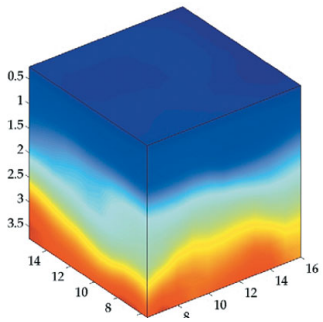
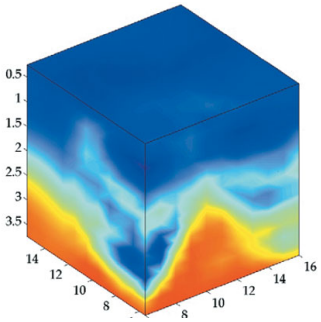
FULLY EXPLOITING A GPU SUPERCOMPUTER FOR SEISMIC IMAGING

Lionel Boillot, Long Qu; TOTAL

Pascal Vezolle; IBM

HPC SEISMIC IMAGING

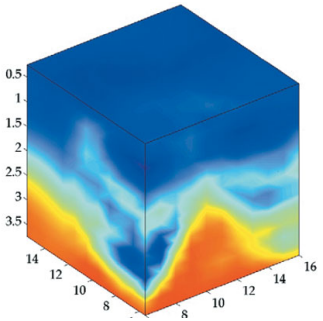


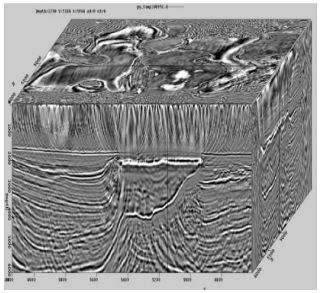


x1 Tomography

x50 Full Waveform Inversion

x200 Wave Equation Tomography





x1 Beam

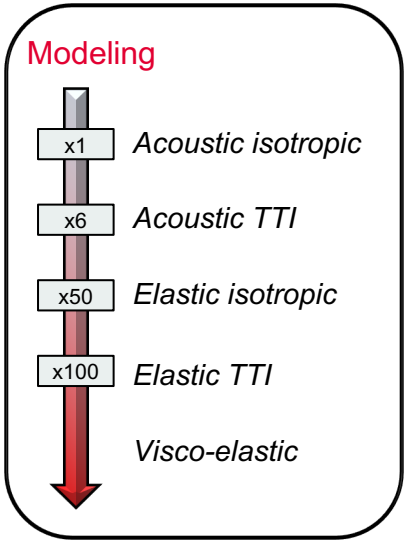
x10 Kirchhoff

x100 Wave Equation Migration

x1000 Reverse Time Migration

x5000 LS-RTM

HPC WAVE EQUATION

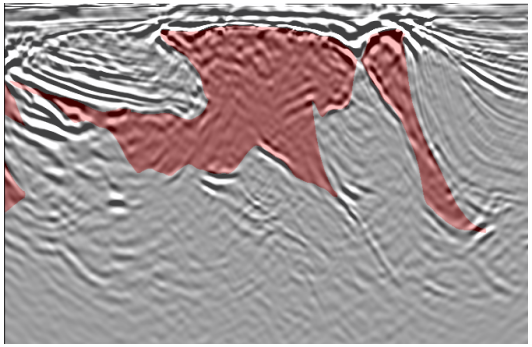


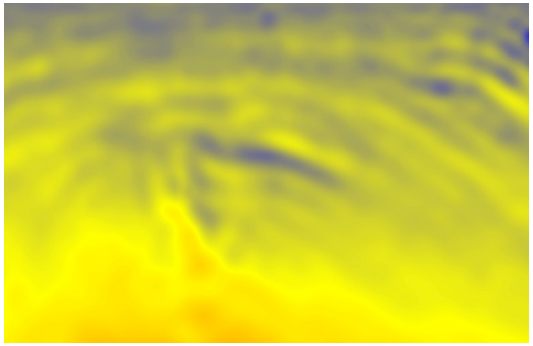
$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = s(t)$$

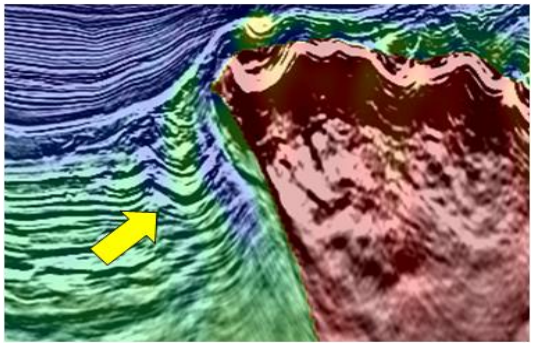
$$\rho \frac{\partial^2 u}{\partial t^2} - \nabla\lambda(\nabla u) - \nabla\mu\left[\nabla u + \left(\nabla u + (\nabla u)^T\right)\right] - (\lambda + 2\mu)\nabla\nabla u - \mu\nabla \times \nabla \times u = s(t)$$

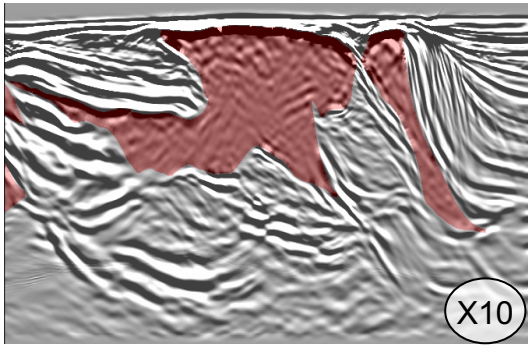
TOTAL

REAL CASES

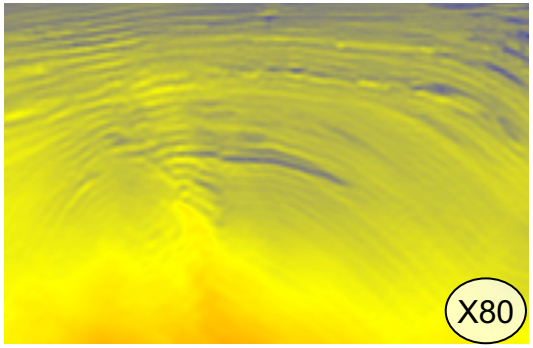




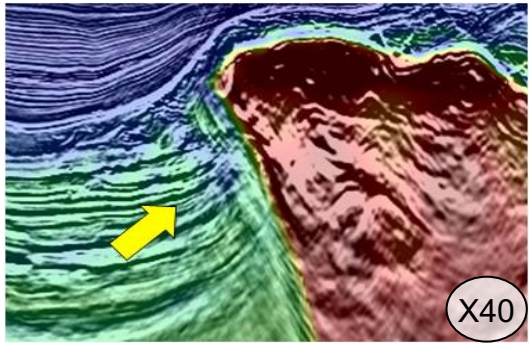




MORE PHYSICS IN IMAGE



HIGHER DEFINITION



MORE PHYSICS IN MODEL

AGENDA

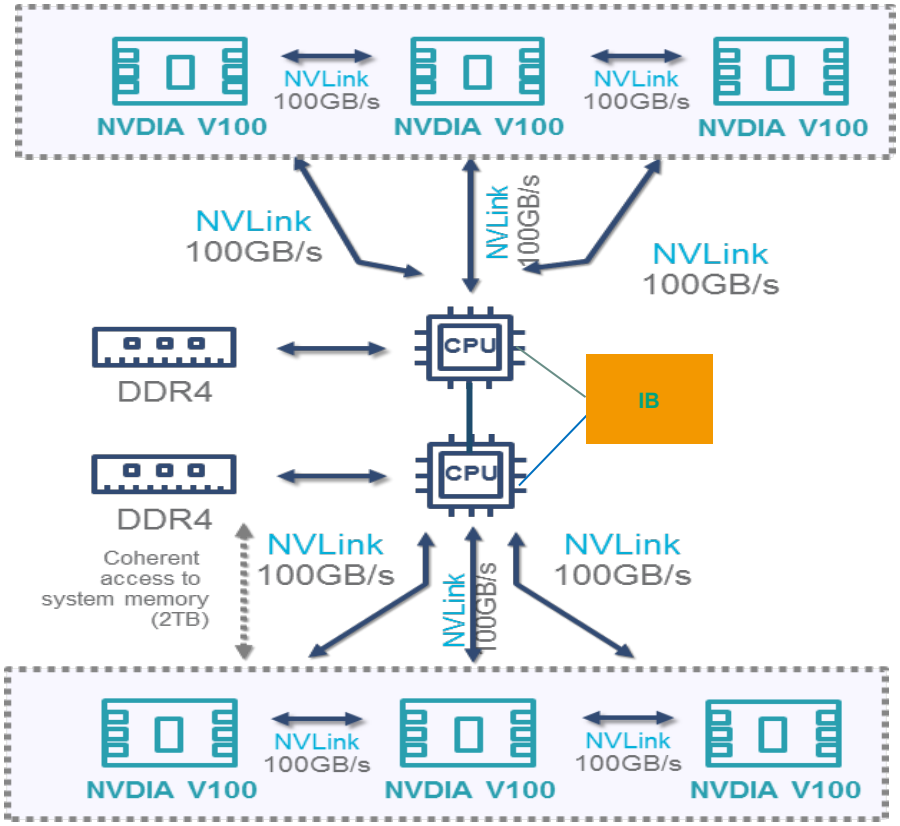
- Machine Specifications
- Seismic Algorithm
- Porting, Optimization & Performance

MACHINE SPECIFICATIONS

NODE DETAILS

- IBM Power System AC922
  - 2 IBM Power9 18C 3.45GHz
  - 6 NVIDIA GV100
  - Dual-rail Mellanox EDR InfiniBand

- Main features
  - NVLink High Bandwidth on sockets
  - UVMA: Unified Virtual Memory Addressing
  - NX on-chip gzip compression accelerators



MACHINE SCALE

"Pangea3" = **558 nodes** = 1116 Power9 CPU + 3348 Nvidia GPU

November 2019 performance:

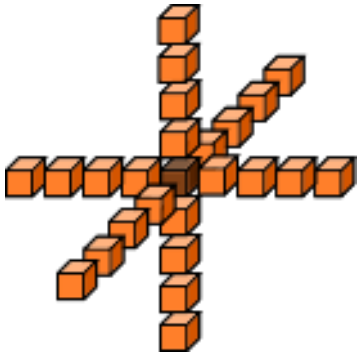
- 11th global ranking
  - Linpack 17.86 PFlop/s
  - Theorical 25.03 PFlop/s

- 9th global ranking
  - 1.37 MWatts
  - 13.07 GFlops/Watt

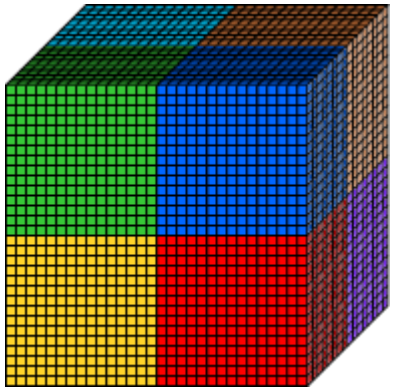
SEISMIC ALGORITHM

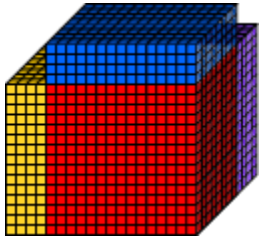
WAVE-BASED ALGORITHMS

- 3D-Wavefield modeling with **Full-Waveform Inversion** (FWI) and **Least-Squares Reverse Time Migration** (LS-RTM)
  - various propagators: acoustic/elastic + isotropic/TTI
  - finite-difference scheme 8th order: e.g. 25-points stencil



  - domain decomposition on regular grids





Local sub-domain:
stencil => halo

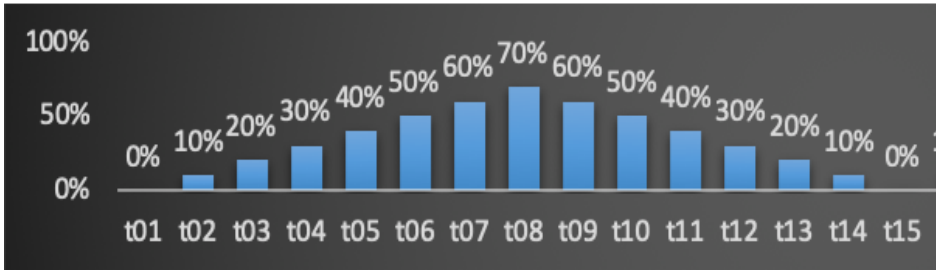
COMMON SKELETON FOR LS-RTM AND FWI METHOD

```
Do, until Inverse Problem convergence
    For shot = 1 : n_shots

      // FORWARD SIMULATION
     For timestep = 1 : n_times
          Computation
          Communication
          Save forward snapshot

      // BACKWARD SIMULATION
     For timestep = n_times : 1
          Computation
          Communication
          Load forward snapshot
          Compute seismic operations on forward & backward snapshots

    Compute Gradient Descent and update on shot summation
```

Timestep: 10 … 20 … 30

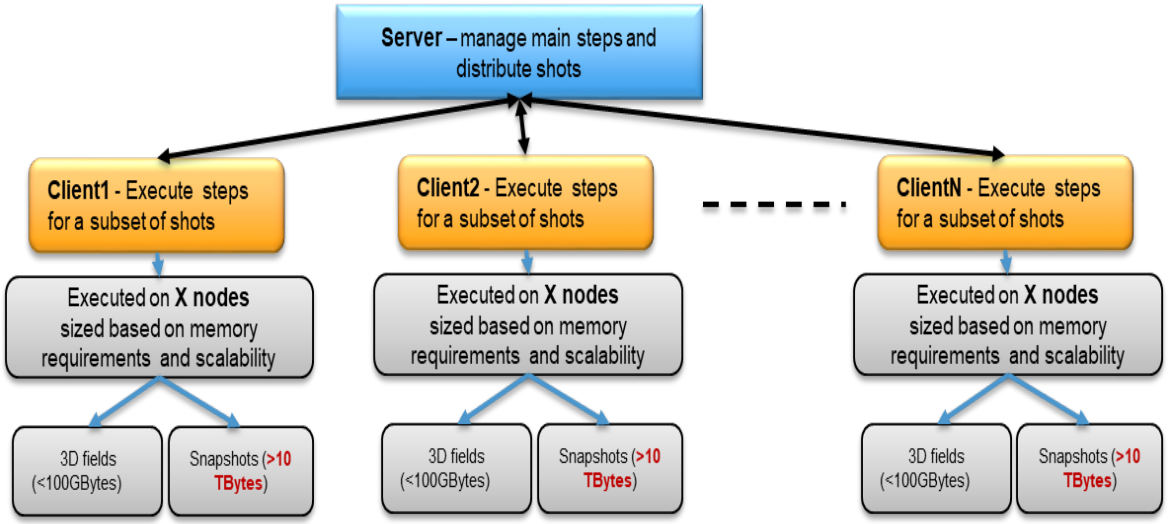
Timestep : 30 … 20 … 10

Forward-Backward memory profile during one shot step



LS-RTM AND FWI SOLVER SCHEME - PRE-REQUISITES

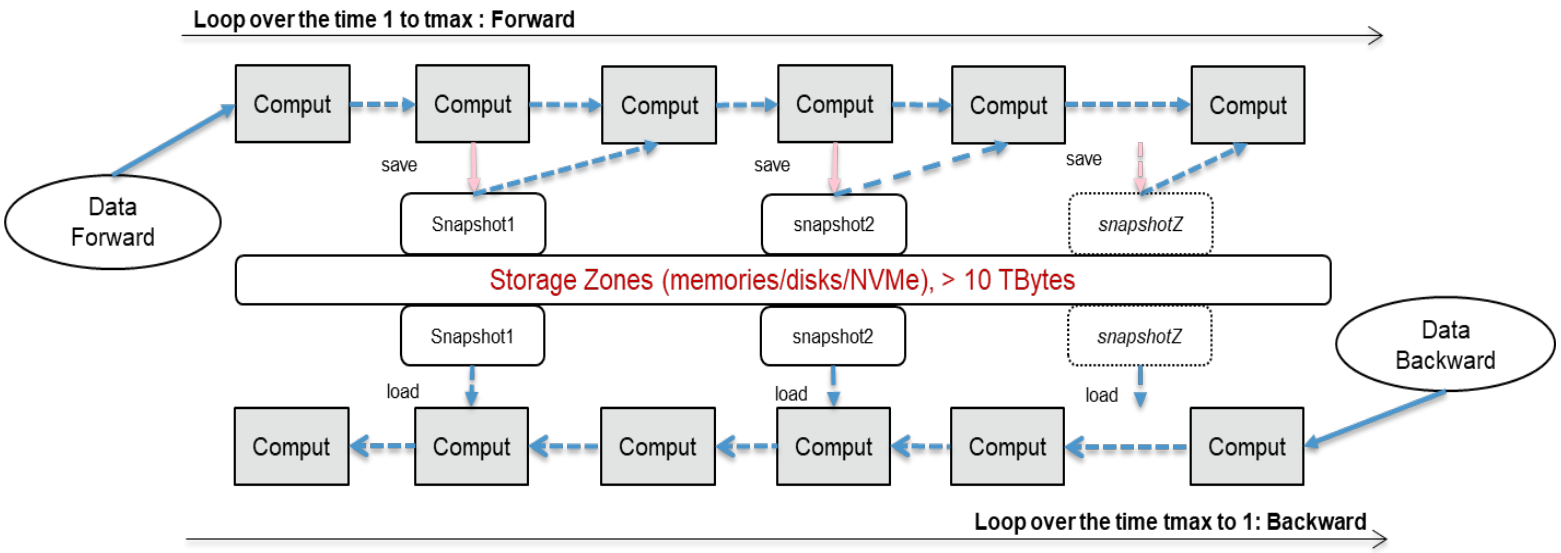
Clients/Server implementation – "massively parallelism"

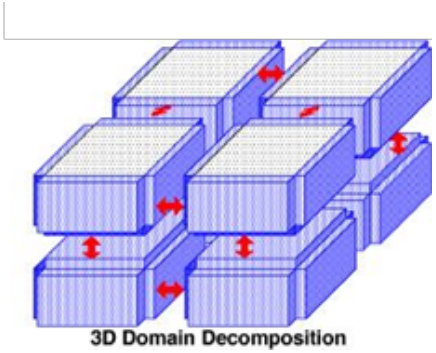


- Data input per shot, >10K shot per run
- 3D space Cartesian domain (up to 1000x1000x1000),
  up to >3GBytes per value (pressure, velocity…)
  partitioned in subdomains (one per MPI task)
- Old CPU cluster configuration:
  - Snapshot options: memory, local disk, parallel FS
    ➔ Keep snapshots in node memories as much as possible
    - Very fast but need a lot of nodes
    - Good scalability, > 40% of CPU efficiency

LS-RTM AND FWI FINITE IMPLEMENTATION PER SHOT SIMILAR TO FOR

Client solver pattern for one shot





Assuming time forward ~= time backward,
the total running time to reduce is defined by:

*Total_time ≈ tmax **
(time_compute + time_communicate +
time_snapshot/step_snapshot +
time_extra)

TECHNICAL GPU CHALLENGES

*Total_time ≈ tmax **
(time_compute + time_communicate +
time_snapshot*/step_snapshot + time_extra)*

Strong reduction of the number of compute nodes per MPI and per shot

=> Per node: more compute but less memory, bandwidth, network, IO and network capabilities

- Both CPU and GPU memory becomes the sizing parameter for parallel executions and production

  ➔ Main challenges:
  - Production improvement by a factor X – average time per seismic shot
  - Keep of the implementation driven by the compute parts and not the snapshot/IO parts

    Old CPU memory snapshot approach not applicable

    (>10TB/s memory bandwidth for snapshot saves and loads, with 100 CPU nodes per client)
  - Optimize TCO and achieve at least a similar efficiency with GPU as CPU cluster (>40% of peak)
  - Maintain a single version of the codes for both CPU and GPU

- Parallel functions **previously running on a large number of MPI tasks must therefore evolve** according to the following strategies:
  - Partial or full GPU porting
  - CPU multi-threading approach
  - Overlaping : GPU-GPU and GPU-CPU
  - New method/algorithm

GPU IMPLEMENTATION MAIN STEPS

1. **New innovative backup/restore approach with GPU and CPU compression**, with asynchronous memory transfers and disk accesses...
2. **GPU and CPU Memory Management**
3. **GPU porting and kernel optimization**
4. **New parallelism and concurrent schemes,** using CUDA Stream, CPU threads, MPI Non-Blocking APIs, CPU/GPU topology, GPU direct features…

5. Productivity and TCO optimization, using NVIDIA MPS

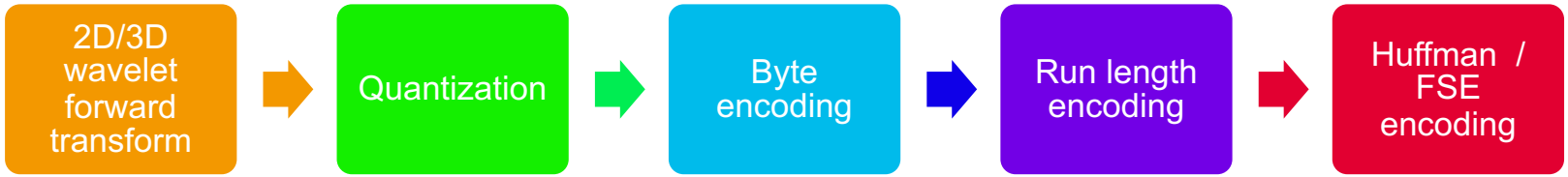
- Increase number of MPI tasks per GPU
- Multiple instance executions

PORTING, OPTIMIZATION & PERFORMANCE

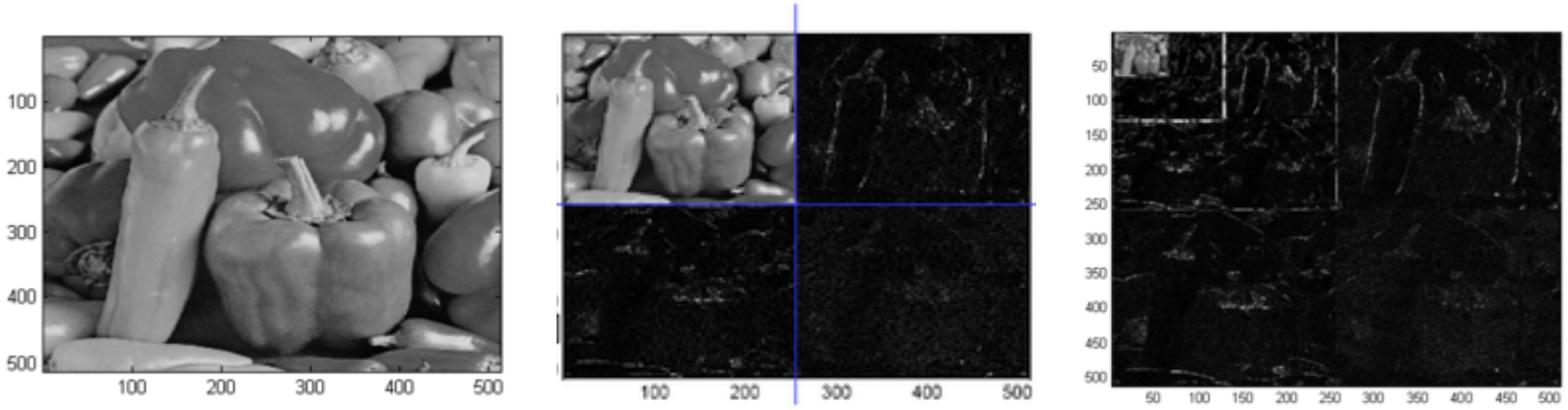
1. NEW INNOVATIVE BACKUP/RESTORE APPROACH WITH GPU AND CPU COMPRESSION

WPC COMPRESSION

- Composed by 5 transformation methods

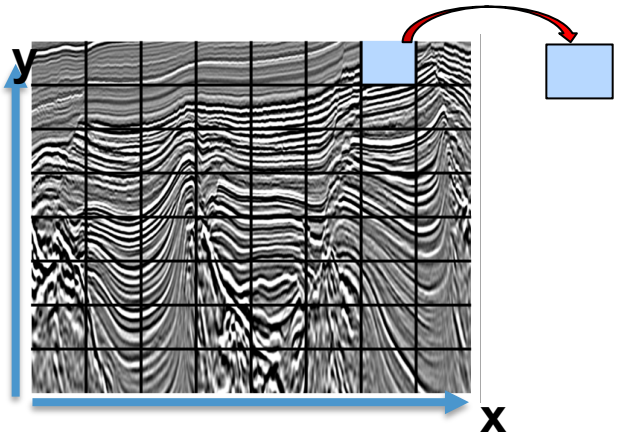


- Step-by-step multilevel 2D wavelet transform illustration



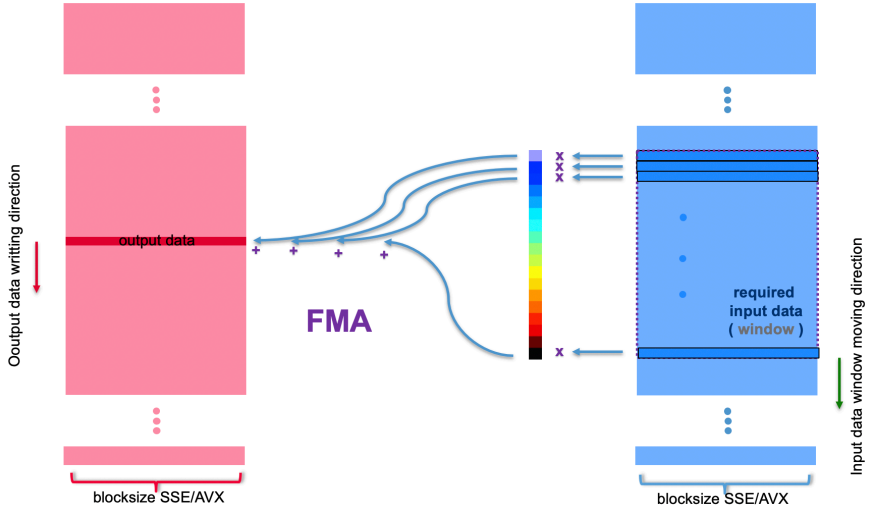
WAVELET TRANSFORM OPTIMIZATION

- Improve data locality



2D or 3D block to fit in cache memory

- Accelerate compute kernel with SIMD (CPU illustration)



CPU (Intel E5-2680v3 2.5GHz): 400MB/s/core --- GPU (NVIDIA GV100) 115GB/s

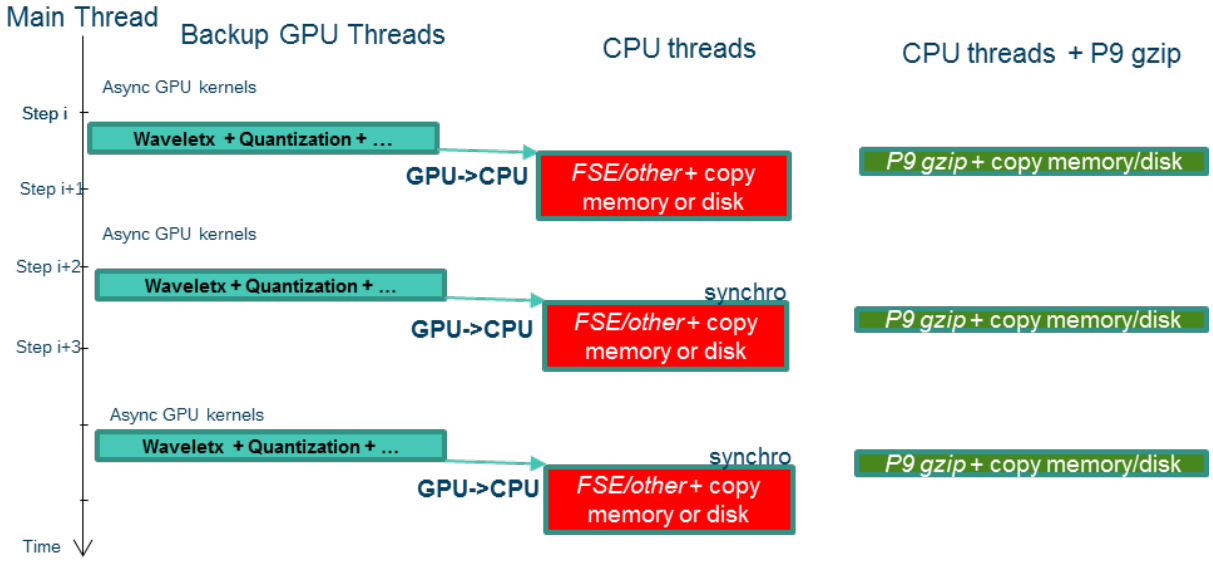
SOLUTION SCHEME

Collaboration between Total IBM, Nvidia and Altimesh

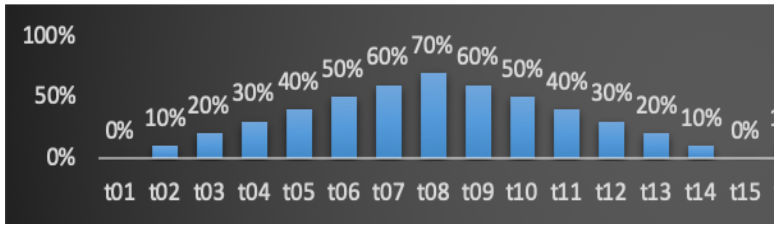
GPU and CPU compression with async read/write for backup/restore overlapping

- Execution time reduction
- Compression rate augmentation
- External library with multiple options : lost rate, memory, transfer, algo, etc…
- Intensive research

- **Performance**
  - **~ 50GB/s/GPU** (Volta GV100)
  - seismic : x30-40 compression ratio



SOLUTION SCHEME BONUS



- **Storage on disk if not enough space in memory**
- **Reversible:**
  - CPU compress can be GPU decompress
  - GPU compress can be CPU decompress

P9 NX on-chip gzip accelerator (2 per system)

- Highest throughput on-chip gzip/gunzip engine in the industry
  - 7.7GB/s compress 15.6GB/s decompress peak
    (perf. number is for a single Deflate stream; not multiple engine aggregates)
  - 80 to 125x speedup over a CPU single thread
- Standard user-mode interfaces
- Option for higher compression ratio with dynamic Deflate block type support

2. GPU AND CPU MEMORY MANAGEMENT

2. GPU AND CPU MEMORY MANAGEMENT

CPU-GPU data allocation and transfers

- zero copy, duplication, memory allocators, async transfers, GPU Direct, topology, pinned memory…

Memory Allocators and management

- OpenACC/OpenMP Data Directives (GPU data allocation, update and transfer)
- CUDA allocator
- CUDA Fortran Managed or/and CUDA Managed Memory
- AC922 UVMA (map, malloc, posix-memalign, shem) including share segment

GPU and CPU data compression (backup), with disk extension

Memory locking and process binding for multi-instances executions

EXEMPLE OF GPU MEMORY ALLOCATIONS AND MANAGEMENT

OpenACC

```
!! field declararation
    real,allocatable,dimension(:,:,:) :: sxx

!! Allocate Fields.
!! ----------------
allocate(sxx(NX,NY,NZ))
!$ACC enter data copyin(sxx)

!! Initialize Fields.
!$ACC kernel present(sxx)
field%sxx=0.

call OpenACC_kernel/CUDA_kernels
If CPU_kernels need !$ACC update host(sxx)

!$ECC exit data delete(sxx)
```

Managed (**CUDA Fortran**)

```
!! field declaration
    real,allocatable,dimension(:,:,:), MANAGED :: sxx

!! Allocate Fields.
!! ----------------
allocate(sxx(NX,NY,NZ))

!! Initialize Fields.
field%sxx=0.      ! PGI compiler  initializes on GPU

Call OpenACC_kernels/CUDA_kernels or call
CPU_kernels
```

MANAGED PERFORMANCE TESTS – IBM POWER8/NVIDIA P100

managed

```
module routines
contains
    attributes(global) subroutine stream(a, b, N)
        real(kind=8), managed  :: a(1:N), b(1:N)
        integer(kind=4), value :: N
        integer(kind=4) :: i

        do i=1 + threadIdx%x + blockIdx%x * blockDim%x, N, blockDim%x * gridDim%x
            a(i) = a(i) + b(i)
        end do
    end subroutine
end module

program main
    use routines
    integer(kind=4) :: N
    real(kind=8), allocatable, managed, dimension(:) :: a, b

    N = 1024*1024*8*8
    allocate(a(1:N))
    allocate(b(1:N))

    print *, "N = ", N
    call stream <<< N / 128, 128 >>> (a, b, N)
    call stream <<< N / 128, 128 >>> (a, b, N)

    cudaError = cudaDeviceSynchronize()
    cudaError = cudaDeviceReset()
end program main
```

CUDA/Fortran

```
module routines
contains
        attributes(global) subroutine stream(a, b, N)
                real(kind=8), device  :: a(1:N), b(1:N)
                integer(kind=4), value :: N
                integer(kind=4) :: i

                do i=1 + threadIdx%x + blockIdx%x * blockDim%x, N, blockDim%x * gridDim%x
                        a(i) = a(i) + b(i)
                end do
        end subroutine
end module

program main
        use routines
        integer(kind=4) :: N
        real(kind=8), allocatable, device, dimension(:) :: a,b

        N = 1024*1024*8*8
        allocate(a(1:N))
        allocate(b(1:N))

        print *, "N = ", N
        call stream <<< N / 128, 128 >>> (a, b, N)
        call stream <<< N / 128, 128 >>> (a, b, N)

        cudaError = cudaDeviceSynchronize()
        cudaError = cudaDeviceReset()
end program main
```

```
 N =      67108864
==57943== Profiling application: ./stream_managed
==57943== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  911.99ms         2  455.99ms  2.6179ms  909.37ms  routines_stream_
```

```
 N =      67108864
==57980== Profiling application: ./stream_dev
==57980== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  5.2374ms         2  2.6187ms  2.6170ms  2.6204ms  routines_stream_
```

On the first call, the data is transferred. The second call is as fast as without managed

3. GPU PORTING AND KERNEL OPTIMIZATION

PROGRAMMATION CHOICES

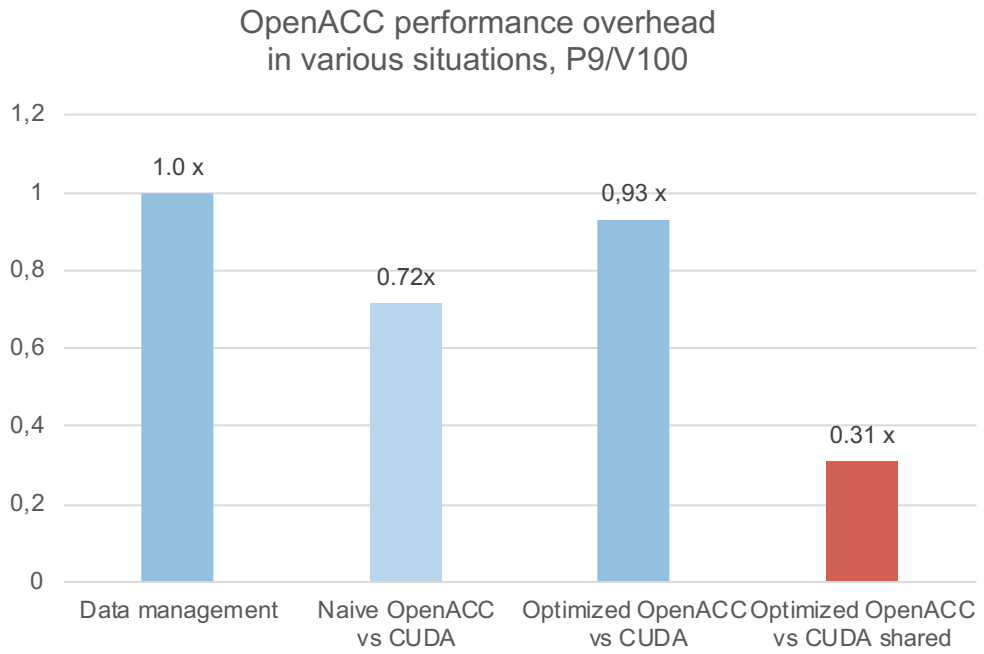
- Programming models
  - OpenACC directives
  - Cuda Fortran and C APIs

- OpenACC main advantages
  - « present » checking
  - Size transfert in Fortran
  - Error checking

- CUDA main advantages
  - Shared memory
  - Low level API

- Data transfert technics
  - OpenACC manual call to « update »
  - CUDA APIs
  - CUDA Fortran « Managed » on declaration

PERFORMANCE



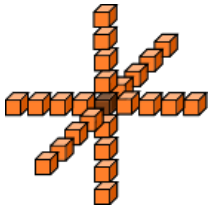
Naive propagator OpenACC GPU porting

```
!$ACC PARALLEL PRESENT(wave_fields,other)
!$ACC LOOP GANG VECTOR COLLAPSE(3)
   do k=zmin,zmax
     do j=ymin,ymax
       do i=xmin,xmax
                ….
```

Not all kernels can take benefit of shared memory!

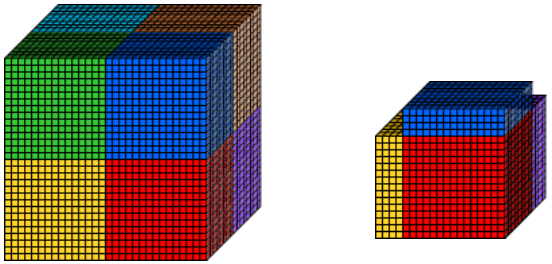
4. NEW PARALLELISM AND CONCURRENT SCHEMES

STARTING POINT



LS-RTM and FWI seismic propagator numerical method: 3D Finite Difference discretization on regular grids

Parallel MPI approach: Domain decomposition with halo (➔ Stencil). One sub-domain per MPI task.



For timestep = 1 : n_times
(1) **Computation (local propagation)**
(2) **MPI halo communication**
Save forward snapshot

Target:

- seismic 3D propagator on GPU
- GPU-GPU MPI halo communication
- overlap MPI communications

GPU technics:

- **OpenACC directives for compute and data**
- **CUDA streams for asynchronism**
- **MPI Non-blocking coms and « GPU Direct »**

BASIC CODE PRINCIPLE

(1) Simple propagator GPU porting using OpenACC directives

```
!$ACC PARALLEL PRESENT(wave_fields,other) ASYNC(steamS)
!$ACC LOOP GANG VECTOR COLLAPSE(3)  independent
   do k=zmin,zmax
     do j=ymin,ymax
       do i=xmin,xmax
            ….
```

Attention: critical to use CUDA streams

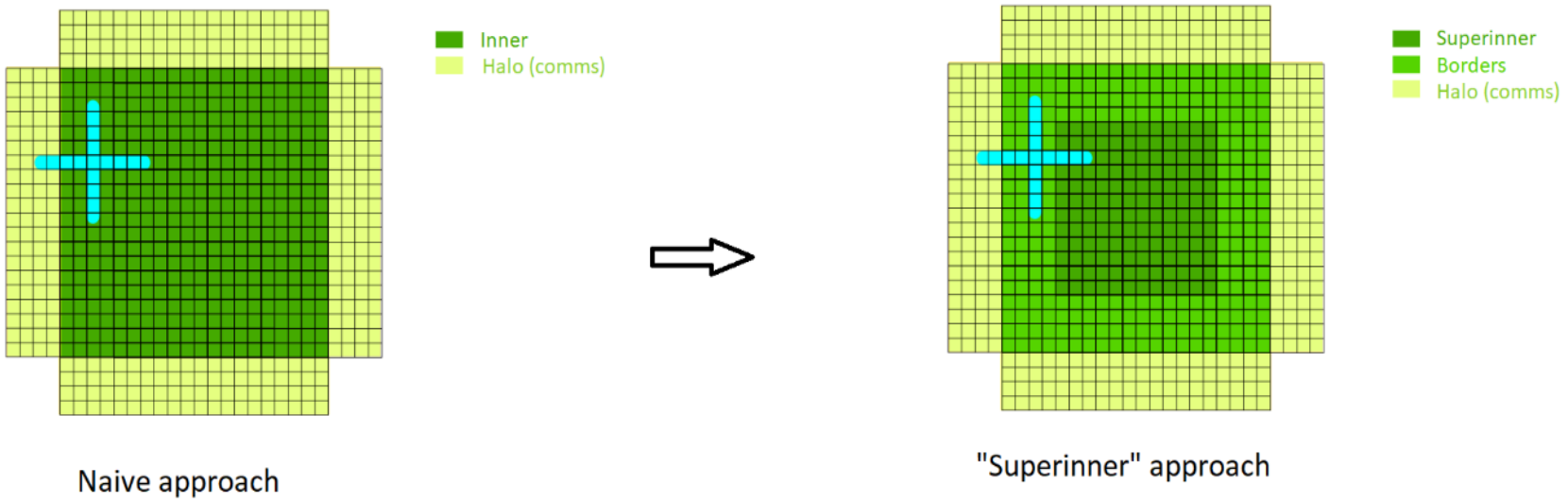
(2) Simple MPI halo coms on GPU, GPU Direct

```
 copy halo on GPU (S buffer), in specific CUDA streams
!$acc wait(steamBj)
!$acc update use_device(S)
  call mpi_isend(S…)
 !$acc update use_device(R)
  call mpi_isend(R…)
copy R in wave fields halo on GPU, in specify CUDA streams
!$acc wait(steamBj)
```

OVERLAPPING COMMUNICATIONS BY COMPUTATIONS 1/2

Local sub-domain new decomposition

- « SuperInner » strategy
  - Split computational part
  - Use priorities on streams



OVERLAPPING COMMUNICATIONS BY COMPUTATIONS 2/2

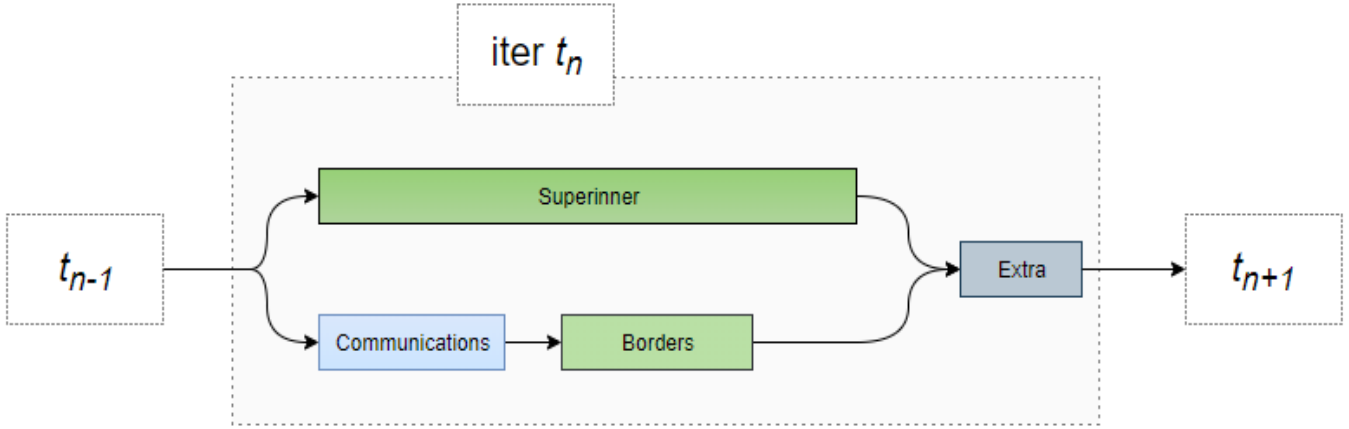
For timestep = 1 : n_times
- **MPI halo communication**
- **Computation local propagation**

Save forward snapshot

⇒

For timestep = 1 : n_times
- **Computation local propagation superinner, stream1**
- **MPI halo communication, stream2**
- **Computation local propagation borders, stream2**

Save forward snapshot

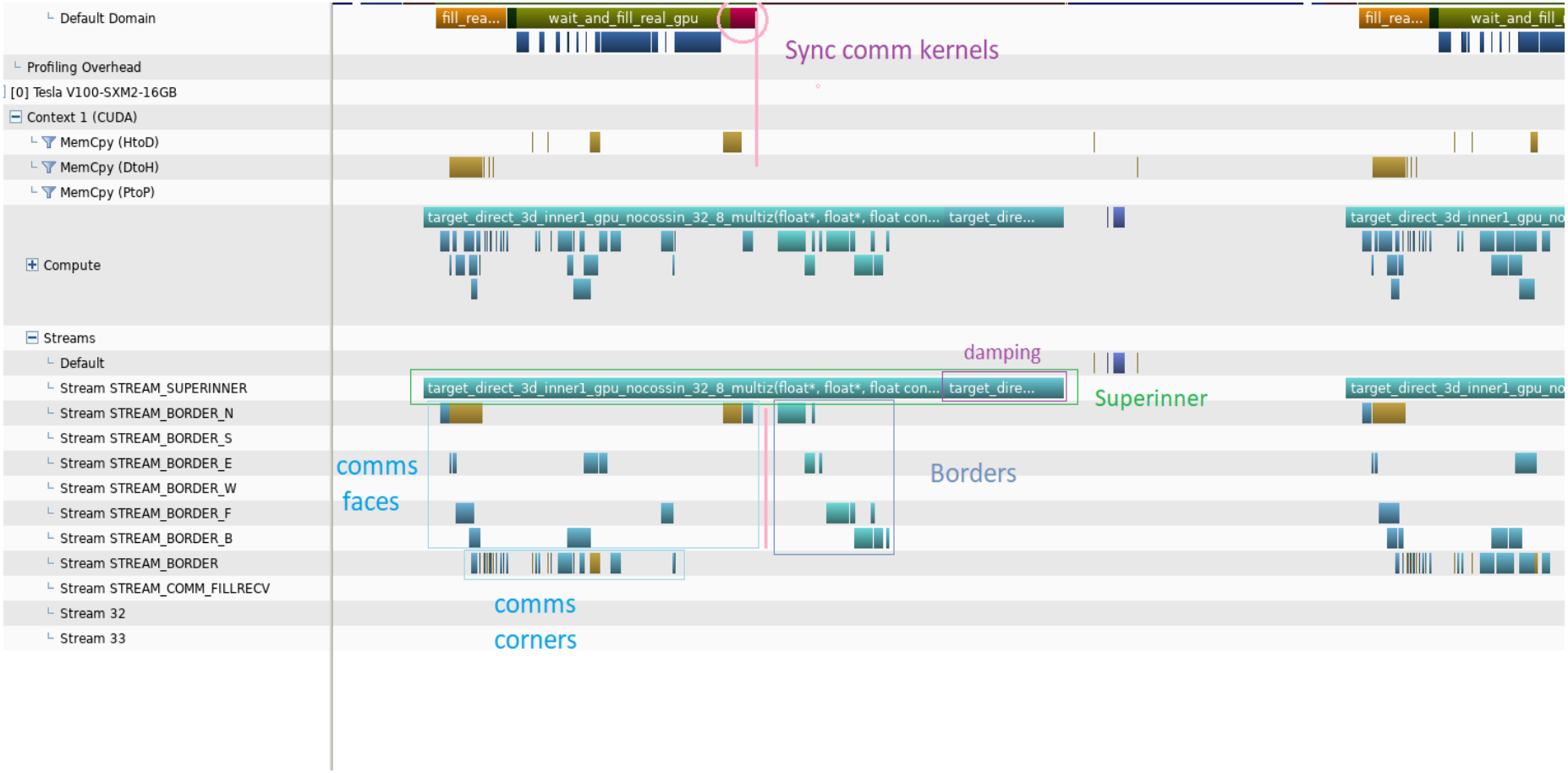


*Total_time ≈ tmax * (* ***time_Kernel + time_coms***
+ *time_snapshot/step_snapshot + time_extra)*

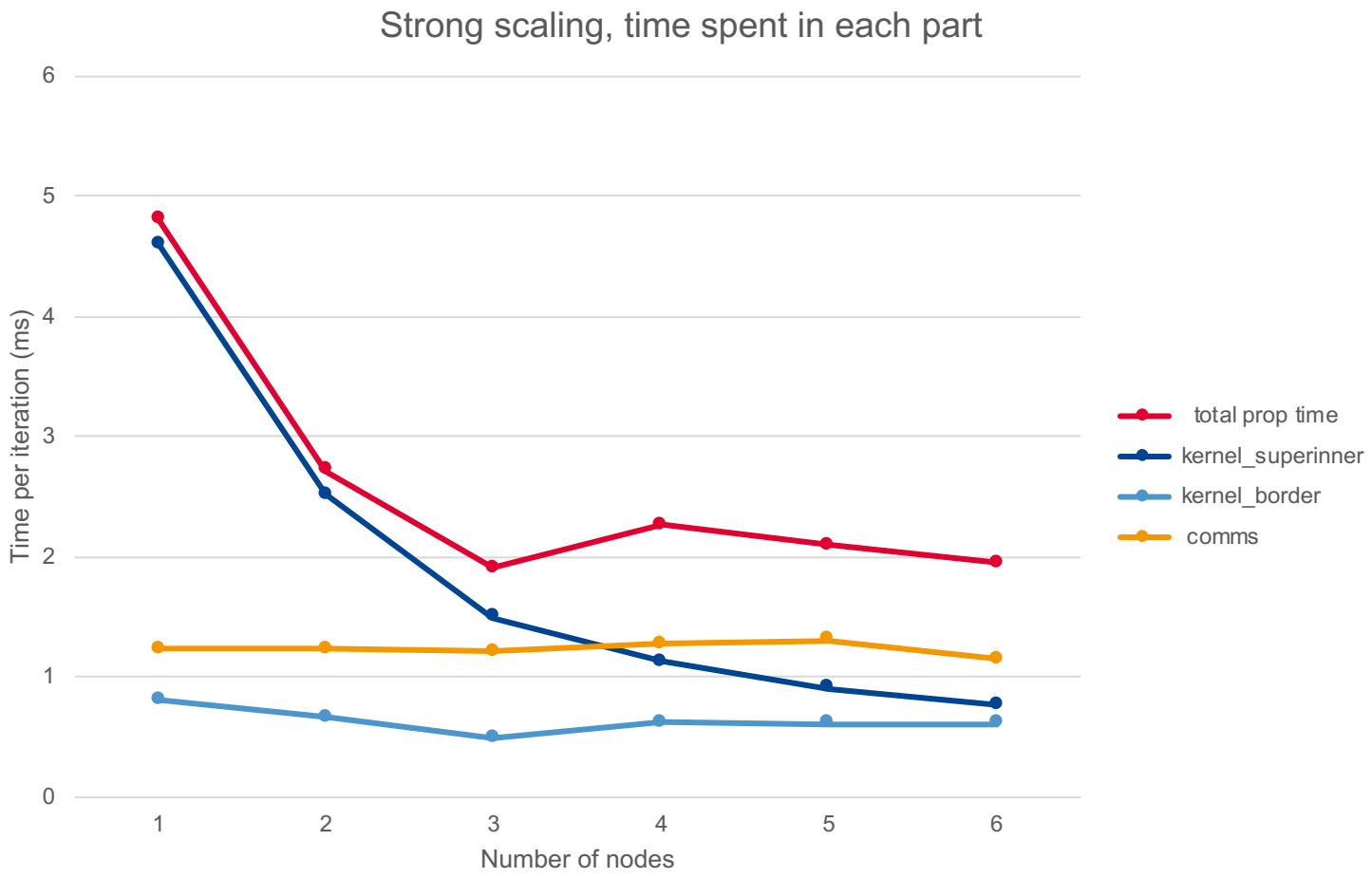
⇒

*Total_time ≈ tmax * (* ***time_internal_Kernel***
+ *time_snapshot/step_snapshot + time_extra)*

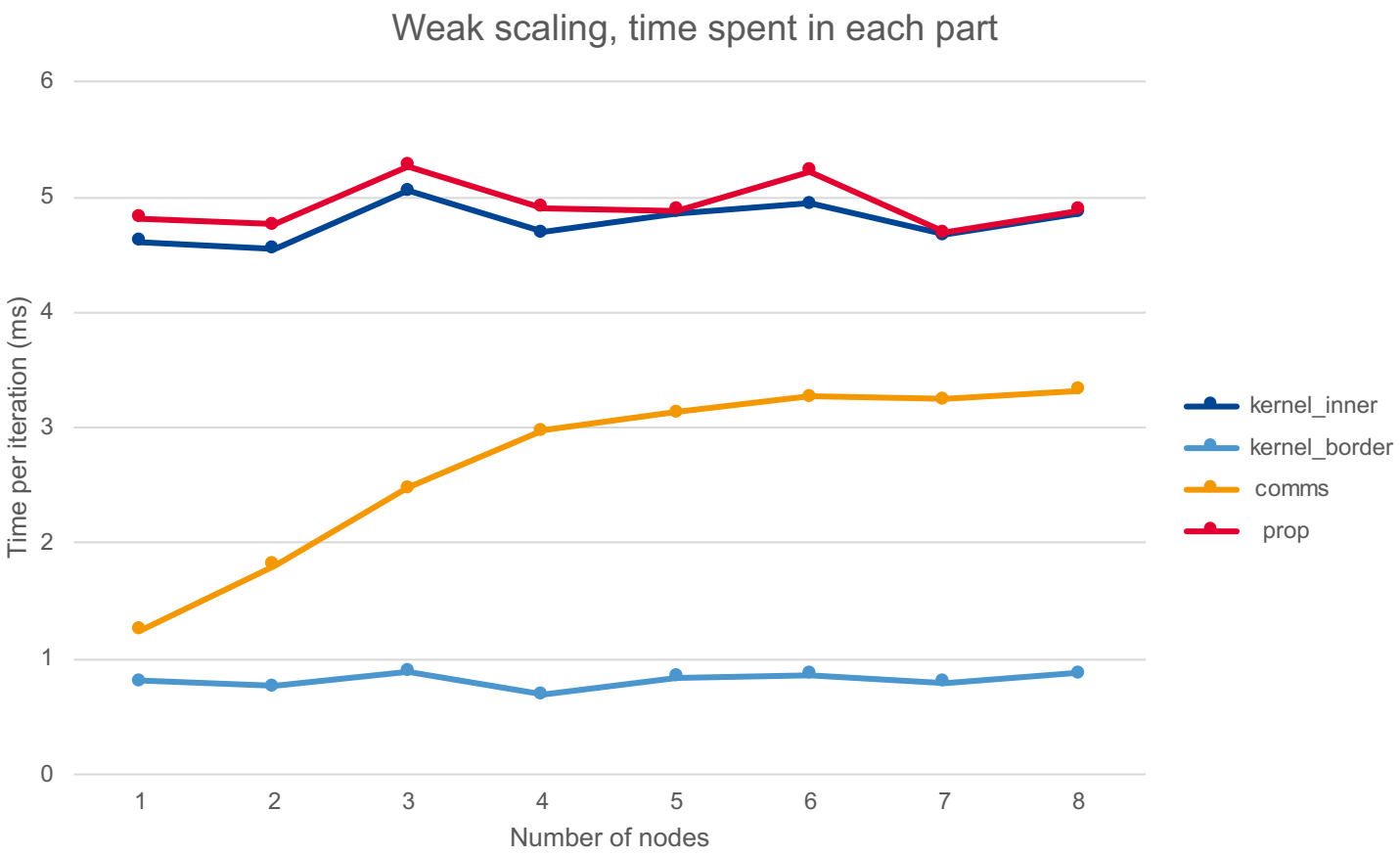
TRACE ILLUSTRATION



SCALABILITY 1/2

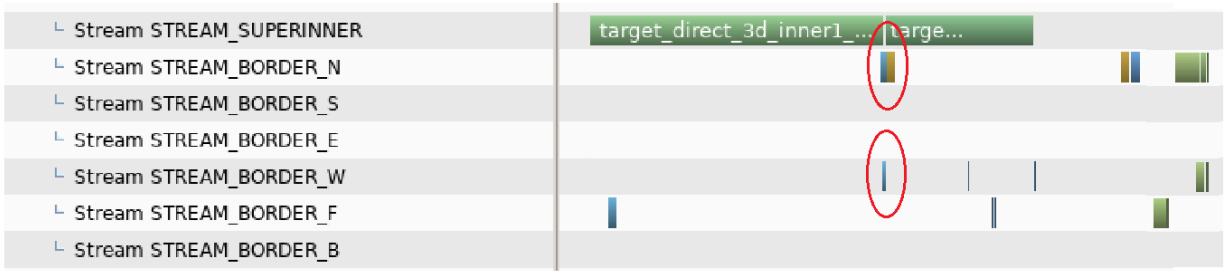


SCALABILITY 2/2

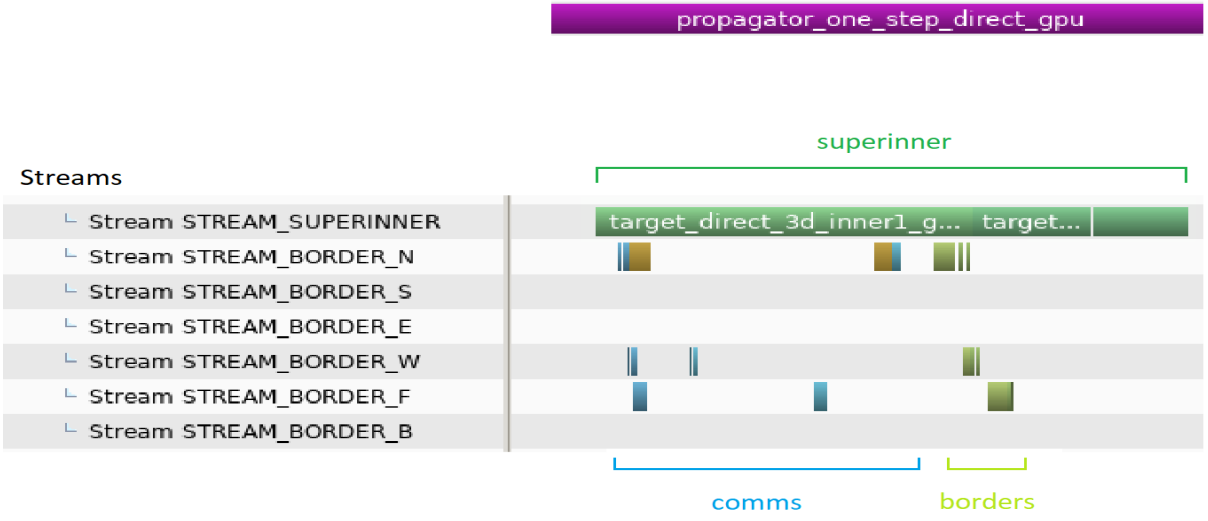


CRITICAL TO INTRODUCE CUDA STREAM PRIORITIES

Cuda streams all with default priority (high) ➔ poor overlapping



Cuda stream for superinner kernel with low priority



HOW TO CREATE CUDA&OPENACC STREAMS WITH PRIORITIES

Fortran example to create cuda streams with different priorities & OpenACC association

```
! declaration
    integer ::stream_acc1, stream_acc2
    integer(kind=cuda_stream_kind)  :: stream_cuda1, stream_cuda2
    Integer :: low_priority, high_prioriy
! Get  Nvidia V100 priorities
    call cudaDeviceGetStreamPriorityRange(LOW_PRIORITY,      HIGH_PRIORITY)
! Create stream with priority
    call cudaStreamCreateWithPriority(stream_cuda1,0,low_priority)
    call cudaStreamCreateWithPriority(stream_cuda2,0,high_priority)
! Associated CUDA and Open_ACC stream
    call acc_set_cuda_stream(stream_acc1, stream_cuda1)
    call acc_set_cuda_stream(stream_acc2, stream_cuda2)


! Now any OpenACC or CUDA Kernels in a stream can synchronized with
    call cudastreamsynchronize(stream_cuda)
    !$acc wait(stream_acc)
```

GPU INTER AND INTRA-NODE COMMUNICATION OPTIMIZATION

GPU Direct not necessary the best based on message size, GPU topology, distance and location, MPI library…

- GPU-GPU MPI communication options
  - GPU-GPU with GPU Direct
  - GPU->CPU, CPU-CPU with MPI, CPU->GPU
    - On MPI CPU buffers, use PINNED memory (no pageable)

- How to allocate host pinned memory
  - CUDA Fortran: `REAL, ALLOCATABLE, DIMENSION(:,:), PINNED :: bufferhost`
  - CudaMallocHost API:

```
!$acc host_data use_device(a)
    cudamemcpyasync(bufferhost(1,jstart),a(1,jstart),n, cudaMemcpyDeviceToHost,st)
    cudamemcpyasync(bufferhost(1,jend),a(1,jend),n, cudaMemcpyDeviceToHost,st1)
    cudastreamsynchronize(st)
    cudastreamsynchronize(st1)

     CALL MPI_Sendrecv(bufferhost(1,jstart), n, MPI_REAL, top   , 0, bufferhost(1,jend+1), …
     CALL MPI_Sendrecv(bufferhost(1,jend), n, MPI_REAL, bottom, 0, bufferhost(1,(jstart-1)), …

    cudamemcpyasync(a(1,jstart-1),bufferhost(1,jstart-1),n, cudaMemcpyHostToDevice,st)
    cudamemcpyasync(a(1,jend+1),bufferhost(1,jend+1),n, cudaMemcpyHostToDevice,st1)
    cudastreamsynchronize(st)
    cudastreamsynchronize(st1)
!$acc end host_data
```

5. PRODUCTIVITY AND TCO OPTIMIZATION

RESOURCE OPTIMIZATION WITH MULTI-INSTANCES EXECUTIONS

Objective : increase # shots per node and per second
➔ best productivity and TCO

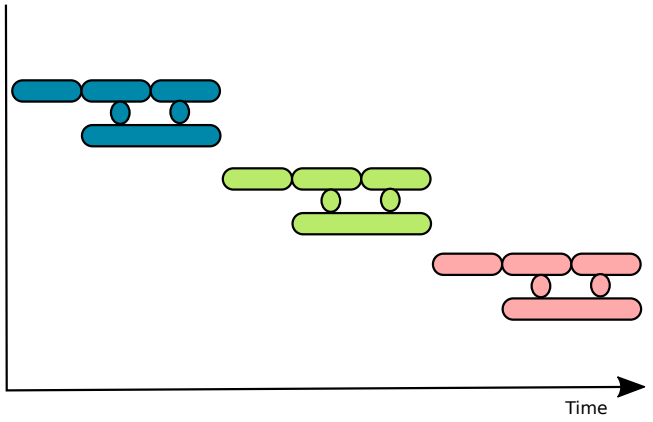
- Hardware resource utilization rate and sustained performance: CPUs, GPUs, memory, network and IO/FS.
- Execution sizing and configuration depending on CPU and GPU resource requirements

Runtime options:

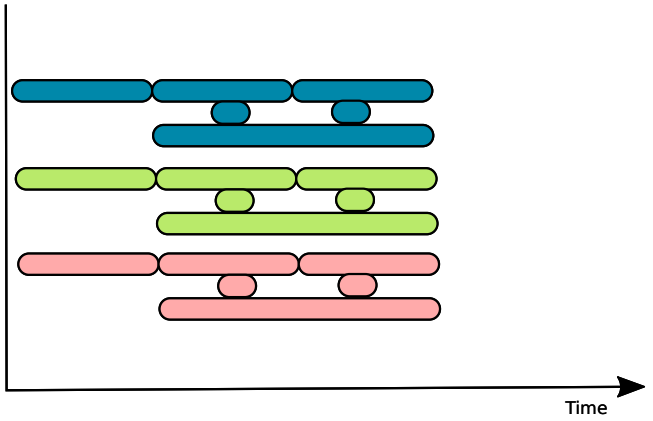
- Increase number of MPIs per GPU, number of CPU threads…
- Concurrent executions per node and per GPU (up to xx% production improvement)
  - Intelligent and adaptive resource sizing and allocations
  - Intelligent process binding depending on domain partitioning and data compression rate
  - Memory locking approach versus backup disk options

INCREASE OCCUPANCY 1/2

- Memory is the limit:
  - Default (doing nothing): Serialization

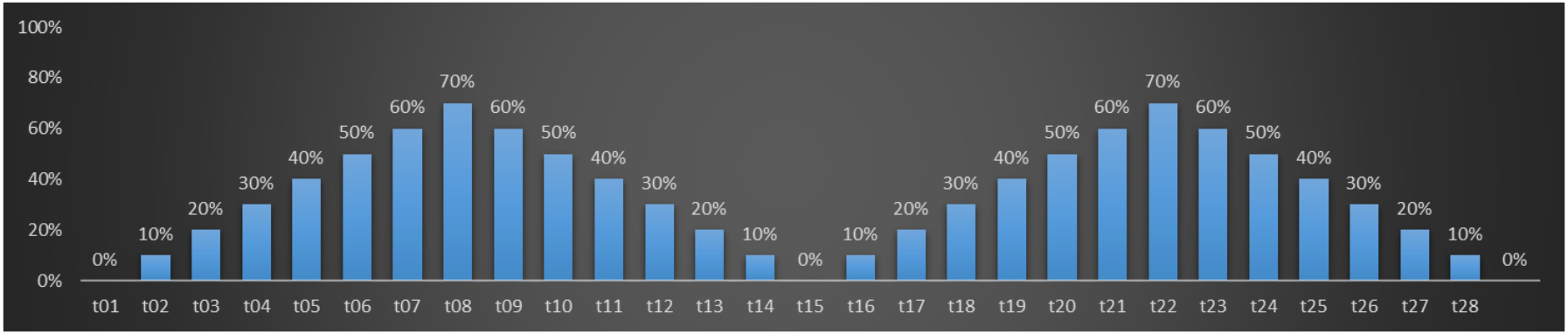


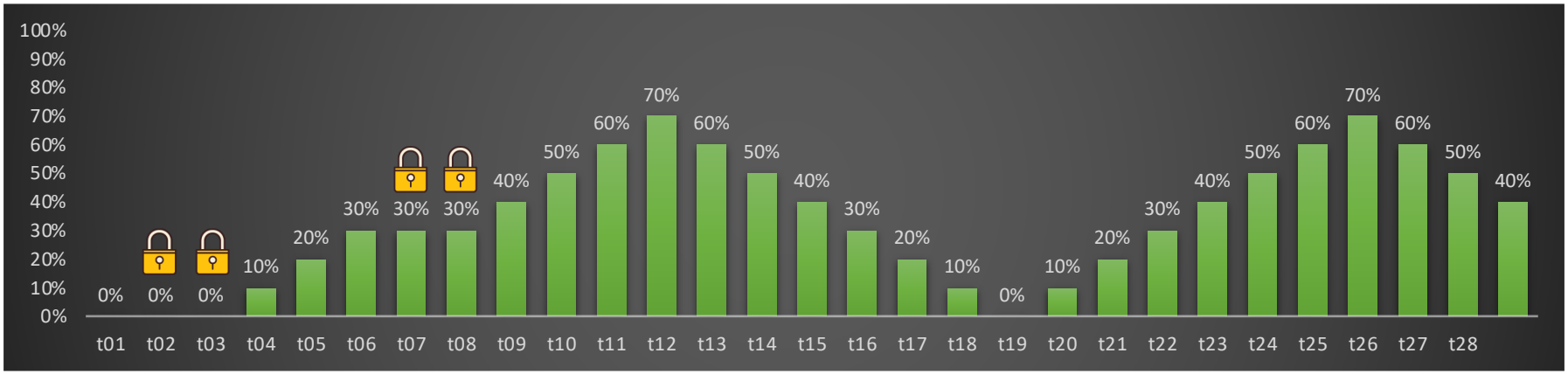
  - Possible to enable NVIDIA MPS



INCREASE OCCUPANCY 2/2

Large shots strategy: memory locking across timesteps





CONCLUSION

FWI/LS-RTM GPU PORTING SUMMARY

CPU-GPU-GPU data allocation and transfers

- zero copy, duplication, memory allocators, async transfers, GPU Direct, topology, pinned memory…

Memory Allocators and management

- OpenACC Data Directives (GPU data allocation, update and transfer)
- local CUDA allocator
- Local CUDA Fortran statements

GPU and CPU data compression for snapshots, with disk extension

Asynchronism and concurrent operations

- GPU vs GPU kernels with CUDA Streams
- GPU kernels vs data transfers
- GPU kernels vs MPI communications
- GPU kernels vs CPU kernels
- CPU kernels vs CPU kernels

Development of CUDA versions for intensive kernels using SM shared memory(vs OpenACC)

Production Optimization: ***enbaling multi-instances executions***

NUMBERS FROM CPU TO GPU

- CPU machine (Pangea2):
  Intel Xeon E5-2680v3 12C 2.5GHz
  Infiniband FDR

- GPU machine (Pangea3):
  IBM Power9 18C 3.45GHz
  Dual-rail Mellanox EDR Infiniband
  NVIDIA Volta GV100

- GPU vs CPU machines
  - ~10x less node numbers
  - 4x more memory per node
  - ~3x less power consumption

  - Seismic RTM/FWI
    - ~8x (machine scale)
    - ~50x (node scale)

PERSPECTIVES