ACCELERATING A C++ CFD CODE WITH OPENACC

NVIDIA APPLICATION LAB AT JÜLICH 2ND ANNUAL WORKSHOP

Jiri Kraus (NVIDIA),

Michael Schlottke (RWTH Aachen)

AGENDA

ZFS

Accelerating ZFS with OpenACC

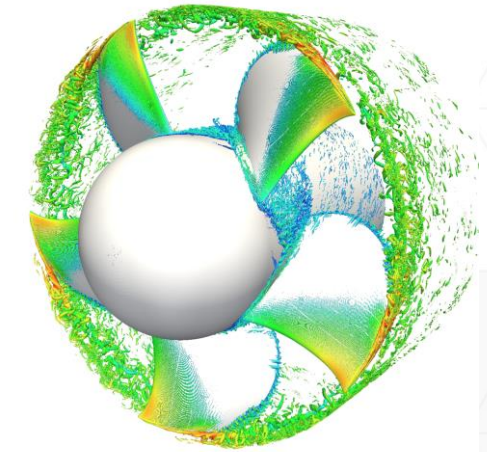
Optimizations

Performance Analysis

Multi-GPU and CUDA-aware MPI

ZFS

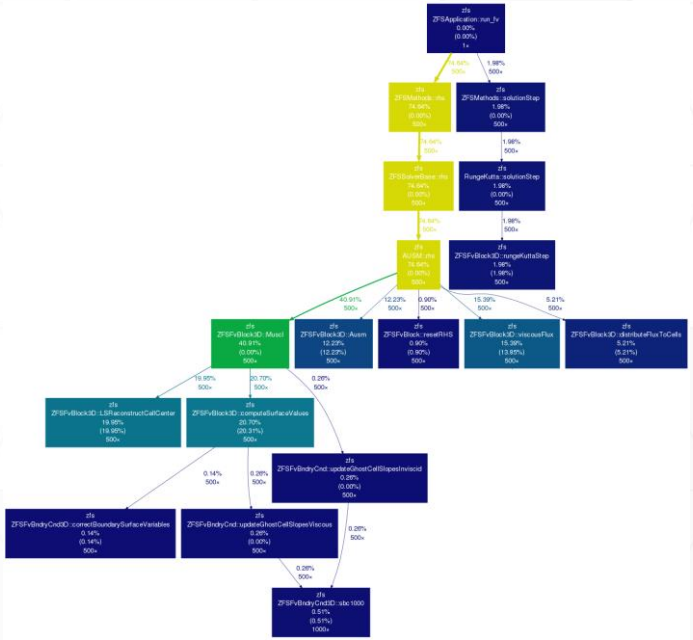
- **Z**onal **F**low **S**olver is a CFD code supporting different solvers:
  - Finite-Volume
  - Dicontinuous Galerkin
  - Lattice Boltzmann
- Explicit time stepping
- MPI parallel C++ with almost 200000 lines of code
- Developed at the Institute of Aerodynamics at RWTH Aachen



ACCELERATING ZFS WITH OPENACC

Assess

- Profile CPU Version to get Calltree
- Focus on keeping data resident on GPU



ACCELERATING ZFS WITH OPENACC

Parallelize I

- Incremental approach staging data in and out in every method
  - Using **present_or_*/p***
- Always working code but very long runtimes
  - Requires small input set
- Tipp:
  - Compile with debug info and use cuda-memcheck

```
#pragma acc data \
  pcopyin(sCellIds[0:noSCells], \
          mCellIds[0:noSCells])) \
  pcopy(rhs[0:noCells*noVarIds])
{
    ...
}
```

ACCELERATING ZFS WITH OPENACC

Parallelize II

- Acceleration of most loops was straight forward:

```
#pragma acc parallel loop collapse(2)
for(ZFSId pc=0; pc<noCellsAtPB; pc++)
  for(ZFSId v=0; v<noVars; v++)
    rhs[cellsAtPB[pc]*noVars+v]=0.0;
```

- Two exceptions:
  - `distributeFluxToCells`
  - `correctMasterCells`

ACCELERATING ZFS WITH OPENACC

Parallelize III

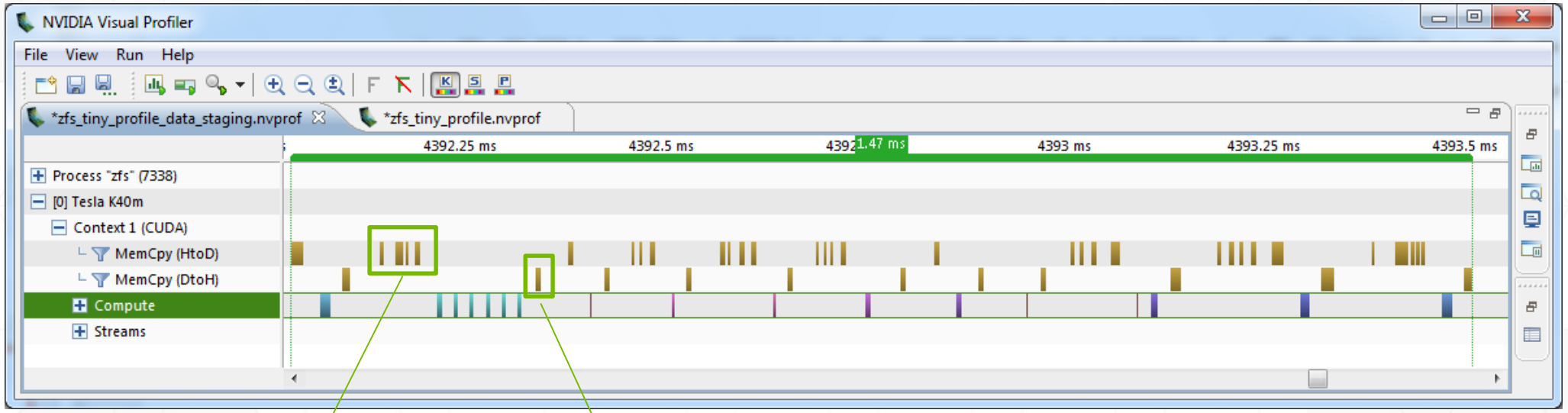
- `correctMasterCells` using atomic directive:

```
#pragma acc parallel loop collapse(2)
for(ZFSId s = 0; s<noSCells; s++) {
  for(ZFSId varId=0; varId<noVarIds; varId++) {
    #pragma acc atomic
    rhs[mCellIds[s]*noVarIds+varId] +=
      rhs[sCellIds[s]*noVarIds+varId];
    rhs[sCellIds[s]*noVarIds+varId]=0.0;
  }
}
```

- `distributeFluxToCells`:
  - Restructure to handle only one orientation and side in each iteration

ACCELERATING ZFS WITH OPENACC

Parallelize IV – unstructured data regions



ACCELERATING ZFS WITH OPENACC

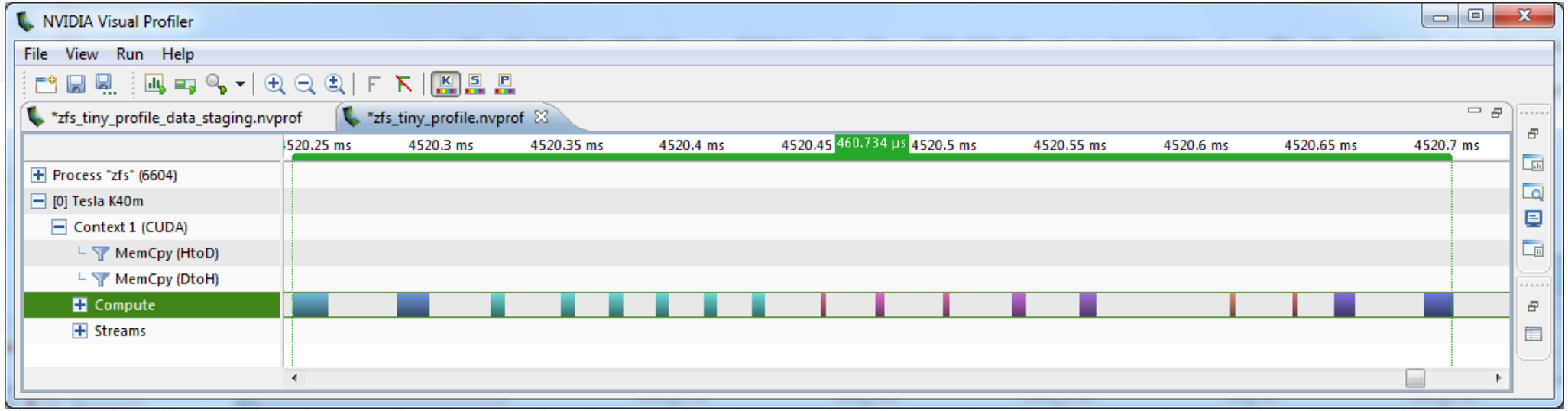
Parallelize IV – unstructured data regions

```
zfsAlloc(hCells,noNghbr,noHCells);
#pragma acc data                    \
create(hCells[0:noNghbr*noHCells])
{
...

}
zfsDeallocate(hCells);
```

```
zfsAlloc(hCells,noNghbr,noHCells);

#pragma acc enter data              \
create(hCells[0:noNghbr*noHCells])
...
#pragma acc exit data               \
delete(hCells[0:noNghbr*noHCells])
zfsDeallocate(hCells);
```

ACCELERATING ZFS WITH OPENACC

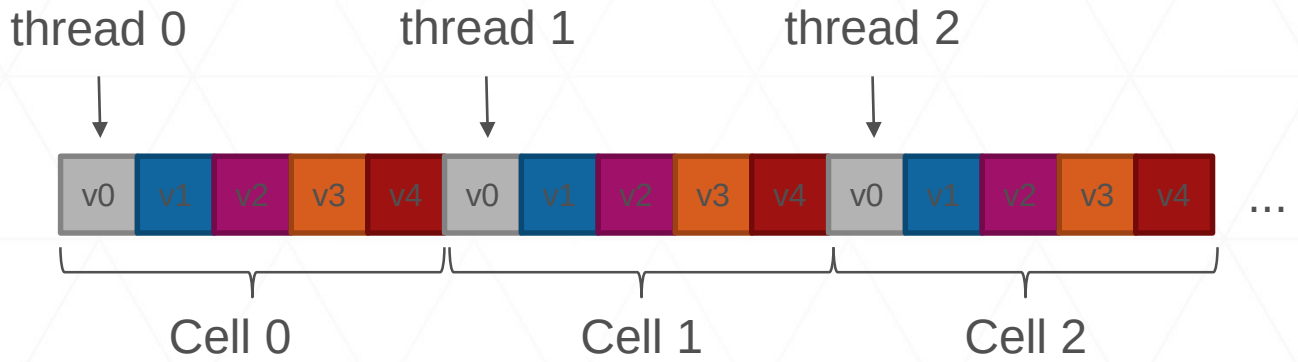
Parallelize IV – unstructured data regions



OPTIMIZATIONS

Texture cache and pointer aliasing

- ZFS uses a Array of Structures (AoS) layout leading to suboptimal memory access patterns with large strides:



- Impact of this is reduced by utilizing the read-only data cache (aka texture cache) on Kepler:

```
const ZFSFloat * const __restrict cells = …
```

OPTIMIZATIONS

Increasing cache hit rates with lower occupancy

- Up to 2048 threads can run on a single Kepler SM
- Sharing 12-48 KB of read-only data (texture) cache per SM and 1536KB L2 per chip
- Decrease occupancy to increase cache hit rates with the vector clause

```
#pragma acc parallel loop gang vector(32)

for( ZFSUint srfcId=0; srfcId<noSurfaces; srfcId++ )
{ ... }
```

OPTIMIZATIONS

Increasing cache hit rates with lower occupancy

```
#pragma acc parallel loop gang vector(32)

for( ZFSUint srfcId=0; srfcId<noSurfaces; srfcId++ )
{... }
```

Effect for computeSurfaceValues (hand tuned for all kernels)

vector size	Time (ms)	Ach. Occ.	Tex$ Hit Rate	L2$ Hit Rate
256 (default)	3.68	0.45	62.08%	46.74%
128	4.07	0.48	63.13%	40.62%
32 (minimum)	2.18	0.25	68.78%	60.36%

OPTIMIZATIONS

Summary

Time (%) (opt)	Calls	Avg. kernel time (no opt)	Avg. kernel time (opt)	Name
28.60%	3000	4.05 ms	3.75 ms	distributeFluxToCells
23.51%	500	27.22 ms	18.52 ms	computeSurfaceValues
15.49%	500	12.25 ms	12.20 ms	LSReconstructCellCenter
12.48%	500	9.82 ms	9.83 ms	viscousFlux
11.44%	500	9.00 ms	9.01 ms	Ausm

PERFORMANCE ANALYSIS

GPU Profile – contributions above 1%

Time (%)	Time	Calls	Avg.	Name
28.60	11.263 s	3000	3.75 ms	distributeFluxToCells
23.51	9.260 s	500	18.52 ms	computeSurfaceValues
15.49	6.100 s	500	12.20 ms	LSReconstructCellCenter
12.48	4.915 s	500	9.83 ms	viscousFlux
11.44	4.504 s	500	9.01 ms	Ausm
2.50	984 ms	501	1.96 ms	computePrimitiveVariables
1.88	739 ms	500	1.48 ms	rungeKuttaStep
1.65	648 ms	500	1.30 ms	LSReconstructCellCenterII
1.64	644 ms	510	1.26 ms	CUDA memcpy DtoH+HtoD

PERFORMANCE ANALYSIS

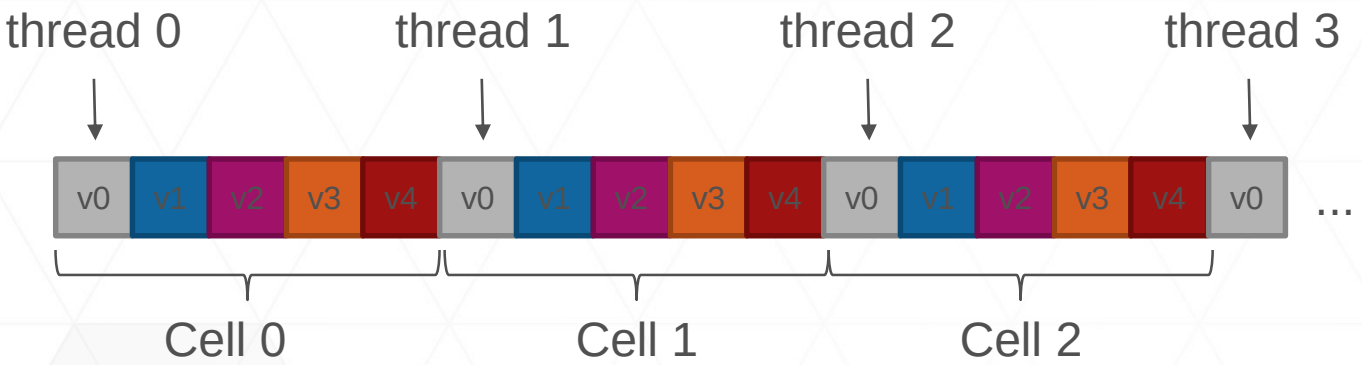
Name	DRAM read BW (GB/s)	DRAM write BW (GB/s)	DRAM BW (GB/s)
distributeFluxToCells	130.99	91.57	222.56
computeSurfaceValues	115.29	101.75	217.04
LSReconstructCellCenter	100.74	72.82	173.56
viscousFlux	178.42	52.14	230.56
Ausm	108.32	34.79	143.11

Top 5 Kernels are all (more or less) stressing the memory system.

PERFORMANCE ANALYSIS

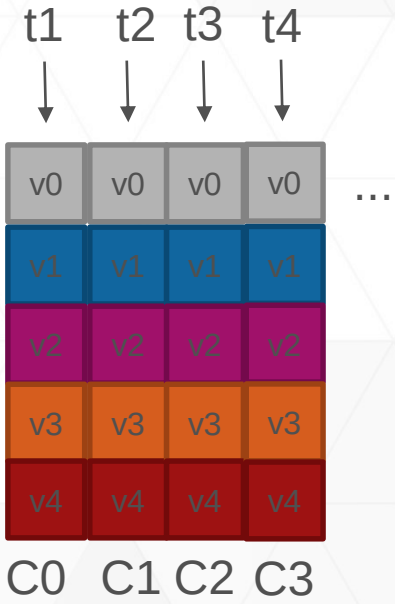
computeSurfaceValues – AoS vs. SoA

Data Accesses (AoS)



Only 8 bytes out of 32byte transaction used

Data Accesses (SoA)



32 bytes out of 32byte transaction used

PERFORMANCE ANALYSIS

computeSurfaceValues – AoS vs. SoA

	SoA	AoS	
Time	1.4201 ms	2.1678 ms	1.53 x
Achieved Occupancy	35%	25%	
DRAM utilization level	High (8)	High (8)	
DRAM BW (r+w)	203.32	223.54	
Tex$ Hit Rate	51.85%	68.92%	
L2$ Hit Rate	55.72%	60.36%	
Texture Cache requests	21605066	30004894	1.38 x
Global st transactions/request	2.50	20	8 x

MULTI-GPU VERSION

- CPU version is already MPI parallel
- Multi GPU version straight forward using CUDA-aware MPI on device buffers:

```
#pragma acc host_data use_device(recvBuffer)
MPI_Recv_init(recvBuffer, ...);
#pragma acc host_data use_device(sendBuffer)
MPI_Send_init(sendBuffer, ...);
```

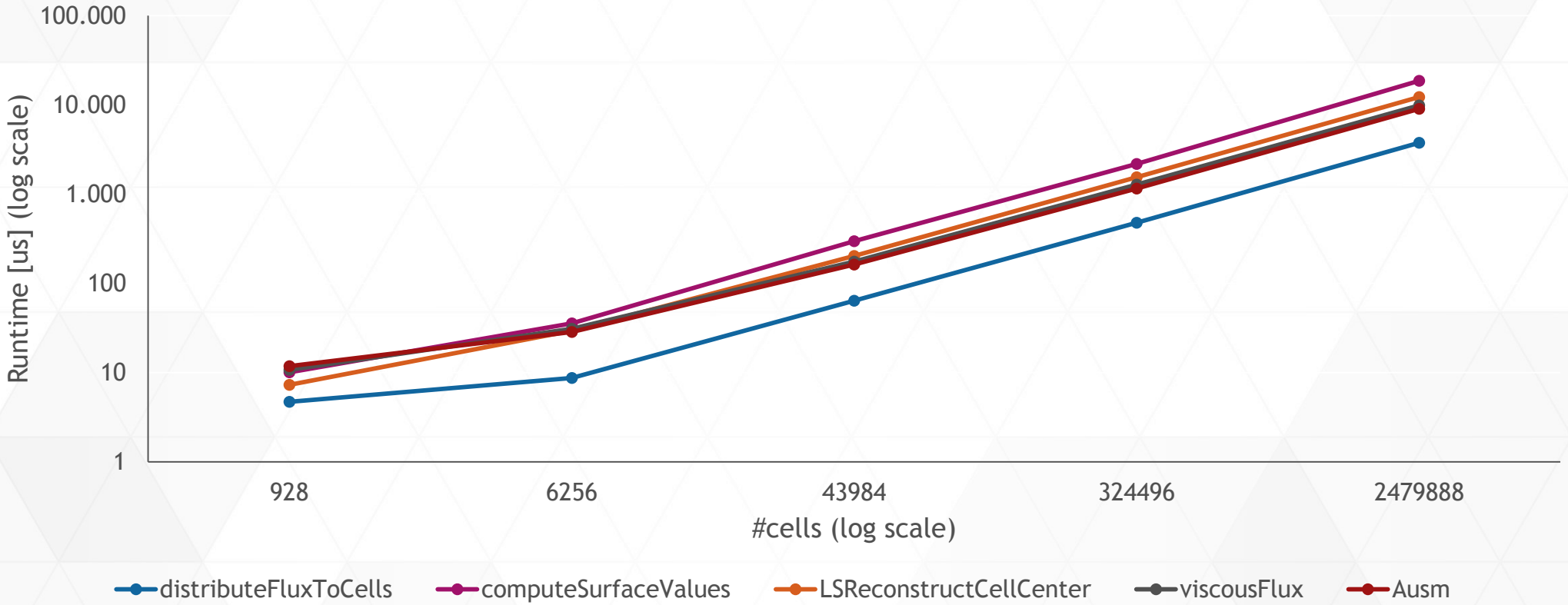
MULTI-GPU VERSION

Handling GPU affinity

```
int rank = 0;
int size = 0;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
#ifdef _OPENACC
int ngpus = acc_get_num_devices(acc_device_nvidia);
int devicenum = rank%ngpus;
acc_set_device_num(devicenum,acc_device_nvidia);
#endif
```

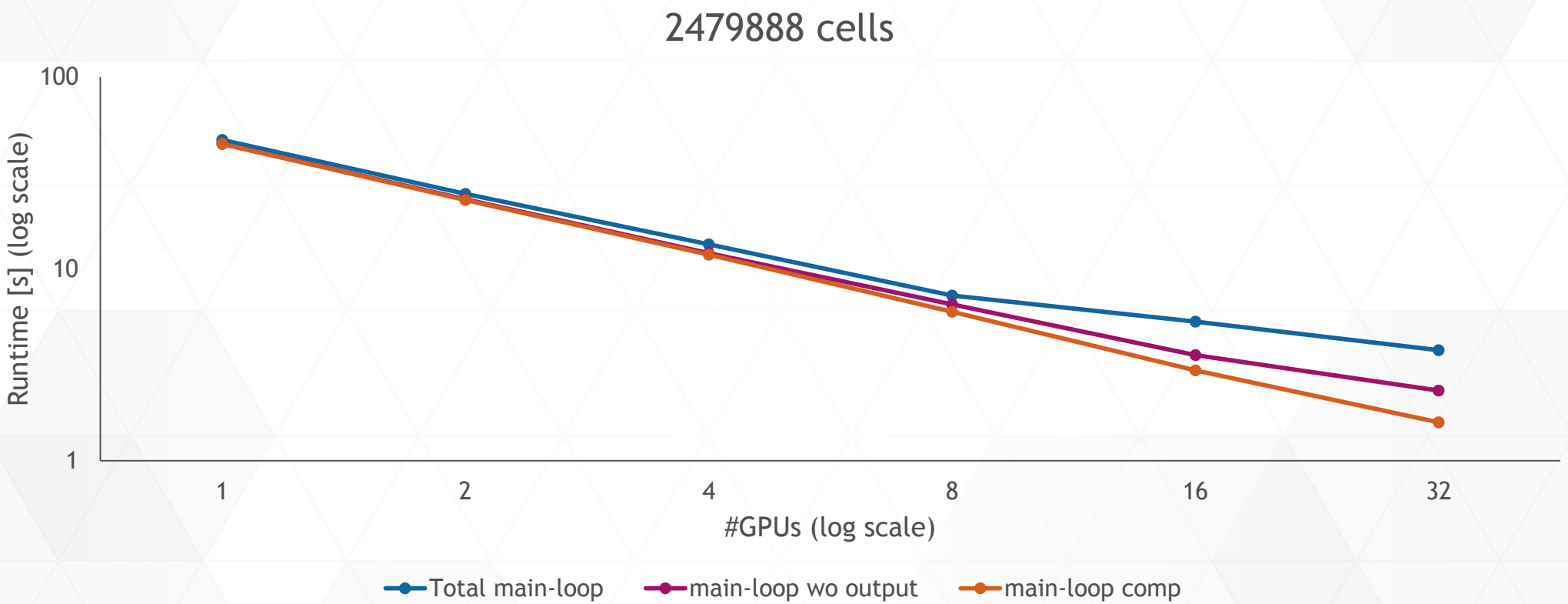
MULTI-GPU VERSION

Scalability – Perf vs. Problem size



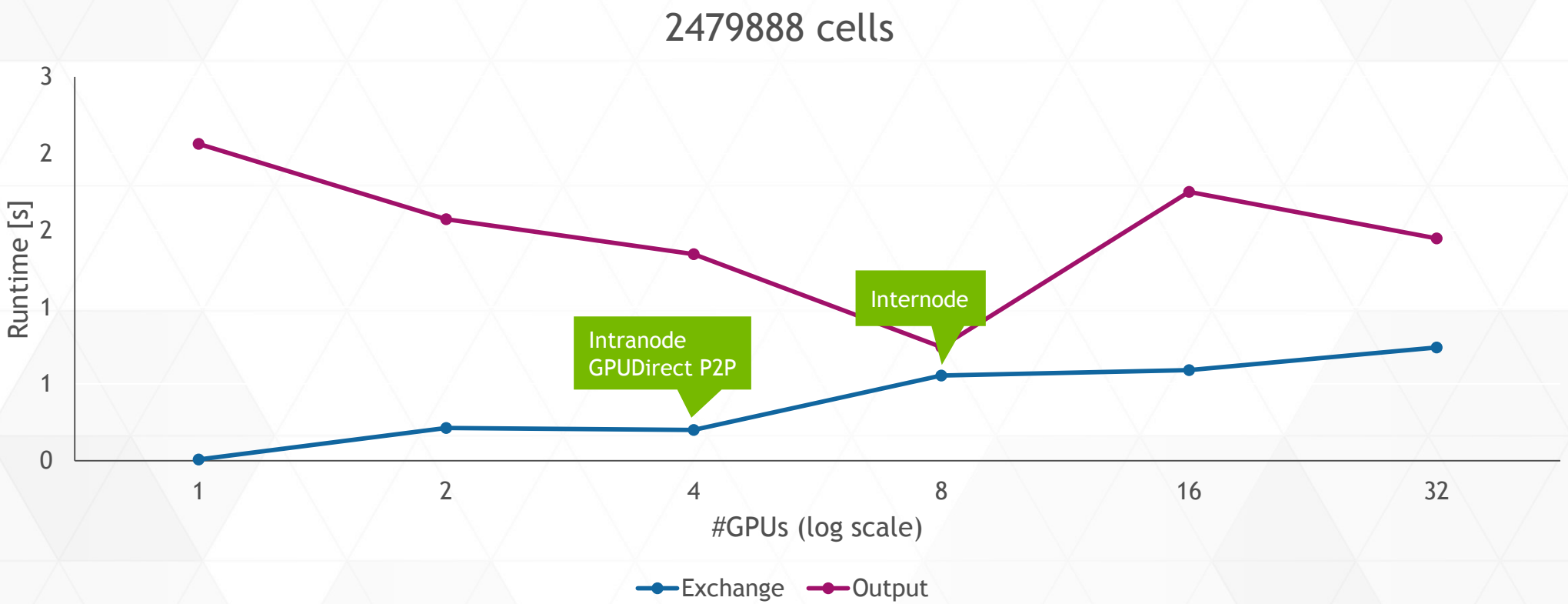
MULTI-GPU VERSION

Scalability – Strong Scaling



MULTI-GPU VERSION

Scalability – Strong Scaling – Output and Exchange



PERFORMANCE COMPARISON

#Devices CPU/GPUs	#Nodes CPU	Time CPU	#Nodes GPU	Time GPU	Speed-up
1	1	110.80 s	1	46.87 s	2.36
2	1	58.22 s	1	24.59 s	2.37
4	2	30.85 s	1	13.39 s	2.30
8	4	17.74 s	2	7.26 s	2.44
16	8	11.22 s	4	5.30 s	2.11
32	n/a	n/a	8	3.77 s	

CONCLUSIONS

- With OpenACC its possible to achieve good speedup with reasonable effort
- Changing data layout from SoA to AoS can improve performance
- Possible to improve scalability by overlapping
  - Communication with computation
  - Writing output while compute next timestep