High-Order Spectral Difference: Validation and Acceleration using GPU Computing

Ben Zimmerman
M.S. Student
Iowa State University
Spectral Difference

- Overview
  - Brief Introduction
  - 1-D Implementation
  - CUDA Implementation
  - CUDA Validation
  - CUDA Acceleration
  - Conclusions
Introduction

- **SD3DN**
  - Spectral Difference 3-D Navier-Stokes Solver
  - High-order
  - Unstructured hexahedral elements
    - Hexahedral cells offer high efficiency
  - Solves equations in differential form
    - No surface / volume integrals
  - Implemented in:
    - MPI FORTRAN
    - CUDA C++
1-D Formulation

• Consider the 1-D wave equation:

\[
\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \quad f(u) = cu \quad c > 0
\]

• Discuss 3\textsuperscript{rd} order spacial discretization

• First assume solution takes the following form:

\[
u(x) = C_0 + C_1 x + C_2 x^2 \quad \rightarrow \quad 3 \text{ points}
\]
1-D Formulation

\[ \frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \]

- From before we have:
  \[ u \in P^2 \quad \quad f(u) \in P^3 \]
- To build flux polynomial → 4 points needed
- We use Gauss and Gauss-Lobatto Points
1-D Formulation

• How do we build the flux?

• Lagrange Polynomials (use flux point locations)

\[ L_1(x) = \frac{(x - \tilde{x}_{i_2})(x - \tilde{x}_{i_3})(x - \tilde{x}_{i_4})}{(\tilde{x}_{i_1} - \tilde{x}_{i_2})(\tilde{x}_{i_1} - \tilde{x}_{i_3})(\tilde{x}_{i_1} - \tilde{x}_{i_4})} \]

\[ L_2(x) = \frac{(x - \tilde{x}_{i_1})(x - \tilde{x}_{i_3})(x - \tilde{x}_{i_4})}{(\tilde{x}_{i_2} - \tilde{x}_{i_1})(\tilde{x}_{i_2} - \tilde{x}_{i_3})(\tilde{x}_{i_2} - \tilde{x}_{i_4})} \]
1-D Formulation

- The Flux polynomial becomes:

\[ f(x) = f_{i1}L_1(x) + f_{i2}L_2(x) + f_{i3}L_3(x) + f_{i4}L_4(x) \]

- We require the derivative and flux values:

\[ \frac{\partial f(x)}{\partial x} = f_{i1}L_1'(x) + f_{i2}L_2'(x) + f_{i3}L_3'(x) + f_{i4}L_4'(x) \]
1-D Formulation

- Spectral Difference Steps
  - Get solution \((u)\) at flux points
  - Evaluate interior flux using \(f(u)\)
  - Set end point flux to common Riemann flux
  - Get flux derivative at solution points
  - Update solution at solution points

\[ u(x) = C_0 + C_1 x + C_2 x^2 \]
CUDA Implementation

• What is CUDA?
  – NVIDIA GPU Computing
    • Compute Unified Device Architecture
  – Massive parallelism of programs
  – Hundreds of “CUDA cores”
  – CUDA functions (“kernels”) contain:
    • Grid
    • Blocks
    • Threads
CUDA Implementation

- Function ("kernal") → Grid
- This Grid contains blocks
  - blockIdx.x
  - blockIdx.y
- Each block contains threads
  - threadIdx.x
  - threadIdx.y
  - threadIdx.z

From NVIDIA Programming Guide
CUDA Implementation

- Memory Importance
  - Global
    - Copied from CPU to GPU and used in calculations
    - Not very fast
  - Texture
    - Bound in CPU code
    - Allows for fast reads
  - Shared
    - Allocated within GPU code
    - 150x faster than global memory
CUDA Implementation

- **1-D Case:**
  - evaluate derivative of flux
    - Gauss points
CUDA Implementation

• Example: Calculate flux derivative:

\[ \frac{\partial f(x)}{\partial x} = f_{i1} L'_1(x) + f_{i2} L'_2(x) + f_{i3} L'_3(x) + f_{i4} L'_4(x) \]

• Let:
  - sp Current solution point
  - fp Current flux point
  - nsp Total number of solution points per cell
  - cell Current cell
  - nc Total number of cells
  - L Array of Lagrange coefficients [#sp * (nsp+1)]
  - F Array of flux values [#cells * (nsp+1)]
CUDA Implementation

\[
\frac{\partial f(x)}{\partial x} = f_{i1} L_1'(x) + f_{i2} L_2'(x) + f_{i3} L_3'(x) + f_{i4} L_4'(x)
\]

- **CPU Code**

```c
for (int cell=0; cell<nc; cell++) {
    //Loop over cells

    for (int sp=0; sp<nsp; sp++) {
        //Loop over solution points
        dfdx[sp] = 0.0;

        for (int fp=0; fp<nsp+1; fp++) {
            //Loop over flux points
            //Flux derivative @ solution point per cell
            dfdx[sp] += F[fp + cell*(nsp+1)] * L[fp + sp*(nsp+1)];
        }
    //End loop over flux points

    dFdX[sp + cell*nsp] = dfdx[sp];  //Save to memory
} //End loop over solution points
} //End loop over Cells
```

- **Requires 2 nested loops on CPU for 1-D**
  - How can we transfer this to CUDA?
CUDA Implementation

\[ \frac{\partial f(x)}{\partial x} = f_{i1} L'_1(x) + f_{i2} L'_2(x) + f_{i3} L'_3(x) + f_{i4} L'_4(x) \]

• GPU Code

```c
int sp = threadIdx.x; //Solution Point
int cell = blockIdx.x; //Current working Cell

double dFdX[nsp*nc];

dFdX[sp + cell*nsp] = 0.0;

for (int fp=0; fp<nsp+1; fp++) { //Loop over flux points
    int id=sp + cell*nsp;
    //Get derivative at solution points
    dFdX[id] += F[fp + cell*(nsp+1)] * L[fp + sp*(nsp+1)];
}
```

• Number of Blocks
  - nc

• Number of Threads
  - nsp

• Only one loop!
  - But what is going on?
CUDA Implementation

```c
for (int fp=0; fp<nsp+1; fp++) {
    int id=sp + cell*nsp;
    //Get derivative at solution points
    dFdX[id] += F[fp + cell*(nsp+1)] * L[fp + sp*(nsp+1)];
}
```

- Recall each cell is a block and nsp threads per block

- Uses global memory
CUDA Implementation

```
int sp = threadIdx.x;     //Solution Point
int fp = threadIdx.y;     //Flux point
int cell = blockIdx.x;    //Cell
int2 ii;                  //Dummy Variable

__shared__ double dfdx[nsp];   //dflux Shared memory
__shared__ double Ls[nsp*(nsp+1)]; //Lagrange Shared memory
__shared__ double Fs[nsp+1];    //flux Shared memory

int idx = fp + sp*(nsp+1);    //Index to read Lagrange from textured memory
int idy = fp + cell*(nsp+1);  //Index to read flux from textured memory

//Fetch data from textured memory
ii = tex1Dfetch(t_L, idx); Ls[idx] = __hiloint2double(ii.y, ii.x);
ii = tex1Dfetch(t_F, idy); Fs[fp] = __hiloint2double(ii.y, ii.x);
__syncthreads(); //Ensure all data is loaded into Shared memory

if (fp == 0) {             //Lock down threads (do not need them)
  dfdx[sp] = 0.0;
}

for (int m=0; m<nsp+1; m++) { //Loop over flux points
  dfdx[sp] += Fs[m] * Ls[m + sp*(nsp+1)];
}
```
CUDA Implementation

• What have we gained?
  – We more than doubled our lines of code
  – But...
    • Code 1 kernal launch time (global memory):
      – 8.412 milliseconds
    • Code 2 kernal launch time (shared+texture memory):
      – 3.917 milliseconds
  – Double performance per iteration!
    • Performed on NVIDIA GEFORCE GT 630M
CUDA Validation

- Isentropic Vortex
  - 2-D case
  - Motion in X
  - Two studies
    - H-refinement
    - P-refinement

5th Order Vortex Run
CUDA Validation

- **H-refinement**
  - 4 Grids
    - 10x10 – 15x15
    - 20x20 – 25x25

<table>
<thead>
<tr>
<th>Order</th>
<th>L Inf Slope</th>
<th>L1 Slope</th>
<th>L2 Slope</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.94</td>
<td>3.96</td>
<td>3.89</td>
</tr>
<tr>
<td>5</td>
<td>4.76</td>
<td>4.98</td>
<td>5.03</td>
</tr>
<tr>
<td>6</td>
<td>5.88</td>
<td>6.08</td>
<td>6.09</td>
</tr>
</tbody>
</table>

4th order refinement
CUDA Validation

- P-refinement
  - Studied on coarse grid (10x10)
  - Polynomial increased to degree 10
  - Exponential decay of error
CUDA Validation

- Couette Flow
  - 3-D case
  - Flow between two parallel plates
  - Viscous validation
  - P-refinement study

Velocity Profile
CUDA Validation

- **P-refinement**
  - Studied on coarse grid (2x2x1)
  - Order increased to 9th
  - Exponential decay of error
CUDA Validation

- Pulse and cylinder – 2074 cells

\[ P = P_{\text{inf}} + \epsilon e^{\ln(-2)\left(\frac{x^2+y^2}{r\theta}\right)} \]
CUDA Validation

- Pressure taken at 3 points
  - Compare with analytic solutions
CUDA Validation
CUDA Validation

Point B

Time
CUDA Validation

Point C

Graph showing data points for different orders of accuracy:
- **Exact**
- **2nd Order**
- **3rd Order**
- **4th Order**
- **5th Order**

Time axis ranges from 8 to 10.
CUDA Validation

- SD7003 Wing
  - 60,000 Re
  - Mach 0.1 at 4 degree AoA
  - Comparison with FDL3DI
  - 293,590 cells

<table>
<thead>
<tr>
<th>Order</th>
<th># DOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd</td>
<td>2,348,720</td>
</tr>
<tr>
<td>3rd</td>
<td>7,926,930</td>
</tr>
</tbody>
</table>
CUDA Validation

Mean U velocity profile (Data at every 0.1c)

FDL3DI

SD3DN

IOWA STATE UNIVERSITY
Department of Aerospace Engineering
CUDA Validation

Mean Pressure Coefficient

Mean Friction Coefficient

FDL3DI  
SD3DN 3rd  
SD3DN 2nd
CUDA Acceleration

• CUDA is fast
  - 10x, 20x, 30x, 40x

• One-to-one
  - # GPU's = # CPU's
  - Use all resources
    • All cores on CPU

• Case I
  - Propagating vortex
  - CPU (8 cores)
    - Xeon 2.27GHz
  - GPU (240 cores)
    - Tesla C1060
CUDA Acceleration

CPU run

GPU run

- 5\textsuperscript{th} order (28,125 DOF)
  - Roughly 13x faster
CUDA Acceleration

- **Case II**
  - Acoustic Pulse

- **CPU**
  - Xeon 2.27GHz
  - 5th order → 3.55 hrs

- **GPU**
  - GTX 550TI
    - 192 CUDA Cores
  - Tesla C2075
    - 448 CUDA Cores
CUDA Acceleration

• 4\textsuperscript{th} order
CUDA Acceleration

- 4\(^{th}\) order
### CUDA Acceleration

<table>
<thead>
<tr>
<th>Type/Order</th>
<th>2(^{nd}) Order</th>
<th>3(^{rd}) Order</th>
<th>4(^{th}) Order</th>
<th>5(^{th}) Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU (GTX)</strong></td>
<td>0.0082766</td>
<td>0.018596</td>
<td>0.041478</td>
<td>0.07455</td>
</tr>
<tr>
<td><strong>GPU (Tesla)</strong></td>
<td>0.0083267</td>
<td>0.019119</td>
<td>0.0321213</td>
<td>0.066518</td>
</tr>
<tr>
<td><strong>CPU (8 Cores)</strong></td>
<td>0.205342</td>
<td>0.57619</td>
<td>1.23375</td>
<td>2.34552</td>
</tr>
<tr>
<td><strong>Speed Up (GTX)</strong></td>
<td>24.81 x</td>
<td>30.98 x</td>
<td>29.74 x</td>
<td>31.46 x</td>
</tr>
<tr>
<td><strong>Speed Up (Tesla)</strong></td>
<td>24.66 x</td>
<td>30.14 x</td>
<td>38.41 x</td>
<td>35.26 x</td>
</tr>
</tbody>
</table>

*Timings in seconds / iteration*
CUDA Acceleration

- **Case III**
  - SD7003 airfoil
- **CPU**
  - Xeon 2.27GHz
  - 4 CPUs → 32 cores
- **GPU**
  - 4 Tesla C2075

<table>
<thead>
<tr>
<th>Order</th>
<th># DOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd</td>
<td>2,348,720</td>
</tr>
<tr>
<td>3rd</td>
<td>7,926,930</td>
</tr>
</tbody>
</table>
# CUDA Acceleration

Q-Criterion
4 NVIDIA Tesla 3\textsuperscript{rd} order
8 degree AoA

<table>
<thead>
<tr>
<th>Order</th>
<th>Required Steps</th>
<th>CPU Total Time</th>
<th>GPU Total Time</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2\textsuperscript{nd} Order</td>
<td>400,000</td>
<td>607.78 hrs (25 days)</td>
<td>27.86 hrs</td>
<td>21.81x</td>
</tr>
<tr>
<td>3\textsuperscript{rd} Order</td>
<td>800,000</td>
<td>3483.29 hrs (145 days)</td>
<td>137.14 hrs (\sim6 days)</td>
<td>25.4x</td>
</tr>
</tbody>
</table>
Conclusions

• GPU CFD Computing
  – Disadvantages
    • Limited by GPU memory
    • Rewrite from scratch → Best performance
    • Copy data to GPU is slow
  – Advantages
    • Huge increase in processing power over CPUs
    • Can out-preform CPU servers
      – Saving time, space, and money
    • Compiler flags to optimize CPU codes for GPU
Questions?