Performance

- 2013 IvyBridge, 2013 Kepler K40, 2016 KNL
- Execution time for a single call to C_SW

C_SW Call Tree
- divergence_corner
- d2a2_vect
- c_sw_partial



C_SW Conclusions - Part 1

- K-I-J variant involves lots of changes with little performance benefit
  - c_sw_partial, divergence k-i-j gave 1.5X benefit over i-j-k, warrants further investigation
- I-J-K variant resulted in a small improvement in CPU performance
  - Few changes to the code
    - Promote 2D arrays to 3D
    - Add K loop
    - Minor changes to OpenMP directives

FV3 dynamics

- dyn_core (100%)
  - c_sw (13%)
    - d2a2_vect
    - divergence_corner
  - update_dz_c (2%)
  - riem_solver_c (14%)
  - d_sw (38%)
    - FV_TP_2D (37%)
      - copy_corners (0.1%)
      - xppm0 (14%)
      - yppm0 (14%)
    - xtp_v
    - xtp_u
  - update_dz_d (10%)
    - FV_TP_2D (37%)
  - riem_solver3 (1%)
  - pg_d (5%)
    - nh_p_grad (5%)
  - tracer_2d (6%)
  - remapping (6%)

<u>Notes</u>

- Model configured for non-hydrostatic, non-nested, with 10 tracers
- Runtime percentages are for Haswell CPU
- Percentages represent aggregate values
- Remapping is done once every 10 timesteps
- Current efforts are shaded

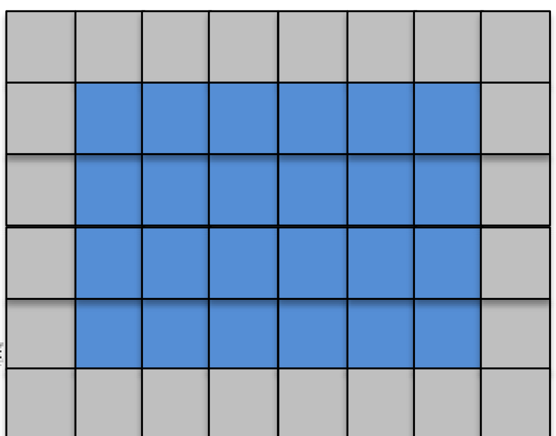
```
module m1                    !Poor Performance, not enough shared memory
        integer, parameter :: isd = -2,  ied = 195, jsd = -2, jed = 195
        integer, parameter ::  is  =   1,   ie =   192, js  =   1, je   = 192
        integer, parameter :: npz = 128
contains
  subroutine s1(a)
        real,intent(INOUT) :: delpc( isd:ied, jsd:jed, npz) delpc,...          ! global arrays
        real, dimension( is-1:ie+1, js-1:je+2 ) :: fy, fy1, fy2,fx, fx1,fx2      ! local arrays
!$acc kernels
!$acc loop private(fx,fx1,fx2,fy,fy1,fy2)
   do k = 1, npz    ! gang loop
!$acc loop collapse(2)
        do j=js-1,je+2          ! vector loop
          do i=is-1,ie+1        ! vector loop
              fy1(i,j) = delp(i,j-1,k);     fy(i,j) =   pt(i,j-1,k);      fy2(i,j) =    w(i,j-1,k)
         enddo
      enddo

! additional calculations with fx,fx1,fx2,   handling of corner points
! dependencies on fx1, fy1, ect require synchronization here
      do j=js-1,je+1
         do i=is-1,ie+1
           delpc(i,j,k) = delp(i,j,k) + (fx1(i,j) - fx1(i+1,j) + fy1(i,j) - fy1(i,j+1)) * rarea(i,j)
```

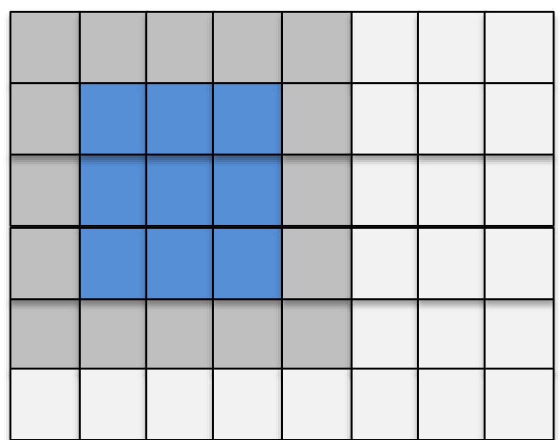
Tiling / Cache Blocking

- Increase utilization of GPU shared / cache memory
  - 48 / 16KB per multiprocessor
- Increased complexity of code
  - Add chunk loops, indexing, etc
- Gave 3X performance boost for simple test case
  - Testing in c_sw

```
do j=1, 192, jchunk          !chunk loop
  do i=1,192, ichunk         !chunk loop
    do jx = 1, jchunk        !tile loop
    do ix = 1, ichunk        !tile loop
      i = ic + ix – 1;   j = jc +jx -1
```

```
do j=1, 192        ! worker loop
  do i=1,192       ! vector loop
```





Conclusion

- NIM work has ended
  - Using NIM for performance & scaling
    - Testing on KNL, Pascal chips
  - Apply knowledge toward FV3
    - Serial, parallel performance, portability
- FV3 parallelization is going fairly well
  - Goal is single source code, performance portability on CPU, GPU & MIC
  - Modifying code to improve performance
    - Push “k” loop into subroutines
    - OpenACC parallelization using PGI compiler
    - Exploring optimizations including tiling