GPU Optimization of the Kripke Neutron-Particle Transport Mini-App

David Appelhans[1], Steve Rennich[2], Adam Kunen[3], Leopold Grinberg[1]

[1]IBM Research, [2]Nvidia, [3]LLNL

April 7, 2016

Neutron Transport

- Solve steady state Boltzmann transport equation for angular flux $\psi(r, \Omega, E)$.

$$[\Omega \cdot \nabla + \sigma(r, E)]\psi(r, \Omega, E) =$$
$$\int dE' \int d\Omega\prime \sigma_s(r, \Omega' \cdot \Omega, E' \to E)\psi(r, \Omega', E') + q(r, \Omega, E)$$

- 6D phase space, leads to high unknown counts.
- Spatial domain, $r \in \mathbb{R}^3$
- Scattering direction in unit sphere, $\Omega \in \mathcal{S}^2$
- Energy, $E \in (0, \infty)$

Discretized Steady State Transport Equation

In discretized matrix form, transport equation becomes,

$$H\Psi = L^{+}\Sigma_s L\Psi + Q. \tag{1}$$

With $Z \cdot D \cdot G$ unknowns,

- (Z) Diamond-Difference discretization of H into spatial zones.
- (D) Quadrature points for integral over directions Ω.
- (G) Energy is binned into G groups.

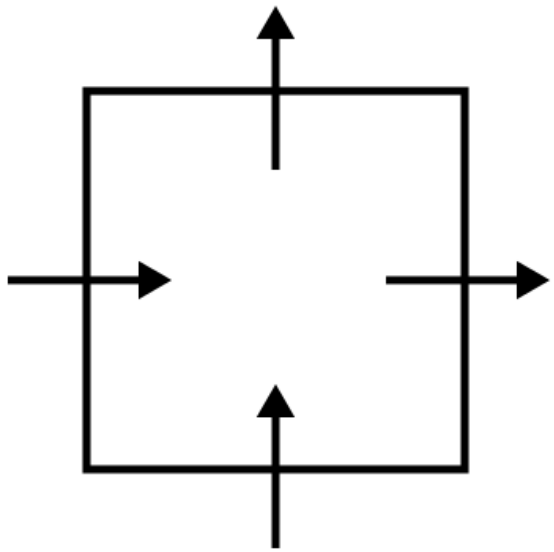
Solution Steps

Iterative solution:

$$H\Psi^{i+1} = L^{+}\Sigma_s L\Psi^{i} + Q \qquad (2)$$

Each iteration requires

- calculation of RHS
  - Embarassingly parallel w.r.t Z
  - Matrix-matrix multiplication
- Solve of $H\Psi = RHS$ (Sweep)
  - H is block-diagonal in G and D.
  - Upwind data dependence in Z.
  - Data dependence limits scaling.



KRIPKE

- mini-app from LLNL (https://codesign.llnl.gov/kripke.php)
- Captures behavior of Ardra, a production transport solver.
- Explores different data ordering of discretization variables, e.g. ZDG vs DGZ.
- Leads to different loop ordering for each operation.
- This talk used ZDG and ZGD ordering.
- Code was ported using CUDA and OpenMP 4 through LLVM compiler for comparison.

Construction of RHS

- Embarassingly parallel in zones.
- Explored hand written CUDA and OpenMP 4 for comparison of differences.
- OpenMP 4 was within 20% of optimized CUDA.
- All implementations (C / OpenMP / CUDA) can use a batched GEMM library for the RHS construction. This achieves 900 GF/s, or 60% of the achievable peak on the GPU.
- Sweep determines scalability.

Simple Sweep

Original algorithm sweeps through zones within a MPI rank sequentially.

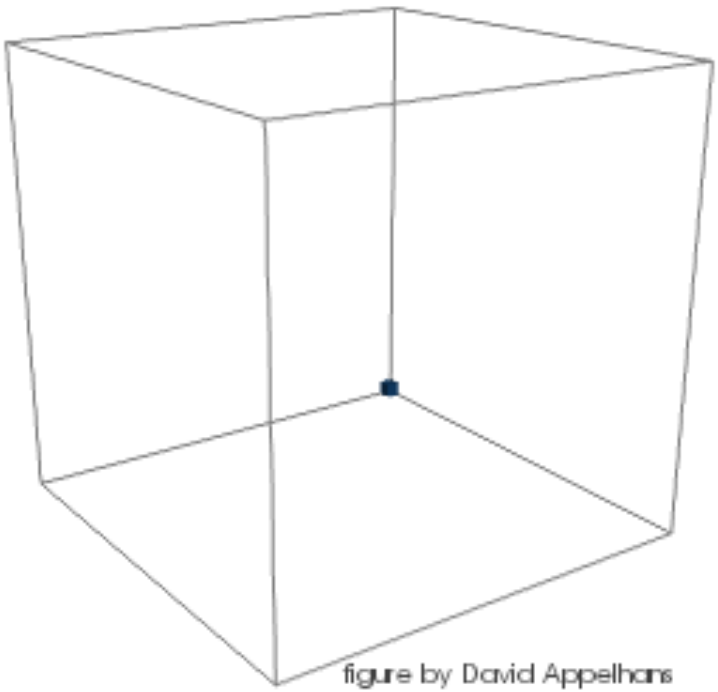


figure by David Appelhans

- Groups and directions processed in parallel.
- Parallelism over groups and directions is not sufficient.
- Typically 32^3 zones, 16 directions, and 128 groups.
- Other MPI ranks can begin only once sweep has completed in a dependent rank.

HYPERPLANE SWEEP

Proceed along hyperplanes of zones where data dependency has been met.

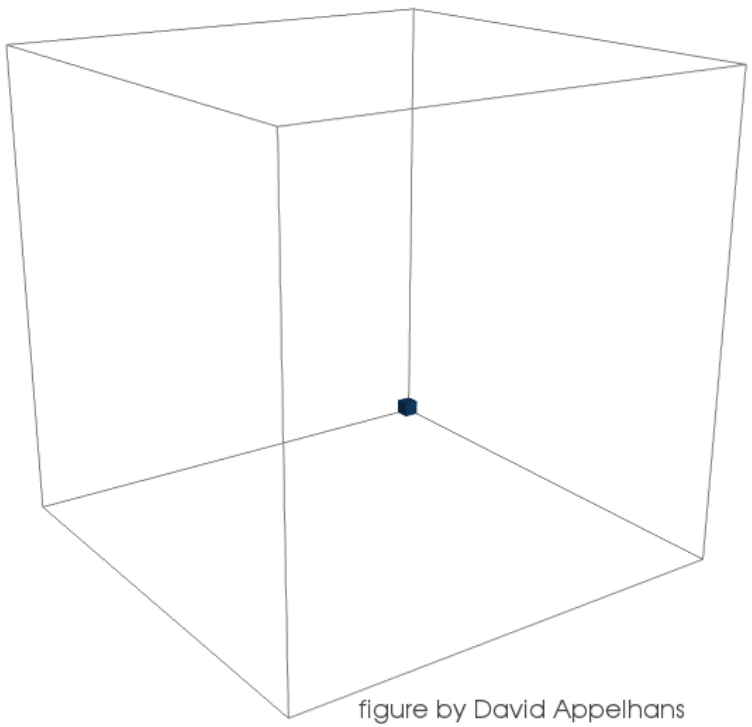


figure by David Appelhans

Advantages:

- Can process zones in a hyperplane in parallel blocks.
- For 32x32x32 local zones, **3 orders of magnitude** more parallelism.
- Groups and directions add further parallelism (thread level).

Hyperplane Sweep

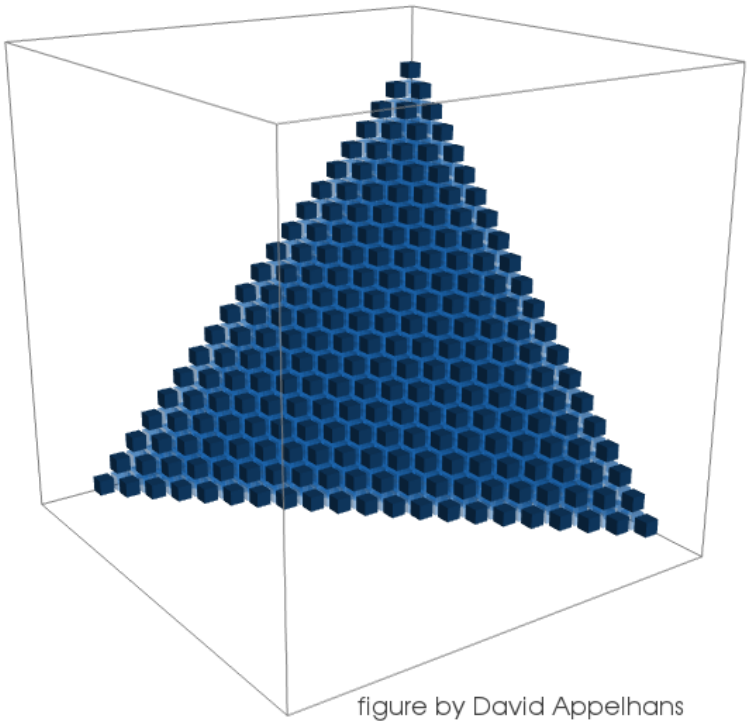


figure by David Appelhans

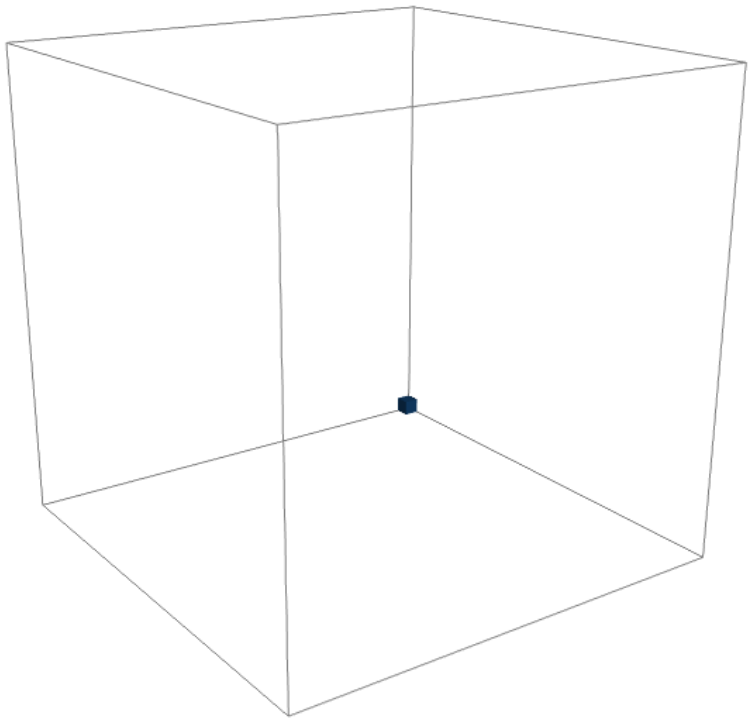
Limitations:

- Each zone stores flux data in global memory. (Bandwidth bound).
- Synchronization through kernel launches for each hplane (expensive).

Tiled Hyperplane Sweep

A thread works on same zone in j-k projection, progresses in i direction.

Tiled Hyperplanes



Regular Hyperplanes

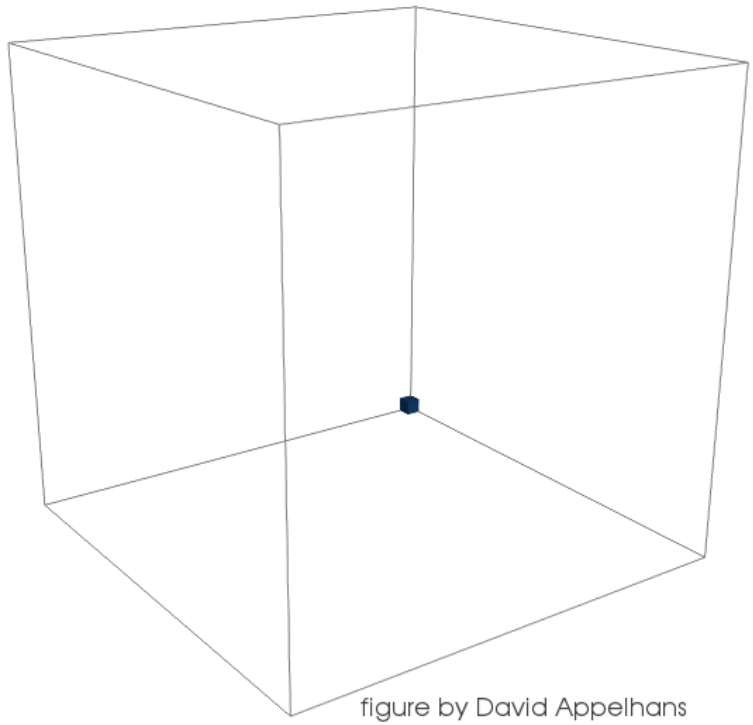
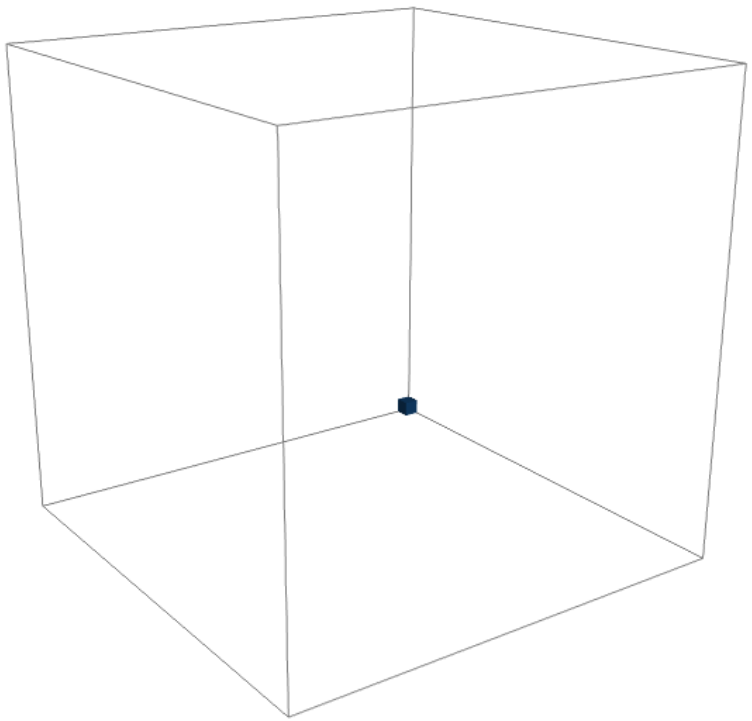


figure by David Appelhans

Tiled Hyperplane Sweep

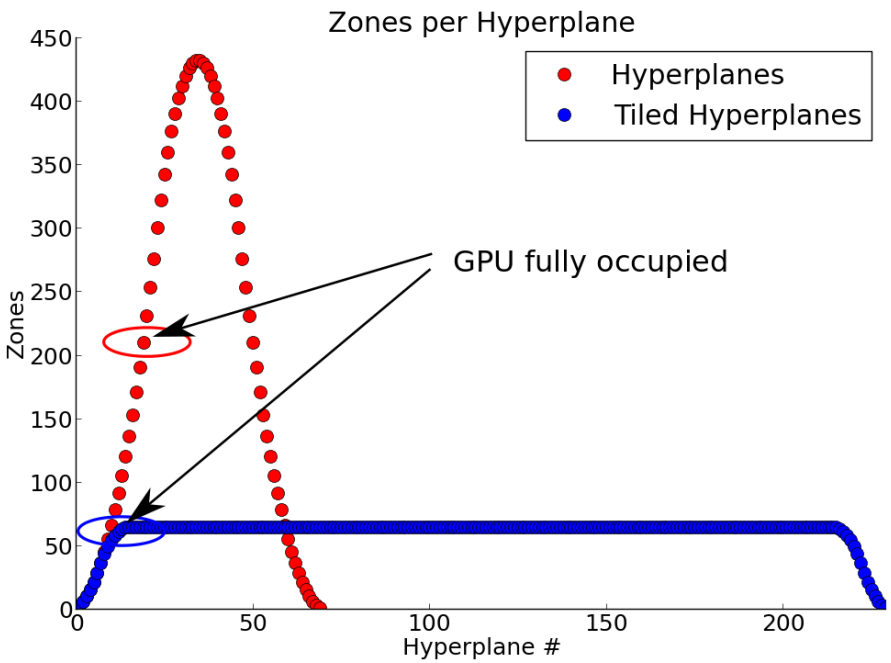
A thread works on same zone in j-k projection, progresses in i direction.

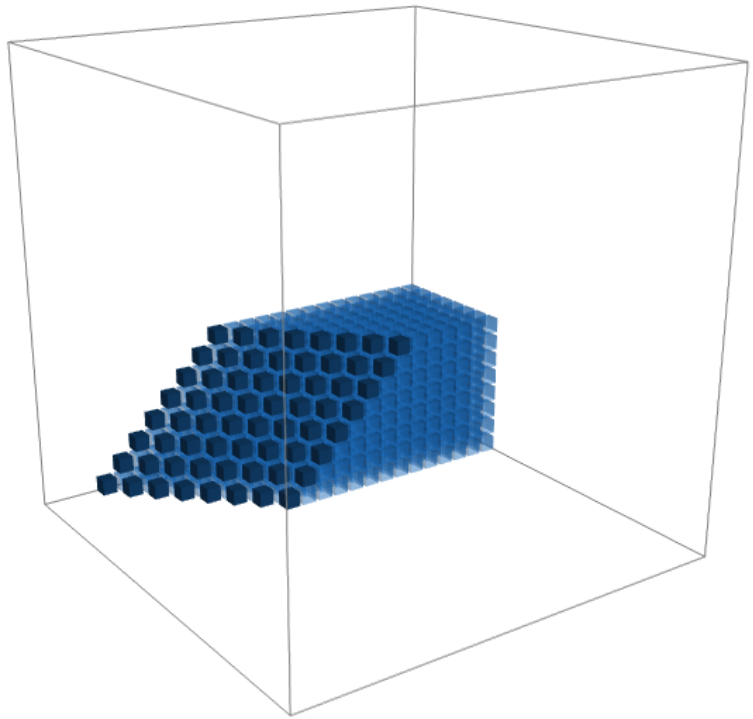


- j,k remains same for each thread.
- i-flux is stored in a register.
- j and k flux in interior of subdomain is stored in shared memory.
- Psi[G,D,Z] accessed in cache line friendly chunks (32 kb or 128 kb).
- Hyperplanes synchronized within a block (more efficent).

Tiled Hyperplane Sweep

- Reduces global memory access by only storing flux on boundary.
- Pipeline is filled sooner: uses a few blocks of 1024 threads per SM, instead of number of blocks = number of zones.
- Can efficiently do small subset of G or D at a time -> faster data generation for MPI communication.





Tiled Hyperplane Sweep: CUDA

CUDA version:

- Uses 58 registers/thread.
- 1024 threads, 16 kb of shared memory per block.
- on k80 two of these blocks can run per SM.
- Theoretical max occupancy: 100%.
- Two configurations:
  - 8x8 subdomain of zones, 16 directions per zone
  - 16x16 zones, 4 directions per zone

Tiled Hyperplane Sweep: OpenMP 4

LLVM OpenMP 4 compiler is young–excessive register allocation problem limits number of threads (for no spilling).

- 162 registers/thread for no spilling.
- **256** threads per block, 4 kb of shared memory per block.
- 3 blocks per SM on K80 (register limited).
- Theoretical max occupancy: 37.5%.
- 8x8 subdomain of zones, **4** directions per zone.

Sweep Timing Normalization

- Different problem parameters benefit different methods.
- Tiled hyperplane algorithm works best on K80 when number of blocks is 26 (CUDA) or 39 (OMP4).
- Normalize reported times by dividing by the degrees of freedom.
- Define grind time

$$t_g = \frac{t_o}{zones * groups * directions}$$

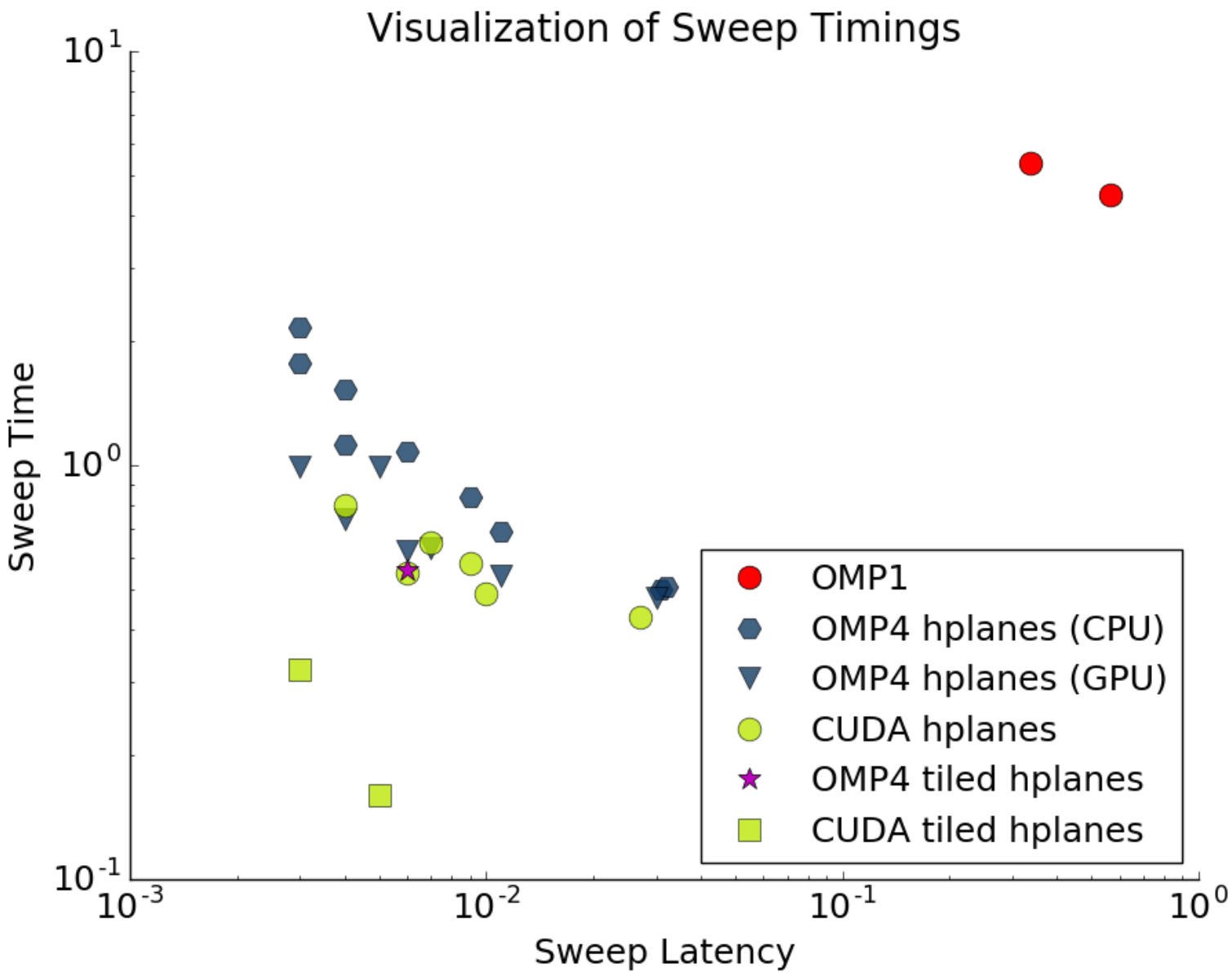
Latency and Throughput

- Sweep Time - Time to apply diamond difference relations to all dof for 1 MPI rank.
- Sweep Latency - Time to sweep subset of groups/directions before communication to waiting MPI rank.
- Group/direction subsets in Kripke allow tuning of group/direction chunk sizes.
- Tradeoff between sweep latency and total sweep time.

Sweep Comparison

Units: ms/million dof	Time	Latency	(gset,dset)
OMP1 regular (CPU)	4.51	0.563	(1,8)
OMP4 hplane (CPU)	0.50	0.031	(2,8)
OMP4 hplane (CPU)	0.69	0.011	(8,8)
OMP4 hplane (GPU)	0.48	0.030	(2,8)
OMP4 hplane (GPU)	0.62	0.006	(4,24)
CUDA hplane (GPU)	0.43	0.027	(2,8)
CUDA hplane (GPU)	0.55	0.006	(4,24)
OMP4 tiled hplane (GPU)	0.56	0.006	(4,24)
CUDA tiled hplane 16x16x4	0.32	0.003	(4,24)
CUDA tiled hplane 8x8x16	0.16	0.005	(4,8)

Table: Table of sweep times for 32x32x32 zone domain, 96-128 directions, 104-156 groups. More gset/dsets results in lower latency but slower overall time. CPU was dual socket, 20 core POWER 8, all GPU results were for a single K80 (half a physical card).


Visualization of Sweep Timings
Sweep Time
Sweep Latency
10^{1}
10^{0}
10^{-1}
10^{-3}
10^{-2}
10^{-1}
10^{0}
OMP1
OMP4 hplanes (CPU)
OMP4 hplanes (GPU)
CUDA hplanes
OMP4 tiled hplanes
CUDA tiled hplanes

CONCLUSIONS

- CUDA tiled hyperplanes
  - Best performance on GPU (2.6x faster than regular hyperplanes).
  - 2x faster communication
  - Requires adjustment of number of blocks based on number of SMS.
- OpenMP 4 tiled sweep
  - Not as fast as CUDA version yet, because of excessive register usage forcing low occupancy and threadcount.
  - When register usage is improved, expect similar performance as CUDA version.
  - Algorithm needs modification to explore performance portability.
- OpenMP 4 regular hyperplanes
  - Performance portable: same code on GPU and CPU.
  - 9x faster than original sweep (GPU & CPU).
  - OpenMP on GPU within 10% of CUDA hyperplane sweep.

QUESTIONS?

David Appelhans - dappelh@us.ibm.com

OpenMP Hyperplane Code Structure

```
1   #pragma omp target data map(........) if(GPU_FLAG)
2      // Loop over the hyperplanes (slices).
3
4      for (int slice = 0; slice < Nslices; slice++){
5        int hplane_size = offset[slice+1]-offset[slice];
6
7   #pragma omp target if(GPU_FLAG)
8   #pragma omp teams distribute num_teams(hplane_size) thread_limit(2)
9         for (int element = offset[slice]; element < offset[slice+1]; ++element) {
10          // load i,j,k,z
11          ...
12          // Pointer initializations
13          ...
14  #pragma omp parallel for
15          for (int d = 0; d < num_directions; ++d) {
16            // calculate cos info depending on d.
17            ...
18  #pragma omp simd
19            for (int group = 0; group < num_groups; ++group) {
20              ...
21              /* Calculate new zonal flux */
22              ...
23              psi_z_d[gd] = psi_z_d_g;
24              /* Apply diamond-difference relationships */
25              ...
26            }
27          }
28        } //end element (distribute)
29     }//end "for (slice"
```