Unstructured-Grid CFD Algorithms on the NVIDIA Pascal and Volta Architectures

Mohammad Zubair and Jason Orender
Old Dominion University

Eric Nielsen and Aaron Walden
NASA Langley Research Center

Justin Luitjens
NVIDIA Corporation

John Wohlbier
DoD HPCMP PETTT
Engility Corporation

John Linford
ParaTools, Inc.
Collaboration across government, industry, and academia:

- NASA ARMD TTT/RCA, NASA Langley CDT and HPC Incubator efforts
- Old Dominion University
- NVIDIA and Portland Group
- Department of Defense HPCMP PETTT
- ParaTools, Inc. and University of Oregon
- ORNL GPU hackathons and others

- Attempted GPU acceleration of FUN3D kernels in the past, but limitations in the programming model and earlier hardware could not compete with traditional CPU technology

- Newer GPU hardware with improved memory bandwidth such as Pascal and Volta, coupled with an optimized CUDA approach, has now made GPU computing a compelling alternative
NASA Langley’s FUN3D solver is used to tackle some of the nation’s most complex aerodynamics problems.
FUN3D solves the Navier-Stokes equations of fluid dynamics using implicit time integration on general unstructured grids.

This approach gives rise to a large block-sparse system of linear equations that must be solved at each time step.

Two kernels are generally the largest contributors to runtime:
- Kernel 1: Construction and storage of the compressible viscous flux Jacobians
- Kernel 2: Multicolor point-implicit linear solver used to solve $Ax=b$

This overview focuses on CUDA implementations of these two kernels.
Initialize $A_{\text{DIAG}} \leftarrow 0$ and $A_{\text{OFF}} \leftarrow 0$

for each cell $\in$ Grid do
  Update $A_{\text{DIAG}}$ for nodes in cell
  Update $A_{\text{OFF}}$ for nodes in cell
end for

$A_{\text{DIAG}}$ : Diagonal block matrix
$A_{\text{OFF}}$ : Off-diagonal block-sparse matrix
$n$ : Number of cells in grid

- Updates to $A_{\text{DIAG}}$ and $A_{\text{OFF}}$ require several loops over the number of edges, faces, and nodes within the current cell

**Parallelization**: The computation can be parallelized over the number of cells, however; atomic updates are required to avoid race conditions when writing to $A_{\text{DIAG}}$ and $A_{\text{OFF}}$

**Challenges**: Traversal of the grid results in irregular memory accesses, complexities related to the underlying physics, and a large number of variables and temp arrays resulting in cache and register pressure
Kernel 1: Initial Implementation

- Assign a thread to a cell and use atomic updates for $A_{\text{DIAG}}$ and $A_{\text{OFF}}$
- Refactor code to minimize the number of variables and temp arrays
  - Also resulted in ~2x Xeon speedup
- Use shared memory for a few critical temp arrays

Issues: Each thread still uses many temp arrays and a large number of registers

Initialize $A_{\text{DIAG}} \leftarrow 0$ and $A_{\text{OFF}} \leftarrow 0$

for each cell $\in$ Grid do
  Update $A_{\text{DIAG}}$ for nodes in cell
  Update $A_{\text{OFF}}$ for nodes in cell
end for
Initialize $A_{\text{DIAG}} \leftarrow 0$ and $A_{\text{OFF}} \leftarrow 0$

for each cell $\in \text{Grid}$ do
  for each node $\in \text{cell}$ do
    // compute cell averages, set local arrays
  end for
  for each face $\in \text{cell}$ do
    // linearize cell gradients
  end for
  for each edge $\in \text{cell}$ do
    // compute edge contributions to Jacobian
      for each node $\in \text{cell}$ do
        // compute gradients at dual face
      end for
  end for
  for each node $\in \text{cell}$ do
    // assemble 17 contributions to Jacobian
  end for
end for

Parallelize across $\text{gridDim}.x \times \text{blockDim}.y$ threads

Parallelize using $\text{blockDim}.x$ threads

Flatten nested loops and parallelize using $\text{blockDim}.x$ threads

Parallelize using $\text{blockDim}.x$ threads

- Assign a CTA of $\text{blockDim}.x \times \text{blockDim}.y$ threads to process $\text{blockDim}.y$ cells
- Increases number of active threads and improves thread utilization
- Coalesce memory access pattern
- Reduces register and shared memory pressure increasing occupancy
- Enable reduction in inner loops using shared memory
- Auto-tuning used to choose $\text{blockDim}.x$ and $\text{blockDim}.y$
FUN3D uses a series of multicolor point-implicit sweeps to form an approximate solution to $Ax = b$

- Color by rows which share no adjacent unknowns; and re-order matrix rows by color contiguously
- Unknowns of the same color carry no data dependency and may be updated in parallel
- Updates of unknowns of each color use the latest updated values at grid points corresponding to other colors
- The overall process may be repeated using several outer sweeps over the entire system
• With an average of few off-diagonal blocks per row, the arithmetic intensity of the computation is quite low ($\approx 0.5$) – memory bound on GPU

• The number of rows associated with a color can vary significantly. Consequently the amount of parallelism available for different colors varies significantly

• To support strong scalability, the single node performance for light workload should also be good

• The solver uses indirect memory addressing

• The implementation must support a broad range of block sizes
Fig. 2: Figure (a) shows the sparsity structure of a matrix $O$. An entry $\times$ indicates a non-zero block. Figure (b) shows $O$ for a block size of $2 \times 2$.

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
13 & 15 & 7 & 15 \\
14 & 15 & 8 & 20 \\
21 & 22 & 0 & 0 \\
22 & 23 & 0 & 0
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
13 & 15 & 7 & 15 \\
14 & 15 & 8 & 20 \\
21 & 22 & 0 & 0 \\
22 & 23 & 0 & 0
\end{array}
\]

\[i_o = [1, 3, 4, 6, 7]\]
\[j_o = [3, 4, 3, 1, 2, 1]\]
\[O = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]\]

Fig. 3: CSR storage for the matrix of Figure 2.
Kernel 2: Naive Single GPU Algorithm

To solve $Ax = b$:

Initialize $x \leftarrow 0$

for $i = 1$ to number_of_sweeps do
  for $k = 1$ to number_of_colors do
    Compute $q^k \leftarrow b^k - A_{OFF}^k x$
    Solve for $y^k$ in $L_D^k y^k = q^k$
    Solve for $x^k$ in $U_D^k x^k = y^k$
  end for
end while

Assign a thread to process a row block. As all rows can be processed independently the parallelism is determined by number of row blocks.

$x$ : Solution vector  
$L_D^k$ : Lower triangular of $A_{DIAG}^k$  
$U_D^k$ : Upper triangular of $A_{DIAG}^k$
kernel 2: use of blas

initialize $x \leftarrow 0$

for $i = 1$ to number_of_sweeps do
  for $k = 1$ to number_of_colors do
    compute $q^k \leftarrow b^k - A_{OFF}^k x$
    solve for $y^k$ in $L_D^k y^k = q^k$
    solve for $x^k$ in $U_D^k x^k = y^k$
  end for
end while

- experiments show that the performance of the BLAS functions available in existing CUDA libraries is suboptimal for matrices representative of those encountered in actual simulations
- instead, optimized versions of these functions are developed$^1$

• An approach is developed for a broad range of block sizes $nb$; here we focus on $1 \leq nb \leq 16$
Consider an example of a block-sparse matrix with a block size of 4:

- A single warp is used to process a row of the matrix in a loop, where $nbk = 2$ consecutive non-zero blocks are processed by the warp at each iteration.
- The warp handles 32 $(2 \times (4 \times 4))$ matrix entries during each iteration – allowing a single warp to load the elements of the matrix in a coalesced fashion.
- The appropriate elements of $x$ are also loaded from the read-only data cache, multiplied by the corresponding elements of $A_{OFF}$, and the results are accumulated.
- After completion of the loop, the partial results are stored in shared memory to be aggregated later.
• The columns of the lower triangular factor of $A_{\text{DIAG}}$ are processed from left to right using a single warp.

• The amount of parallelism available to the warp decreases as we move from left to right.

• Shuffle instruction broadcasts the value from the previous column.

• Upper triangular portion processed in a similar fashion.

\[
\begin{bmatrix}
    x \\
    x & x \\
    x & x & x \\
    x & x & x & x \\
    x & x & x & x & x
\end{bmatrix}
\]
ELL-S (Variant of ELLPACK): Sort the rows by the number of non-zeros and then make groups of 32 consecutive rows and pad them to have equal number of non-zeros. Increase in number of non-zeros < 1%

A group of 32 rows with 4 non-zeros in a row. For a matrix with a single group with max non-zeros as 4, nzs = 4.

Another example, where there are two groups, one group with max non-zeros as 4, and the other group with max non-zeros as 6, the value of nzs = 4 + 6 = 10.

a_offELL(32, 5, 5, nzs)
Kernel 2: Implementation
Using Variant of ELLPACK

Performance of ELL-S Storage Scheme Vs Block CSR for Kernel 2 (Solver)

- With ELL-S we observed a 1.32x improvement compared to block-CSR on Pascal P100. No significant improvement on K40.
- On Volta V100, block-CSR is already achieving close to peak memory bandwidth.
- Additionally, it is not clear the impact of new storage on Kernel 1. For now our integrated kernel is based on block-CSR.

Algorithm: Assign a warp for a group of 32 rows

```
    | t1 process row 1 |
   +------------------+
   | t2 process row 2 |
   +------------------+
   | t32 process row 32 |
```

```
    do i = iamr(qid), iamr(qid+1)-1
    do j = 1, 5
        dqs = dq(j, iamr(xid))
        x1 = x1 - a_offELL(lrow,1,j,i)*dqs
        x2 = x2 - a_offELL(lrow,2,j,i)*dqs
        x3 = x3 - a_offELL(lrow,3,j,i)*dqs
        x4 = x4 - a_offELL(lrow,4,j,i)*dqs
        x5 = x5 - a_offELL(lrow,5,j,i)*dqs
    end do
    xid = xid+32
```

Kernel 2: Multi GPU Algorithm

\[ x \]: Solution vector (initialize to zero)
\[ L_D^k \]: Lower triangular of \( A_{DIAG}^k \)
\[ U_D^k \]: Upper triangular of \( A_{DIAG}^k \)

\[
\text{for } i = 1 \text{ to number_of_sweeps do} \\
\text{for } k = 1 \text{ to number_of_colors do} \\
\quad \text{// Compute halo values} \\
\quad \text{Compute } q^k \leftarrow b^k - A_{OFF}^k x \\
\quad \text{Solve for } y^k \text{ in } L_D^k y^k = q^k \\
\quad \text{Solve for } x^k \text{ in } U_D^k x^k = y^k \\
\quad \text{// Non-blocking MPI send/rec halo} \\
\quad \text{// Compute interior values} \\
\quad \text{Compute } q^k \leftarrow b^k - A_{OFF}^k x \\
\quad \text{Solve for } y^k \text{ in } L_D^k y^k = q^k \\
\quad \text{Solve for } x^k \text{ in } U_D^k x^k = y^k \\
\quad \text{// MPI_Waitall} \\
\text{end for} \\
\text{end for}
\]

- Halo rows ordered and processed first, then call non-blocking MPI send/receive for halos
- Computation of interior values proceeds as halos are exchanged
- Blocking MPI_Waitall follows interior computation
- Strong scaling heavily dependent upon interior computation effectively hiding comm. latency
Fun3D Scalability on DGX1V

- **1M**
  - Iterations/Second: 40
  - Num GPUs: 1, 2, 4, 8, 16, 32, 64

- **3.6M**
  - Iterations/Second: 25
  - Num GPUs: 1, 2, 4, 8, 16, 32, 64

- **7M**
  - Iterations/Second: 20
  - Num GPUs: 1, 2, 4, 8, 16, 32, 64
Fun3D - Throughput - 3.6M

Node Configurations
- NASA Pleiades: Broadwell
- NASA Electra: Skylake
- ORNL Summit-Dev: P100
- DGX1-V: V100
Fun3D - Scalability

Number of GPUs (P100 or V100)

- DGX1V-3.6M
- DGX1V-7M
- SumDev-14.6M
- SumDev-60.7M
Attempted GPU acceleration of FUN3D kernels in the past, but limitations in the programming model and earlier hardware could not compete with traditional CPU technology.

Newer GPU hardware with improved memory bandwidth such as Pascal and Volta, coupled with an optimized CUDA approach, has now made GPU computing a compelling alternative.

Further development and optimization is currently being performed on Summit.