



Direct Numerical Simulations of Fluid Flow on Roadrunner

Jamaludin Mohd-Yusof
CCS-2 Roadrunner Applications Team
with
Nehal Desai, Tim Kelley, Daniel Livescu



Overview

- **Previous accelerator experience**
- **DNS basics**
- **Original code overview**
- **Serial CBE version**
- **Modified original code**
- **Roadrunner version**
- **Observations**
- **Conclusions**



Advanced Architectures project

- **Demonstrated the potential of accelerator architectures**
- **GPGPU (General Purpose computing on Graphics Processing Units) at LANL**
 - Scout language implementation
 - extension of visualization language to scientific computing
 - 100x speedup of simple compute kernels (dendrite growth, shallow water equations)
 - limited looping prevented implementation of more complex codes at that time
 - single precision
- **Acceleration of complex FEM codes on GPU-enhanced clusters**
 - precision issues
 - data movement
 - work block sizes
 - Amdahls Law, etc.



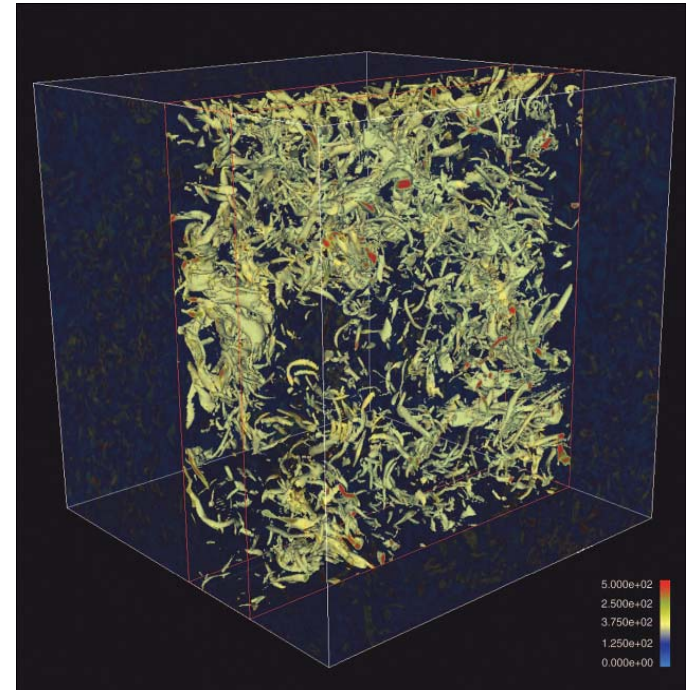
Direct Numerical Simulation

- **Solution of the governing equations *without any modeled terms***
 - “numerical experiments”
 - accurate numerical methods (spectral methods, higher order central or compact finite differences)
 - all length and time scales must be resolved
 - turbulence theory depends on separation of scales
 - large grids, long run times, lots of data storage
- **Provide data for**
 - closure models; needed for coarse ‘engineering’ simulations
 - physical insight; more detailed information than can be provided by experiments
 - validation of e.g. LES simulations
 - scaling laws



CFDNS Code Overview

- **Fortran 90**
- **compressible Navier-Stokes solver**
 - multiple species
 - realistic equation of state
- **2 or 3 dimensional**
- **Cartesian, cylindrical or spherical geometry**
 - logically Cartesian mesh
- **periodic or non-periodic BCs**
- **spectral or compact finite-difference derivatives**
- **scales well up to 8192 processors**
 - more than 96% parallel efficiency





Derivatives are precomputed

For either spectral or compact FD schemes;

$$\frac{\partial u}{\partial t} = -u \frac{\partial u}{\partial x} - v \frac{\partial v}{\partial y} - \frac{1}{\rho} \frac{\partial p}{\partial x} + v \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right)$$

becomes simply;

$$\frac{\partial u}{\partial t} = -u^* u_x - v^* v_y - \frac{1}{\rho} p_x + v(u_{xx} + u_{yy})$$

- u_x, v_y etc are now just arrays
- stencil operations (e.g. for centered finite differences) become element-wise multiply



Two main elements of a timestep

- **Derivative calculation**

- Pade scheme (6th order compact finite differences)

$$a_1 f''_{i-1} + a_2 f''_i + a_3 f''_{i+1} = b_1 f_{i-2} + b_2 f_{i-1} + b_3 f_i + b_4 f_{i+1} + b_5 f_{i+2}$$

- requires tridiagonal solver
- parallel version requires communication, either;
 - data transpose (original solver)
 - ghost cells, intermediate values (distributed solve)

- **Update calculation**

- point-wise, requires no communication
- easily vectorized for SPU



Initial observations and misconceptions

- **Splitting the timestep between Opteron and Cell not an option**
 - too much data movement across the PCI bus
 - concern that the PPU would be a bottleneck

- **Arithmetic intensity is low for both portions**
 - ~1 for tridiagonal solver
 - ~5 for update equations

- **Uncertainty due to prior experience with GPUs**
 - difficulty implementing the tridiagonal solver on Cell
 - strided data loads would be unfavorable to the CBE



Memory layout comparison

- **Original code; Fortran**
 - single large array containing all variables
 - $sr(n,i,j,k)$; $n = 1 \rightarrow$ density etc.
 - array of structures (note Fortran array ordering)
 - not amenable to vectorizing
 - optimized for the update step
 - all variables at a single grid point are contiguous in memory

- **Cell code; C with SPU intrinsics**
 - multiple variable arrays
 - $\rho(i,j,k)$, $u(i,j,k)$, $v(i,j,k)$, etc.
 - contiguous in k direction $(i,j,k) \rightarrow N_z \times N_y \times i + N_z \times j + k$
 - optimized for vector DMA and math operations



Vectorized Tridiagonal Solve

- **Original algorithm**

- loops look like;

```
do i=1,N
  x(i) = (r(i) - a*x(i-1))*b(i)
enddo
```

- **Rewrite algorithm to use data in place**

$$x_{m1} = (r - a[i]*x_{m1})*b[i]$$

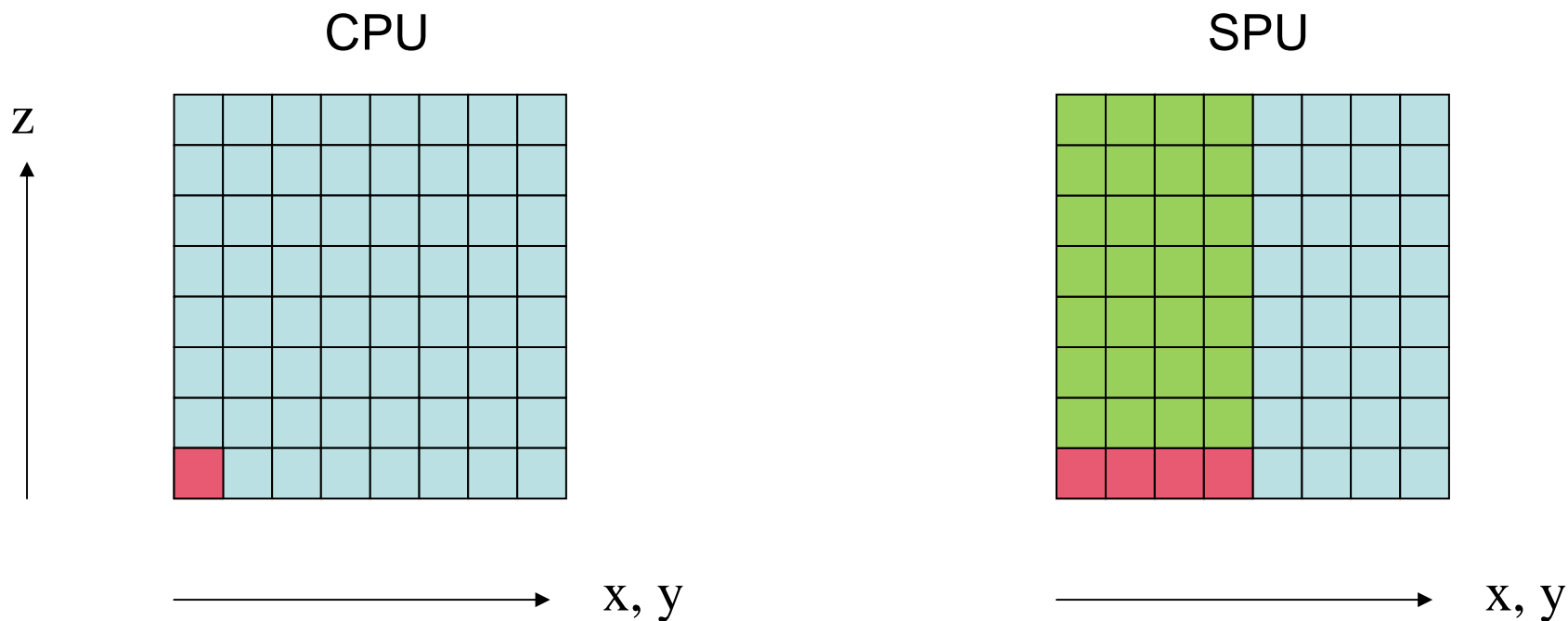
only 2 working data arrays, x_{m1} and r
 $a[i]$, $b[i]$ precomputed, x_{m1} , r , are long vectors

- **Vectorize naturally along z-vectors for x- and y- derivatives (transverse derivative)**
- **Separate vectorized version for z-direction (co-linear derivative)**
 - load multiple columns and form vectors from them



Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)



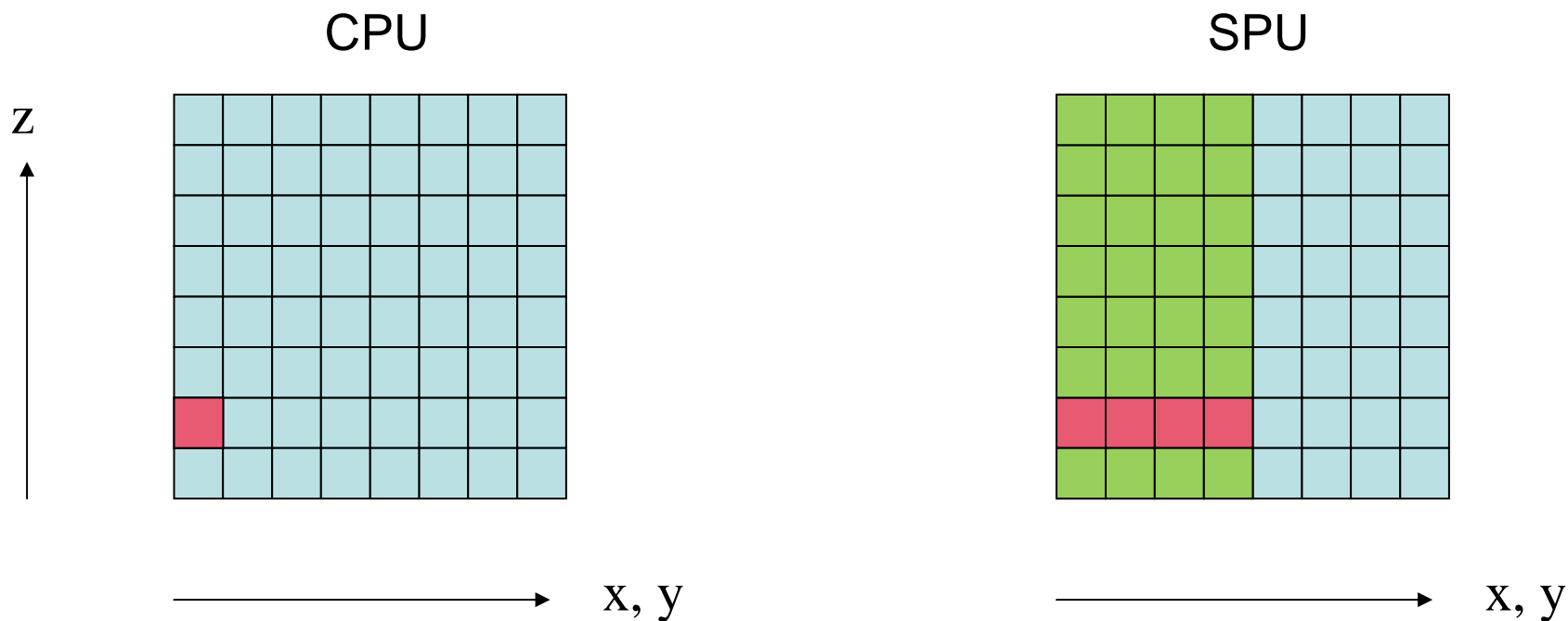
Red is working data

Green (SPU only) shows data loaded into local store



Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)



Red is working data

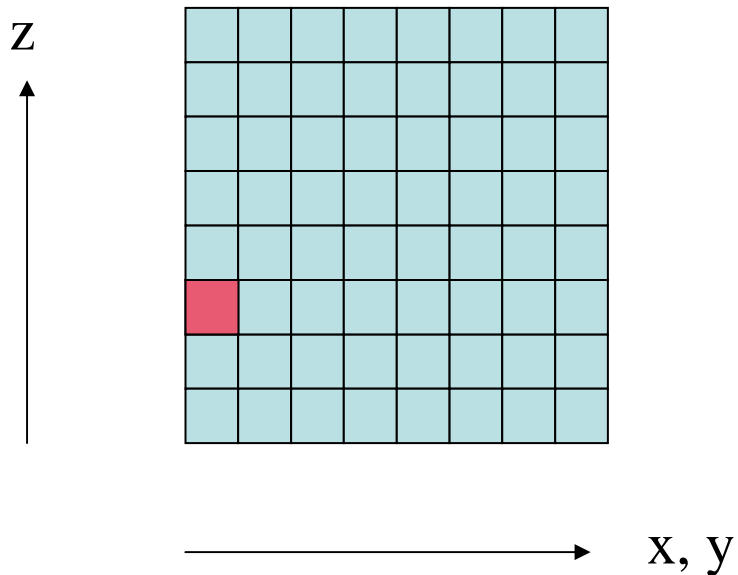
Green (SPU only) shows data loaded into local store



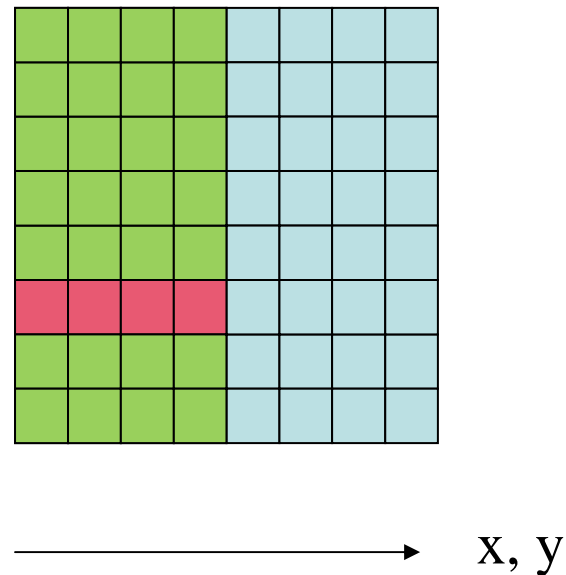
Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)

CPU



SPU



Red is working data

Green (SPU only) shows data loaded into local store

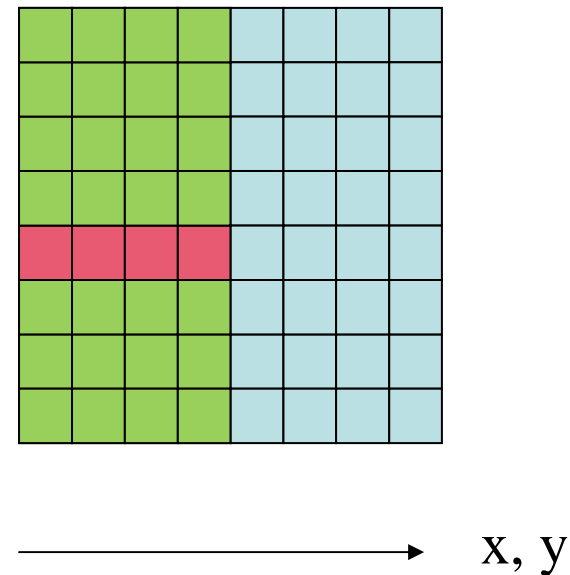
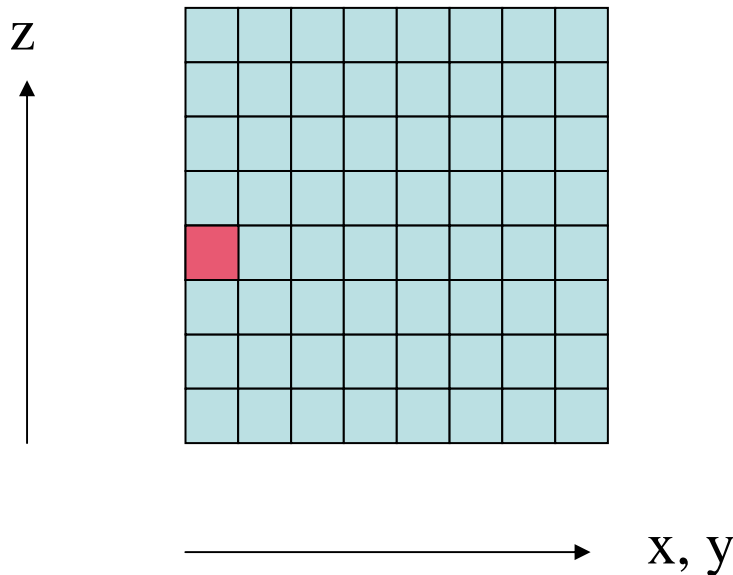


Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

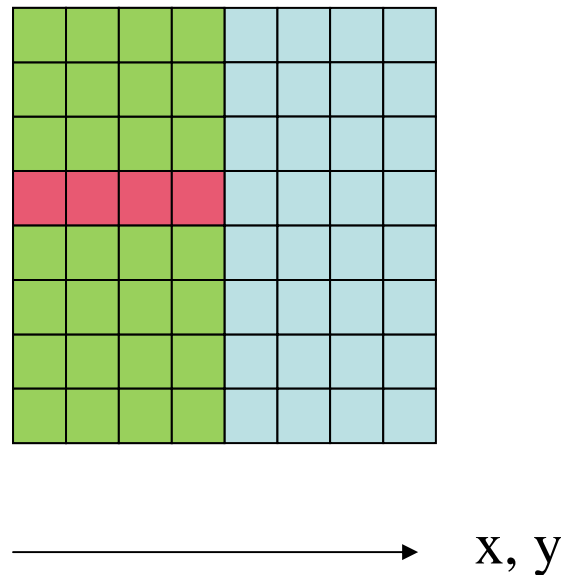
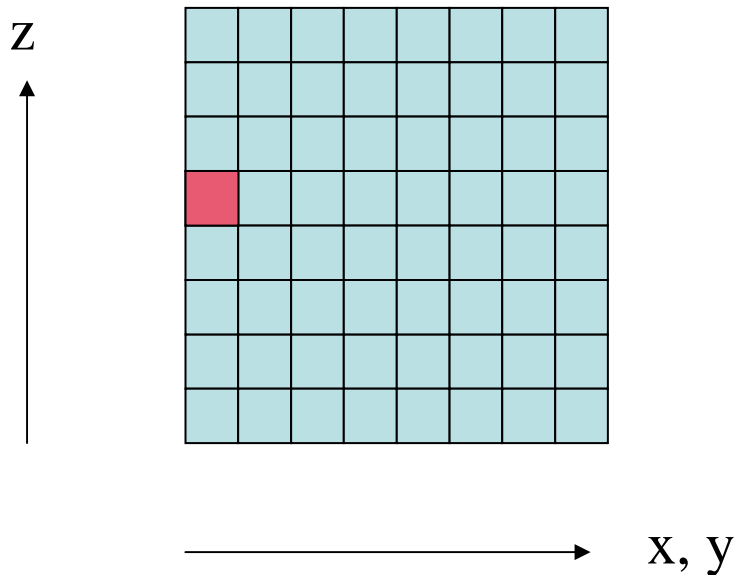


Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)

CPU

SPU



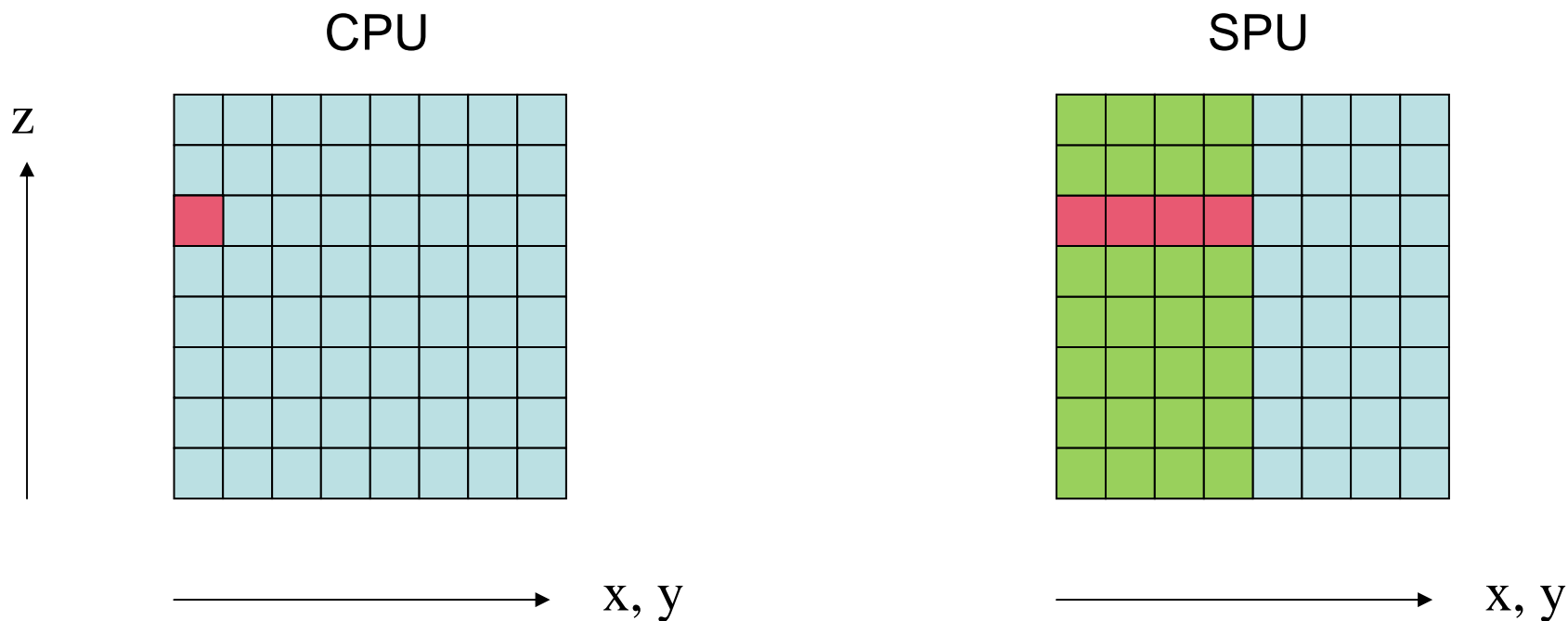
Red is working data

Green (SPU only) shows data loaded into local store



Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)



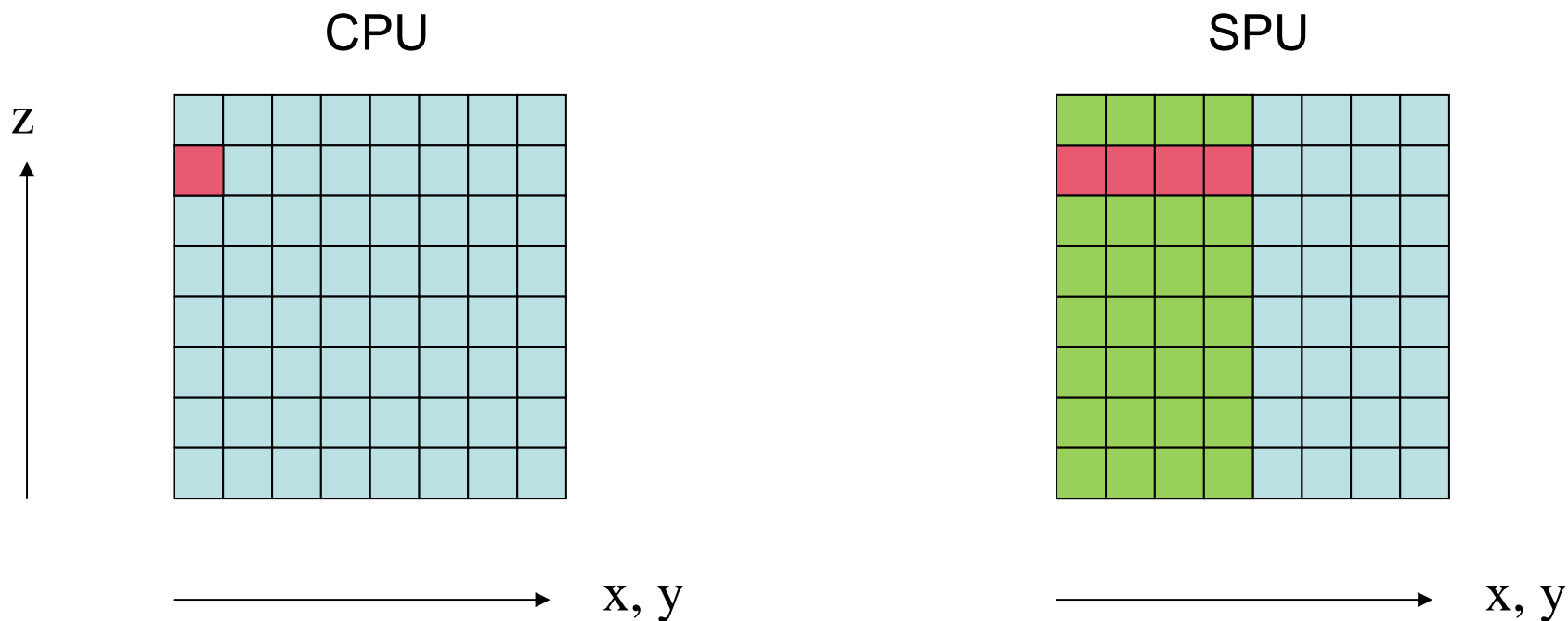
Red is working data

Green (SPU only) shows data loaded into local store



Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)



Red is working data
Green (SPU only) shows data loaded into local store

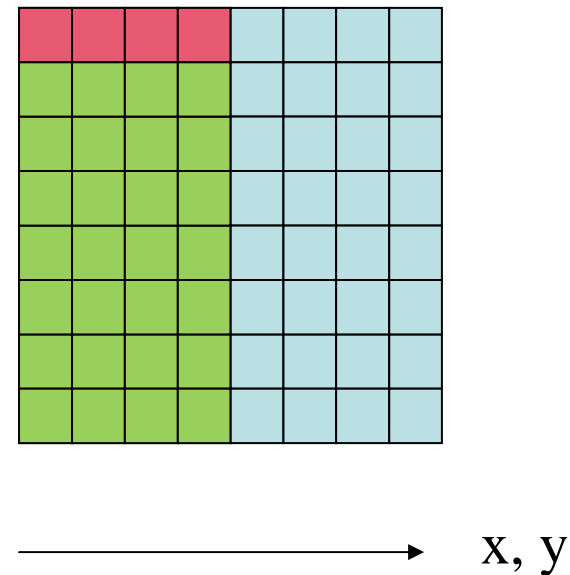
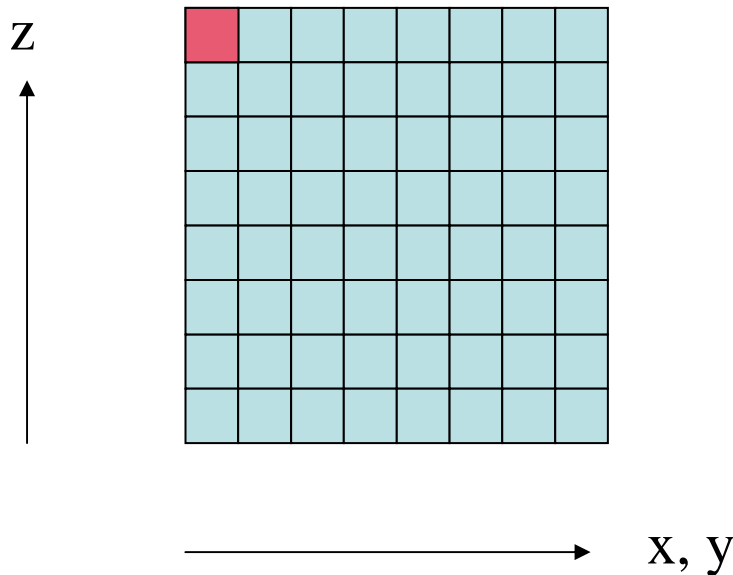


Comparison of serial vs vectorized derivative

Co-linear (contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

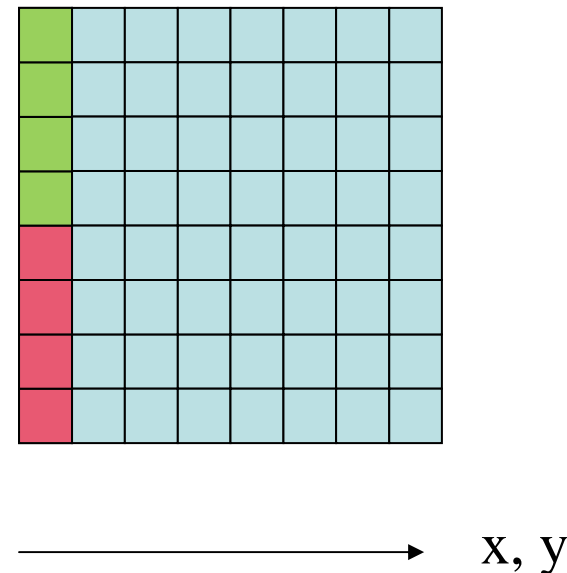
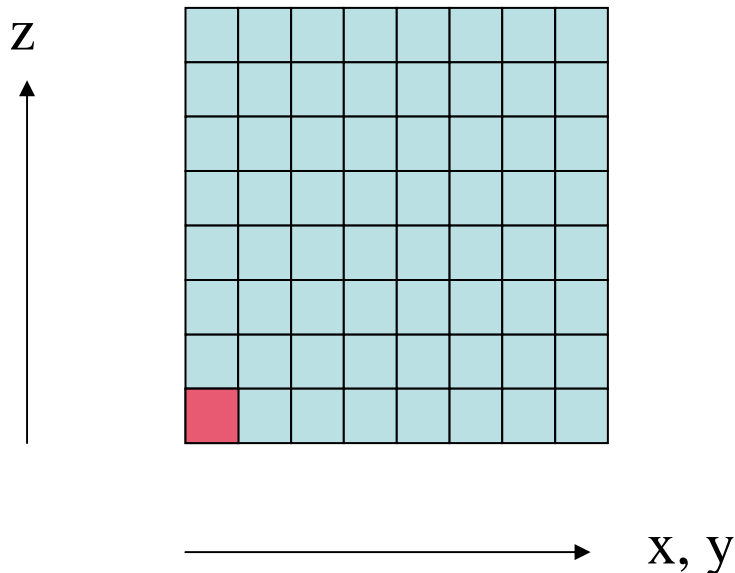


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

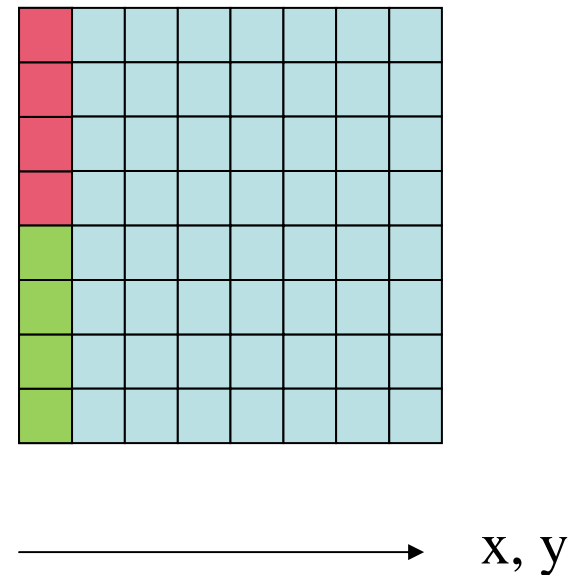
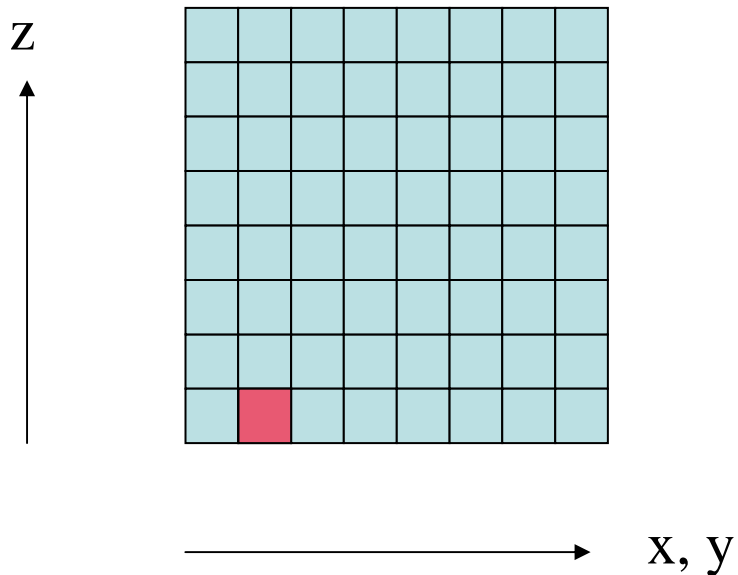


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

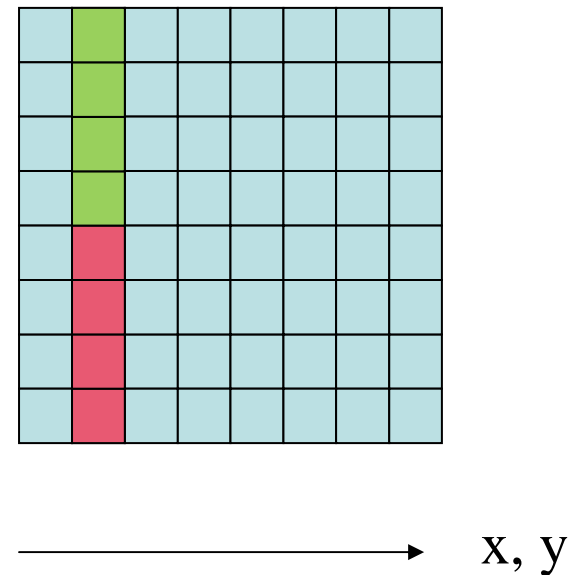
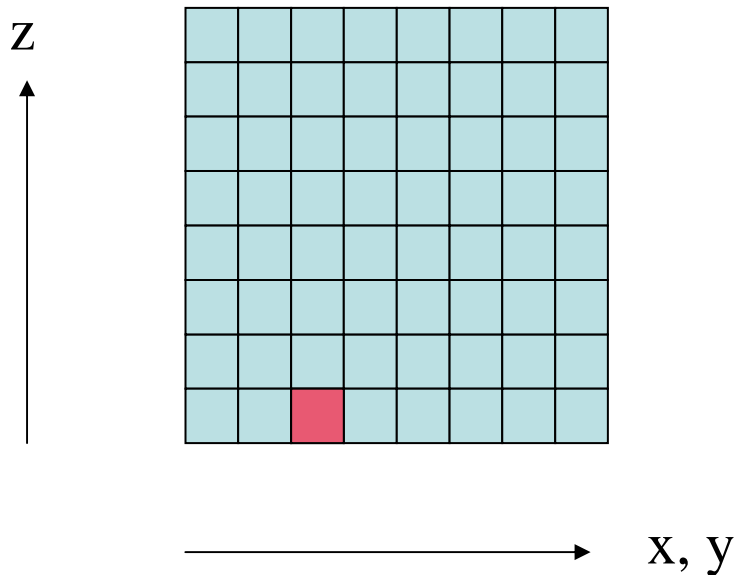


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

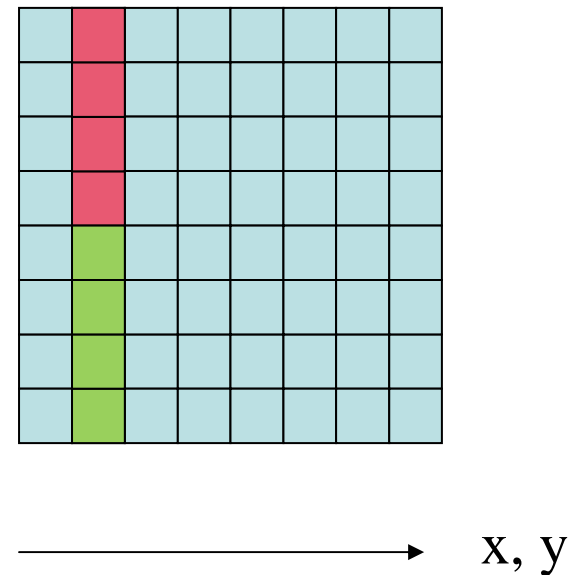
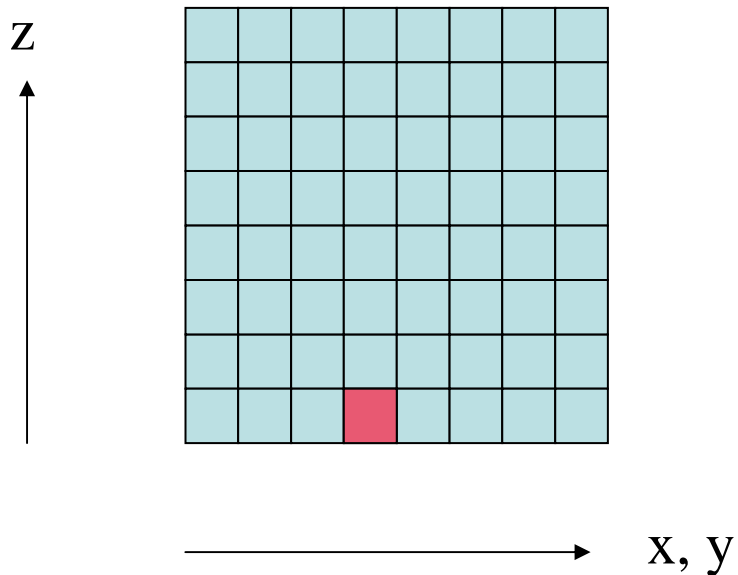


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

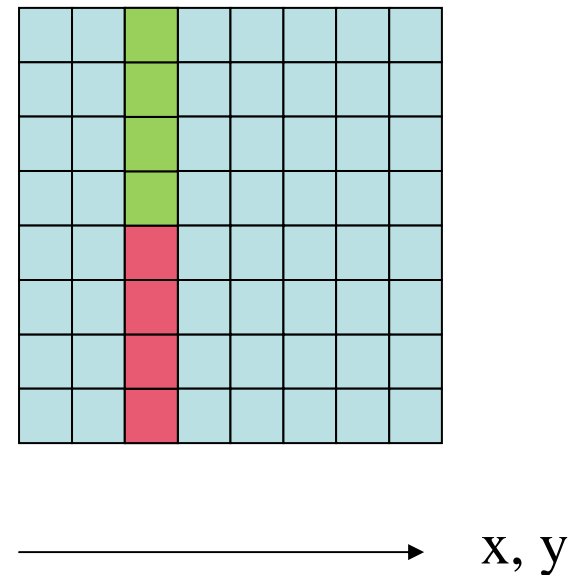
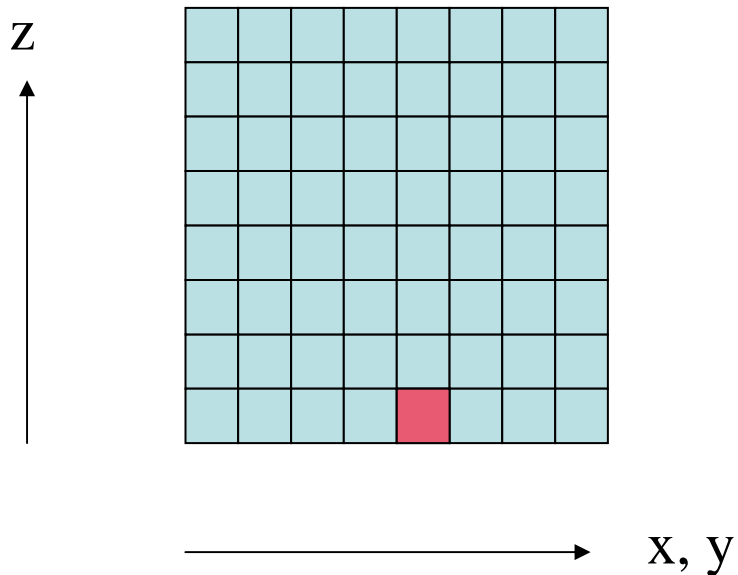


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

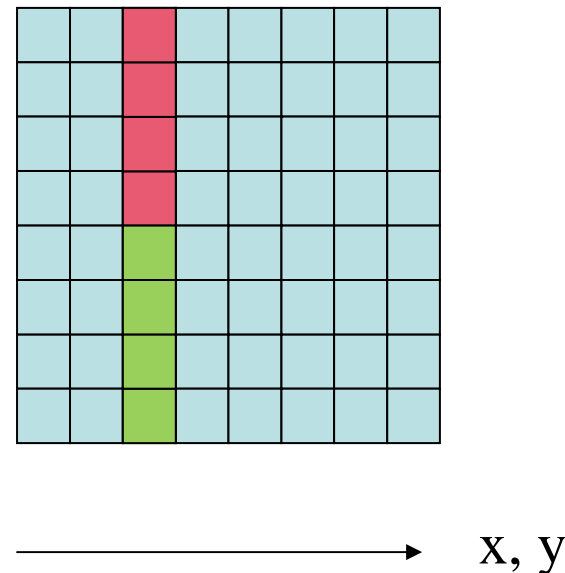
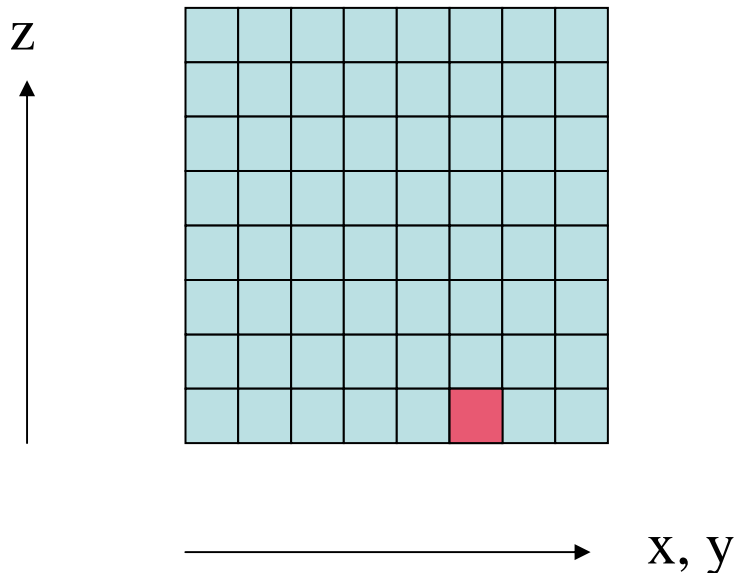


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

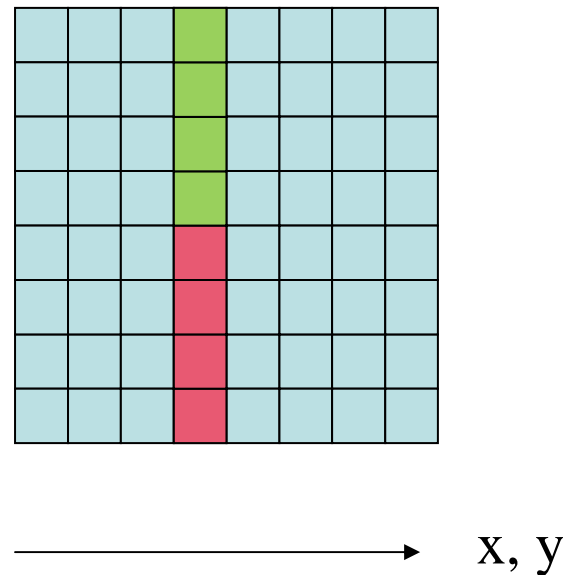
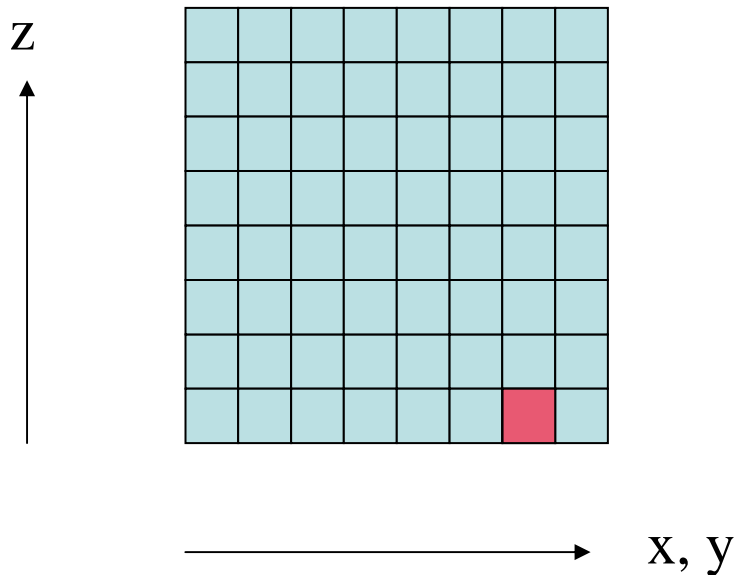


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

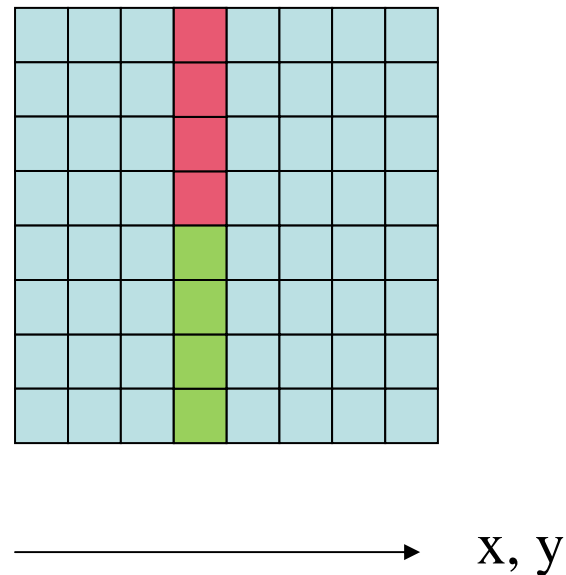
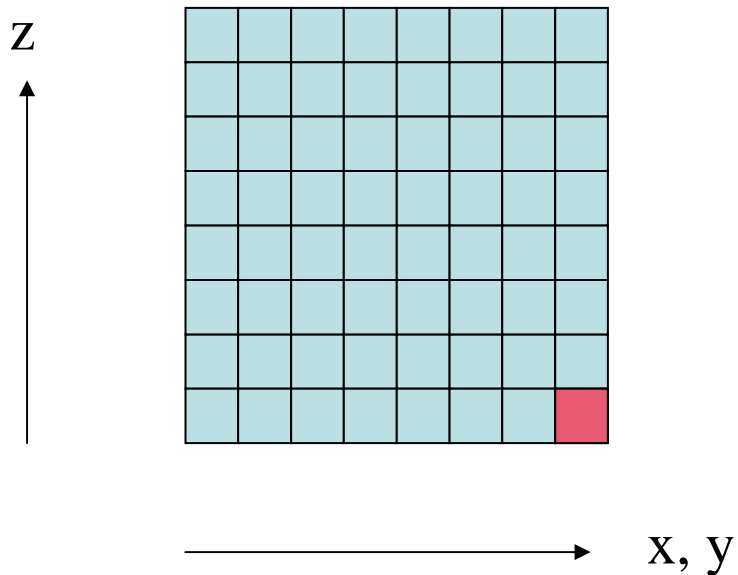


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

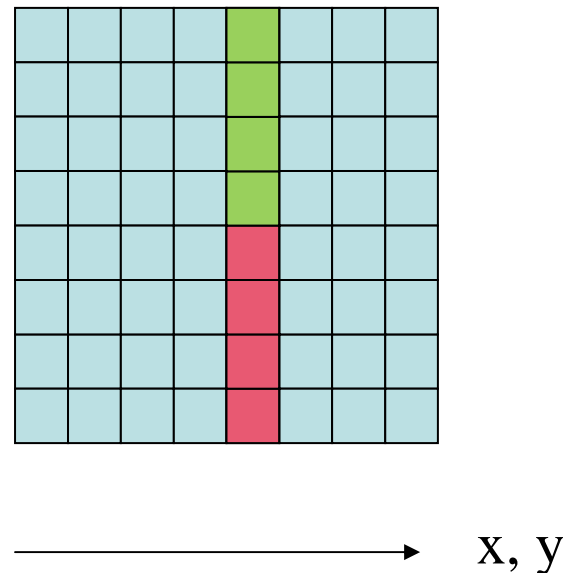
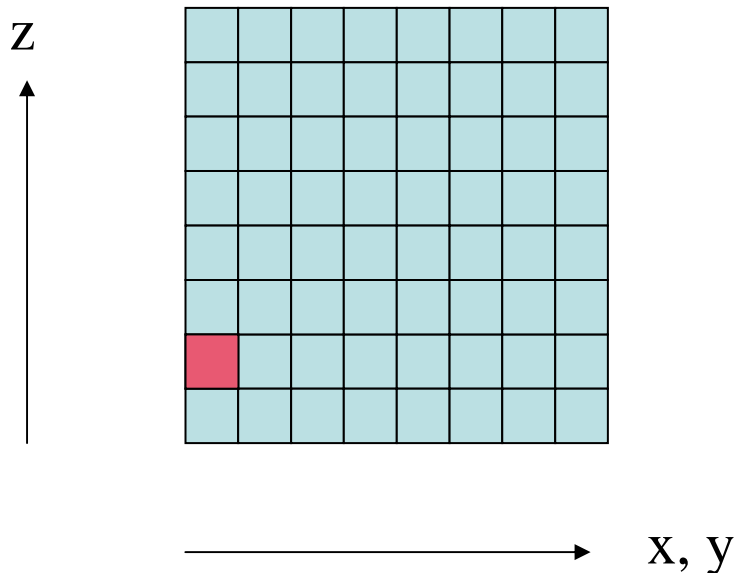


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

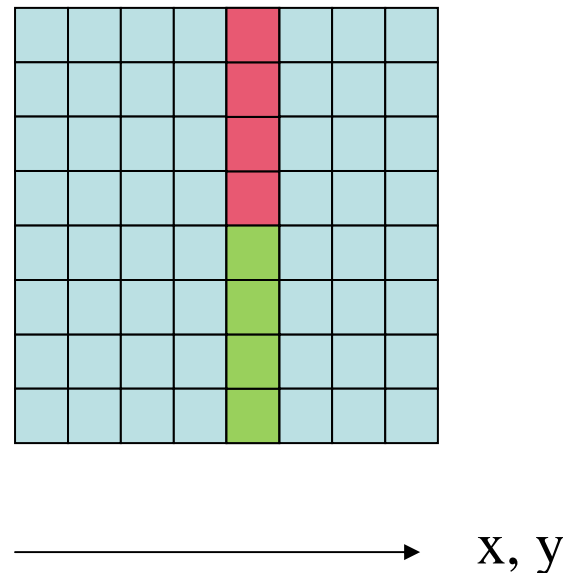
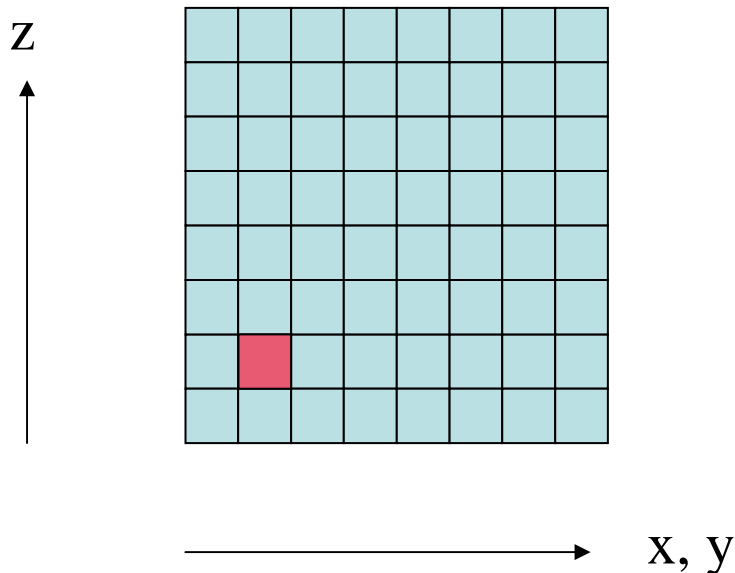


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

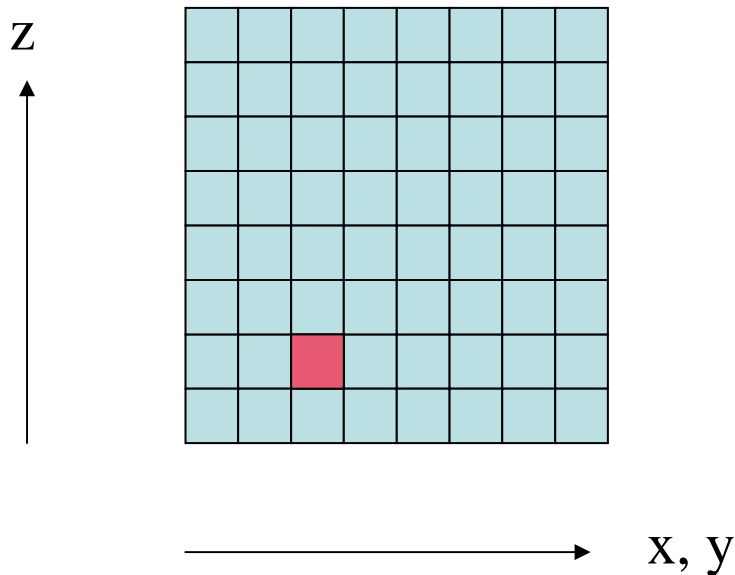
Green (SPU only) shows data loaded into local store



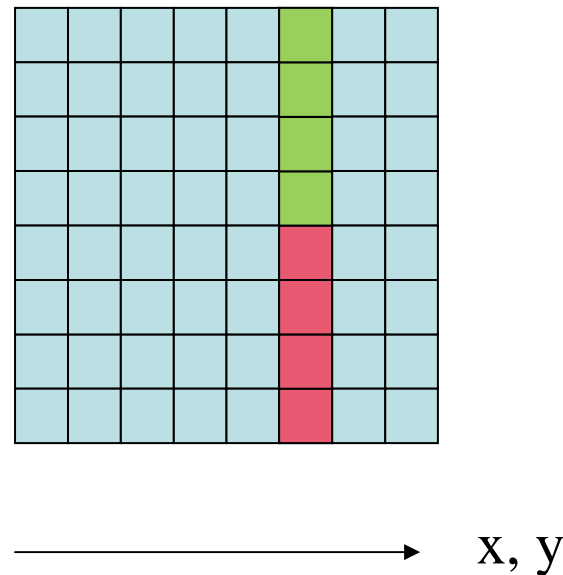
Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU



SPU



Red is working data

Green (SPU only) shows data loaded into local store

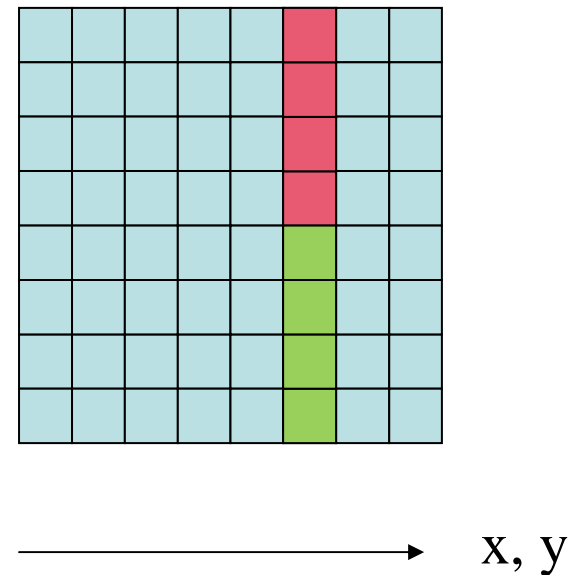
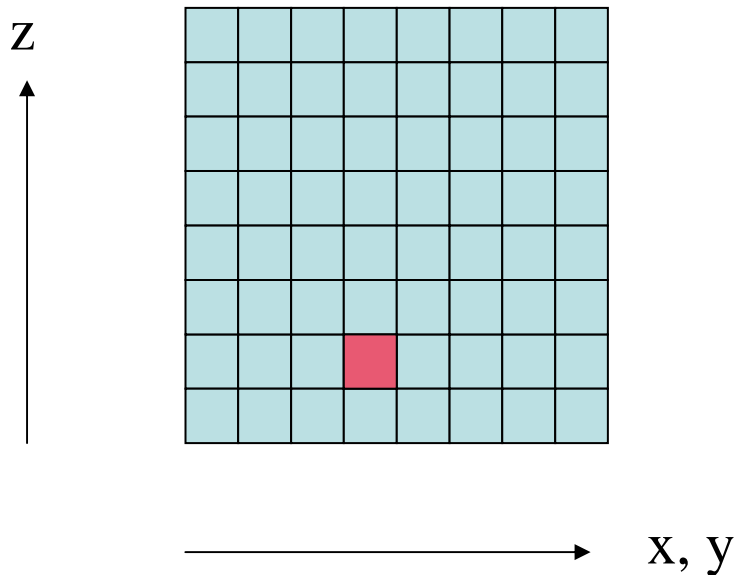


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

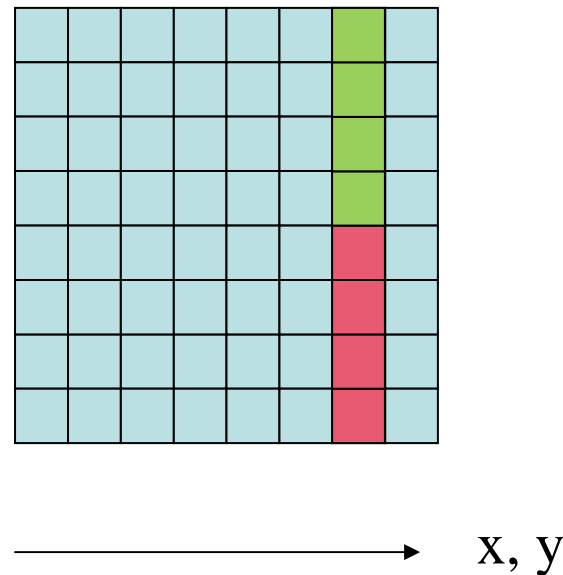
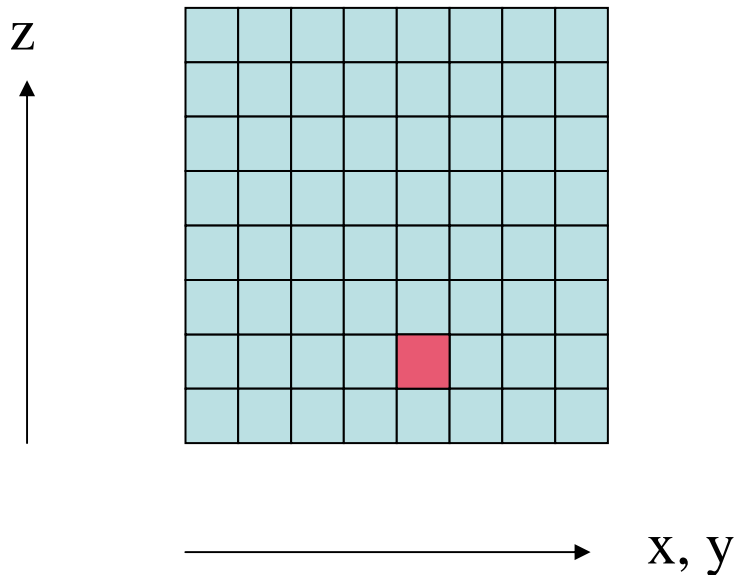


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

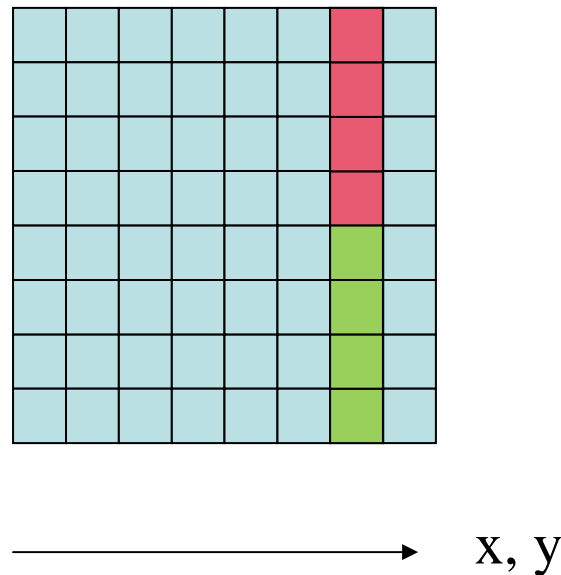
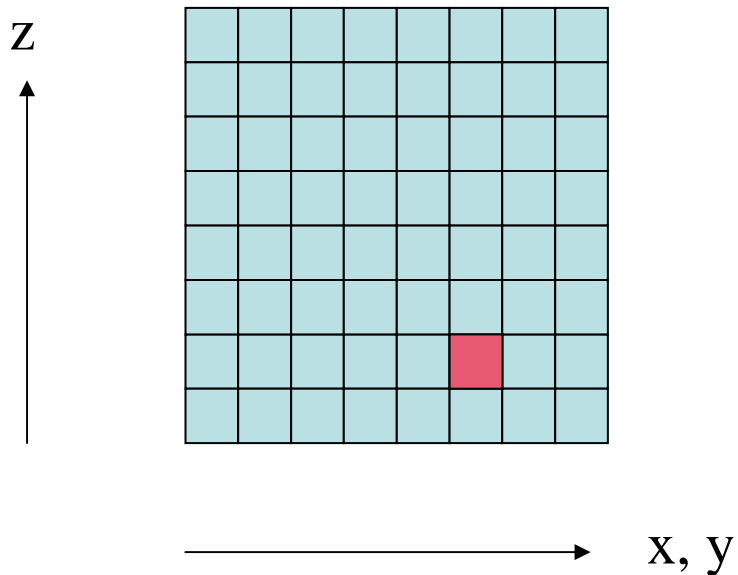


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

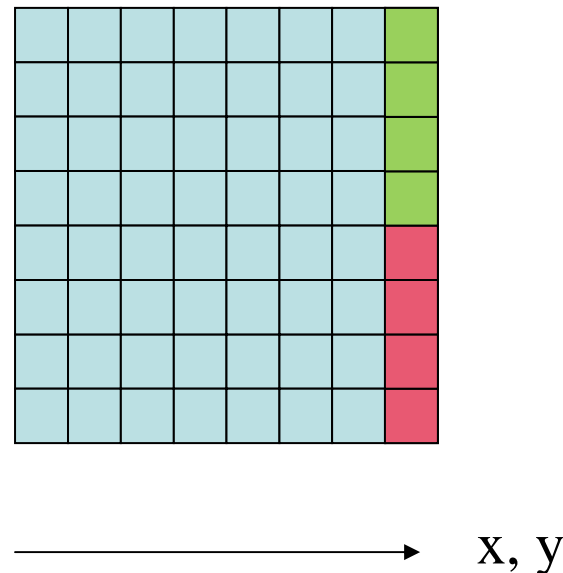
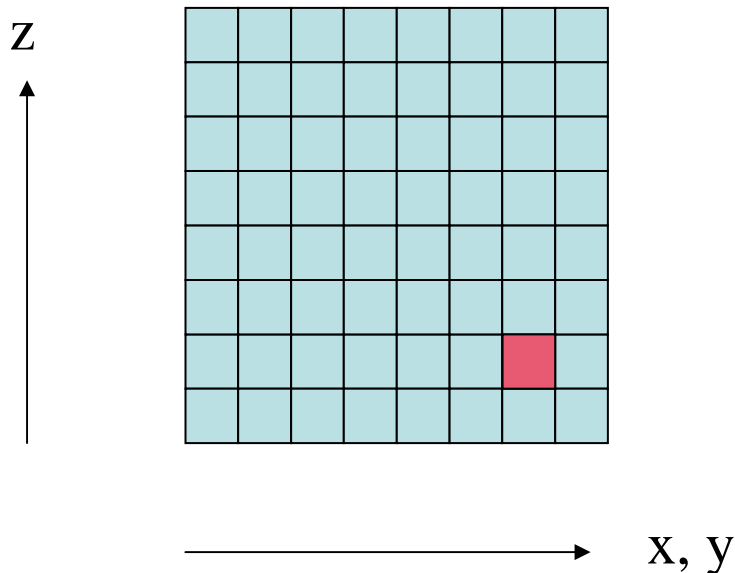


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



Red is working data

Green (SPU only) shows data loaded into local store

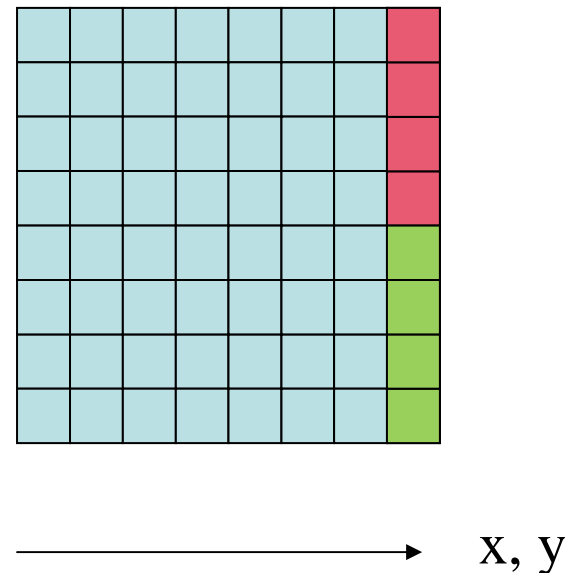
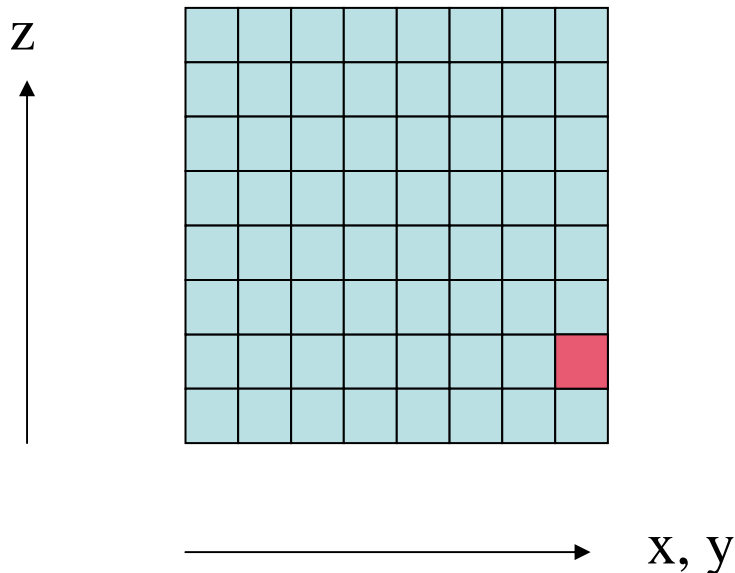


Comparison of serial vs vectorized derivative

Transverse (non-contiguous direction)

CPU

SPU



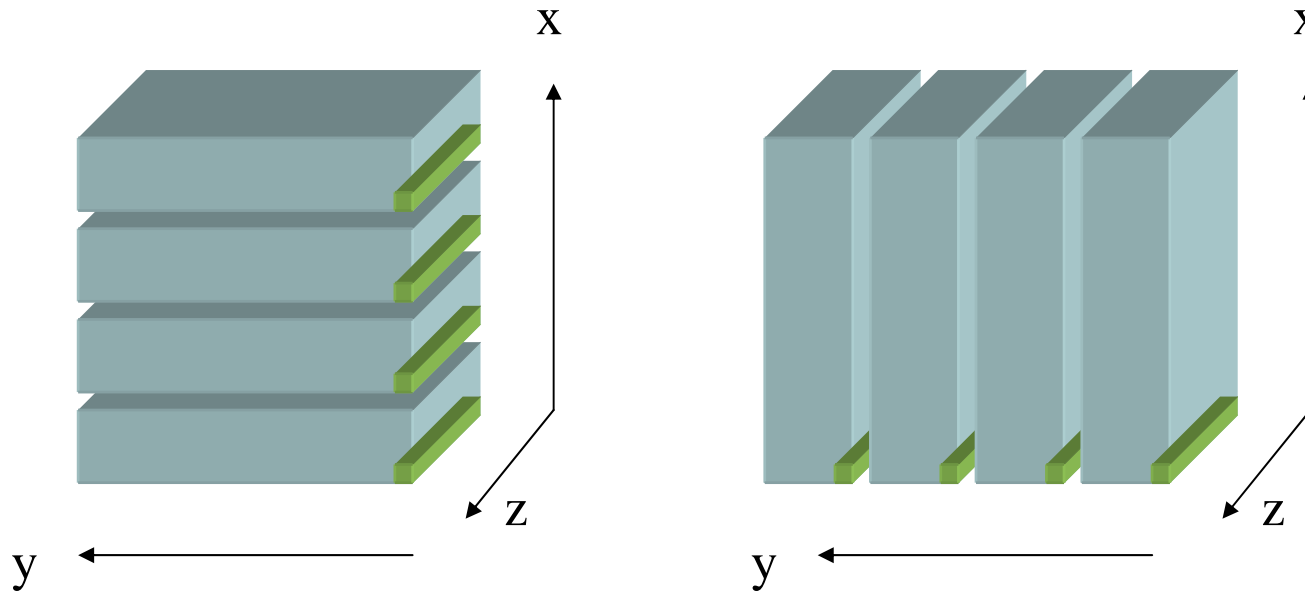
Red is working data

Green (SPU only) shows data loaded into local store



Data access in Cell memory

- **Each SPE accesses memory independently**
 - only 4 shown for clarity
- **Different offsets, strides, starting addresses for y and z derivatives**
 - always load contiguous z-vectors

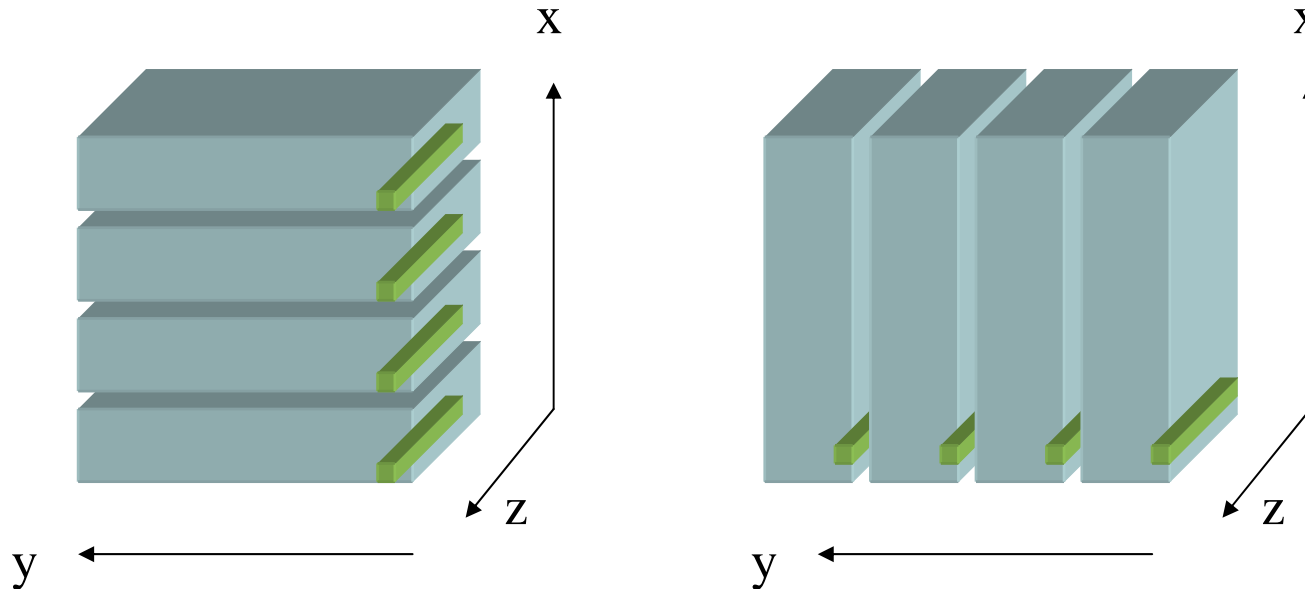


- **Data in cell memory never re-ordered**
- **Either access pattern is fine for update equation**



Data access in Cell memory

- **Each SPE accesses memory independently**
 - only 4 shown for clarity
- **Different offsets, strides, starting addresses for y and z derivatives**
 - always load contiguous z-vectors

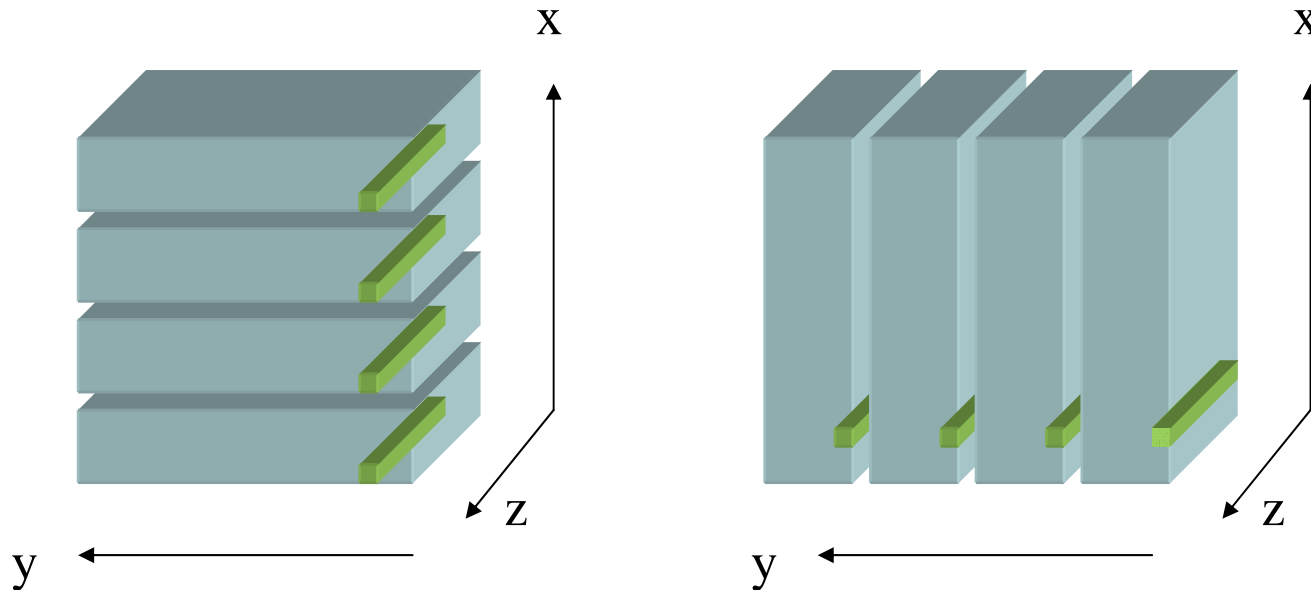


- **Data in cell memory never re-ordered**
- **Either access pattern is fine for update equation**



Data access in Cell memory

- **Each SPE accesses memory independently**
 - only 4 shown for clarity
- **Different offsets, strides, starting addresses for y and z derivatives**
 - always load contiguous z-vectors



- **Data in cell memory never re-ordered**
- **Either access pattern is fine for update equation**



Preliminary Timing

- **Comparison against purpose-built code**
 - same modifications made to both
 - precomputed coefficients, separate arrays etc.
- **For 128^3 test cases, single CBE**
 - Single precision (AAIS)
 - du/dy 0.0044s, 11.7 GB/s
 - du/dx 0.0052s, 8.7 GB/s
 - du/dz 0.008s, 6.1 GB/s
 - Total derivative time ~ 0.19 s
 - Update kernel 0.05sec
 - Total time .24 sec (vs. 11.9s for Opteron)
 - ~ 50 x speedup
 - Single buffered
 - no overlap of compute and memory fetches



Sanity check

- **Is this reasonable?**
 - 8 cores
 - 4-way vectorization (single precision) *
 - clock speed ratio is $3.2/2.2 = 1.45$
 - $8 * 4 * 1.45 = 46.54$

- **Excellent memory performance**
 - each SPU can issue DMAs independently

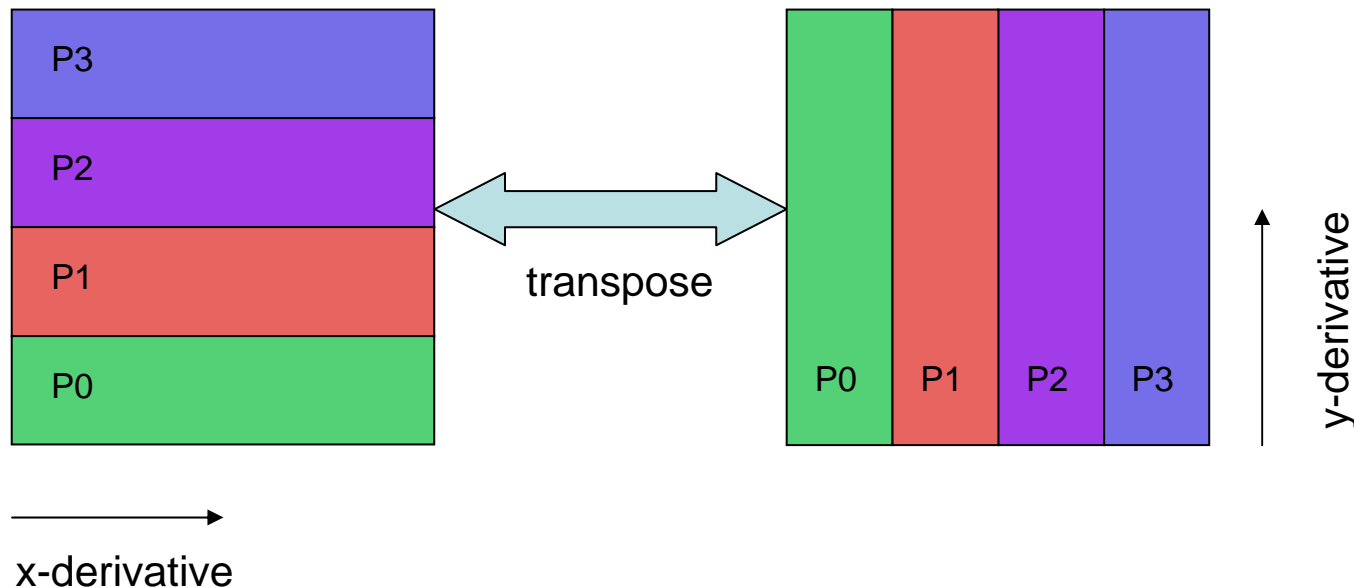
- *** How do we keep this for double precision?**
 - Overlap compute and communication
 - Lots of optimization to be had
 - Does it really matter?

- **Overall speedup not limited by CBE performance**
 - Need to revisit original code



Modifications to the original code

- **The CBE creates an imbalance between compute and communication**
- **Original tridiagonal solver required all data on the line to be on-processor**
 - All-to-all transpose required for each field derivative
 - All-to-all transpose back for the result array
 - Each processor must move $N_x \times N_y \times N_z$ (all local data for that field)
- **20% or more of runtime spent in MPI**
 - maximum speedup is at best 5x (Amdahls law)





Distributed Tridiagonal Solver

- **2 passes through the data in each direction**
 - only 1 pass for the original solver
- **Requires ghost cell data to compute RHS**
 - `mpi_sendrecv`; $N_x \times N_y \times 4$ (2 layers of ghost cells on each end)
- **After 1st pass in each direction, requires endpoint data passing**
 - `mpi_allgather`; $N_x \times N_y$, twice
- **After 2nd L-R pass, requires endpoint data to compute sum (for periodic case only)**
 - `mpi_allreduce`; $N_x \times N_y$, `mpi_scatter`; $N_x \times N_y$
- **Total data movement per processor is;**
 - $8 \times N_x \times N_y$; 6 MPI calls (periodic derivative)
 - $6 \times N_x \times N_y$; 4 MPI calls (non-periodic)



Distributed solver schematic



Pass ghost cells



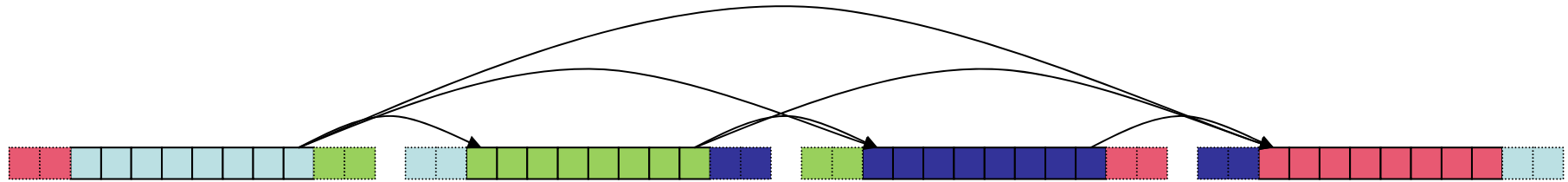
Distributed solver schematic



Pass ghost cells
First L-R pass



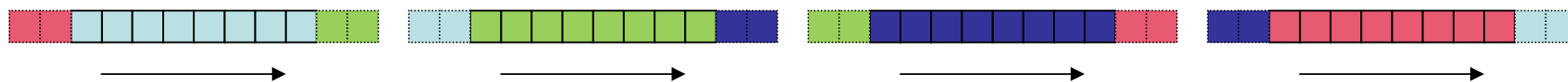
Distributed solver schematic



- Pass ghost cells**
- First L-R pass**
- Pass correction terms**



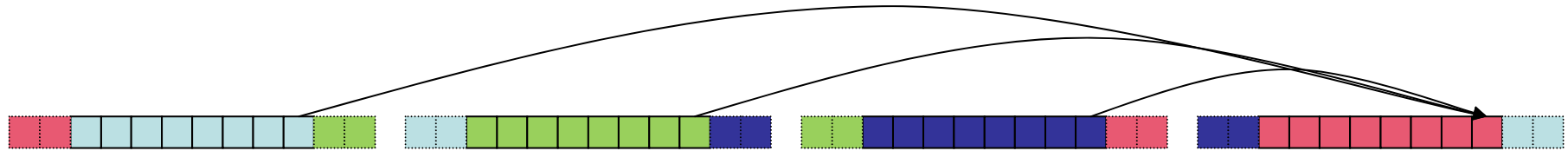
Distributed solver schematic



- Pass ghost cells**
- First L-R pass**
- Pass correction terms**
- Second L-R pass**



Distributed solver schematic



Pass ghost cells

First L-R pass

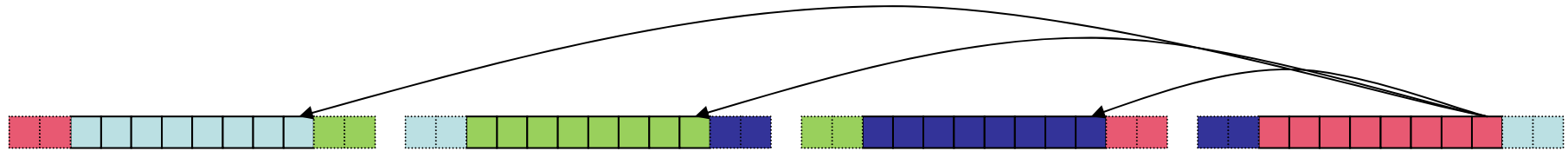
Pass correction terms

Second L-R pass

Pass sum to rightmost node (periodic case only)



Distributed solver schematic



Pass ghost cells

First L-R pass

Pass correction terms

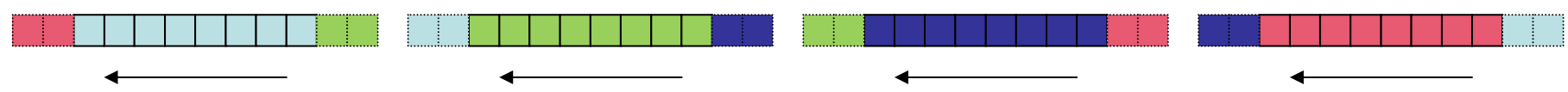
Second L-R pass

Pass sum to rightmost node (periodic case only)

Pass rightmost value to all nodes (periodic only)



Distributed solver schematic



Pass ghost cells

First L-R pass

Pass correction terms

Second L-R pass

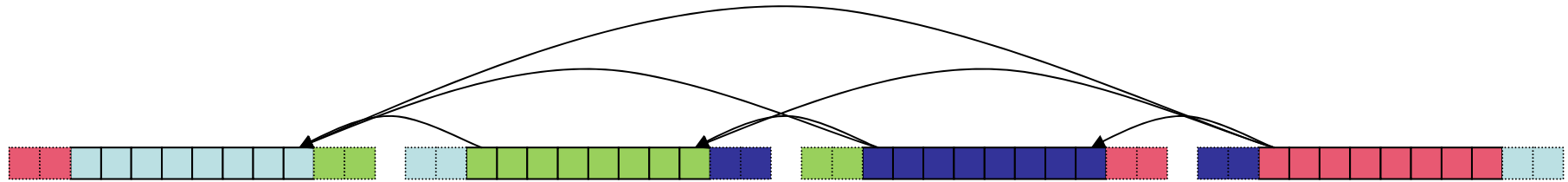
Pass sum to rightmost node (periodic case only)

Pass rightmost value to all nodes (periodic only)

First R-L pass



Distributed solver schematic



Pass ghost cells

First L-R pass

Pass correction terms

Second L-R pass

Pass sum to rightmost node (periodic case only)

Pass rightmost value to all nodes (periodic only)

First R-L pass

Pass correction terms



Distributed solver schematic



Pass ghost cells

First L-R pass

Pass correction terms

Second L-R pass

Pass sum to rightmost node (periodic case only)

Pass rightmost value to all nodes (periodic only)

First R-L pass

Pass correction terms

Second R-L pass



Solver timing comparison

- **Original solver**
 - communication is > 20% of total runtime
 - maximum potential speedup ~ 5x
- **Distributed solver**
 - communication is ~ 4% of runtime
 - maximum potential speedup ~ 25x
- **Ratio improves for larger data sets**
 - better surface area to volume ratio
 - production grid size is 256^3 per node
 - 64x less data movement
- **Opteron code does gain some performance**
 - doubling of local work partially offsets communication gain
 - still can gain ~ 20% for some configurations



Roadrunner code - choices to be made

- **The obvious ones**
 - Bulk of compute on the SPEs
 - Minimize PPU involvement
 - Granularity of MPI data movement
 - planes of data, not cells
- **The less obvious ones**
 - Rank-dependent compute
 - cells or Operons
 - Granularity/encapsulation of the derivatives
 - code readability
 - code performance
 - code size
 - Absolute performance of SPU compute portion
 - not the dominant factor



Hybrid code layout

- **Opteron**

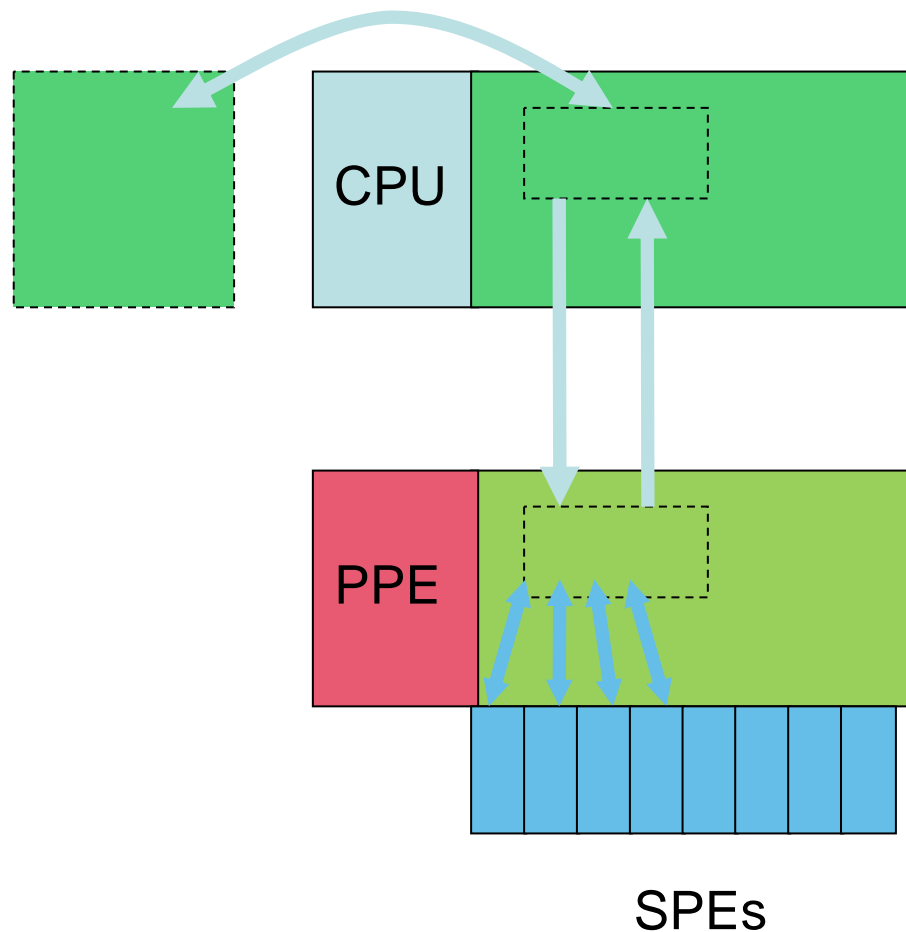
- Load correct Cell executable
 - based on MPI rank
- Handle MPI calls
- Handle disk access
- DaCS put/get to Cell memory

- **PPU**

- Share Cell memory with Opteron
- Pass messages between SPUs and CPU

- **SPEs**

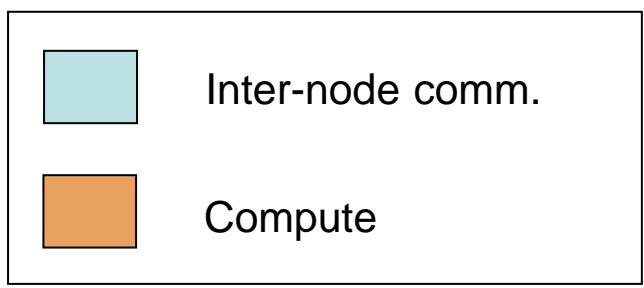
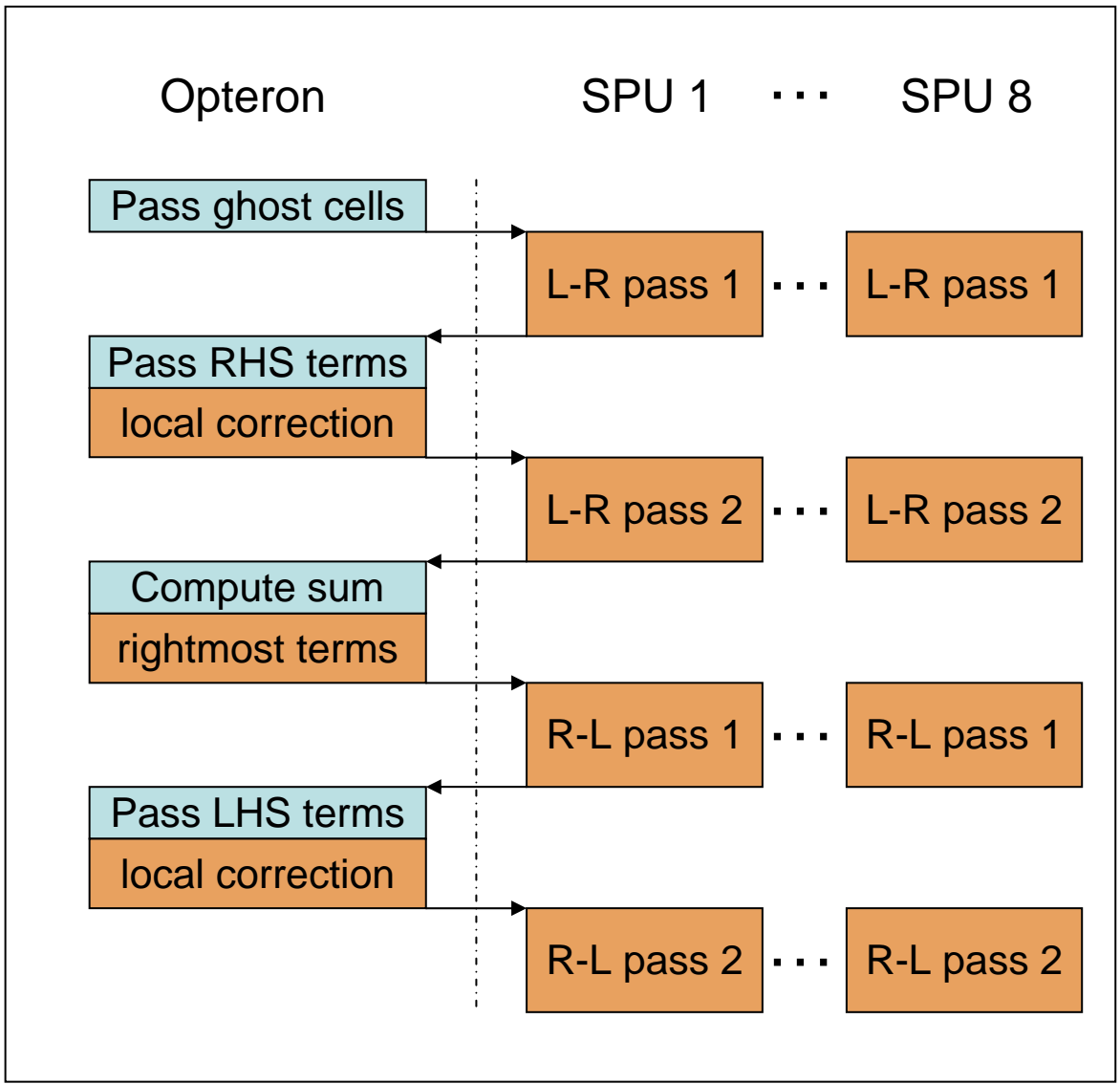
- DMA in/out of Cell memory
- Crunch





Distributed Tri-diagonal Solver (V1)

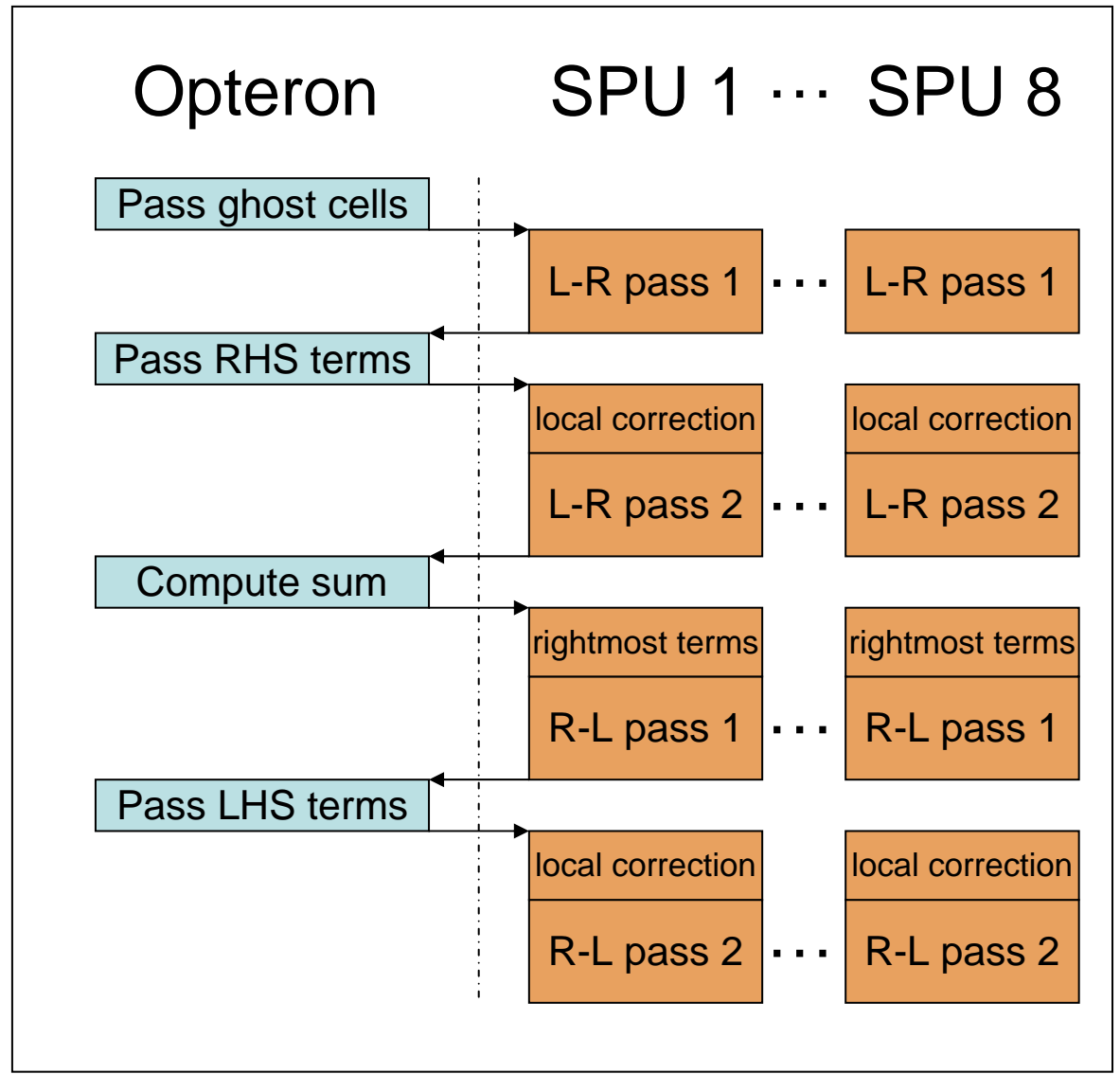
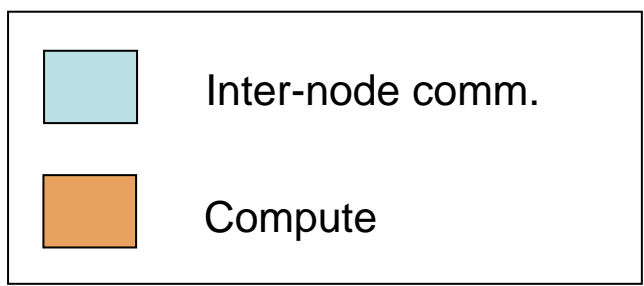
- Less data movement, more calls
- Local correction computed on Opteron (need to know rank in communicator)
- SPU compute is rank-independent
- Endian-conversion overhead





Distributed Tri-diagonal Solver (V2)

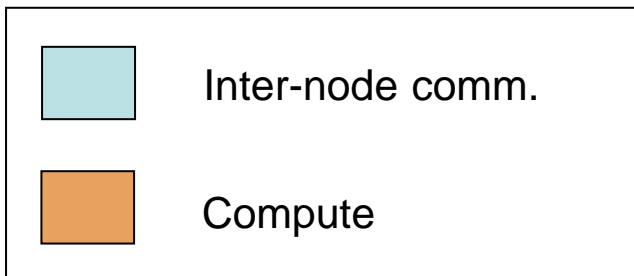
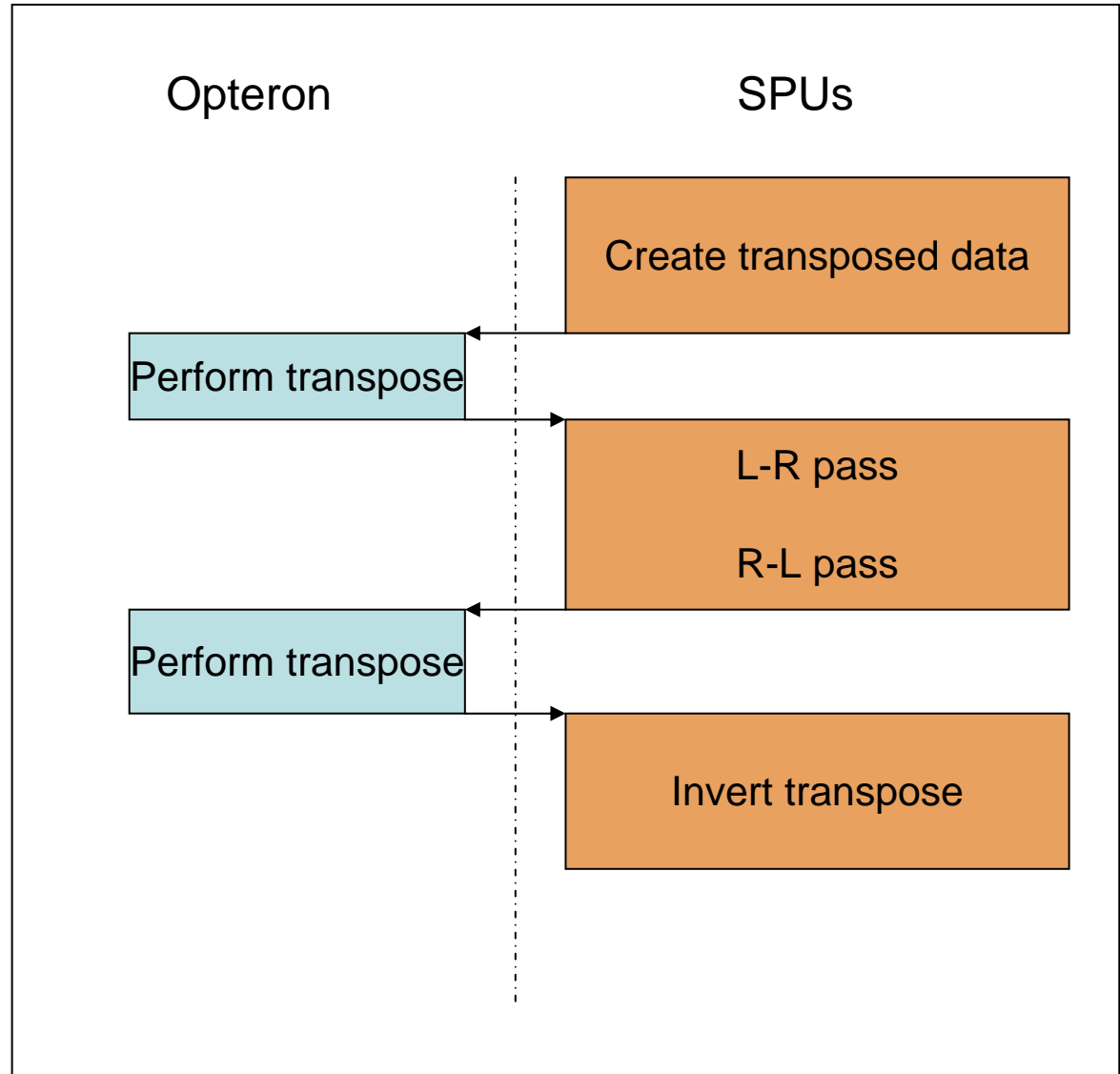
- Trade logic for compute
- Local correction computed on SPUs
- Coefficients pre-computed on Opteron (based on rank)
- SPU compute is rank-independent
- Compute balanced across all nodes
- Avoid byte-swap overhead





Original Tri-diagonal Solver

- Lots of data movement, few MPI calls
- Data transpose can occur on Opteron or Cell
- SPU compute is rank-independent





Memory Access/Movement Considerations

- **DMA's take time, regardless of size**
 - Make vector loads as long as possible
 - Minimize number of DMA's
 - $64 \times 64 \times 512$ is 20% faster than $128 \times 128 \times 128$
- **For ghost cell movement, cubic domains give best surface area/volume ratio**
 - need to balance compute with communication
- **For DNS, we want to fully utilize local memory**
 - $256^3 \times 50$ arrays = 6.25 GB (80% of local memory)
 - 256×50 arrays = 100 kB (40% of local store)
 - 512×50 arrays = 200 kB (80% of local store)



8GB of Opteron memory is a nice sized buffer

- **MPI data buffer**

- after fields are computed, immediately begin moving data so ghost cells/transposed data is available at destination
 - compute ghost cell layers first?
- push to Cell memory as requested
- minimize latency

- **Disk I/O buffer**

- data dumps are pulled to Opteron memory as needed
- timestep counter is on Opteron
 - at worst the SPU sees a barrier_wait until fields are read
- only 0.625 GB/restart (5 fields)



Conclusions

- **For this example**
 - Serial tests show speedup of $> 30x$, double precision
 - Significant restructuring of the code needed
 - New parallel tridiagonal solver reduces data movement
 - latency vs data volume tradeoff
 - more complicated scheduling and synchronization
- **In general**
 - For large structured data sets, the Cell acceleration potential is very good
 - excellent memory performance is largely responsible
 - overall speedup will depend on communication overhead for parallel codes
 - optimal compute/communication ratio is essentially ratio of bus speeds (EIB vs PCI/IB)