Efficient Multi-GPU CUDA Linear Solvers for OpenFOAM

Alexander Monakov, amonakov@ispras.ru

Institute for System Programming of Russian Academy of Sciences

March 20, 2013
In OpenFOAM, run time is usually dominated by time in linear solvers:

- Preconditioned conjugate gradient
- Algebraic multigrid

Mostly memory bound, can benefit from GPU acceleration

Our goal: develop efficient CUDA solvers for OpenFOAM

- Only conjugate gradient for now
- Focus on optimization techniques
- Multi-process and multi-GPU capable
Building a CUDA-MPI Solver in Three Steps

1. Single-threaded preconditioned conjugate gradient on the GPU
2. Multi-process OpenFOAM with a single GPU
3. Multi-node/Multi-GPU
Building a CUDA-MPI Solver in Three Steps

1. Single-threaded preconditioned conjugate gradient on the GPU

2. Multi-process OpenFOAM with a single GPU

3. Multi-node/Multi-GPU
Building a CUDA-MPI Solver in Three Steps

1. Single-threaded preconditioned conjugate gradient on the GPU
2. Multi-process OpenFOAM with a single GPU
3. Multi-node/Multi-GPU
Preconditioned conjugate gradient is:

1. BLAS-1 vector ops (dot product, linear combination)
   - CUBLAS

2. Sparse matrix – vector multiplication
   - CUSPARSE or custom SpMV library

3. Preconditioning
   - No obvious choice
Preconditioned conjugate gradient is:

1. BLAS-1 vector ops (dot product, linear combination)
   CUBLAS

2. Sparse matrix – vector multiplication
   CUSPARSE or custom SpMV library

3. Preconditioning
   No obvious choice
Preconditioned conjugate gradient is:

1. BLAS-1 vector ops (dot product, linear combination)
   - CUBLAS

2. Sparse matrix – vector multiplication
   - CUSPARSE or custom SpMV library

3. Preconditioning
   - No obvious choice
Preconditioned conjugate gradient is:

1. BLAS-1 vector ops (dot product, linear combination)  
   CUBLAS

2. Sparse matrix – vector multiplication  
   CUSPARSE or custom SpMV library

3. Preconditioning  
   No obvious choice
Consider:

- Both structured and unstructured grids are important
- Grid geometry defines the sparsity pattern
- Geometry is usually unchanging

This allows to:

- Use GPU sparse matrix format with complex setup phase
- Reduce PCIe uploads: update the coefficients only
Preconditioning

DILU: the usual choice in OpenFOAM
- Setup and application: two triangular solves
- Cheap, strong, inherently sequential
- Reordering can extract parallelism, but reduces strength

Diagonal
- Even cheaper: no setup, divide by diagonal coefficients to apply
- Parallel!
- Weak

Cheap, strong, parallel: pick any two
DILU: the usual choice in OpenFOAM

- Setup and application: two triangular solves
- Cheap, strong, inherently sequential
- Reordering can extract parallelism, but reduces strength

Diagonal

- Even cheaper: no setup, divide by diagonal coefficients to apply
- Parallel!
- Weak
Preconditioning

Cheap, strong, parallel: pick any two

DILU: the usual choice in OpenFOAM
- Setup and application: two triangular solves
- Cheap, strong, inherently sequential
- Reordering can extract parallelism, but reduces strength

Diagonal
- Even cheaper: no setup, divide by diagonal coefficients to apply
- Parallel!
- Weak
Approximate Inverse (AINV):

\[ W^T AZ = D + e \]

- Express approximation to the inverse matrix with two triangular factors
- Costly, complex setup with limited parallelism
- Applying: two SpMV operations
- Parameterized by *drop tolerance*
AINV setup phase drops factors’ coefficients below given threshold
A tradeoff between sparseness and approximation error

- $\text{droptol} \rightarrow 1$: diagonal preconditioner
  
  Cheap to apply, slow convergence

- $\text{droptol} \rightarrow 0$: exact inverse
  
  Slow (impractical), fast convergence

Need to find a sweet spot
By varying $\delta$ (drop tolerance), minimize

$$T_{\text{solution}}(\delta) = (T_{\text{precond}}(\delta) + T_{\text{rest}})N_{\text{iter}}(\delta)$$

Assuming that $T_{\text{solution}}$ is monotonic on both sides of global minimum, can use ternary search by $\delta$ to optimize it.

Ideally, want to update drop tolerance, not choose once, but tricky:

- Can’t compare time to solution or convergence rate for different systems
- Need to solve one system with two tolerance values in parallel and compare
Mitigating the AINV Setup Cost

Time to compute AINV factorization on the CPU $\approx$ time to solve a system on the GPU

Badly need to hide that cost

- Matrices on adjacent time-steps have similar coefficients
- The preconditioner from the previous time-step: almost as good as new
- Can reuse preconditioners to amortize the cost

Pushing that further:

- Use an asynchronous CPU worker to recompute preconditioners
- Solve the system with the last preconditioner produced by the worker
- Almost “free” preconditioners, except on first time-step
Mitigating the AINV Setup Cost

Time to compute AINV factorization on the CPU \( \approx \) time to solve a system on the GPU
Badly need to hide that cost

- Matrices on adjacent time-steps have similar coefficients
- The preconditioner from the previous time-step: almost as good as new
- Can reuse preconditioners to amortize the cost

Pushing that further:

- Use an asynchronous CPU worker to recompute preconditioners
- Solve the system with the last preconditioner produced by the worker
- Almost “free” preconditioners, except on first time-step
Time to compute AINV factorization on the CPU \(\approx\) time to solve a system on the GPU

Badly need to hide that cost

- Matrices on adjacent time-steps have similar coefficients
- The preconditioner from the previous time-step: almost as good as new
- Can reuse preconditioners to amortize the cost

Pushing that further:

- Use an asynchronous CPU worker to recompute preconditioners
- Solve the system with the last preconditioner produced by the worker
- Almost “free” preconditioners, except on first time-step
Flip AINV factors into upper-left triangular form
  Avoid bad CUDA load balancing in lower-triangular factor

Use single-precision floating point to store the preconditioner
  No need for precision, save memory bandwidth

Keep CUBLAS coefficients in device memory
  Do not ping-pong them over PCIe
Going Parallel: Multiple Cores, One GPU

\( N \) CPUs + 1 GPU: unusual, but important:

- For users: small-scale runs on their PC/laptop before submitting to a cluster
- For the implementor: a stepping stone between single-process and multi-node multi-GPU
Implementation:

- OpenFOAM invoked as $N$ MPI ranks
- Each rank determines sparsity pattern of its slice of the combined matrix (once)
- Rank 0 gathers sparsity pattern of the combined matrix
- To solve a system, rank 0 process:
  - Gathers non-zero coefficients, RHS, initial LHS
  - Updates preconditioners
  - Solves the system on GPU (no MPI communication inside the loop)
  - Distributes combined solution vector to other ranks
Already at this stage, CUDA-aware MPI with OS-assisted intra-node communication is needed:

- Send preconditioner data directly into GPU memory
- Gather matrix coefficients into GPU RAM

Need to adjust affinity of rank-0 preconditioner worker

Keeping track of basic performance figures helps (CPU time, GPU time, GPU memory bandwidth)
Multi-node Multi-GPU

We aim to efficiently implement solving when $N_{\text{GPUs}} < N_{\text{CPU cores}}$

- Run $N_{\text{CPU cores}}$ MPI ranks
- Each GPU is used from only one process
Multi-node Multi-GPU

Implementation

- Partition MPI ranks by GPUs
  Need hwloc integration to respect affinity
- Decide GPU leaders and build combined submatrices like in single-GPU case
- Build combined halo regions
- Changes in solver code:
  - CUBLAS reduction ops are followed by MPI global reduction
    Need support for MPI asynchronous reduction, in MVAPICH2
  - SpMV is preceded by MPI sends, followed by a small kernel to update boundary cells
Future Work

1. Comprehensive performance and scalability evaluation
2. Engage with OpenFOAM community
3. Pursue optimization opportunities
4. Other solvers: block-coupled solver, multigrid, or iterative methods with better convergence (IDR(s), RPM)