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
Zonal Flow Solver is a CFD code supporting different solvers:

- Finite-Volume
- Discontinuous 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`

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

Parallelize II

Acceleration of most loops was straightforward:

```c
#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:

```c
#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

copyin
copyout
ACCELERATING ZFS WITH OPENACC

Parallelize IV - unstructured data regions

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

```c
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:

```c
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

```c
#pragma acc parallel loop gang vector(32)
for ( ZFSUint srfcId=0; srfcId<noSurfaces; srfcId++ )
{ ... }
```
OPTIMIZATIONS

Increasing cache hit rates with lower occupancy

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

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

Effect for computeSurfaceValues (hand tuned for all kernels)

<table>
<thead>
<tr>
<th>vector size</th>
<th>Time (ms)</th>
<th>Ach. Occ.</th>
<th>Tex$ Hit Rate</th>
<th>L2$ Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 (default)</td>
<td>3.68</td>
<td>0.45</td>
<td>62.08%</td>
<td>46.74%</td>
</tr>
<tr>
<td>128</td>
<td>4.07</td>
<td>0.48</td>
<td>63.13%</td>
<td>40.62%</td>
</tr>
<tr>
<td>32 (minimum)</td>
<td>2.18</td>
<td>0.25</td>
<td>68.78%</td>
<td>60.36%</td>
</tr>
</tbody>
</table>
## OPTIMIZATIONS

### Summary

<table>
<thead>
<tr>
<th>Time (%) (opt)</th>
<th>Calls</th>
<th>Avg. kernel time (no opt)</th>
<th>Avg. kernel time (opt)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.60%</td>
<td>3000</td>
<td>4.05 ms</td>
<td>3.75 ms</td>
<td>distributeFluxToCells</td>
</tr>
<tr>
<td>23.51%</td>
<td>500</td>
<td>27.22 ms</td>
<td>18.52 ms</td>
<td>computeSurfaceValues</td>
</tr>
<tr>
<td>15.49%</td>
<td>500</td>
<td>12.25 ms</td>
<td>12.20 ms</td>
<td>LSReconstructCellCenter</td>
</tr>
<tr>
<td>12.48%</td>
<td>500</td>
<td>9.82 ms</td>
<td>9.83 ms</td>
<td>viscousFlux</td>
</tr>
<tr>
<td>11.44%</td>
<td>500</td>
<td>9.00 ms</td>
<td>9.01 ms</td>
<td>Ausm</td>
</tr>
</tbody>
</table>
## PERFORMANCE ANALYSIS

**GPU Profile - contributions above 1%**

<table>
<thead>
<tr>
<th>Time (%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg.</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.60</td>
<td>11.263 s</td>
<td>3000</td>
<td>3.75 ms</td>
<td>distributeFluxToCells</td>
</tr>
<tr>
<td>23.51</td>
<td>9.260 s</td>
<td>500</td>
<td>18.52 ms</td>
<td>computeSurfaceValues</td>
</tr>
<tr>
<td>15.49</td>
<td>6.100 s</td>
<td>500</td>
<td>12.20 ms</td>
<td>LSReconstructCellCenter</td>
</tr>
<tr>
<td>12.48</td>
<td>4.915 s</td>
<td>500</td>
<td>9.83 ms</td>
<td>viscousFlux</td>
</tr>
<tr>
<td>11.44</td>
<td>4.504 s</td>
<td>500</td>
<td>9.01 ms</td>
<td>Ausm</td>
</tr>
<tr>
<td>2.50</td>
<td>984 ms</td>
<td>501</td>
<td>1.96 ms</td>
<td>computePrimitiveVariables</td>
</tr>
<tr>
<td>1.88</td>
<td>739 ms</td>
<td>500</td>
<td>1.48 ms</td>
<td>rungeKuttaStep</td>
</tr>
<tr>
<td>1.65</td>
<td>648 ms</td>
<td>500</td>
<td>1.30 ms</td>
<td>LSReconstructCellCenterII</td>
</tr>
<tr>
<td>1.64</td>
<td>644 ms</td>
<td>510</td>
<td>1.26 ms</td>
<td>CUDA memcpyDtoH+HtoD</td>
</tr>
</tbody>
</table>
PERFORMANCE ANALYSIS

<table>
<thead>
<tr>
<th>Name</th>
<th>DRAM read BW (GB/s)</th>
<th>DRAM write BW (GB/s)</th>
<th>DRAM BW (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributeFluxToCells</td>
<td>130.99</td>
<td>91.57</td>
<td>222.56</td>
</tr>
<tr>
<td>computeSurfaceValues</td>
<td>115.29</td>
<td>101.75</td>
<td>217.04</td>
</tr>
<tr>
<td>LSReconstructCellCenter</td>
<td>100.74</td>
<td>72.82</td>
<td>173.56</td>
</tr>
<tr>
<td>viscousFlux</td>
<td>178.42</td>
<td>52.14</td>
<td>230.56</td>
</tr>
<tr>
<td>Ausm</td>
<td>108.32</td>
<td>34.79</td>
<td>143.11</td>
</tr>
</tbody>
</table>

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

computeSurfaceValues - AoS vs. SoA

Data Accesses (AoS)

thread 0

v0 v1 v2 v3 v4

thread 1

v0 v1 v2 v3 v4

thread 2

v0 v1 v2 v3 v4

thread 3

v0 v1 v2 v3 v4

Cell 0

Cell 1

Cell 2

Only 8 bytes out of 32byte transaction used

Data Accesses (SoA)

t1 t2 t3 t4

v0 v0 v0 v0

v1 v1 v1 v1

v2 v2 v2 v2

v3 v3 v3 v3

v4 v4 v4 v4

C0 C1 C2 C3

32 bytes out of 32byte transaction used
## PERFORMANCE ANALYSIS

**computeSurfaceValues - AoS vs. SoA**

<table>
<thead>
<tr>
<th></th>
<th>SoA</th>
<th>AoS</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>1.4201 ms</td>
<td>2.1678 ms</td>
<td>1.53 x</td>
</tr>
<tr>
<td><strong>Achieved Occupancy</strong></td>
<td>35%</td>
<td>25%</td>
<td></td>
</tr>
<tr>
<td><strong>DRAM utilization level</strong></td>
<td>High (8)</td>
<td>High (8)</td>
<td></td>
</tr>
<tr>
<td><strong>DRAM BW (r+w)</strong></td>
<td>203.32</td>
<td>223.54</td>
<td></td>
</tr>
<tr>
<td><strong>Tex$ Hit Rate</strong></td>
<td>51.85%</td>
<td>68.92%</td>
<td></td>
</tr>
<tr>
<td><strong>L2$ Hit Rate</strong></td>
<td>55.72%</td>
<td>60.36%</td>
<td></td>
</tr>
<tr>
<td><strong>Texture Cache requests</strong></td>
<td>21605066</td>
<td>30004894</td>
<td>1.38 x</td>
</tr>
<tr>
<td><strong>Global st transactions/request</strong></td>
<td>2.50</td>
<td>20</td>
<td>8 x</td>
</tr>
</tbody>
</table>
CPU version is already MPI parallel

Multi GPU version straight forward using CUDA-aware MPI on device buffers:

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

2479888 cells
MULTI-GPU VERSION

Scalability - Strong Scaling - Output and Exchange

2479888 cells

![Graph showing scalability and strong scaling for output and exchange with multi-GPU version. The graph includes data points for exchange and output across different numbers of GPUs (log scale), with annotations for intranode and internode GPUDirect P2P.]
## PERFORMANCE COMPARISON

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