Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2013

The Efficient Implementation of Correction Procedure via Reconstruction with GPU Computing

Ben James Zimmerman

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Aerospace Engineering Commons

Recommended Citation

Zimmerman, Ben James, "The Efficient Implementation of Correction Procedure via Reconstruction with GPU Computing" (2013). *Graduate Theses and Dissertations*. 13102.
https://lib.dr.iastate.edu/etd/13102

The efficient implementation of

correction procedure via reconstruction with GPU computing

by

Ben J. Zimmerman

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Aerospace Engineering

Program of Study Committee:

Zhi J. Wang, Major Professor

Alric P. Rothmayer

Amber K. Mitra

Iowa State University

Ames, Iowa

2013

DEDICATION

To my Mother and Father,

whose support made this possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

ABSTRACT

Computational fluid dynamics (CFD) has long been a useful tool to model fluid flow problems across many engineering disciplines, and while problem size, complexity, and difficulty continue to expand, the demands for robustness and accuracy grow. Furthermore, generating high-order accurate solutions has escalated the required computational resources, and as problems continue to increase in complexity, so will computational needs such as memory requirements and calculation time for accurate flow field prediction. To improve upon computational time, vast amounts of computational power and resources are employed, but even over dozens to hundreds of central processing units (CPUs), the required computational time to formulate solutions can be weeks, months, or longer, which is particularly true when generating high-order accurate solutions over large computational domains. One response to lower the computational time for CFD problems is to implement graphical processing units (GPUs) with current CFD solvers. GPUs have illustrated the ability to solve problems orders of magnitude faster than their CPU counterparts with identical accuracy. The goal of the presented work is to combine a CFD solver and GPU computing with the intent to solve complex problems at a high-order of accuracy while lowering the computational time required to generate the solution. The CFD solver should have high-order spacial capabilities to evaluate small fluctuations and fluid structures not generally captured by lower-order methods (2nd and 1st order) and be efficient for the GPU architecture. This research combines the high-order Correction Procedure via Reconstruction (CPR) method with compute unified device architecture (CUDA) from NVIDIA to reach these goals. In addition, the study demonstrates accuracy of the developed solver by comparing results with other solvers and exact solutions. Solving CFD problems accurately and quickly are two factors to consider for the next generation of solvers. GPU computing is a step forward for the CFD community in solving both current and up-coming problems fast and with high accuracy.

CHAPTER 1.   INTRODUCTION

Computational fluid dynamics (CFD) is widely used throughout engineering and science disciplines for fluid flow evaluation and analysis. The use of low-order methods is popular in industrial settings due to robustness of the methods, but they are less accurate and can require large amounts of grid points to reach a set error criteria. These grids continue to exponentially grow as the problems to be solved become increasingly complex in terms of geometry and flow resolution requirements, increasing the computational cost. High-order methods (higher than 2nd order) are more accurate and can reach a set error criteria faster than lower order methods [1], but are more complicated and not as robust. These methods are a necessity, however, when considering aeroacoustic problems, where the numerical dissipation associated with low-order methods is infeasible for evaluation and a high number of grid points coupled with small time-steps are demanded when simulating [28]. Furthermore, complicated fluid structures in flows are uncapturable by low-order schemes unless high grid resolution is employed in specific areas within the domain. Interest is apparent in continued development of high-order methods to improve on robustness, efficiency, and implementation. In particular, the efficiency and implementation aspects of high-order methods is considered in this work.

Running CFD solvers is traditionally completed on servers of central processing units (CPUs). When complicated geometries or high-resolution requirements demand large numbers of grid points, the computational cost cannot be ignored. Large problems, even when partitioned and ran across multiple processors, can require a significant amount of computational time to complete while consuming considerable amounts of CPU resources. The current work focuses on implementation with graphic processing units (GPUs) to calculate said problems quickly. Recently, interests have shifted to NVIDIA's compute unified device architecture (CUDA). CUDA has already shown promising results when applied to aerospace sciences

[27, 13, 8], achieving considerable speed-ups when compared to existing CPU codes. The results indicate that large scale problems, which required CPU servers to generate solutions, can be completed with GPU workstations, consuming less power and computational resources while generating the solution in a comparable, or even faster time frame. Implementing a solver efficiently with GPU CUDA computing to achieve similar results is the goal of the current work.

The high-order method for CUDA implementation should be compact and efficient for use with GPU architecture. The industry standard finite volume (FV) method [2, 9] is robust and easy to implement, but the solution reconstruction is not local. It involves a least-squares formulation using neighboring cell data, and since each unknown has a unique stencil, the least squares inversion must be either completed every time step or stored. Applying high-order accuracy to the method implies that completing the inversion every step will consume large amounts of computational time, whereas storing the data results in a large memory requirement. Hence, a method whose solution reconstruction is completed locally per cell is preferable. Recently, the three-dimensional correction procedure via reconstruction (CPR) method has been developed [32] for mixed grids, including tetrahedrons, triangular prism, and even more recently, hexahedrals. The CPR method was developed to improved the efficiency of other high-order methods, including discontinuous Galerkin (DG) [4, 5, 7], spectral volume (SV) [30, 17], staggered grid (SG) multi-domain [15, 14], and spectral difference (SD) [16, 25] methods. Additionally, it unified all these methods [29]. Due to the efficiency of the CPR method, it is chosen for three-dimensional implementation with GPU CUDA computing. Element types are also considered for CUDA implementation. Triangular cells for the two-dimensional CPR method is currently implemented with CUDA [11] where exceptional increase in performance is demonstrated when compared to the CPU implementation. Operations across quadrilateral cells are more efficient, however, because the operations are completed in a one-dimensional manner. Thus, when investigating elements for three-dimensional efficiency, hexahedral cells are the obvious choice. In addition, when compared to other elements, such as tetrahedral cells, hexahedrals have illustrated higher efficiency and accuracy for viscous boundary layers [24]. Therefore, this thesis is focused on the implementation of the CPR method with hexahedral

cells.

The thesis is composed as follows, Chapter 2 covers GPU CUDA computing, explaining both GPU architecture and code optimization. Chapter 3 derives the CPR method, applies high-order elements, and outlines the Riemann solvers and time-stepping implemented. Chapter 4 discusses the implementation of the CPR method to the GPU. Chapter 5 covers results of the GPU code, and Chapter 6 draws the conclusions and outlines potential future work.

CHAPTER 2.   GPU CUDA COMPUTING

Graphic processing units (GPUs) were mostly used for graphics acceleration, calculating images shown on a computer screen. Recently, GPUs have shown the ability to tackle more general problems at much faster computing speeds than its central processing unit (CPU) counterpart. In 2006, NVIDIA released compute unified device architecture, or CUDA, for a few of their cards [19]. Since then, NVIDIA has continued to update their GPU hardware and CUDA capabilities, enabling CUDA to handle larger problems and generate solutions extremely fast. This chapter focuses on the GPU computing architecture and the optimizations to consider when implementing. Section 2.1 covers the architecture, while Section 2.2 discusses the optimization strategy.

2.1   GPU architecture

The architecture and capabilities of GPUs varies from card to card. Older GPUs (such as the GeForce GT and 8000 series) only support single precision computing and have limited memory. Newer cards (such as the Tesla C2070) contain over 10 times the memory as some older cards and support double precision. Another aspect is the shared multi-processor (SM) count. A GPUs SM count determines how many tasks the GPU can preform at once. Figure 2.1 illustrates the importance of SMs. The GPU on the left, with 2 SMs, can only run computations on two of the tasks in parallel, while the GPU on the right can run four tasks in parallel. Hence, a GPU with more SMs will complete a problem faster than a GPU with less SMs.

The GPU is composed of grids, blocks, and threads. When a GPU function (called a kernel) is launched and executed by CUDA threads, the GPU forms a grid. The grid is either one or two-dimensions, composed of blocks, while the blocks are one, two, or three-dimensions,
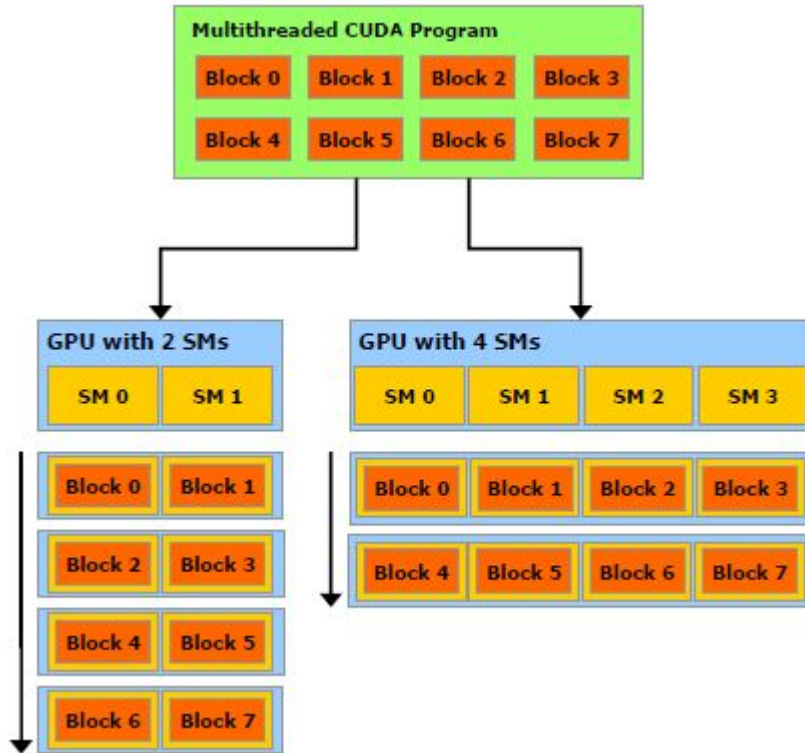
Figure 2.1   CUDA cards and shared multiprocessors [19]

composed of threads. The number of threads allowed per block is 512 on older cards, and 1024 on newer cards. As an example of the GPU architecture, consider figure 2.2. The grid is two-dimensional, (3 x 2), and contains a total of six blocks. Each block is also two-dimensional, (4 x 3), and contains twelve total threads. Total threads for the CUDA grid number the threads per block times the total blocks, or 72 in the example. Each thread and block have unique identification which can be accessed within the kernels by built-in threadIdx and blockIdx variables with a $.x$, $.y$, or $.z$ extension for each of the three dimensions. Additionally, the variables blockDim and gridDim access the dimension of the blocks and grids. In the example from the figure, gridDim.$x = 3$, gridDim.$y = 2$, blockDim.$x = 4$, blockDim.$y = 3$, blockIdx.$x = 0, 1, 2$, blockIdx.$y = 0, 1$, threadIdx.$x = 0, 1, 2, 3$, and threadIdx.$y = 0, 1, 2$ (thread indexing starts at 0). For an example of threads and blocks applied in a CUDA kernel, see Appendix A.
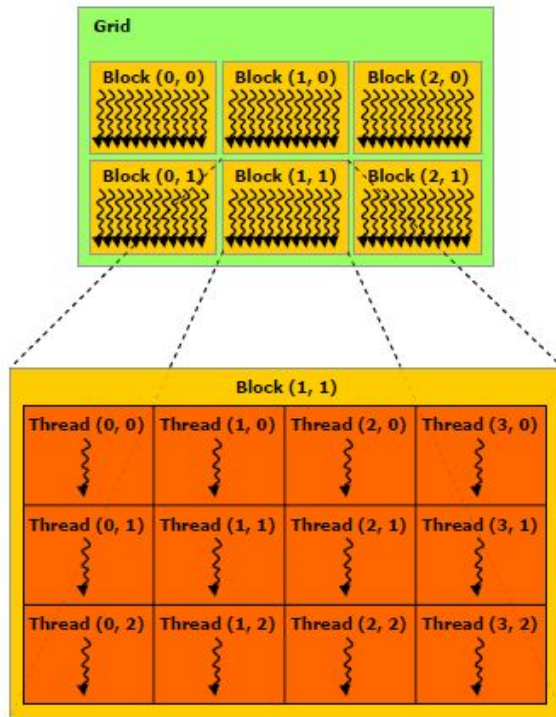
Figure 2.2   CUDA threads and blocks [19]

2.2   GPU optimization

Optimizing the CUDA application is extremely important, and will enable the GPU to achieve best performance from the code. First, GPU memory access and memory transfer must be coalesced and minimized. To ensure coalesced memory access, neighboring threads access neighboring cells in memory, which allows fast memory access. If neighboring threads access misaligned or scattered cells in memory, non-coalesced access can occur, and decayed performance will be seen. Coalesced memory access is a high priority memory optimization [18] and is imperative for an optimized CUDA program. Minimizing memory transfer is equally important, since copying memory to and from the GPU is computationally expensive. Thus, all required data for calculations are transfered into the GPU memory before starting any GPU calculations. Once the necessary data is transfered, GPU calculations begin and data transfer back to the CPU is minimized (the residual is transfered to the CPU and monitored rarely). Only after the calculations are complete is all the necessary memory transfered back to the

CPU for post-processing procedures.

The GPU contains multiple types of memory. Four important memory types for optimization are global, textured, local, and shared. Global memory can be accessed by all threads and is bounded to textured memory, but computations in global memory are slow. In addition, global memory requires coalesced memory access for best performance, hence, global memory is only accessed at the end of a GPU kernel for data transfer, so computations can be completed in other kernels on this data. Once global memory is updated, the corresponding bounded texture memory is updated, allowing for texture memory access on the data in later kernels. Texture memory is cached and read only [18] allowing fast memory access within a GPU kernel, even if the read is not coalesced, hence optimal performance is achieved when reading from textured memory instead of global memory. The next memory type is local, which is local to the thread, and access is as expensive as global memory [18], but computations are fast. Storage into local memory is ideal when a thread will access the same location of local memory storage in a later computation. The final memory is shared memory, which is the most important memory for optimization purposes, as proper use of this memory can drastically improve performance. Shared memory should be implemented when data is required by many threads within the same block, or when data needs re-ordering for coalesced access. Additionally, shared memory has a lifetime of the block, meaning allocation of this memory is done at a per block basis, whereas textured and global memory lifetime is the application itself, and local memory has the lifetime of a thread.

The proper use of GPU memory is key to developing CUDA applications. The performance differences between optimized and unoptimized code is apparent, and is explored in Chapter 5. Correct use of the memory types explained are illustrated in Appendix A and throughout Chapter 4.

CHAPTER 3. HIGH-ORDER CPR METHOD

This chapter explains the high-order Correction Procedure via Reconstruction (CPR) formulation. Section 3.1 derives the CPR method, Section 3.2 covers the Riemann solvers implemented, and Section 3.3 shows the time integration scheme.

3.1 Correction procedure via reconstruction

The CPR method combines high-accuracy and compactness while retaining the efficiency and simplicity of the finite difference method [31]. Section 3.1.1 derives the CPR method for both the inviscid and viscous flux, and Section 3.1.2 covers the extension of the method to high-order elements.

3.1.1 CPR formulation

This section is broken into two subsections. The first subsection will derive the CPR method for the inviscid flux, or Euler equations. Then in the second subsection the CPR method for the viscous flux, or Navier-Stokes equations, will be derived.

Inviscid flux

Consider the hyperbolic equation law given by,

$$\frac{\partial Q}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q) = 0, \tag{3.1.1}$$

where Q is the state vector and $\vec{F}(Q) = (F(Q), G(Q), H(Q))$ is the flux vector. The solution vector, Q, takes the form,

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{bmatrix}, \tag{3.1.2}$$

where ρ is the density, u, v, and w are the velocities in the x, y and z-directions, and e is the total energy per unit volume. The inviscid-flux vector, $\vec{F}(Q)$, is,

$$\vec{F}(Q) = (F(Q), G(Q), H(Q)) = \left(\begin{bmatrix} \rho u \\ p + \rho u^2 \\ \rho uv \\ \rho uw \\ u(e+p) \end{bmatrix}, \begin{bmatrix} \rho v \\ \rho uv \\ p + \rho v^2 \\ \rho vw \\ v(e+p) \end{bmatrix}, \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ p + \rho w^2 \\ w(e+p) \end{bmatrix} \right), \tag{3.1.3}$$

where $p = (\gamma - 1)(e - \frac{1}{2}\rho(u^2 + v^2 + w^2))$ is the pressure and γ is the ratio of specific heats. Equation (3.1.1) can be written in the form,

$$\frac{\partial Q}{\partial t} + \frac{\partial F(Q)}{\partial x} + \frac{\partial G(Q)}{\partial y} + \frac{\partial H(Q)}{\partial z} = 0. \tag{3.1.4}$$

The computational domain Ω is split into N non-overlapping elements, where element i has volume V_i. Equation (3.1.1) is integrated over the domain and multiplied by an arbitrary weighting function W,

$$\int_{V_i} \left[\frac{\partial Q}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q) \right] W dV. \tag{3.1.5}$$

Applying integration by parts and the Gauss Divergence theorem to the above equation yields the weighted residual form of equation 3.1.1,

$$\int_{V_i} \frac{\partial Q}{\partial t} W dV + \int_{\partial V_i} W \vec{F}(Q) \cdot \vec{n} dS - \int_{V_i} \vec{\nabla} W \cdot \vec{F}(Q) dV = 0. \tag{3.1.6}$$

To approximate the solution to Q on an element V, introduce Q_i. The solution is assumed to belong in the space of polynomials of degree k or less, meaning $Q_i \in P^k(V_i)$, and no continuity
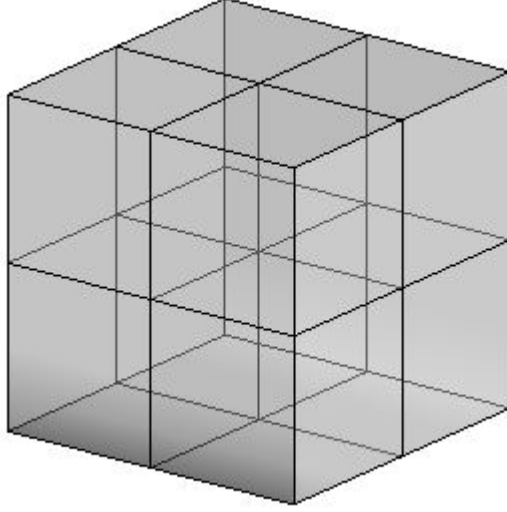
Figure 3.1　Computational domain Ω

requirement across element interfaces is employed. Then, the numerical solution Q_i must satisfy (3.1.6),

$$\int_{V_i} \frac{\partial Q_i}{\partial t} W dV + \int_{\partial V_i} W \vec{F}(Q_i) \cdot \vec{n} dS - \int_{V_i} \vec{\nabla} W \cdot \vec{F}(Q_i) dV = 0. \qquad (3.1.7)$$

At element interfaces, $\int_{\partial V_i} W \vec{F}(Q_i) \cdot \vec{n} dS$ is not directly defined due to discontinuities in the solution. To provide element coupling, a common Riemann flux is used to replace the normal flux, i.e.,

$$F^n(Q_i) \equiv \vec{F}(Q_i) \cdot \vec{n} \approx F_{com}^n(Q_i, Q_{i+}, \vec{n}), \qquad (3.1.8)$$

where Q_{i+} is the solution outside of element V_i. Equation (3.1.7) becomes,

$$\int_{V_i} \frac{\partial Q_i}{\partial t} W dV + \int_{\partial V_i} W F_{com}^n dS - \int_{V_i} \vec{\nabla} W \cdot \vec{F}(Q_i) dV = 0. \qquad (3.1.9)$$

Once again, apply integration by parts and the Gauss Divergence theroem to the final term in equation 3.1.9 to yeild,

$$\begin{aligned}
\int_{V_i} \vec{\nabla} W \cdot \vec{F}(Q_i) dV &= \int_{V_i} \vec{\nabla} \cdot \left[\vec{F}(Q_i) W \right] dV - \int_{V_i} \vec{\nabla} \cdot \vec{F}(Q_i) W dV \\
&= \int_{\partial V_i} \vec{F}(Q_i) W \cdot \vec{n} dS - \int_{V_i} \vec{\nabla} \cdot \vec{F}(Q_i) W dV.
\end{aligned} \qquad (3.1.10)$$

Combining the result in equation (3.1.10) with equation (3.1.9) gives,

$$\int_{V_i} \frac{\partial Q_i}{\partial t} W dV + \int_{V_i} \vec{\nabla} \cdot \vec{F}(Q_i) W dV + \int_{\partial V_i} \left[F_{com}^n - F^n(Q_i) \right] W dS = 0. \tag{3.1.11}$$

The final term in equation (3.1.11) "penalizes" the normal flux difference, $[F^n] = F_{com}^n - F^n(Q_i)$, and can be viewed as a penalty term. Next, to change the surface integral into a volume integral in equation (3.1.11), a "correction field", $\delta_i \in P^k(V_i)$, is introduced which is evaluated from a "lifting operator" equation defined as,

$$\int_{\partial V_i} W \left[F^n \right] dS = \int_{V_i} W \delta_i dV. \tag{3.1.12}$$

Substituting equation (3.1.12) into equation (3.1.11) yeilds,

$$\int_{V_i} \left(\frac{\partial Q_i}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q_i) + \delta_i \right) W dV = 0. \tag{3.1.13}$$

The flux divergence is approximated by polynomials of degree k or less. This simplifies the derivation, but $\vec{\nabla} \cdot \vec{F}(Q_i)$ does not necessarily reside in the space $P^k(V_i)$, hence a projection term is employed to project $\vec{\nabla} \cdot \vec{F}(Q_i)$ into the proper space,

$$\int_{V_i} \vec{\nabla} \cdot \vec{F}(Q_i) dV = \int_{V_i} \Pi \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right] dV. \tag{3.1.14}$$

If the weighting function, W, is selected such that a unique solution exists, equation (3.1.13) becomes,

$$\frac{\partial Q_i}{\partial t} + \Pi \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right] + \delta_i = 0. \tag{3.1.15}$$

The definition of a correction field reduces the weighted residual formulation into a differential formulation. To find the approximate solution Q_i, define the degrees of freedom to be solution values located at solution points (SP) at each element. Then equation (3.1.15) must hold at every solution point in the domain, i.e.,

$$\frac{\partial Q_{i,j}^h}{\partial t} + \Pi_j \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right] + \delta_{i,j} = 0, \tag{3.1.16}$$

where $\Pi_j \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right]$ is the value of $\Pi \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right]$ at solution point j. The efficiency of the method lies in calculating the correction function, δ_i. For linear triangles with straight faces,

Table 3.1   Lifting coefficients for $P^1 - P^5$ for linear element [-1,1].

SP	P^1	P^2	P^3	P^4	P^5
j	$\alpha_{L,j}$	$\alpha_{L,j}$	$\alpha_{L,j}$	$\alpha_{L,j}$	$\alpha_{L,j}$
1	2.0	4.5	8.0	12.5	18.0
2	-1.0	-0.75	-0.5938	-0.2612	0.2513
3	-	1.5	0.9688	0.9375	0.8518
4	-	-	-2.0	-1.1451	-1.1244
5	-	-	-	0.5	1.3103
6	-	-	-	-	-3.0

once the solution points and flux points have been chosen, the correction at the solution points is,

$$\delta_{i,j} = \frac{1}{|V_i|} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} \left[F^n\right]_{f,l} S_f, \tag{3.1.17}$$

where $\alpha_{j,f,l}$ are constant coefficients independent of the solution and shape of the simplex, $|V_i|$ is the cell volume, S_f is the face area, and l runs through the flux points on the faces. In the case of quadrilateral or hexahedral elements, the extension is straightforward as all the operations are carried out in a one-dimensional manner using a tensor product basis. For one-dimensional conservation laws, equation (3.1.17) reduces to,

$$\delta_{i,j} = \frac{1}{h_i} \left(\alpha_{L,j} \left[F^n\right]_L + \alpha_{R,j} \left[F^n\right]_R \right), \tag{3.1.18}$$

where the element i has two faces (a left and a right one) and has length h_i. The elements sides are unit in area, and have unit face normals of 1 and -1 such that $[F^n]_L = -[F]_L$ and $[F^n]_R = [F]_R$. The terms $\alpha_{R,j}$ and $\alpha_{L,j}$ are constant lifting coefficients in one-dimension (see table 3.1) which penalizes the normal flux difference at the faces for every solution point j. Due to symmetry, $\alpha_{L,j} = \alpha_{R,k+2-j}$ for the one-dimensional case [12], where k is the value of the solution reconstruction order, or the value of P^k. Derivation of the coefficients for linear elements is covered in Appendix B, specifically for P^2 reconstruction. The chain rule approach computes $\Pi_j \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right]$ efficiently, i.e.,

Table 3.2   Lagrange interpolation coefficients for P^2.

$c_{j,m}$	$m = 1$	$m = 2$	$m = 3$
$j = 1$	-1.5	2.0	-0.5
$j = 2$	-0.5	0.0	0.5
$j = 3$	0.5	-2.0	1.5

$$
\begin{aligned}
\vec{\nabla} \cdot \vec{F}(Q_{i,j}^h) &= \frac{\partial F(Q_{i,j})}{\partial x} + \frac{\partial G(Q_{i,j})}{\partial y} + \frac{\partial H(Q_{i,j})}{\partial z} \\
&= \frac{\partial F(Q_{i,j})}{\partial Q} \frac{\partial Q_{i,j}}{\partial x} + \frac{\partial G(Q_{i,j})}{\partial Q} \frac{\partial Q_{i,j}}{\partial y} + \frac{\partial H(Q_{i,j})}{\partial Q} \frac{\partial Q_{i,j}}{\partial z} \\
&= \frac{\partial \vec{F}(Q_{i,j})}{\partial Q} \cdot \vec{\nabla} Q_{i,j}.
\end{aligned}
\tag{3.1.19}
$$

The $\frac{\partial \vec{F}(Q_{i,j})}{\partial Q}$ term can be computed analytically [26],

$$
\frac{\partial F(Q_{i,j})}{\partial x} = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
\frac{\gamma-1}{2}(v^2 + w^2) + \frac{\gamma-3}{2}u^2 & (3-\gamma)u & -(\gamma-1)v & -(\gamma-1)w & \gamma-1 \\
-uv & v & u & 0 & 0 \\
-uw & w & 0 & u & 0 \\
\left[-\frac{\gamma e}{\rho} + (\gamma-1)(u^2 + v^2 + w^2)\right]u & \frac{\gamma e}{\rho} - \frac{(\gamma-1)}{2}(3u^2 + v^2 + w^2) & -(\gamma-1)uv & -(\gamma-1)uw & \gamma u
\end{bmatrix},
$$

$$
\frac{\partial G(Q_{i,j})}{\partial y} = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
-uv & v & u & 0 & 0 \\
\frac{\gamma-1}{2}(u^2 + w^2) + \frac{\gamma-3}{2}v^2 & -(\gamma-1)u & (3-\gamma)v & -(\gamma-1)w & \gamma-1 \\
-vw & 0 & w & v & 0 \\
\left[-\frac{\gamma e}{\rho} + (\gamma-1)(u^2 + v^2 + w^2)\right]v & -(\gamma-1)uv & \frac{\gamma e}{\rho} - \frac{(\gamma-1)}{2}(u^2 + 3v^2 + w^2) & (\gamma+1)vw & \gamma uv
\end{bmatrix},
$$

$$
\frac{\partial H(Q_{i,j})}{\partial z} = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 \\
-uw & w & 0 & u & 0 \\
-vw & 0 & w & v & 0 \\
\frac{\gamma-1}{2}(u^2 + v^2) + \frac{\gamma-3}{2}w^2 & -(\gamma-1)u & -(\gamma-1)v & (3-\gamma)w & \gamma-1 \\
\left[-\frac{\gamma e}{\rho} + (\gamma-1)(u^2 + v^2 + w^2)\right]w & -(\gamma-1)uw & (\gamma+1)vw & \frac{\gamma e}{\rho} - \frac{(\gamma-1)}{2}(u^2 + v^2 + 3w^2) & \gamma w
\end{bmatrix},
$$

while the solution derivative, $\vec{\nabla} Q_{i,j}$, is computed using a Lagrange polynomial interpolation. The Lagrange polynomial is expressed in the form,

$$
L_j^{SP}(X) = \prod_{s=1, s \neq i}^{n} \left(\frac{X - X_s}{X_i - X_s} \right),
\tag{3.1.20}
$$
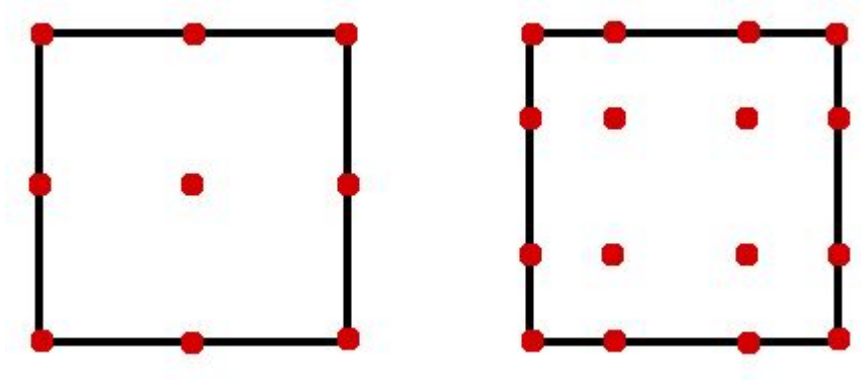
Figure 3.2   Solution points on a hexahedral face for P^2 and P^3

where X_s is the location of the solution points in the domain, which are Gauss-Lobatto points defined by,

$$X_s = -cos\left[\frac{(s-1)\pi}{k}\right] \cdot s = 1, 2, ..., k+1 \qquad (3.1.21)$$

The gradient of Q is then calculated with,

$$\vec{\nabla}Q_{i,j} = \sum_j Q_{i,j}\vec{\nabla}L_j^{SP}.$$

Hence, the solution gradient is formulated from derivatives of the Lagrange polynomials (table 3.2). Applying the projection and correction function formulation into equation (3.1.15), the CPR formulation for the inviscid flux for simplex elements is,

$$\frac{\partial Q_{i,j}^h}{\partial t} + \Pi_j\left(\vec{\nabla}\cdot\vec{F}(Q)\right) + \frac{1}{|V_i|}\sum_{f\in\partial V_i}\sum_l \alpha_{j,f,l}\left[F_{com}^n - F^n(Q_i)\right]_{f,l} S_f = 0. \qquad (3.1.22)$$

For one-dimensional conservation laws, equation (3.1.22) reduces to,

$$\frac{\partial Q_{i,j}^h}{\partial t} + \Pi_j\left(\frac{\partial F(Q_i^h)}{\partial x}\right) + \frac{1}{h_i}\left(\alpha_{L,j}\left[F^n\right]_L + \alpha_{R,j}\left[F^n\right]_R\right) = 0, \qquad (3.1.23)$$

To extend the one-dimensional CPR method to three-dimensions, let $Q_{i:,j,m,l}$ denote the degrees of freedom (cell i and solution point indexes j, m, and l) within hexahedral elements, where each element i has six faces and volume $|V_i|$. The CPR formulation from equation (3.1.22) becomes,

$$\frac{\partial Q^h_{i:j,m,l}}{\partial t} + \Pi_{j,m,l} \left[\vec{\nabla} \cdot \vec{F}(Q_i) \right]$$

$$+ \frac{1}{|V_i|} (\alpha_{R,j}[F^n]_{R,j} S_1 + \alpha_{R,m}[G^n]_{R,m} S_2 + \alpha_{R,l}[H^n]_{R,l} S_3 \qquad (3.1.24)$$

$$+ \alpha_{L,j}[F^n]_{L,j} S_4 + \alpha_{L,m}[G^n]_{L,m} S_5 + \alpha_{L,l}[H^n]_{L,l} S_6) = 0,$$

where S_1 through S_6 is the value of each face area. For efficiency, let the flux point (FP) locations coincide with the solution points. Thus computational cost is reduced since no data interpolation between solution points and flux points is needed.

Viscous flux

The Navier-Stokes equations take the form,

$$\frac{\partial Q}{\partial t} + \vec{\nabla} \cdot \vec{F}(Q) - \vec{\nabla} \cdot \vec{F}^v(Q, \vec{\nabla}Q) = 0, \qquad (3.1.25)$$

where $\vec{F}$ is the inviscid flux vector as described before and $\vec{F}^v$ is the viscous flux vector given by,

$$\vec{F}^v(Q, \vec{\nabla}Q) = \left(F^v(Q, \vec{\nabla}Q), G^v(Q, \vec{\nabla}Q), H^v(Q, \vec{\nabla}Q) \right) \qquad (3.1.26)$$

$$\vec{F}^v = \left(\begin{bmatrix} 0 \\ \tau_{xx} \\ \tau_{xy} \\ \tau_{xz} \\ u\tau_{xx} + v\tau_{xy} + w\tau_{xz} - q_x \end{bmatrix}, \begin{bmatrix} 0 \\ \tau_{yx} \\ \tau_{yy} \\ \tau_{yz} \\ u\tau_{yx} + v\tau_{yy} + w\tau_{yz} - q_y \end{bmatrix}, \begin{bmatrix} 0 \\ \tau_{zx} \\ \tau_{zy} \\ \tau_{zz} \\ u\tau_{zx} + v\tau_{zy} + w\tau_{zz} - q_z \end{bmatrix} \right).$$

$$(3.1.27)$$

The stress tensor, τ, is,

$$\tau = \mu \left[\vec{\nabla}\vec{u} + \left(\vec{\nabla}\vec{u} \right)^T - \frac{2}{3} \left(\vec{\nabla} \cdot \vec{u} \right) I \right], \qquad (3.1.28)$$

where I is the identity matrix, μ is the molecular viscosity coefficient, and $\vec{u}$ contains the velocity components. The individual components of the viscous stress tensor are,

$$\tau_{xx} = \frac{2}{3}\mu\left(2\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} - \frac{\partial w}{\partial z}\right),$$

$$\tau_{yy} = \frac{2}{3}\mu\left(2\frac{\partial v}{\partial y} - \frac{\partial u}{\partial x} - \frac{\partial w}{\partial z}\right),$$

$$\tau_{zz} = \frac{2}{3}\mu\left(2\frac{\partial w}{\partial z} - \frac{\partial u}{\partial x} - \frac{\partial v}{\partial y}\right),$$

$$\tau_{xy} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) = \tau_{yx},$$

$$\tau_{xz} = \mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) = \tau_{zx},$$

$$\tau_{yz} = \mu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) = \tau_{zy}.$$

The heat flux is,

$$\vec{q} = -c_p\frac{\mu}{\text{Pr}}\vec{\nabla}T, \tag{3.1.29}$$

where c_p is the specific heat capacity at constant pressure, Pr is the Prandtl number, and T is the temperature. The components of the heat flux are,

$$q_x = -c_p\frac{\mu}{Pr}\frac{\partial T}{\partial x},$$

$$q_y = -c_p\frac{\mu}{Pr}\frac{\partial T}{\partial y},$$

$$q_z = -c_p\frac{\mu}{Pr}\frac{\partial T}{\partial z}.$$

A variable $\vec{R}$ is introduced such that,

$$\vec{R} = \vec{\nabla}Q. \tag{3.1.30}$$

Let $\vec{R}_i$ be an approximation of $\vec{R}$ on V_i, and $\vec{R}_i \in \left(P^k, P^k, P^k\right)$ for three dimensions and $\vec{R}_i \in \left(P^k, P^k\right)$ for two-dimensions. The obvious choice of $\vec{R}_i = \vec{\nabla}Q_i$ is not appropriate, and the computation of $\vec{R}_i$ must involve data from neighboring cells. Discretizing the viscous terms using CPR for simplex elements yields the equation,

$$\frac{\partial Q_{i,j}^h}{\partial t} + \Pi_j\left[\vec{\nabla}\cdot\vec{F}(Q_i^h)\right] - \Pi_j^v\left[\vec{\nabla}\cdot\vec{F}_v(Q_i^h, \vec{R}_i^h)\right]$$
$$+ \frac{1}{|V_i|}\sum_{f\in\partial V_i}\sum_l \alpha_{j,f,l}\left([F]_{f,l}^n - [F^v]_{f,l}^n\right)S_f, \tag{3.1.31}$$

Table 3.3 Viscous DG correction coefficient.

P^k	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
β	2.0	6.0	10.0	18.0	21.0

where $[F]^n \equiv F^n_{com} - F^n(Q^h_i, \vec{n})$ and,

$$[F^v]^n \equiv \vec{F}^v(Q^{com}_{f,l}, \vec{\nabla}Q^{com}_{f,l}) \cdot \vec{n}_{f,l} - \left. \vec{F}^v(Q^h_i, \vec{R}^h_i) \right|_{f,l} \cdot \vec{n}_{f,l}.$$

The calculation of $\vec{R}^h_{i,j}$ is completed as follows,

$$\vec{R}^h_{i,j} = \left(\vec{\nabla}Q^h_i\right) + \frac{1}{|V_i|} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} \left[Q^h_{com} - Q^h_i\right]_{f,l} \vec{n}_f S_f, \tag{3.1.32}$$

Extension to hexahedral elements is straight forward, like that of equation (3.1.24). All the corrections are completed in a one-dimensional manner across the 6 faces of the element. For the current study, the Bassi-Rebay 2 scheme (BR2) [3] is implemented to discretize the viscous flux. The value of Q^h_{com} is simply the average of the solution on both sides of face f. The computation of the viscous flux vector, $\Pi^v_j \left[\vec{\nabla} \cdot \vec{F}_v(Q^h_i, \vec{R}^h_i)\right]$, follows the same Lagrange polynomial approach as described in the inviscid flux formulation. First, the viscous flux is evaluated at the solution points,

$$\vec{F}^v_{i,j} = \vec{F}^v\left(Q_{i,j}, \vec{R}_{i,j}\right).$$

Then, Lagrange interpolation formulates a viscous flux polynomial, and the divergence of the polynomial is used as the projection,

$$\Pi^v_j \left[\vec{\nabla} \cdot \vec{F}^v(Q^h_i, \vec{R}^h_i)\right] \approx \sum_j \vec{F}^v_{i,j} \cdot \vec{\nabla}L^{SP}_j \tag{3.1.33}$$

In the correction term, the common viscous flux term, $F^n_{v,com}(Q^h_{com}, \vec{\nabla}Q^h_{com}, \vec{n})$, is required. For the BR2 scheme, the common solution is simply the average of the solutions at both sides of the flux points,

$$\left. Q^h_{com} \right|_{f,l} = \frac{Q_i|_{f,l} + Q_{i+}|_{f,l}}{2},$$

while common gradient on face f and flux point l is evaluated as,

$$\left. \vec{\nabla}Q^h_{com} \right|_{f,l} = \frac{1}{2}\left(\vec{\nabla}Q^-_{f,l} + \vec{r}^-_{f,l} + \vec{\nabla}Q^+_{f,l} + \vec{r}^+_{f,l}\right), \tag{3.1.34}$$
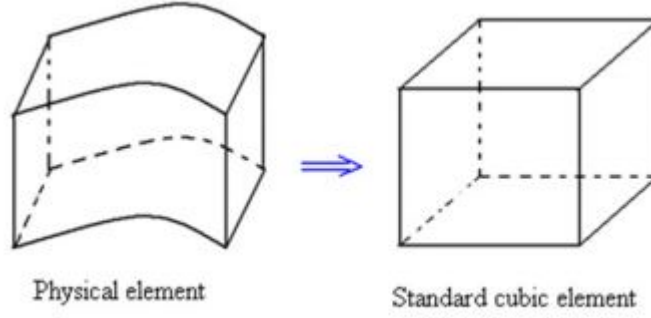
Figure 3.3   Transformation from physical to standard element

where $\vec{\nabla} Q_{f,l}^{-}$ and $\vec{\nabla} Q_{f,l}^{+}$ are the gradients of the solution from both the left and right cells, and the $\vec{r}_{f,l}^{-}$ and $\vec{r}_{f,l}^{+}$ terms are the local lifting corrections to the gradients due to the common solution on face f. They are calculated using,

$$r_{f,l}^{\pm} = \frac{1}{|V_i^{\pm}|} \sum_m \beta \left[Q_{com}^h - Q_i^h \right]_{f,m} (\mp \vec{n}_f) \, S_f, \qquad (3.1.35)$$

where m is the index of the flux point on faces f, $\vec{n}_f$ is the unit normal vector directing from left to right, and β is the viscous discontinuous Galerkin (DG) correction coefficient (see table 3.3). Note that in equation (3.1.35) there is no summation over the faces so that the BR2 scheme maintains a compact face neighbor stencil.

3.1.2   High-order elements

The computational domain is filled with non-overlapping hexahedral elements. The elements are transformed from the standard coordinate systems, (x, y, z), to a standard cubic element, $(\xi, \eta, \zeta) \in [0, 1] \ \text{x} \ [0, 1] \ \text{x} \ [0, 1]$, as shown in figure 3.3. The transformation takes the form,

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \sum_{i=1}^{K} M_i(\xi, \eta, \zeta) \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}, \qquad (3.1.36)$$

where K is the number of points used to define the physical element, (x_i, y_i, z_i) are the Cartesian coordinates of those points, and $M_i(\xi, \eta, \zeta)$ are the shape functions determined by node locations [34]. The Jacobian matrix, J, is,

$$J = \frac{\partial(x, y, z)}{\partial(\xi, \eta, \zeta)} = \begin{bmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{bmatrix}. \tag{3.1.37}$$

When the transformation is non-singular, the inverse transformation must exist. The Jacobian matrices are related to one-another according to,

$$\frac{\partial(\xi, \eta, \zeta)}{\partial(x, y, z)} = \begin{bmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{bmatrix} = J^{-1}. \tag{3.1.38}$$

Hence, the metrics can be computed as follows:

$$
\begin{aligned}
\xi_x &= \frac{y_\eta z_\zeta - y_\zeta z_\eta}{|J|}, & \xi_y &= \frac{x_\zeta z_\eta - x_\eta z_\zeta}{|J|}, & \xi_z &= \frac{x_\eta y_\zeta - x_\zeta y_\eta}{|J|}, \\
\eta_x &= \frac{y_\zeta z_\xi - y_\xi z_\zeta}{|J|}, & \eta_y &= \frac{x_\xi z_\zeta - x_\zeta z_\xi}{|J|}, & \eta_z &= \frac{x_\zeta y_\xi - x_\xi y_\zeta}{|J|}, \\
\zeta_x &= \frac{y_\xi z_\eta - y_\eta z_\xi}{|J|}, & \zeta_y &= \frac{x_\eta z_\xi - x_\xi z_\eta}{|J|}, & \zeta_z &= \frac{x_\xi y_\eta - x_\eta y_\xi}{|J|}.
\end{aligned} \tag{3.1.39}
$$

The governing equation for the inviscid flux is transformed from the physical domain to the computational domain and becomes,

$$\frac{\partial \tilde{Q}}{\partial t} + \frac{\partial \tilde{F}}{\partial \xi} + \frac{\partial \tilde{G}}{\partial \eta} + \frac{\partial \tilde{H}}{\partial \zeta} = 0. \tag{3.1.40}$$

Likewise, the viscous flux equation is,

$$\frac{\partial \tilde{Q}}{\partial t} + \frac{\partial \left(\tilde{F} - \tilde{F}_v \right)}{\partial \xi} + \frac{\partial \left(\tilde{G} - \tilde{G}_v \right)}{\partial \eta} + \frac{\partial \left(\tilde{H} - \tilde{H}_v \right)}{\partial \zeta} = 0. \tag{3.1.41}$$

The transformed flux variables are,

$$
\begin{pmatrix} \tilde{F} \\ \tilde{G} \\ \tilde{H} \end{pmatrix} = |J| \begin{bmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{bmatrix} \cdot \begin{pmatrix} F \\ G \\ H \end{pmatrix}, \tag{3.1.42}
$$

$$
\begin{pmatrix} \tilde{F}_v \\ \tilde{G}_v \\ \tilde{H}_v \end{pmatrix} = |J| \begin{bmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{bmatrix} \cdot \begin{pmatrix} F_v \\ G_v \\ H_v \end{pmatrix}. \tag{3.1.43}
$$

The following subsections will detail the extension of high-order elements to the inviscid and viscous flux for the CPR method.

Inviscid flux

Let $\vec{S}_\xi = |J|(\xi_x, \xi_y, \xi_z)$, $\vec{S}_\eta = |J|(\eta_x, \eta_y, \eta_z)$, and $\vec{S}_\zeta = |J|(\zeta_x, \zeta_y, \zeta_z)$. We obtain $\tilde{F} = \vec{F} \cdot \vec{S}_\xi$, $\tilde{G} = \vec{F} \cdot \vec{S}_\eta$, and $\tilde{H} = \vec{F} \cdot \vec{S}_\zeta$. Then equation (3.1.40) becomes,

$$
\frac{\partial \tilde{Q}}{\partial t} + \vec{\nabla}^\xi \cdot \vec{\tilde{F}} = 0, \tag{3.1.44}
$$

where $\vec{\tilde{F}} = \left(\tilde{F}, \tilde{G}, \tilde{H} \right)$ and $\vec{\nabla}^\xi$ is the divergence operator in the computational domain. Following from the CPR formulation for a simplex element,

$$
\frac{\partial \tilde{Q}_{i,j}^h}{\partial t} + \Pi_j \left[\vec{\nabla}^\xi \cdot \vec{\tilde{F}}(\tilde{Q}_i) \right] + \frac{1}{|V_i^\xi|} \sum_{f \in \partial V_i} \sum_l \alpha_{j,f,l} \left[\tilde{F}^n \right]_{f,l} S_f^\xi, = 0, \tag{3.1.45}
$$

where the ξ subscript indicates the variables which are evaluated in the computational domain. The transformed normal flux is further expressed in terms of the flux in physical space as,

$$
\begin{aligned}
\left[\tilde{F} \right]_{f,l}^n S_f^\xi &= \left(\left[\tilde{F} \right]_{f,l}^n \cdot \vec{n}_f^\xi \right) S_f^\xi \\
&= \left(\left[\tilde{F} \right]_{f,l}^n \cdot \vec{S}_\xi \Big|_{f,l} \, n_\xi |_{f,l} \right) S_f^\xi + \left(\left[\tilde{F} \right]_{f,l}^n \cdot \vec{S}_\eta \Big|_{f,l} \, n_\eta |_{f,l} \right) S_f^\xi + \left(\left[\tilde{F} \right]_{f,l}^n \cdot \vec{S}_\zeta \Big|_{f,l} \, n_\zeta |_{f,l} \right) S_f^\xi \\
&= \left[\tilde{F} \right]_{f,l}^n \cdot \vec{S}^n \Big|_{f,l} \\
&= \left[\tilde{F} \right]_{f,l}^n |\vec{S}^n| \Big|_{f,l}
\end{aligned}
$$

$$
\tag{3.1.46}
$$

where $\vec{n}_f^\xi = (n_\xi, n_\eta, n_\zeta)$ is a unit normal vector on a straight face of the standard element and $\vec{S}^n = \vec{S}_\xi n^\xi + \vec{S}_\eta n^\eta + \vec{S}_\zeta n^\zeta$ is a normal vector on a face in the physical space. For a hexahedral element with indexes (j, m, l) to denote the solution points, the CPR formulation for the inviscid flux becomes,

$$
\begin{aligned}
\frac{\partial \tilde{Q}_{i:j,m,l}^h}{\partial t} + &\Pi_{j,m,l} \left[\vec{\nabla}^\xi \cdot \vec{F}(\tilde{Q}_i) \right] \\
&+ \frac{1}{|V_i|} (\alpha_{R,j} [\tilde{F}_{com}(1, \eta_{j,m,l}, \zeta_{j,m,l}) - \tilde{F}_i(1, \eta_{j,m,l}, \zeta_{j,m,l})]_j^n S_{1,j} \\
&+ \alpha_{L,j} [\tilde{F}_{com}(-1, \eta_{j,m,l}, \zeta_{j,m,l}) - \tilde{F}_i(-1, \eta_{j,m,l}, \zeta_{j,m,l})]_j^n S_{2,j} \\
&+ \alpha_{R,m} [\tilde{G}_{com}(\xi_{j,m,l}, 1, \zeta_{j,m,l}) - \tilde{G}_i(\xi_{j,m,l}, 1, \zeta_{j,m,l})]_m^n S_{3,m} \\
&+ \alpha_{L,m} [\tilde{G}_{com}(\xi_{j,m,l}, -1, \zeta_{j,m,l}) - \tilde{G}_i(\xi_{j,m,l}, -1, \zeta_{j,m,l})]_m^n S_{4,m} \\
&+ \alpha_{R,l} [\tilde{H}_{com}(\xi_{j,m,l}, \eta_{j,m,l}, 1) - \tilde{H}_i(\xi_{j,m,l}, \eta_{j,m,l}, 1)]_l^n S_{5,l} \\
&+ \alpha_{L,l} [\tilde{H}_{com}(\xi_{j,m,l}, \eta_{j,m,l}, -1) - \tilde{H}_i(\xi_{j,m,l}, \eta_{j,m,l}, -1)]_l^n S_{6,l}) = 0.
\end{aligned}
\tag{3.1.47}
$$

Note that the correction is completed in a one-dimensional manner which makes the method more efficient per degree of freedom when compared to tetrahedral or prism cells.

Viscous flux

Let $\vec{S}_\xi = |J|(\xi_x, \xi_y, \xi_z)$, $\vec{S}_\eta = |J|(\eta_x, \eta_y, \eta_z)$, and $\vec{S}_\zeta = |J|(\zeta_x, \zeta_y, \zeta_z)$. We obtain $\tilde{F} = \vec{F} \cdot \vec{S}_\xi$, $\tilde{G} = \vec{F} \cdot \vec{S}_\eta$, and $\tilde{H} = \vec{F} \cdot \vec{S}_\zeta$. Then equation (3.1.41) becomes,

$$
\frac{\partial \tilde{Q}}{\partial t} + \vec{\nabla}^\xi \cdot \vec{\tilde{F}} - \vec{\nabla}^\xi \cdot \vec{\tilde{F}}^v = 0,
\tag{3.1.48}
$$

where $\vec{\tilde{F}} = \left(\tilde{F}, \tilde{G}, \tilde{H} \right)$, $\vec{\tilde{F}}^v = \left(\tilde{F}^v, \tilde{G}^v, \tilde{H}^v \right)$, and $\vec{\nabla}^\xi$ is the divergence operator in the computational domain. Following from the CPR formulation for a simplex element, we obtain,

$$
\begin{aligned}
\frac{\partial \tilde{Q}_{i,j}^h}{\partial t} + &\Pi_j \left[\vec{\nabla}^\xi \cdot \vec{\tilde{F}}_c(Q_i^h) \right] - \Pi_j^v \left[\vec{\nabla}^\xi \cdot \vec{\tilde{F}}_v(\tilde{Q}_i^h, \vec{\tilde{R}}_i^h) \right] \\
&+ \frac{1}{|V_i^\xi|} \sum_{f \in \partial V_i^\xi} \sum_l \alpha_{j,f,l} \left(\left[\tilde{F}_{com}^n - \tilde{F}_c \right]_{f,l} - \left[\tilde{F}_{com}^{v,n} - \tilde{F}^{v,n}\left(\tilde{Q}_i, \vec{\tilde{R}}_i \right) \right]_{f,l} \right) S_f^\xi,
\end{aligned}
\tag{3.1.49}
$$

$$
\vec{\tilde{R}}_{i,j}^h = \left(\vec{\nabla}^\xi \tilde{Q}_i^h \right) + \frac{1}{|V_i^\xi|} \sum_{f \in \partial V_i^\xi} \sum_l \alpha_{j,f,l} \left[\tilde{Q}_{com}^h - \tilde{Q}_i^h \right]_{f,l} \vec{n}_f^\xi S_f^\xi.
\tag{3.1.50}
$$

Again, extending the above to hexahedral elements is straight forward, and the same approach for the inviscid flux can be used.
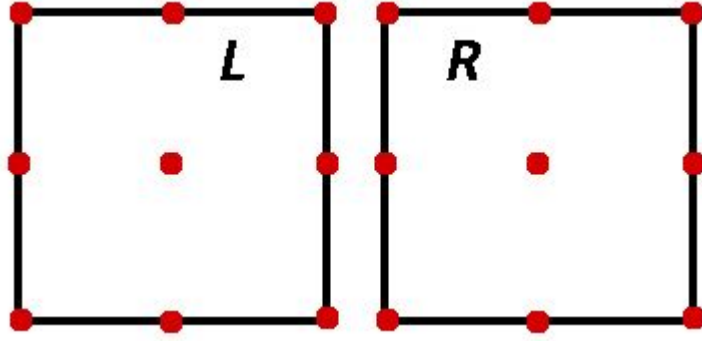
Figure 3.4   Shared face and points between two cells

3.2   Riemann solver

This section covers the calculation of the common flux, $F_{com}^n(Q_i, Q_{i+}, \vec{n})$. The left and right states at cell interfaces are not equal to each other, i.e,

$$Q_{f,j}^L = \begin{bmatrix} \rho_L \\ \rho_L u_L \\ \rho_L v_L \\ \rho_L w_L \\ e_L \end{bmatrix} \neq \begin{bmatrix} \rho_R \\ \rho_R u_R \\ \rho_R v_R \\ \rho_R w_R \\ e_R \end{bmatrix} = Q_{f,j}^R. \tag{3.2.1}$$

Hence, a common flux is found by solving a Riemann problem to calculate the continuous solution over the interfaces. The two solvers implemented are the Rusanov flux and the Roe flux. Both are described in this section.

3.2.1   Rusanov flux

The Rusanov flux [21] requires an average speed of sound to be calculated first,

$$\bar{a} = \sqrt{\frac{\gamma \bar{p}}{\bar{\rho}}}, \tag{3.2.2}$$

where $\bar{p} = \frac{p_L + p_R}{2}$ and $\bar{\rho} = \frac{\rho_L + \rho_R}{2}$. Next, the average speed of the flow in the normal direction is needed,

$$\bar{u}_n = \frac{|u_{L,n} + u_{R,n}|}{2}, \tag{3.2.3}$$

where $u_n = \vec{u} \cdot \vec{n}$. Then, the following equation calculates the Rusanov flux,

$$F_{com}^n = \frac{1}{2} \left(\vec{F}(Q^R) \cdot \vec{n} + \vec{F}(Q^L) \cdot \vec{n} \right) - \frac{1}{2} \left(\bar{u}_n + \bar{a} \right) \left(Q^R - Q^L \right). \tag{3.2.4}$$

3.2.2 Roe flux

The Roe flux [20] attempts to solve the Riemann problem with linearization. Consider the equation,

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} = 0. \tag{3.2.5}$$

Roe uses a linear approximation to the Riemann problem to yield,

$$\frac{\partial Q}{\partial x} + [\bar{A}]^n \frac{\partial Q}{\partial x} = 0, \tag{3.2.6}$$

where $[\bar{A}]^n = [\bar{A}^n(Q_L, Q_R)]$ is Roe's averaged matrix and is evaluated using averaged values of Q at an interface separating the states. The Jacobian is now defined as,

$$[A] = \frac{\partial F}{\partial Q}, \tag{3.2.7}$$

which is replaced by $[\bar{A}]$, or Roe's averaged matrix, in this formulation. Certain conditions must be satisfied for the solution of the linear problem to become an approximate solution to the nonlinear Riemann problem presented. These conditions are [26]:

- Q is related to F by a linear mapping.

- As the left state approaches the right state $(Q_L \Rightarrow Q_R)$, $[\bar{A}(Q_L, Q_R)] \Rightarrow [A]^n$. Where $[A]^n$ is the Jacobian of the original system.

- For any two values of Q_L and Q_R, $F_R^n - F_L^n = [\bar{A}]^n(Q_R - Q_L)$.

- $[\bar{A}]$ must have real and linearly independent eigenvalues.

The averaged values [20], which fill the Jacobian matrix $[\bar{A}]^n$, are given as the following,

$$\bar{\rho} = \frac{\rho_R + \rho_L}{\sqrt{\rho_R} + \sqrt{\rho_L}}, \quad \bar{u} = \frac{\sqrt{\rho_R} u_R + \sqrt{\rho_L} u_L}{\sqrt{\rho_R} + \sqrt{\rho_L}}$$

$$\bar{v} = \frac{\sqrt{\rho_R} v_R + \sqrt{\rho_L} v_L}{\sqrt{\rho_R} + \sqrt{\rho_L}}, \quad \bar{w} = \frac{\sqrt{\rho_R} w_R + \sqrt{\rho_L} w_L}{\sqrt{\rho_R} + \sqrt{\rho_L}}$$

$$\bar{h} = \frac{\sqrt{\rho_R} h_R + \sqrt{\rho_L} h_L}{\sqrt{\rho_R} + \sqrt{\rho_L}}, \quad \bar{a}^2 = (\gamma - 1)\left(\bar{h} - \frac{1}{2}(\bar{u}^2 + \bar{v}^2 + \bar{w}^2)\right).$$

where $a = \frac{\gamma P}{\rho}$ is the speed of sounds and $h = \frac{e+P}{\rho}$ is the total enthalpy. The Jacobian matrices (let $A = \frac{\partial F}{\partial Q}$, $B = \frac{\partial G}{\partial Q}$, and $C = \frac{\partial H}{\partial Q}$ such that $A^n = An^x + Bn^y + Cn^z$) become,

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ (\gamma-1)h - u^2 - a^2 & (3-\gamma)u & -(\gamma-1)v & -(\gamma-1)w & \gamma-1 \\ -uv & v & u & 0 & 0 \\ -uw & w & 0 & u & 0 \\ u[(\gamma-2)h - a^2] & h-(\gamma-1)u^2 & -(\gamma-1)uv & -(\gamma-1)uw & \gamma u \end{bmatrix},$$

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ -uv & v & u & 0 & 0 \\ (\gamma-1)h - v^2 - a^2 & -(\gamma-1)u & (3-\gamma)v & -(\gamma-1)w & \gamma-1 \\ -vw & 0 & w & v & 0 \\ v[(\gamma-2)h - a^2] & -(\gamma-1)uv & h-(\gamma-1)v^2 & (\gamma+1)vw & \gamma uv \end{bmatrix},$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ -uw & w & 0 & u & 0 \\ -vw & 0 & w & v & 0 \\ (\gamma-1)h - w^2 - a^2 & -(\gamma-1)u & -(\gamma-1)v & (3-\gamma)w & \gamma-1 \\ w[(\gamma-2)h - a^2] & -(\gamma-1)uw & (\gamma+1)vw & h-(\gamma-1)w^2 & \gamma w \end{bmatrix}.$$

Then, the numerical flux given at a particular face becomes,

$$F_{com}^n = \frac{1}{2}\left(F^n(Q_R) + F^n(Q_L)\right) - [\bar{A}]^n(Q_R - Q_L). \tag{3.2.8}$$

To solve $[\bar{A}]^n(Q_R - Q_L)$ the eigenvectors of $[\bar{A}]^n$ are required,

$$\bar{\lambda}_1 = \bar{u}_n - \bar{a}, \quad \bar{\lambda}_2 = \bar{\lambda}_3 = \bar{\lambda}_4 = \bar{u}_n, \quad \bar{\lambda}_4 = \bar{u}_n + \bar{a},$$

where $\bar{u}_n = \bar{u}n^x + \bar{v}n^y + \bar{w}n^z$. Next, let $\bar{r}$ be the eigenvectors of the system, in order to completely determine the Roe numerical flux, wave strengths $\bar{\omega}_i$ are required. These can be solved using,

$$Q_R - Q_L = \sum_{i=1}^{5} \bar{\omega}_i \bar{r}_i, \tag{3.2.9}$$

hence the wave strengths can be found, and the Roe flux can be formed,

$$F_{com}^n = \frac{1}{2}\left(F^n(Q_R) + F^n(Q_L)\right) - \frac{1}{2}\sum_{i=1}^{5} |\bar{\lambda}_i|\bar{\omega}_i\bar{r}_i \tag{3.2.10}$$

3.3 Time-stepping

This last section explains the time discretization implemented. The hyperbolic conservation law for the inviscid flux is,

$$\frac{\partial Q}{\partial t} = -\vec{\nabla} \cdot \vec{F}(Q), \tag{3.3.1}$$

and similarly, the viscous flux is,

$$\frac{\partial Q}{\partial t} = -\vec{\nabla} \cdot \vec{F}(Q) + \vec{\nabla} \cdot \vec{F}^v(Q, \vec{\nabla}Q). \tag{3.3.2}$$

The right hand side of both equations can be viewed as the residual and is required for time-stepping. For explicit time integration, a three-stage Runge-Kutta scheme [23] is used. In order to march the solution forward in time $(t + 1)$, the following is done,

$$\begin{aligned}
Q_i^{(1)} &= Q_i^t + \Delta t * \mathrm{Res}_i(Q^t), \\
Q_i^{(2)} &= \frac{3}{4}Q_i^t + \frac{1}{4}Q_i^{(1)} + \frac{1}{4}\Delta t * \mathrm{Res}_i(Q^{(1)}), \\
Q_i^{(t+1)} &= \frac{1}{3}Q_i^t + \frac{2}{3}Q_i^{(2)} + \frac{2}{3}\Delta t * \mathrm{Res}_i(Q^{(2)}),
\end{aligned} \tag{3.3.3}$$

where Δt is the chosen time-step and $\mathrm{Res}_i(Q)$ is the right hand side of equation (3.3.1) or (3.3.2).

CHAPTER 4.   CUDA IMPLEMENTATION

This chapter explains the implementation of the CPR method on hexahedral cells for both the viscous and inviscid flux with GPU computing. The organization of the chapter is as follows: Section 4.1 discusses the data initialization for the GPU device while Section 4.2 shows the implementation of the inviscid flux, viscous flux, and additional kernels for explicit time-stepping. It should be noted that the data initialization is completed on the CPU side, and then transfered to the GPU once the data is structured properly.

4.1   Data initialization

This section describes the initialization of the GPU data, which is completed in C++ on the CPU side and transfered for calculations in CUDA C++. First, the solution array, Q_g, the old solution array, Q_g^o, and the residual, Res_g, are all stored within the GPU's global memory. They are stored in the following manner:

$$Q_g[i + j * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v] = Q_{i,j,k},$$

Where n_{sp} is the number of solution points in a cell, which depends on the order of accuracy, and n_v is the number of state vectors (which is five for three-dimensional problems). The three indexes i, j, and k, represent the solution points, state vectors, and cells respectfully. The index i will run from 0 to $(\mathrm{n}_{sp} - 1)$, j will run from 0 to $(\mathrm{n}_v - 1)$, and k will run from 0 to $(n_c - 1)$ where n_c is the total number of cells in the domain (CUDA arrays will start at index 0). The arrays are copied to the GPU device in a one-dimensional format to improve memory access speed. Next, the metric terms for transformations are copied into the global GPU memory,

$$T_g[i + j * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = T_{i,j,k},$$

where j runs through each metric term (0 to 10) for every solution point (i) and every cell (k) in the domain. The data is structured as such:

$$j = 0 \rightarrow T_g[i + 0 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \xi}{\partial x}\right|_{i,k},$$

$$j = 1 \rightarrow T_g[i + 1 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \xi}{\partial y}\right|_{i,k},$$

$$j = 2 \rightarrow T_g[i + 2 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \xi}{\partial z}\right|_{i,k},$$

$$j = 3 \rightarrow T_g[i + 3 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \eta}{\partial x}\right|_{i,k},$$

$$j = 4 \rightarrow T_g[i + 4 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \eta}{\partial y}\right|_{i,k},$$

$$j = 5 \rightarrow T_g[i + 5 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \eta}{\partial z}\right|_{i,k},$$

$$j = 6 \rightarrow T_g[i + 6 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \zeta}{\partial x}\right|_{i,k},$$

$$j = 7 \rightarrow T_g[i + 7 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \zeta}{\partial y}\right|_{i,k},$$

$$j = 8 \rightarrow T_g[i + 8 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{\partial \zeta}{\partial z}\right|_{i,k},$$

$$j = 9 \rightarrow T_g[i + 9 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.J\right|_{i,k},$$

$$j = 10 \rightarrow T_g[i + 10 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11] = \left.\frac{1}{J}\right|_{i,k}.$$

The final two terms in the transformation data include the Jacobian and its inverse (the inverse is stored to reduce division cost). Next, the boundary conditions are stored as,

$$BC_g[i + f_{bf} * 8] = BC_{i,f_{bf}},$$

where f_{bf} runs through all faces at the boundaries and i runs from 0 to 7. The following information is stored,

$$i = 0 \rightarrow BC_g[0 + f_{bf} * 8] = \text{Boundary condition type,}$$

$$i = 1 \rightarrow BC_g[1 + f_{bf} * 8] = \text{Location of cell with boundary condition,}$$

$$i = 2 \rightarrow BC_g[2 + f_{bf} * 8] = \text{Location of face in cell with boundary condition,}$$

$$i = 3 \rightarrow BC_g[3 + f_{bf} * 8] = \rho_{bf}^{fix},$$

$$i = 4 \rightarrow BC_g[4 + f_{bf} * 8] = \rho_{bf}^{fix} * u_{bf}^{fix},$$

$$i = 5 \rightarrow BC_g[5 + f_{bf} * 8] = \rho_{bf}^{fix} * v_{bf}^{fix},$$

$$i = 6 \rightarrow BC_g[6 + f_{bf} * 8] = \rho_{bf}^{fix} * w_{bf}^{fix},$$

$$i = 7 \rightarrow BC_g[7 + f_{bf} * 8] = e_{bf}^{fix},$$

where the values ρ_{bf}^{fix}, u_{bf}^{fix}, v_{bf}^{fix}, w_{bf}^{fix}, and e_{bf}^{fix} are fixed values at boundary faces in the domain for a specific boundary condition (they are not necessarily used). Next we require allocation of solution information at the boundary faces,

$$Q_g^{bf}[i + j * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_v] = Q_{i,j,k}^{bf},$$

where i runs through the flux points (n_{fp}), j runs through the state variables, and k runs over the boundary faces. The vector stores the following values,

$$j = 0 \rightarrow Q_g^{bf}[i + 0 * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_v] = \rho_{bf},$$

$$j = 1 \rightarrow Q_g^{bf}[i + 1 * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_v] = \rho_{bf} * u_{bf},$$

$$j = 2 \rightarrow Q_g^{bf}[i + 2 * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_v] = \rho_{bf} * v_{bf},$$

$$j = 3 \rightarrow Q_g^{bf}[i + 3 * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_v] = \rho_{bf} * w_{bf},$$

$$j = 4 \rightarrow Q_g^{bf}[i + 4 * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_v] = e_{bf}.$$

The array stores the state vectors at the solution points on the boundary faces, whose values depend on the boundary condition type. Next, we require information on the flux points for face areas and normals,

$$H_g[i + j * \mathrm{n}_{fp} + m * \mathrm{n}_{fp} * \mathrm{n}_f + k * \mathrm{n}_{fp} * \mathrm{n}_f * 6] = H_{i,j,m,k},$$

where i, j, and k, run through the flux points, faces, and cells respectfully, m runs from 0 to 5, and n_f is the total number of faces per cell (which is six for hexahedral cells). The information

stored in the array is,

$$m = 0 \rightarrow H_g[id + 0 * \mathrm{n}_{fp} * \mathrm{n}_f] = \text{Current cell and SP number corresponding to FP,}$$

$$m = 1 \rightarrow H_g[id + 1 * \mathrm{n}_{fp} * \mathrm{n}_f] = \text{Neighbor cell and SP number corresponding to FP,}$$

$$= \text{or face and FP number at boundary,}$$

$$m = 2 \rightarrow H_g[id + 2 * \mathrm{n}_{fp} * \mathrm{n}_f] = n^x_{i,j,k},$$

$$m = 3 \rightarrow H_g[id + 3 * \mathrm{n}_{fp} * \mathrm{n}_f] = n^y_{i,j,k},$$

$$m = 4 \rightarrow H_g[id + 4 * \mathrm{n}_{fp} * \mathrm{n}_f] = n^z_{i,j,k},$$

$$m = 5 \rightarrow H_g[id + 5 * \mathrm{n}_{fp} * \mathrm{n}_f] = |\vec{S}^n|_{i,j,k},$$

where $id = i + j * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_f * 6$, while SP and FP correspond to solution point and flux point respectfully. In addition, another array is needed for information at the flux points to decide whether or not the point corresponds to a wall type of boundary condition,

$$V_g[i + j * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_f] = V_{i,j,k}.$$

The data stored in this array is,

$$V_g[i + j * \mathrm{n}_{fp} + k * \mathrm{n}_{fp} * \mathrm{n}_f] = 0, \text{ If not a wall boundary,}$$

$$= 1, \text{ If wall boundary,}$$

where the indexes are the same as the previous array. The next set of information needed involves updating the residual. Two separate arrays are required,

$$R^{idx}_g[m + i * 2 + k * 2 * \mathrm{n}_{sp}] = R^{idx}_{m,i,k},$$

where i and k loop through solution points and cells, while m is either 0 or 1. The array contains the data for accessing the proper memory when updating the residual, specifically,

$$m = 0 \rightarrow R^{idx}_g[0 + i * 2 + k * 2 * \mathrm{n}_{sp}] = \text{The number of updates at current solution point,}$$

$$m = 1 \rightarrow R^{idx}_g[1 + i * 2 + k * 2 * \mathrm{n}_{sp}] = \text{Number to jump in the memory access.}$$

In addition, another array is needed for accessing the correct solution point, flux point, and face.

$$R^{loc}_g[m + i * 3 + j * 3 * \mathrm{n}_{fp} + l * 3 * n_{np} * \mathrm{n}_f + k * 3 * n_{np} * \mathrm{n}_f * \mathrm{n}^{1d}_{sp}] = R^{loc}_{m,i,j,l,k}.$$

Here, m varies from 0 to 2, while i, j, l, and k, loop through flux points, faces per cell, solution points in one-dimension, and cells respectfully. The term n_{sp}^{1d} refers to the solution points in one-dimension which is equal to the order of accuracy (i.e for P^2 the value is $\mathrm{n}_{sp}^{1d} = 3$). The contents of the array are the index location of the following,

$$m = 0 \rightarrow R_g^{loc}[0 + i * 3 + j * 3 * \mathrm{n}_{fp} + l * 3 * n_{np} * \mathrm{n}_f + k * 3 * n_{np} * \mathrm{n}_f * \mathrm{n}_{sp}^{1d}] = \text{Correction},$$

$$m = 1 \rightarrow R_g^{loc}[1 + i * 3 + j * 3 * \mathrm{n}_{fp} + l * 3 * n_{np} * \mathrm{n}_f + k * 3 * n_{np} * \mathrm{n}_f * \mathrm{n}_{sp}^{1d}] = \text{Flux point},$$

$$m = 2 \rightarrow R_g^{loc}[2 + i * 3 + j * 3 * \mathrm{n}_{fp} + l * 3 * n_{np} * \mathrm{n}_f + k * 3 * n_{np} * \mathrm{n}_f * \mathrm{n}_{sp}^{1d}] = \text{Face}.$$

The contents and use of these two arrays will be further explained in the next section, when the residual update is discussed. Another two additional arrays are needed for the computation of the viscous flux. The first array stores the solution gradient,

$$Q_g^{xyz}[i + m * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = Q_{i,m,k}^{xyz},$$

where m varies from 0 to 14 while i and k run through the solution points and cells accordingly. The contents of the array are as follows,

$$m = 0 \rightarrow Q_g^{xyz}[i + 0 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho}{\partial x}\right|_{i,k},$$

$$m = 1 \rightarrow Q_g^{xyz}[i + 1 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho u}{\partial x}\right|_{i,k},$$

$$m = 2 \rightarrow Q_g^{xyz}[i + 2 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho v}{\partial x}\right|_{i,k},$$

$$m = 3 \rightarrow Q_g^{xyz}[i + 3 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho w}{\partial x}\right|_{i,k},$$

$$m = 4 \rightarrow Q_g^{xyz}[i + 4 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial e}{\partial x}\right|_{i,k},$$

$$m = 5 \rightarrow Q_g^{xyz}[i + 5 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho}{\partial y}\right|_{i,k},$$

$$m = 6 \rightarrow Q_g^{xyz}[i + 6 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho u}{\partial y}\right|_{i,k},$$

$$m = 7 \rightarrow Q_g^{xyz}[i + 7 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho v}{\partial y}\right|_{i,k},$$

$$m = 8 \rightarrow Q_g^{xyz}[i + 8 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho w}{\partial y}\right|_{i,k},$$

$$m = 9 \rightarrow Q_g^{xyz}[i + 9 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial e}{\partial y}\right|_{i,k},$$

$$m = 10 \rightarrow Q_g^{xyz}[i + 10 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho}{\partial z}\right|_{i,k},$$

$$m = 11 \rightarrow Q_g^{xyz}[i + 11 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho u}{\partial z}\right|_{i,k},$$

$$m = 12 \rightarrow Q_g^{xyz}[i + 12 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho v}{\partial z}\right|_{i,k},$$

$$m = 13 \rightarrow Q_g^{xyz}[i + 13 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial \rho w}{\partial z}\right|_{i,k},$$

$$m = 14 \rightarrow Q_g^{xyz}[i + 14 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v * 3] = \left.\frac{\partial e}{\partial z}\right|_{i,k}.$$

The second array stores the viscous flux at the solution points,

$$Fv_g^{sp}[i + m * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 12] = Fv_{i,m,k}^{sp},$$

where m varies from 0 to 11 while i and k run through the solution points and cells like before. The contents of the array are as follows,

$$m = 0 \rightarrow Fv_g^{sp}[i + 0 * n_{sp} + k * n_{sp} * 12] = \tau_{xx},$$

$$m = 1 \rightarrow Fv_g^{sp}[i + 1 * n_{sp} + k * n_{sp} * 12] = \tau_{yx},$$

$$m = 2 \rightarrow Fv_g^{sp}[i + 2 * n_{sp} + k * n_{sp} * 12] = \tau_{zx},$$

$$m = 3 \rightarrow Fv_g^{sp}[i + 3 * n_{sp} + k * n_{sp} * 12] = u * \tau_{xx} + v * \tau_{xy} + w * \tau_{xz} - q_x,$$

$$m = 4 \rightarrow Fv_g^{sp}[i + 4 * n_{sp} + k * n_{sp} * 12] = \tau_{xy},$$

$$m = 5 \rightarrow Fv_g^{sp}[i + 5 * n_{sp} + k * n_{sp} * 12] = \tau_{yy},$$

$$m = 6 \rightarrow Fv_g^{sp}[i + 6 * n_{sp} + k * n_{sp} * 12] = \tau_{zy},$$

$$m = 7 \rightarrow Fv_g^{sp}[i + 7 * n_{sp} + k * n_{sp} * 12] = u * \tau_{yx} + v * \tau_{yy} + w * \tau_{yz} - q_y,$$

$$m = 8 \rightarrow Fv_g^{sp}[i + 8 * n_{sp} + k * n_{sp} * 12] = \tau_{xz},$$

$$m = 9 \rightarrow Fv_g^{sp}[i + 9 * n_{sp} + k * n_{sp} * 12] = \tau_{yz},$$

$$m = 10 \rightarrow Fv_g^{sp}[i + 10 * n_{sp} + k * n_{sp} * 12] = \tau_{zz},$$

$$m = 11 \rightarrow Fv_g^{sp}[i + 11 * n_{sp} + k * n_{sp} * 12] = u * \tau_{zx} + v * \tau_{zy} + w * \tau_{zz} - q_z.$$

To compute the derivatives $\vec{\nabla} Q^\xi$, the values of the coefficients are stored in the array,

$$c_g[i + j * n_{sp}^{1d}] = c_{i,j},$$

where i and j both vary from 0 to $(n_{sp}^{1d} - 1)$ (see table 3.2). Next, to compute the correction term, δ, coefficients (table 3.1) are stored in the following array,

$$\alpha_g[i] = \alpha_i,$$

where i varies from 0 to $(n_{sp}^{1d} - 1)$. The final coefficient to be loaded is the viscous DG correction term as shown in table 3.3,

$$\beta_g = \beta.$$

Additional arrays are needed for specific simulation commands. For averaging the solution, we require two more arrays, one for the averaged primitive variables, and one for averaging the fluctuations, such as $\overline{u'u'}$ and $\overline{v'v'}$,

$$Q_g^{avg}[i + j * n_{sp} + k * n_{sp} * 5] = Q_{i,j,k}^{avg},$$

$$Qm_g^{avg}[i + j * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = Qm_{i,j,k}^{avg}.$$

In both arrays, i and k vary through solution points and cells, but j runs from 0 to 4 in the first array, and from 0 to 5 in the second. The contents of the arrays are as follows,

$$j = 0 \rightarrow Q_g^{avg}[i + 0 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 5] = \overline{\rho},$$

$$j = 1 \rightarrow Q_g^{avg}[i + 1 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 5] = \overline{u},$$

$$j = 2 \rightarrow Q_g^{avg}[i + 2 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 5] = \overline{v},$$

$$j = 3 \rightarrow Q_g^{avg}[i + 3 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 5] = \overline{w},$$

$$j = 4 \rightarrow Q_g^{avg}[i + 4 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 5] = \overline{p},$$

$$j = 0 \rightarrow Qm_g^{avg}[i + 0 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = \overline{u'u'},$$

$$j = 1 \rightarrow Qm_g^{avg}[i + 1 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = \overline{v'v'},$$

$$j = 2 \rightarrow Qm_g^{avg}[i + 2 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = \overline{w'w'},$$

$$j = 3 \rightarrow Qm_g^{avg}[i + 3 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = \overline{u'v'},$$

$$j = 4 \rightarrow Qm_g^{avg}[i + 4 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = \overline{v'w'},$$

$$j = 5 \rightarrow Qm_g^{avg}[i + 5 * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 6] = \overline{w'u'}.$$

The final addition to the data initialization is the insertion of a probe in the domain to measure the pressure. The probes position is defined in the CPU code, and is sent to the GPU with the array,

$$P_g[i + j * 2] = P_{i,j},$$

where i is 0 or 1 and j runs from 0 to the number of probes desired (n_p). The contents of the array are,

$$i = 0 \rightarrow P_g[0 + j * 2] = \text{Cell location of probe},$$

$$i = 1 \rightarrow P_g[1 + j * 2] = \text{SP location of probe}.$$

The locations are sent to the device, and are easily read from GPU memory for finding the pressure. After initializing the above arrays, each array is bound in textured memory on the

GPU, and one can calculate the memory usage in global memory,

$$Q_g \rightarrow Q_t \rightarrow [\text{n}_{sp} * \text{n}_v * \text{n}_c] * 8 \text{ Byte},$$

$$Q_g^o \rightarrow Q_t^o \rightarrow [\text{n}_{sp} * \text{n}_v * \text{n}_c] * 8 \text{ Byte},$$

$$Res_g \rightarrow Res_t \rightarrow [\text{n}_{sp} * \text{n}_v * \text{n}_c] * 8 \text{ Byte},$$

$$M_g \rightarrow M_t \rightarrow [\text{n}_{sp} * 11 * \text{n}_c] * 8 \text{ Byte},$$

$$BC_g \rightarrow BC_t \rightarrow [\text{n}_{bf} * 8] * 8 \text{ Byte},$$

$$Q_g^{bf} \rightarrow Q_t^{bf} \rightarrow [\text{n}_{bf} * \text{n}_v * \text{n}_{fp}] * 8 \text{ Byte},$$

$$H_g \rightarrow H_t \rightarrow [\text{n}_{fp} * \text{n}_f * \text{n}_c * 6] * 8 \text{ Byte},$$

$$V_g \rightarrow V_t \rightarrow [\text{n}_{fp} * \text{n}_f * \text{n}_c] * 1 \text{ Byte},$$

$$R_g^{idx} \rightarrow R_t^{idx} \rightarrow [\text{n}_{sp} * 2 * \text{n}_c] * 1 \text{ Byte},$$

$$R_g^{loc} \rightarrow R_t^{loc} \rightarrow [\text{n}_{fp} * \text{n}_{sp}^{1d} * 3 * 6 * \text{n}_c] * 1 \text{ Byte},$$

$$c_g \rightarrow c_t \rightarrow [\text{n}_{sp}^{1d}] * 8 \text{ Byte},$$

$$\alpha_g \rightarrow \alpha_t \rightarrow [\text{n}_{fp}] * 8 \text{ Byte},$$

$$\beta_g \rightarrow \beta_t \rightarrow [1] * 8 \text{ Byte},$$

$$Q_g^{xyz} \rightarrow Q_t^{xyz} \rightarrow [\text{n}_{sp} * \text{n}_v * \text{n}_c * 3] * 8 \text{ Byte (Viscous only)},$$

$$Fv_g^{sp} \rightarrow Fv_t^{sp} \rightarrow [\text{n}_{sp} * 12 * \text{n}_c] * 8 \text{ Byte (Viscous only)},$$

$$Q_g^{avg} \rightarrow Q_t^{avg} \rightarrow [\text{n}_{sp} * \text{n}_v * \text{n}_c] * 8 \text{ Byte (Average only)},$$

$$Qm_g^{avg} \rightarrow Qm_t^{avg} \rightarrow [\text{n}_{sp} * 6 * \text{n}_c] * 8 \text{ Byte (Average only)},$$

$$P_g \rightarrow P_t \rightarrow [\text{n}_p * 2] * 1 \text{ (Pressure probe only)}.$$

The arrays which are multiplied by 8 bytes are double precision valued, while those multiplied by 1 byte are integer valued.

4.2 CUDA implementation

All GPU code illustrated is written in CUDA C++. Exact code commands for CUDA are not included in the algorithms displayed, but the basic idea is presented. To calculate the solution for every time step, four GPU kernels are required for the inviscid flux while seven

kernels are needed for the viscous flux. The first algorithm illustrates the host CPU code for third-order Runge-Kutta time-stepping scheme, where t_{start} and t_{end} represent the starting and ending time. Each GPU kernel will be discussed in detail, giving complete dimensions for the grid and blocks used, and displaying an accurate code description.

Algorithm GPU kernel Launch Order

---
    for $i = t_{start}$ to t_{end} **do**
        Launch kernel $\rightarrow$ GPU_BC
        Launch kernel $\rightarrow$ GPU_COPY
        UpdateResiduals(1.0,1.0)
        Launch kernel $\rightarrow$ GPU_BC
        UpdateResiduals(0.75,0.25)
        Launch kernel $\rightarrow$ GPU_BC
        UpdateResiduals(1.0/3.0,2.0/3.0)
    end for

4.2.1   General CUDA kernels

Three CUDA kernels used by both the inviscid and viscous algorithms are discussed here. The kernel **GPU_COPY** copies the values from the Q_g array to the Q_g^o array, the kernel **GPU_BC** calculates the values at the boundary faces, and the **GPU_RK** kernel updates the solution in time.

GPU_COPY

This kernel simply copies values from one array to another, both of which are the same size. The threads are set-up as $\vec{t} = [t^x, t^y, t^z] = \left[n_{sp}, n_v, \frac{128}{n_{sp}} \right]$ (the number 128 was chosen because testing showed this value to give the best performance), and the block grid is defined as $b = \left[\left(\left(\frac{n_c}{t^z} \right) * \left(\frac{1}{14} \right) + 1 \right) * 14 \right]$ (14 is selected because the code primarily was run on a Tesla C2070 which has 14 streaming multiprocessors, but the number can be changed for any GPU, and the addition of 1 was selected because the divisions are integer division, rounded down). The thread grid is divided up such that t^x will calculate on the solution points, t^y will calculate on the state variables, and t^z determines how many cells are calculated per block. For example, consider a mesh with $31,255$ cells and it is desired to have a P^2 reconstruction. The kernel

thread grid would yeild $\vec{t} = [27, 5, 4]$ and the number of blocks would be $b = [7826]$. If the addition of 1 was not present in the block calculation, then the number of blocks would be $b = [7812]$. Since t^z is the number of cells calculated per block, the total number of cells in the calculation with the addition of the 1 is $t^z * b = 4 * 7826 = 31304$, which is greater than the number of cells in our domain. However, if the addition of 1 is not present, then the total number of cells in the calculation is $t^z * b = 4 * 7812 = 31248$, which is less then the number of cells in the domain, and the GPU grid is not large enough. For larger orders of P^k, t^z will be reduced to calculating 1 cell per block, due to the increase in memory required per cell. However, this presents a problem when the number of cells is greater than the maximum grid size of the GPU (65,535 blocks for a Tesla C2070). In such a case, the number of cells calculated per block is increased until the GPU grid encompasses the domain, further increasing the amount of shared memory required per block.

Algorithm GPU_COPY

$\quad i = \text{threadIdx.x}$
$\quad j = \text{threadIdx.y}$
$\quad tmp = \text{threadIdx.z}$
$\quad k = \text{blockIdx}.x * \text{blockDim}.z + tmp$
$\quad \textbf{if } k < \text{n}_c \textbf{ then}$
$\qquad Q_g^o[i + j * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v] = Q_t[i + j * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
$\quad \textbf{end if}$

In this kernel, the index i runs over all the solution points of one cell, while j runs through all the state vectors on these solution points, and tmp counts the number of cells calculated on one GPU multiprocessor. In the prior discussed example, each multiprocessor will run on 4 cells in parallel. Index k calculates the current cell in the domain, and is used in every kernel for this purpose. The kernel will continue to run and copy data from cell k while $k < \text{n}_c$.

GPU_BC

This kernel calculates the value of the solution at the boundary faces in the domain. The threads are set-up as $\vec{t} = \left[\text{n}_{fp}, \frac{128}{\text{n}_{fp}}\right]$ and the block grid is defined to be $b = \left[\left(\left(\frac{\text{n}_{bf}}{t^y}\right) * \left(\frac{1}{14}\right) + 1\right) * 14\right]$. As an example, consider a domain with 850 boundary faces for P^2. The threads become $\vec{t} = [9, 14]$ and the blocks become $b = [70]$. In the segment presented, two boundary conditions

are shown, a symmetric and fix all boundary. Here, the information from neighboring cells is required, and is read at the start. Then the solution and normals are read from the neighboring cell face, and the condition for the current cells face is evaluated. Other boundary conditions are also implemented, but they follow similar operations from those already discussed, so they shall be omitted.

Algorithm GPU_BC

$j = \text{threadIdx.x}$
$tmp = \text{threadIdx.y}$
$f = \text{blockIdx}.x * \text{blockDim}.y + tmp$
if $f < \text{n}_{bf}$ **then**
    ▷ Read type from textured memory
    $type = BC_t[f * 8]$
    if $type = \text{FIX\_ALL}$ **then**
        ▷ Read solution information from textured memory
        $Q_g^{bf}[j + (0...4) * \text{n}_{fp} + f * \text{n}_{fp} * \text{n}_v] = BC_t[(3...7) + f * 8]$
    else if $type = \text{SYMMETRY}$ **then**
        ▷ Also condition for inviscid wall
        ▷ Read cell and face location of boundary
        $cell_b = BC_t[1 + f * 8]$
        $face_b = BC_t[2 + f * 8]$
        ▷ Get index of the left cell
        $idu = H_t[j + face_b * \text{n}_{fp} + 0 * \text{n}_{fp} * \text{n}_f + cell_b * \text{n}_{fp} * \text{n}_f * 6]$
        ▷ Get normal directions
        $(n^x, n^y, n^z) = H_t[j + face_b * \text{n}_{fp} + (2...4) * \text{n}_{fp} * \text{n}_f + cell_b * \text{n}_{fp} * \text{n}_f * 6]$
        ▷ Read solution information from left cell
        $q[0...4] = Q_t[idu + (0...4) * \text{n}_{sp}]$
        ▷ Evaluate symmetric condition
        $\rho * (\vec{v} \cdot \vec{n}) = q[1] * n^x + q[2] * n^y + q[3] * n^z$
        $Q_g^{bf}[j + 0 * \text{n}_{fp} + f * \text{n}_{fp} * \text{n}_v] = q[0]$
        $Q_g^{bf}[j + 1 * \text{n}_{fp} + f * \text{n}_{fp} * \text{n}_v] = q[1] - 2 * \rho * (\vec{v} \cdot n) * n^x$
        $Q_g^{bf}[j + 2 * \text{n}_{fp} + f * \text{n}_{fp} * \text{n}_v] = q[2] - 2 * \rho * (\vec{v} \cdot n) * n^y$
        $Q_g^{bf}[j + 3 * \text{n}_{fp} + f * \text{n}_{fp} * \text{n}_v] = q[3] - 2 * \rho * (\vec{v} \cdot n) * n^z$
        $Q_g^{bf}[j + 4 * \text{n}_{fp} + f * \text{n}_{fp} * \text{n}_v] = q[4]$
    end if
end if

Table 4.1   Thread switching

P^k	n_{sp}	$n_{fp} * n_f$	Thread difference
$k = 1$	8	24	16
$k = 2$	27	54	27
$k = 3$	64	96	32
$k = 4$	125	150	25

GPU_RK

The purpose of this kernel is to update the solution in time. The thread grid is set up such that $\vec{t} = \left[n_{sp}, n_v, \frac{64}{n_{sp}} \right]$ and the blocks as $b = \left[\left(\left(\frac{n_c}{t^z} \right) * \left(\frac{1}{14} \right) + 1 \right) * 14 \right]$. As an example, consider a domain with $31,255$ cells as before, but suppose the reconstruction is P^3. Then the thread grid will become $\vec{t} = [64, 5, 1]$ and the number of blocks will be $b = [31262]$. Inputs to the kernel (ξ_1 and ξ_2) depend on which Runge-Kutta stage is calculated.

Algorithm GPU_RK(ξ_1, ξ_2)

$i = $ threadIdx.x
$j = $ threadIdx.y
$tmp = $ threadIdx.z
$k = $ blockIdx.x $*$ blockDim.z $+ tmp$
$id = i + j * n_{sp} + k * n_{sp} * n_v$
if $k < n_c$ **then**
$\quad Q_g[id] = \xi_1 * Q_t^o[id] + (1 - \xi_1) * Q_t[id] + \xi_2 * \Delta t * Res_t[id]$
end if

4.2.2   Inviscid CUDA code

This section explains the inviscid section of the CUDA code where only one additional kernel to those outlined in section 4.2.1 is required to update the inviscid residual.

Algorithm UpdateResiduals(ξ_1, ξ_2) (Inviscid only)

Launch kernel $\rightarrow$ GPU_INV_Flux
Launch kernel $\rightarrow$ GPU_RK(ξ_1, ξ_2)

GPU_INV_Flux

Here, the kernel for calculating the residual for the inviscid flux is explained. The threads of this kernel must be able to switch freely between solution points per cell and the flux points on all the faces per cell, hence the threads must always be the greater of the two values, which will be the flux points on all the faces for P^1 to P^4. The thread grid for this kernel is given as $\vec{t} = \left[n_{fp} * n_f, \frac{64}{n_{sp}} \right]$, and the blocks are given as $b = \left[\left(\left(\frac{n_c}{t^y} \right) * \left(\frac{1}{14} \right) + 1 \right) * 14 \right]$. When the kernel switches between solution points and flux points, it is computationally expensive. As an example consider table 4.1 with accuracy P^2, where the number of solution points per cell is $n_{sp} = 27$ while the number of flux points on each face is $n_{fp} * n_f = 54$. When the kernel switches from flux points on each face to solution points, there are $(n_{fp} * n_f - n_{sp}) = 54 - 27 = 27$ threads which are not active, or 50% of the block waiting. However, splitting the kernel into several smaller kernels to avoid the thread switching decreases performance, as too much data is re-loaded, hence the current configuration is optimal with the current GPU architecture. The first part of **GPU_INV_Flux** involves setting up the threads properly and loading in shared memory. The three variables ix, iy, and iz are indexes in each direction for the solution points. For P^2, $n_{sp}^{1d} = 3$, $n_{fp} = 9$, and n varies from 0 to 26. So for $n = 2$, $(ix, iy, iz) = (2, 0, 0)$, for $n = 8$, $(ix, iy, iz) = (0, 2, 0)$, and for $n = 22$, $(ix, iy, iz) = (1, 1, 2)$. The use of modular arithmetic allows the use of only one-dimensional blocks (threads in only one direction). The above can easily be done with three dimensional blocks (threads in all three directions) but yields a decrease in performance. The shared memory array is allocated with $size$, where the value of $size$ depends on the order of accuracy. For best performance, shared memory should be exactly allocated, but can be allocated for a maximum value so the code can be ran without re-compiling. The shared memory will hold data on both solution points and the flux points on the faces, so the allocation must be the maximum of the two. As an example, consider P^2 with two cells per block. Then $size$ must be the maximum of $n_{sp} * n_v * 2 = 270$ or $n_{fp} * n_f * n_v * 2 = 540$. Thus, if $size < 540$ the code will not work for P^2 with two cells calculated per block. Next, the if-statement for $(n < n_{sp})$ keeps the threads operating on the solution points while shared memory is loaded with the solution from textured memory. The

shared memory will hold information for the number of cells calculated per block, so for the P^2 example described before, the memory will contain information for two cells. Finally, the transformations from textured memory are loaded into the local memory (T_l), and the threads are synchronized to ensure all data is loaded properly.

Algorithm GPU_INV_Flux (Part 1)

$n = \text{threadIdx.x}$
$tmp = \text{threadIdx.y}$
$ix = (n \bmod \text{n}_{sp}^{1d})$
$iy = (n/\text{n}_{sp}^{1d}) \bmod \text{n}_{sp}^{1d}$
$iz = (n \bmod \text{n}_{fp})$
$k = \text{blockIdx}.x * \text{blockDim}.y + tmp$
$\_\_\text{shared}\_\_ \textbf{ double } tmp_s[size]$
$jmp_{sp} = \text{n}_{sp} * tmp$
$jmp_{fp} = \text{n}_{fp} * \text{n}_f * tmp$
if $k < \text{n}_c$ **then**
    if $n < \text{n}_{sp}$ **then**
        ▷ Load solution into shared memory
        $id = \text{n}_v * (n + jmp_{sp})$
        $tmp_s[(0...4) + id] = Q_t[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
        ▷ Load transformation information into local memory
        $T_l[0...10] = T_t[n + (0...10) * \text{n}_{sp} + k * \text{n}_{sp} * 11]$
        $\_\_$syncthreads
    ...

Part 2 computes the derivatives ($\frac{\partial Q}{\partial x}$, $\frac{\partial Q}{\partial y}$, and $\frac{\partial Q}{\partial z}$) and stores them into the local memory of each thread. The values in the computational domain of $\frac{\partial Q}{\partial \xi}$, $\frac{\partial Q}{\partial \eta}$, and $\frac{\partial Q}{\partial \zeta}$ are initialized with zeros, then every thread through the if-statement of part 1 loops through the solution points in one-dimension to calculate the computational domain derivatives and stores them in local memory. Finally, the derivatives in the computational domain are transformed into the derivatives in the physical domain and stored in local memory.

Part 3 computes the projection terms, $\Pi_j\left(\vec{\nabla} \cdot \vec{F}(Q_{i,j})\right)$, and stores them into local memory. The necessary values for calculating $\frac{\partial \vec{F}(Q_{i,j})}{\partial Q}$ are read from the shared memory array and the if-statement for $(n < \text{n}_{sp})$ is completed. The threads require synchronization from the command $\_\_$syncthreads to ensure all threads complete calculations across the solution points before proceeding to calculations through the flux points.

Algorithm GPU_INV_Flux (Part 2)

...

$\frac{\partial Q}{\partial \xi}_l[0...4] = 0, \ \frac{\partial Q}{\partial \eta}_l[0...4] = 0, \ \frac{\partial Q}{\partial \zeta}_l[0...4] = 0$

for $m = 0$ to $(\mathrm{n}_{sp}^{1d} - 1)$ **do**

    ▷ Indeces for shared memory

    $idx = \mathrm{n}_v * (m + iy * \mathrm{n}_{sp}^{1d} + iz * \mathrm{n}_{fp} + jmp_{sp})$

    $idy = \mathrm{n}_v * (ix + m * \mathrm{n}_{sp}^{1d} + iz * \mathrm{n}_{fp} + jmp_{sp})$

    $idz = \mathrm{n}_v * (ix + iy * \mathrm{n}_{sp}^{1d} + m * \mathrm{n}_{fp} + jmp_{sp})$

    ▷ Now calculate the derivatives

    $\frac{\partial Q}{\partial \xi}_l[0...4] = \frac{\partial Q}{\partial \xi}_l[0...4] + c_t[m + ix * \mathrm{n}_{sp}^{1d}] * tmp_s[(0...4) + idx]$

    $\frac{\partial Q}{\partial \eta}_l[0...4] = \frac{\partial Q}{\partial \eta}_l[0...4] + c_t[m + iy * \mathrm{n}_{sp}^{1d}] * tmp_s[(0...4) + idy]$

    $\frac{\partial Q}{\partial \zeta}_l[0...4] = \frac{\partial Q}{\partial \zeta}_l[0...4] + c_t[m + iz * \mathrm{n}_{sp}^{1d}] * tmp_s[(0...4) + idz]$

end for

▷ Transform into the physical domain

$\frac{\partial Q}{\partial x}_l[0...4] = \frac{\partial Q}{\partial \xi}_l[0...4] * T_l[0] + \frac{\partial Q}{\partial \eta}_l[0...4] * T_l[3] + \frac{\partial Q}{\partial \zeta}_l[0...4] * T_l[6]$

$\frac{\partial Q}{\partial y}_l[0...4] = \frac{\partial Q}{\partial \xi}_l[0...4] * T_l[1] + \frac{\partial Q}{\partial \eta}_l[0...4] * T_l[4] + \frac{\partial Q}{\partial \zeta}_l[0...4] * T_l[7]$

$\frac{\partial Q}{\partial z}_l[0...4] = \frac{\partial Q}{\partial \xi}_l[0...4] * T_l[2] + \frac{\partial Q}{\partial \eta}_l[0...4] * T_l[5] + \frac{\partial Q}{\partial \zeta}_l[0...4] * T_l[8]$

...

Algorithm GPU_INV_Flux (Part 3)

...

▷ Compute the projections

$\Pi_l[0] = \frac{\partial F(Q_{i,j})}{\partial \rho} * \frac{\partial Q}{\partial x}_l[0] + \frac{\partial G(Q_{i,j})}{\partial \rho} * \frac{\partial Q}{\partial y}_l[0] + \frac{\partial H(Q_{i,j})}{\partial \rho} * \frac{\partial Q}{\partial z}_l[0]$

$\Pi_l[1] = \frac{\partial F(Q_{i,j})}{\partial \rho u} * \frac{\partial Q}{\partial x}_l[1] + \frac{\partial G(Q_{i,j})}{\partial \rho u} * \frac{\partial Q}{\partial y}_l[1] + \frac{\partial H(Q_{i,j})}{\partial \rho u} * \frac{\partial Q}{\partial z}_l[1]$

$\Pi_l[2] = \frac{\partial F(Q_{i,j})}{\partial \rho v} * \frac{\partial Q}{\partial x}_l[2] + \frac{\partial G(Q_{i,j})}{\partial \rho v} * \frac{\partial Q}{\partial y}_l[2] + \frac{\partial H(Q_{i,j})}{\partial \rho v} * \frac{\partial Q}{\partial z}_l[2]$

$\Pi_l[3] = \frac{\partial F(Q_{i,j})}{\partial \rho w} * \frac{\partial Q}{\partial x}_l[3] + \frac{\partial G(Q_{i,j})}{\partial \rho w} * \frac{\partial Q}{\partial y}_l[3] + \frac{\partial H(Q_{i,j})}{\partial \rho w} * \frac{\partial Q}{\partial z}_l[3]$

$\Pi_l[4] = \frac{\partial F(Q_{i,j})}{\partial e} * \frac{\partial Q}{\partial x}_l[4] + \frac{\partial G(Q_{i,j})}{\partial e} * \frac{\partial Q}{\partial y}_l[4] + \frac{\partial H(Q_{i,j})}{\partial e} * \frac{\partial Q}{\partial z}_l[4]$

end if

__syncthreads

...

In part 4 of the kernel, the code first switches the threads to the flux points, then reads in indexes, values of the normals and face area, and solution information all from textured memory and stores then into the local memory. If the index $iP_l < 0$, then the left cell face is a boundary face, and the information must be read from Q_t^{bf}.

Algorithm GPU_INV_Flux (Part 4)

...
if $(n < n_{fp} * n_f)$ **then**
    ▷ Read information from flux points
    $iM_l = H_t[n + 0 * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
    $iP_l = H_t[n + 1 * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
    $(n_l^x, n_l^y, n_l^z) = H_t[n + (2,3,4) * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
    $|\vec{S}^n|_l = H_t[n + 5 * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
    ▷ Read information from the left solution
    $Q_l^L[0...4] = Q_t[iM_l + (0...4) * n_{sp}]$
    if $(iP_l < 0)$ **then**
        ▷ Located at a boundary
        $Q_l^R[0...4] = Q_t^{bf}[-1 - iP_l + (0...4) * n_{sp}]$
    else
        ▷ Just the neighbor cell
        $Q_l^R[0...4] = Q_t[iP_l + (0...4) * n_{sp}]$
    end if
...

Part 5 of the kernel calculates the values for ρ_L, u_L, v_L, w_L, and p_L from the Q_l^L array from part 4 and uses the data to formulate the fluxes, $F_L(Q_L)$, $G_L(Q_L)$, and $H_L(Q_L)$. Then the common flux $F_{com_l}^n(Q_l^L, Q_l^R)$ is calculated using either the Rusanov or Roe approach as described in Section 3.2, and the values are stored into the local memory of each thread. Finally, the flux difference calculation is run in local memory and saved into the shared memory array allocated previously. Storing the flux difference in shared memory will allow the threads to access the information when the kernel switches back to solution points in part 6.

The final section of **GPU_INV_Flux** computes the corrections using the lifting coefficients and the flux difference. The lifting coefficients, α_l, are read from textured memory and stored in local memory while the flux difference resides in shared memory. The indexes read from R^{idx} and R^{loc} locate the appropriate indexes for the flux points, faces, and corrections. The indexes are then used to gather the necessary information from the textured and shared memory. Prior

---

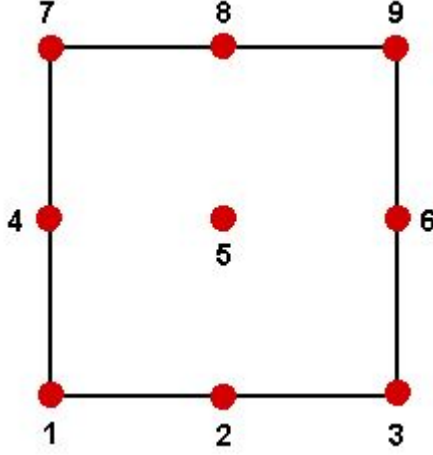**Algorithm** GPU_INV_Flux (Part 5)

   ...
   ▷ Now calculate the fluxes
   $F_l(Q_l^L)$ is calculated (see equation 3.1.3)
   $G_l(Q_l^L)$ is calculated (see equation 3.1.3)
   $H_l(Q_l^L)$ is calculated (see equation 3.1.3)
   $tmp_f[0...4] = F_l[0...4] * n_l^x + G_l[0...4] * n_l^y + H_l[0...4] * n_l^z$
   ▷ Now calculate the common flux using Roe or Rusanov
   $F_{com_l}^n(Q_l^L, Q_l^R)$ is calculated (see section 3.2)
   ▷ Formulate the flux difference
   $id = n_v * (n + jmp_{fp})$
   $tmp_s[(0...4) + id] = \left(F_{com_l}^n[0...4] - tmp_f[0...4]\right) * |\vec{S}^n|_l$
 end if
 _syncthreads
   ...

---

to this algorithm, the residual calculation was inefficient and resulted in large computational cost. On the CPU code, the residual uses a step array to jump to the appropriate residual location. Consider the algorithm **CPU_RES_Update** and let the projection terms already reside in the residual array. In addition, the example CPU code only shows the update for the corrections at one cell. The location in memory of the residual update is controlled by the step array as shown, since flux points are shared per face, but the solution point is the same. Consider figure 4.1, which shows one face of a three dimensional hexahedral cell with solution point numbering. Solution point 5 shares no common faces, hence the solution point is only updated once. However, points 2, 4, 6, and 8 all have another face in common, so the solution update depends on flux point values at two different faces. Finally, points 1, 3, 7, and 9 have three faces in common, further complicating the update. A direct implementation of this algorithm in the GPU code results in poor performance, due to non-coalesced global memory writes. Migrating to the algorithm **GPU_INV_Flux** Part 6 yields a 15 times performance increase when compared to a direct conversion of **CPU_RES_Update** on the GPU.

4.2.3 Viscous CUDA code

For the calculation of the viscous flux, three extra kernels are required in addition to those specified in Section 4.2.1. These kernels calculate the solution gradient, $\vec{\nabla}Q$, formulate the

Figure 4.1 Face flux point numbering for P^2

Algorithm GPU_INV_Flux (Part 6)

...
    if $(n < \mathrm{n}_{sp})$ **then**
        ▷ Read count per point and jump information
        $cnt_l = R_t^{idx}[0 + n*2 + k*2*\mathrm{n}_{sp}]$
        $jmp_l = R_t^{idx}[1 + n*2 + k*2*\mathrm{n}_{sp}]$
        $\delta_l[0...4] = 0$
        for $j = 0$ to $(cnt_l - 1)$ **do**
            ▷ Read each points flux, correction, and face values
            $cor_l = R_t^{loc}[0 + 3*(j + jmp_l)]$
            $fp_l = R_t^{loc}[1 + 3*(j + jmp_l)]$
            $face_l = R_t^{loc}[2 + 3*(j + jmp_l)]$
            $id = \mathrm{n}_v * (fp_l + face_l * \mathrm{n}_{fp} + jmp_{fp})$
            ▷ Formulate the corrections
            $\delta_l[0...4] = \delta_l[0...4] + \alpha_t[cor_l] * tmp_s[(0...4) + id]$
        end for
        ▷ Update the residual
        $Res_g[n + (0...4)*\mathrm{n}_{sp} + k*\mathrm{n}_{sp}*\mathrm{n}_v] = -\Pi_l[0...4] - \delta_l[0...4] * T_l[10]$
    end if
  end if

Algorithm CPU_RES_Update

$\triangleright$ Step array for jumping
step$[0...5] = (\mathrm{n}_{sp}^{1d}, -\mathrm{n}_{sp}^{1d}, -1, 1, \mathrm{n}_{fp}, -\mathrm{n}_{fp})$
for $i = 0$ to $(\mathrm{n}_f - 1)$ **do**
    for $j = 0$ to $(\mathrm{n}_{fp} - 1)$ **do**
        $ip = j * \mathrm{n}_v$
        for $m = 0$ to (n_{sp}^{1d}) **do**
            $id = j + i * \mathrm{n}_{fp} + m * \mathrm{step}[i]$
            $Res[(0....4) + id * \mathrm{n}_v] = Res[(0....4) + id * \mathrm{n}_v] - \alpha[m] * F^n[ip] * T[10 + id * 11]$
        end for
    end for
end for

---

viscous flux polynomial across the solution points, and update the residual. The calculation of $\vec{\nabla}Q$ and the viscous flux polynomial are completed in separate kernels due to performance benefits. The solution derivative is required multiple times and across neighboring cells, while computing the viscous flux polynomial with the residual involves a large memory requirement and excessive switching between solution points and flux points on faces within one kernel, limiting code performance. These reasons require the calculations to be completed separately and read from textured memory when required. Next, the three viscous flux kernels will be described in detail.

Algorithm UpdateResiduals(ξ_1,ξ_2) (Viscous only)

Launch kernel $\rightarrow$ GPU_GRAD_Q
Launch kernel $\rightarrow$ GPU_VIS_Fxyz
Launch kernel $\rightarrow$ GPU_VIS_Flux
Launch kernel $\rightarrow$ GPU_RK(ξ_1,ξ_2)

GPU_GRAD_Q

In **GPU_GRAD_Q**, the solution derivative, $\vec{\nabla}Q$, is calculated. The thread grid for this kernel is $\vec{t} = \left[\mathrm{n}_{sp}, \frac{256}{\mathrm{n}_{sp}}\right]$, and the blocks are given as $b = \left[\left(\left(\frac{\mathrm{n}_c}{t^y}\right) * \left(\frac{1}{14}\right) + 1\right) * 14\right]$. The algorithm is very similar to Part 1 and Part 2 of **GPU_INV_Flux**, except the allocation size of shared memory is larger, and the transformation to the physical domain is stored into global memory, rather than local memory. If the reconstruction is P^3 then the number of cells per block is four and the size of allocation of shared memory is, $size = \mathrm{n}_{sp} * \mathrm{n}_v * 4 = 1280$. In addition,

no if-statement is required in this kernel, since the threads are specified to run only through solution points in cells.

Algorithm GPU_GRAD_Q

$n = \text{threadIdx.x}$

$tmp = \text{threadIdx.y}$

$ix = (n \bmod \text{n}_{sp}^{1d})$

$iy = (n/\text{n}_{sp}^{1d}) \bmod \text{n}_{sp}^{1d}$

$iz = (n \bmod \text{n}_{fp})$

$k = \text{blockIdx}.x * \text{blockDim}.y + tmp$

__shared__ **double** $tmp_s[size]$

$jmp_{sp} = n_{sp} * tmp$

if $k < \text{n}_c$ **then**

   ▷ Load solution into shared memory

   $tmp_s[(0...4) + \text{n}_v * (n + jmp_{sp})] = Q_t[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$

   ▷ Load transformation information into local memory

   $T_l[0...10] = T_t[n + (0...10) * \text{n}_{sp} + k * \text{n}_{sp} * 11]$

   ▷ Ensure everything is loaded

   __syncthreads

   $\frac{\partial Q}{\partial \xi}_l = 0, \ \frac{\partial Q}{\partial \eta}_l = 0, \ \frac{\partial Q}{\partial \zeta}_l = 0$

   for $m = 0$ to $(\text{n}_{sp}^{1d} - 1)$ **do**

      ▷ Indeces for shared memory

      $idx = \text{n}_v * (m + iy * \text{n}_{sp}^{1d} + iz * \text{n}_{fp} + jmp_{sp})$

      $idy = \text{n}_v * (ix + m * \text{n}_{sp}^{1d} + iz * \text{n}_{fp} + jmp_{sp})$

      $idz = \text{n}_v * (ix + iy * \text{n}_{sp}^{1d} + m * \text{n}_{fp} + jmp_{sp})$

      ▷ Now calculate the derivatives

      $\frac{\partial Q}{\partial \xi}_l[0...4] = \frac{\partial Q}{\partial \xi}_l[0...4] + c_t[m + ix * \text{n}_{sp}^{1d}] * tmp_s[(0...4) + idx]$

      $\frac{\partial Q}{\partial \eta}_l[0...4] = \frac{\partial Q}{\partial \eta}_l[0...4] + c_t[m + iy * \text{n}_{sp}^{1d}] * tmp_s[(0...4) + idy]$

      $\frac{\partial Q}{\partial \zeta}_l[0...4] = \frac{\partial Q}{\partial \zeta}_l[0...4] + c_t[m + iz * \text{n}_{sp}^{1d}] * tmp_s[(0...4) + idz]$

   end for

   $id = n + k * 3 * \text{n}_v * \text{n}_{sp}$

   ▷ Transform into the physical domain and store in global memory

   $Q_g^{x,y,z}[id + (0...4) * \text{n}_{sp}] = \frac{\partial Q}{\partial \xi}_l[0...4] * T_l[0] + \frac{\partial Q}{\partial \eta}_l[0...4] * T_l[3] + \frac{\partial Q}{\partial \zeta}_l[0...4] * T_l[6]$

   $Q_g^{x,y,z}[id + (5...9) * \text{n}_{sp}] = \frac{\partial Q}{\partial \xi}_l[0...4] * T_l[1] + \frac{\partial Q}{\partial \eta}_l[0...4] * T_l[4] + \frac{\partial Q}{\partial \zeta}_l[0...4] * T_l[7]$

   $Q_g^{x,y,z}[id + (10...14) * \text{n}_{sp}] = \frac{\partial Q}{\partial \xi}_l[0...4] * T_l[2] + \frac{\partial Q}{\partial \eta}_l[0...4] * T_l[5] + \frac{\partial Q}{\partial \zeta}_l[0...4] * T_l[8]$

end if

GPU_VIS_Fxyz

The next kernel, **GPU_VIS_Fxyz**, computes the viscous flux at the solution points and stores the data in global memory. This thread grid is $\vec{t} = \left[\text{n}_{fp} * \text{n}_f, \frac{64}{\text{n}_{sp}}\right]$, and the blocks are

$b = \left[\left(\left(\frac{n_c}{t^y}\right) * \left(\frac{1}{14}\right) + 1\right) * 14\right]$. This kernel, like **GPU_INV_Flux**, is explained in several parts, due to the size and complexity. Part 1 starts the kernel on the flux points over the faces of the cell. Index information, face areas, and the solutions are read from textured memory, like that of **GPU_INV_Flux** Part 4. Once all the data is read, the solution difference is stored into the shared memory array. Here, the size of shared memory will be, $size = n_v * n_f * n_{fp} * t_y$, where t_y is the number of cells computed per block. Shared memory is required because data needs to be shared between the solution points and flux points. The if-statement is then ended over the flux points, and the if-statement over solution points will start in Part 2.

Algorithm GPU_VIS_Fxyz (Part 1)

   $n = \text{threadIdx.x}$
   $tmp = \text{threadIdx.y}$
   $k = \text{blockIdx}.x * \text{blockDim}.y + tmp$
   __shared__ **double** $tmp_s[size]$
   $jmp_{fp} = n_{fp} * n_f * tmp$
   if $k < n_c$ **then**
      if $(n < n_{fp} * n_f)$ **then**
         ▷ Read information from flux points
         $\text{iM}_l = H_t[n + 0 * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
         $\text{iP}_l = H_t[n + 1 * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
         $|\vec{S^n}|_l = H_t[n + 5 * n_{fp} * n_f + k * n_{fp} * n_f * 6]$
         ▷ Read information from the left solution
         $Q_l^L[0...4] = Q_t[\text{iM}_l + (0...4) * n_{sp}]$
         if $(\text{iP}_l < 0)$ **then**
            ▷ Located at a boundary
            $Q_l^R[0...4] = Q_t^{bf}[-1 - \text{iP}_l + (0...4) * n_{sp}]$
         else
            ▷ Just the nieghbor cell
            $Q_l^R[0...4] = Q_t[\text{iP}_l + (0...4) * n_{sp}]$
         end if
         $id = n_v * (n + jump_{fp})$
         ▷ Store the Q-difference
         $tmp_s[(0...4) + id] = \frac{1}{2} * \left(Q_l^R[0...4] - Q_l^L[0...4]\right) * |\vec{S^n}|_l$
      end if
      __syncthreads
      ...

Part 2 of the kernel begins with fetching the solution variables over all the solution points in the cell, then data from R_t^{idx} is read for an identical operation as described in updating the

inviscid residual. The Jacobian inverse and the solution gradient are required as well, which are stored under local memory in $\frac{1}{|J|}_l$ and $\vec{R}_l$ respectfully.

Algorithm   GPU_VIS_Fxyz (Part 2)

  ...
if $(n < \text{n}_{sp})$ **then**
    ▷ Get solution data
    $\rho_l = Q_t[n + 0 * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
    $u_l = \frac{1}{\rho_l} * Q_t[n + 1 * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
    $v_l = \frac{1}{\rho_l} * Q_t[n + 2 * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
    $w_l = \frac{1}{\rho_l} * Q_t[n + 3 * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
    $e_l = Q_t[n + 4 * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$
    ▷ Read count per point and jump information
    $cnt_l = R_t^{idx}[0 + n * 2 + k * 2 * \text{n}_{sp}]$
    $jmp_l = R_t^{idx}[1 + n * 2 + k * 2 * \text{n}_{sp}]$
    ▷ Read Jacobian inverse
    $\frac{1}{|J|}_l = T_t[n + 10 * \text{n}_{sp} + k * \text{n}_{sp} * 11]$
    ▷ Now get gradient from previous calculation
    $id = n + k * \text{n}_{sp} * \text{n}_v * 3$
    $\vec{R}_l[0...14] = Q_t^{x,y,z}[id + (0...14) * \text{n}_{sp}]$
  ...

The next part follows an identical operation from Part 6 of **GPU_INV_Flux**. Each solution point loops through cnt_l times and reads the appropriate index locations of the correction, flux point, and face. The memory location of the correction term is found with cor_l, and the normals are location with the flux point and face locations (all from textured memory and stored locally). Finally, the gradient corrections are computed from the solution difference stored in shared memory from earlier, the correction coefficient, the inverse Jacobian, and the normal directions.

Part 4 of **GPU_VIS_Fxyz** is the final section and computes the viscous flux at the solution points. All prior computations into local memory are employed to formulate the viscous flux and store the result into global memory for later use. The first four entries are $F_v(Q, \vec{\nabla}Q)$, the second four $G_v(Q, \vec{\nabla}Q)$, and the final four $H_v(Q, \vec{\nabla}Q)$. They are stored into global memory so the data can be accessed in the next kernel when the viscous flux residual is computed.

Algorithm GPU_VIS_Fxyz (Part 3)

...

for $j = 0$ to $(cnt_l - 1)$ **do**

    ▷ Read each points flux, correction, and face values

    $cor_l = R_t^{loc}[0 + 3*(j + jmp)]$

    $fp_l = R_t^{loc}[1 + 3*(j + jmp)]$

    $face_l = R_t^{loc}[2 + 3*(j + jmp)]$

    ▷ Now the normals

    $(n^x, n^y, n^z)_l = H_t[fp_l + face_l * \mathrm{n}_{fp} + (2,3,4) * \mathrm{n}_{fp} * \mathrm{n}_f + k * \mathrm{n}_{fp} * \mathrm{n}_f * 6]$

    ▷ Gradient corrections

    $id = \mathrm{n}_v * (fp_l + face_l * \mathrm{n}_{fp} + jmp_{fp})$

    $\vec{R}_l[0...4] = \vec{R}_l[0...4] + tmp_s[(0...4) + id] * \alpha_t[cor_l] * \frac{1}{|J|_l} * n_l^x$

    $\vec{R}_l[5...9] = \vec{R}_l[5...9] + tmp_s[(0...4) + id] * \alpha_t[cor_l] * \frac{1}{|J|_l} * n_l^y$

    $\vec{R}_l[10...14] = \vec{R}_l[10...14] + tmp_s[(0...4) + id] * \alpha_t[cor_l] * \frac{1}{|J|_l} * n_l^z$

end for

...

Algorithm GPU_VIS_Fxyz (Part 4)

...

▷ Compute the stress components (equation 3.1.28 and equation 3.1.29)

▷ Formulate viscous flux at solution points (F_v, G_v, H_v)

▷ Only 4 values per direction, since $\vec{F}_v[0] = 0$

$Fv_g^{sp}[n + (0...3) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 12] = F_v(Q_l, \vec{R}_l)$ (see equation 3.1.26)

$Fv_g^{sp}[n + (4...7) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 12] = G_v(Q_l, \vec{R}_l)$ (see equation 3.1.26)

$Fv_g^{sp}[n + (8...11) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 12] = H_v(Q_l, \vec{R}_l)$ (see equation 3.1.26)

    end if

end if

GPU_VIS_Flux

The final kernel for computing the viscous flux is discussed here. This thread grid and blocks are set-up identical to the previous kernel, $\vec{t} = \left[\mathrm{n}_{fp} * \mathrm{n}_f, \frac{64}{\mathrm{n}_{sp}} \right]$ and $b = \left[\left(\left(\frac{\mathrm{n}_c}{t^y} \right) * \left(\frac{1}{14} \right) + 1 \right) * 14 \right]$. The discussion of this kernel will also be completed in multiple parts due to length and complexity. Modular arithmetic is employed in this kernel, like that of **GPU_INV_Flux** Part 1. Two arrays in shared memory are required for this kernel, the size of tmp_s is $size = \mathrm{n}_v * \mathrm{n}_f * \mathrm{n}_{fp} * t_y$, same as before. The new shared memory array, f_s, will contain the viscous flux information at the solution points. Its allocation size will be $size_f = \mathrm{n}_{sp} * t_y * 12$. So for a P^2 reconstruction, $size = 540$ and $size_f = 648$ with $t_y = 2$, $\mathrm{n}_{sp} = 27$, and $\mathrm{n}_{fp} = 9$. The kernel starts with the threads operating on the flux points, reading in index locations, normals, and face areas. Then, the *bdf* term will inform the code the type of boundary at interfaces. Next, the left and right states are found, identical to previous kernels, with the addition of the inverse Jacobian at the interface of the two cells. Finally, the solution difference and addition are stored into the local memory of each thread.

In Part 2 of the kernel, the common gradient for the BR2 scheme is calculated. First, the left gradient must be read and the viscous DG correction (see table 3.3) to the gradient due to the common solution at the face must be added to it. The right solution gradient is also required, and if the face is at a boundary, then the right solution gradient is the same as the left solution gradient, otherwise, the gradient must be read in and corrected with the local lifting correction. The common gradient is then calculated and stored into local memory.

Part 3 of the kernel contains similar operations to that of **GPU_VIS_Fxyz** Part 4, with the exception that when calculating the stresses, the common gradient, $\vec{\nabla} Q_l^{com}$, will be used. The *bdf* term controls the temperature gradient ($\vec{\nabla} T$) at the interfaces for adiabatic and isothermal conditions. Once the common viscous flux is calculated, the inviscid flux and common Riemann flux must be formulated, similar to **GPU_INV_Flux** Part 5. The viscous flux found at the solution points is then read from textured memory, the total flux difference is calculated, and the result is stored into shared memory. Finally the threads are synchronized, as the kernel prepares for computations on solution points.

Algorithm GPU_VIS_Flux (Part 1)

$n = \text{threadIdx.x}$

$tmp = \text{threadIdx.y}$

$ix = (n \bmod \text{n}_{sp}^{1d})$

$iy = (n/\text{n}_{sp}^{1d}) \bmod \text{n}_{sp}^{1d}$

$iz = (n \bmod \text{n}_{fp})$

$k = \text{blockIdx}.x * \text{blockDim}.y + tmp$

__shared__ **double** $tmp_s[size]$

__shared__ **double** $f_s[size_f]$

$jmp_{sp} = \text{n}_{sp} * tmp$

$jmp_{fp} = \text{n}_{fp} * \text{n}_f * tmp$

if $k < \text{n}_c$ **then**

    if $n < \text{n}_{fp} * n_f$ **then**

        ▷ Read information from flux points, viscous boundary, and inverse Jacobian

        $\text{iM}_l = H_t[n + 0 * \text{n}_{fp} * \text{n}_f + k * \text{n}_{fp} * \text{n}_f * 6]$

        $\text{iP}_l = H_t[n + 1 * \text{n}_{fp} * \text{n}_f + k * \text{n}_{fp} * \text{n}_f * 6]$

        $(n_l^x, n_l^y, n_l^z) = H_t[n + (2, 3, 4) * \text{n}_{fp} * \text{n}_f + k * \text{n}_{fp} * \text{n}_f * 6]$

        $|\vec{S^n}|_l = H_t[n + 5 * \text{n}_{fp} * \text{n}_f + k * \text{n}_{fp} * \text{n}_f * 6]$

        $bdf = V_t[n + k * \text{n}_{fp} * \text{n}_f]$

        $\frac{1}{|J|}\Big|_l^L = T_t[\text{iM}_l - k * \text{n}_{sp} * \text{n}_v + 10 * \text{n}_{sp} + k * \text{n}_{sp} * 11]$

        ▷ Read information from the left solution

        $Q_l^L[0...4] = Q_t[\text{iM}_l + (0...4) * \text{n}_{sp}]$

        if $(\text{iP}_l < 0)$ **then**

            ▷ Located at a boundary

            $Q_l^R[0...4] = Q_t^{bf}[-1 - \text{iP}_l + (0...4) * \text{n}_{sp}]$

        else

            ▷ Just the nieghbor cell

            $Q_l^R[0...4] = Q_t[\text{iP}_l + (0...4) * \text{n}_{sp}]$

            $id = \text{iP}_l/(\text{n}_{sp} * \text{n}_v)$

            $\frac{1}{|J|}\Big|_l^R = T_t[\text{iP}_l - id * \text{n}_v * \text{n}_{sp} + 10 * \text{n}_{sp} + id * \text{n}_{sp} * 11]$

        end if

        ▷ Store the Q-difference and Q-addition

        $\delta Q_l[0...4] = \frac{1}{2}\left(Q_l^R[0...4] - Q_l^L[0...4]\right)$

        $\rho_l^p = \frac{1}{2}\left(Q_l^R[0] + Q_l^L[0]\right)$

        $(u_l^p, v_l^p, w_l^p) = \frac{1}{2}\left(Q_l^R[1, 2, 3] + Q_l^L[1, 2, 3]\right)\frac{1}{\rho_l^p}$

        $e_l^p = \frac{1}{2}\left(Q_l^R[4] + Q_l^L[4]\right)$

        ...

Algorithm GPU_VIS_Flux (Part 2)

...
▷ Read in the left gradient
$m = \text{iM}_l - k * \text{n}_{sp} * \text{n}_v + k * \text{n}_v * 3 * \text{n}_{sp}$
$Q_l^{x,L}[0...4] = Q_t^{xyz}[m + (0...4) * \text{n}_{sp}] + \beta_t * \delta Q_l[0...4] * \frac{1}{|J|}\Big|_l^L * |\vec{S^n}|_l * n_l^x$
$Q_l^{y,L}[0...4] = Q_t^{xyz}[m + (5...9) * \text{n}_{sp}] + \beta_t * \delta Q_l[0...4] * \frac{1}{|J|}\Big|_l^L * |\vec{S^n}|_l * n_l^y$
$Q_l^{z,L}[0...4] = Q_t^{xyz}[m + (10...14) * \text{n}_{sp}] + \beta_t * \delta Q_l[0...4] * \frac{1}{|J|}\Big|_l^L * |\vec{S^n}|_l * n_l^z$
if ($\text{iP}_l < 0$) **then**
    ▷ Located at a boundary
    $Q_l^{x,R}[0...4] = Q_l^{x,L}[0...4]$
    $Q_l^{y,R}[0...4] = Q_l^{y,L}[0...4]$
    $Q_l^{z,R}[0...4] = Q_l^{z,L}[0...4]$
else
    ▷ Just the nieghbor cell
    $id = \text{iP}_l/(\text{n}_{sp} * \text{n}_v)$
    $m = \text{iP}_l - id * \text{n}_v * \text{n}_{sp} + id * \text{n}_v * 3 * \text{n}_{sp}$
    $Q_l^{x,R}[0...4] = Q_t^{xyz}[m + (0...4) * \text{n}_{sp}] + \beta * \delta Q_l[0...4] * \frac{1}{|J|}\Big|_l^R * |\vec{S^n}|_l * n_l^x$
    $Q_l^{y,R}[0...4] = Q_t^{xyz}[m + (5...9) * \text{n}_{sp}] + \beta * \delta Q_l[0...4] * \frac{1}{|J|}\Big|_l^R * |\vec{S^n}|_l * n_l^y$
    $Q_l^{z,R}[0...4] = Q_t^{xyz}[m + (10...14) * \text{n}_{sp}] + \beta * \delta Q_l[0...4] * \frac{1}{|J|}\Big|_l^R * |\vec{S^n}|_l * n_l^z$
end if
▷ Now average the gradient (BR2)
$\vec{\nabla} Q_l^{com}[0...4] = \frac{Q_l^{x,R}[0...4] + Q_l^{x,L}[0...4]}{2}$
$\vec{\nabla} Q_l^{com}[5...9] = \frac{Q_l^{y,R}[0...4] + Q_l^{y,L}[0...4]}{2}$
$\vec{\nabla} Q_l^{com}[10...14] = \frac{Q_l^{z,R}[0...4] + Q_l^{z,L}[0...4]}{2}$
...

Algorithm GPU_VIS_Flux (Part 3)

...

$\triangleright$ Compute the stress components

$\triangleright$ (see equation 3.1.28 and equation 3.1.29)

if $(bdf == 1)$ **then**

$\quad \triangleright$ Adiabatic wall

$\quad \left(\frac{\partial T}{\partial x}_l, \frac{\partial T}{\partial y}_l, \frac{\partial T}{\partial z}_l\right) = (0,0,0)$

end if

$\triangleright$ Formulate the common viscous flux

$Fv_l^{com}[(0...3)] = F_v(Q_l^{com}, \vec{\nabla}Q_l^{com})$ (see equation 3.1.26)

$Gv_l^{com}[(0...3)] = G_v(Q_l^{com}, \vec{\nabla}Q_l^{com})$ (see equation 3.1.26)

$Hv_l^{com}[(0...3)] = H_v(Q_l^{com}, \vec{\nabla}Q_l^{com})$ (see equation 3.1.26)

$\triangleright$ Now get the normal direction of the common viscous flux

$Fv_{com,l}^n[(0...3)] = Fv_l^{com}[(0...3)] * n_l^x + Gv_l^{com}[(0...3)] * n_l^y + Hv_l^{com}[(0...3)] * n_l^z$

$\triangleright$ Now calculate the fluxes

$F_l(Q_l^L)$ is calculated (see equation 3.1.3)

$G_l(Q_l^L)$ is calculated (see equation 3.1.3)

$H_l(Q_l^L)$ is calculated (see equation 3.1.3)

$\triangleright$ Now calculate the common flux using Roe or Rusanov

$F_{com_l}^n(Q_l^L, Q_l^R)$ is calculated (see section 3.2)

$tmpF_l[0...4] = F_{com_l}^n[0] - (F_l[0...4] * n_l^x + G_l[0..4] * n_l^y + H_l[0...4] * n_l^z)$

$\triangleright$ Read in previous calculation of viscous flux at solution points

$Fv_l^{sp}[0...11] = Fv_t^{sp}[\text{iM} - k * \text{n}_{sp} * \text{n}_v + (0...11) * \text{n}_{sp} + k * \text{n}_{sp} * 12]$

$tmpF_l^v[0...3] = -Fv_l^{sp}[0...3] * n_l^x - Fv_l^{sp}[4...7] * n_l^y - Fv_l^{sp}[8...11] * n_l^z$

$\triangleright$ Compute and store the flux difference into shared memory

$id = n * \text{n}_v + k * \text{n}_v * \text{n}_{fp} * \text{n}_f$

$tmp_s[0 + id] = tmpF_l[0] * |\vec{S^n}|_l$

$tmp_s[1 + id] = (-Fv_{com,l}^n[0] + tmpF_l[1] - tmpF_l^v[0]) * |\vec{S^n}|_l$

$tmp_s[2 + id] = (-Fv_{com,l}^n[1] + tmpF_l[2] - tmpF_l^v[1]) * |\vec{S^n}|_l$

$tmp_s[3 + id] = (-Fv_{com,l}^n[2] + tmpF_l[3] - tmpF_l^v[2]) * |\vec{S^n}|_l$

$tmp_s[4 + id] = (-Fv_{com,l}^n[3] + tmpF_l[4] - tmpF_l^v[3]) * |\vec{S^n}|_l$

end if

__syncthreads

...

In part 4 of the kernel, all necessary data is loaded from textured memory into the local thread memory. Then the flux is transformed from the physical domain into the computational domain (i.e. $\vec{F}^v$ to $\vec{\tilde{F}}^v$) and stored into the shared memory array f_s. The threads are again synchronized to ensure all threads have written to the shared memory array before continuing.

Algorithm GPU_VIS_Flux (Part 4)

...
if $(n < \mathrm{n}_{sp})$ **then**
   ▷ Load from textured memory
   $T_l[0...10] = T_t[n + (0...10) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 11]$
   $cnt_l = R_t^{idx}[0 + n * 2 + k * 2 * \mathrm{n}_{sp}]$
   $jmp_l = R_t^{idx}[1 + n * 2 + k * 2 * \mathrm{n}_{sp}]$
   $\frac{\partial Q}{\partial x}_l[0...4] = Q_t^{x,y,z}[n + (0...4) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 3]$
   $\frac{\partial Q}{\partial y}_l[0...4] = Q_t^{x,y,z}[n + (5...9) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 3]$
   $\frac{\partial Q}{\partial z}_l[0...4] = Q_t^{x,y,z}[n + (10...14) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 3]$
   $Q_l[0...5] = Q_t[n + (0...4) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v]$
   $Fv_l^{sp}[0...11] = Fv_t^{sp}[n + (0...11) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * 12]$
   ▷ Transform the flux
   $id = 12 * (n + jmp_{sp})$
   $f_s[(0...3) + id] = (Fv_l^{sp}[0...3] * T_l[0] + Fv_l^{sp}[4...7] * T_l[1] + Fv_l^{sp}[8...11] * T_l[2]) * T_l[9]$
   $f_s[(4...7) + id] = (Fv_l^{sp}[0...3] * T_l[3] + Fv_l^{sp}[4...7] * T_l[4] + Fv_l^{sp}[8...11] * T_l[5]) * T_l[9]$
   $f_s[(8...11) + id] = (Fv_l^{sp}[0...3] * T_l[6] + Fv_l^{sp}[4...7] * T_l[7] + Fv_l^{sp}[8...11] * T_l[8]) * T_l[9]$
   __syncthreads
   ...

The next part of the viscous flux kernel uses the transformed flux variables from shared memory and forms the flux polynomial in the computational domain with a Lagrange interpolation polynomial from textured memory (see equaton 3.1.34). The data is stored in the local memory of each thread, and only has four entries in each coordinate direction, since $\vec{\tilde{F}}^v[0] = 0$.

The final part of the kernel formulates both the projections and corrections to update the residual. To compute the projections, the values for calculating $\frac{\partial \vec{F}(Q_i,j)}{\partial Q}$ are read from the local memory array, Q_l, and the solution gradient terms are read from textured memory (completed earlier). The viscous flux polynomial computed in the previous kernel is also included in the calculation (stored in $tmp_f[0...3]$). Once the projections are complete, the correction terms are calculated in the same way as discussed in part 6 of **GPU_INV_Flux**, the residual is computed, and the solution is updated with the kernel **GPU_RK**.

Algorithm GPU_VIS_Flux (Part 5)

...

$\frac{\partial F}{\partial \xi}{}_l = 0, \; \frac{\partial F}{\partial \eta}{}_l = 0, \; \frac{\partial F}{\partial \zeta}{}_l = 0$

for $m = 0$ to $(\mathrm{n}_{sp}^{1d} - 1)$ **do**

    ▷ Indeces for shared memory

    $idx = \mathrm{n}_v * (m + iy * \mathrm{n}_{sp}^{1d} + iz * \mathrm{n}_{fp} + jmp_{sp})$

    $idy = \mathrm{n}_v * (ix + m * \mathrm{n}_{sp}^{1d} + iz * \mathrm{n}_{fp} + jmp_{sp})$

    $idz = \mathrm{n}_v * (ix + iy * \mathrm{n}_{sp}^{1d} + m * \mathrm{n}_{fp} + jmp_{sp})$

    $\frac{\partial F}{\partial \xi}{}_l[0...3] = \frac{\partial F}{\partial \xi}{}_l[0...3] + c_t[m + ix * \mathrm{n}_{sp}^{1d}] * f_s[(0...3) + idx]$

    $\frac{\partial F}{\partial \eta}{}_l[0...3] = \frac{\partial F}{\partial \eta}{}_l[0...3] + c_t[m + iy * \mathrm{n}_{sp}^{1d}] * f_s[(4...7) + idy]$

    $\frac{\partial F}{\partial \zeta}{}_l[0...3] = \frac{\partial F}{\partial \zeta}{}_l[0...3] + c_t[m + iz * \mathrm{n}_{sp}^{1d}] * f_s[(8...11) + idz]$

end for

...

Algorithm GPU_VIS_Flux (Part 6)

...

$tmp_f[0...3] = \left(\frac{\partial F}{\partial \xi}{}_l[0...3] + \frac{\partial F}{\partial \eta}{}_l[0...3] + \frac{\partial F}{\partial \zeta}{}_l[0...3] \right) * T_l[10]$

▷ Projections

$\Pi_l[0] = \frac{\partial F(Q_{i,j})}{\partial \rho} \frac{\partial Q}{\partial x}{}_l[0] + \frac{\partial G(Q_{i,j})}{\partial \rho} \frac{\partial Q}{\partial y}{}_l[0] + \frac{\partial H(Q_{i,j})}{\partial \rho} \frac{\partial Q}{\partial z}{}_l[0]$

$\Pi_l[1] = \frac{\partial F(Q_{i,j})}{\partial \rho u} \frac{\partial Q}{\partial x}{}_l[1] + \frac{\partial G(Q_{i,j})}{\partial \rho u} \frac{\partial Q}{\partial y}{}_l[1] + \frac{\partial H(Q_{i,j})}{\partial \rho u} \frac{\partial Q}{\partial z}{}_l[1] - tmp_f[0]$

$\Pi_l[2] = \frac{\partial F(Q_{i,j})}{\partial \rho v} \frac{\partial Q}{\partial x}{}_l[2] + \frac{\partial G(Q_{i,j})}{\partial \rho v} \frac{\partial Q}{\partial y}{}_l[2] + \frac{\partial H(Q_{i,j})}{\partial \rho v} \frac{\partial Q}{\partial z}{}_l[2] - tmp_f[1]$

$\Pi_l[3] = \frac{\partial F(Q_{i,j})}{\partial \rho w} \frac{\partial Q}{\partial x}{}_l[3] + \frac{\partial G(Q_{i,j})}{\partial \rho w} \frac{\partial Q}{\partial y}{}_l[3] + \frac{\partial H(Q_{i,j})}{\partial \rho w} \frac{\partial Q}{\partial z}{}_l[3] - tmp_f[2]$

$\Pi_l[4] = \frac{\partial F(Q_{i,j})}{\partial e} \frac{\partial Q}{\partial x}{}_l[4] + \frac{\partial G(Q_{i,j})}{\partial e} \frac{\partial Q}{\partial y}{}_l[4] + \frac{\partial H(Q_{i,j})}{\partial e} \frac{\partial Q}{\partial z}{}_l[4] - tmp_f[3]$

$\delta_l[0...4] = 0$

for $j = 0$ to $(cnt_l - 1)$ **do**

    ▷ Read each points flux, correction, and face values

    $cor_l = R_t^{loc}[0 + 3 * (j + jmp_l)]$

    $fp_l = R_t^{loc}[1 + 3 * (j + jmp_l)]$

    $face_l = R_t^{loc}[2 + 3 * (j + jmp_l)]$

    $id = \mathrm{n}_v * (fp_l + face_l * \mathrm{n}_{fp} + jmp_{fp})$

    ▷ Formulate the corrections

    $\delta_l[0...4] = \delta_l[0...4] + \alpha_t[cor_l] * tmp_s[(0...4) + id]$

end for

▷ Update the residual

$Res_g[n + (0...4) * \mathrm{n}_{sp} + k * \mathrm{n}_{sp} * \mathrm{n}_v] = -\Pi_l[0...4] - \delta_l[0...4] * T_l[10]$

  end if

end if

To achieve good performance from the GPU code, the optimization strategy outlined in section 2.2 is followed. The following list summarizes the optimizations present in every kernel discussed.

- Keep shared memory usage low, and try to reuse shared memory

- Storage order of data in shared, textured, and global memory

- Multiple cells calculated on one multi-processor (where it is possible)

- Position of thread synchronization

- One global write per thread (where it is possible)

4.2.4 Additional CUDA kernels

In addition to the kernels discussed, three more CUDA kernels are implemented for additional data requirements. These kernels are explained in this section. Two of the three compute the mean solution and the mean fluctuations, while the third grabs pressure data at prescribed points.

GPU_Pressure

This kernel computes pressure data at prescribed positions in the flow so the pressure can be transfered to the CPU and stored. The threads are $\vec{t} = [\mathrm{n}_p]$ and the number of blocks are $b = [1]$, where n_p is the number of pressure probes in the domain. Each pressure probe is located in a cell k, with solution point index j. The cell and solution point location are found in the CPU code, and sent into textured memory to be read in this kernel. The variable idx runs through each pressure probe by reading the cell and solution point location from memory, grabbing the necessary variables from textured memory from the indexes, and computing the pressure.

Algorithm GPU_Pressure

$idx = \text{threadIdx}.x$

$k = P_t[0 + 2 * idx]$

$j = P_t[1 + 2 * idx]$

$Q_l[0...4] = Q_t[j + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$

▷ Compute pressure from Q_l and store in global memory (equation 3.1.3)

GPU_Mean

The following kernel means the flow variables through time. The thread grid for this kernel is $\vec{t} = \left[\text{n}_{sp}, \frac{256}{\text{n}_{sp}}\right]$, and the blocks are given as $b = \left[\left(\left(\frac{\text{n}_c}{t^y}\right) * \left(\frac{1}{14}\right) + 1\right) * 14\right]$. The current number of averages, n_{avg}, is input to the kernel and is incremented one when the kernel is completed. State variables and averaged variables are read into local memory from textured memory, then the variables are averaged and stored into global memory. The averaged variables are needed to compute the fluctuations in the kernel following.

Algorithm GPU_Mean

$n = \text{threadIdx.x}$

$tmp = \text{threadIdx.y}$

$k = \text{blockIdx}.x * \text{blockDim}.y + tmp$

if $k < \text{n}_c$ **then**

    $Q_l[0...4] = Q_t[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$

    $Q_l^{avg}[0...4] = Q_t^{avg}[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$

    ▷ Calculated the state variables, ρ, u, v, w, and p from Q_l

    $Q_g^{avg}[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v] = \frac{\text{n}_{avg} * Q_l^{avg}[0...4] + (\rho, u, v, w, p)}{\text{n}_{avg} + 1}$

end if

GPU_Mean

The final kernel discussed averages the fluctuations in the flow. The threads and blocks are set-up exactly as the previous kernel. The required variables are read from textured memory, the fluctuations are calculated, and the proper values are averaged and stored in the global memory array Qm_g^{avg} for post-processing.

Algorithm GPU_Mean_Fluctuation

$n = \text{threadIdx.x}$

$tmp = \text{threadIdx.y}$

$k = \text{blockIdx}.x * \text{blockDim}.y + tmp$

if $k < \text{n}_c$ **then**

$\quad Q_l[0...4] = Q_t[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$

$\quad Q_l^{avg}[0...4] = Q_t^{avg}[n + (0...4) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v]$

$\quad Qm_l^{avg}[0...5] = Qm_t^{avg}[n + (0...5) * \text{n}_{sp} + k * \text{n}_{sp} * 6]$

$\quad \triangleright$ Calculated the state variables, ρ, u, v, w, and p from Q_l

$\quad \triangleright$ Calculated the averaged state variables, $\overline{\rho}$, $\overline{u}$, $\overline{v}$, $\overline{w}$, and $\overline{p}$ from Q_l^{avg}

$\quad u' = u - \overline{u},\ v' = v - \overline{v},\ w' = w - \overline{w}$

$\quad Qm_g^{avg}[n + (0...5) * \text{n}_{sp} + k * \text{n}_{sp} * \text{n}_v] = \frac{\text{n}_{avg}*Qm_l^{avg}[0...5]+(u'u',\ v'v',\ w'w',\ u'v',\ v'w',\ w'u')}{\text{n}_{avg}+1}$

end if

CHAPTER 5. RESULTS

This chapter covers the results from the GPU CUDA CPR code. Section 1 verifies the GPU code by comparing with both the CPU code and case studies while section 2 details the performance increase from the CPU to the GPU code.

5.1 CUDA verification

The verification of the CUDA code with the CPU version is completed by monitoring the residuals at several time steps with each boundary condition implemented for both viscous and inviscid implementations. Table 5.1 shows the results from the inviscid calculation while table 5.2 shows results from viscous calculation on an arbitrary grid initialized with free stream conditions throughout the domain. In both cases, double precision accuracy was used and the polynomial order was P^2. The residual at all time is 13 plus digits accurate, ensuring the GPU code is exact to the CPU code. Additional orders of accuracy were also tested, and the residual was verified to be 13 plus digits accurate as well (not shown here).

Table 5.1   Inviscid GPU code verification (P^2)

Iteration	CPU Residual	GPU Residual
1	1.8773777499999569e-01	1.8773777499999569e-01
15	1.248354017340828e-01	1.248354017340825e-01
75	7.5362490125664539e-02	7.5362490125664455e-02
99	6.9173038873660234e-02	6.9173038873660234e-02

Table 5.2   Viscous GPU code verification (P^2)

Iteration	CPU Residual	GPU Residual
10	1.2105356644868599e-01	1.2105356644868595e-01
50	1.042350000835133e-01	1.042350000835137e-01
99	9.1455714995369808e-02	9.1455714995369711e-02
200	8.0756080475461303e-02	8.0756080475461345e-02

To test the high-order capability of the developed CUDA GPU code, the following case is considered. A cylinder is placed in the center of a domain with a pressure pulse [28] initialized to monitor wave reflections from the body. Equation 5.1.1 is used to start an initial pulse with $b = 0.2$, $\epsilon = 0.1$, $x_c = 4$, and $y_c = 0$.

$$p = p_\infty + \epsilon e^{ln2 \frac{(x-x_c)^2 + (y-y_c)^2}{b^2}} \tag{5.1.1}$$

The computational domain is taken to be [-15,15] x [0,15]. The symmetry condition is employed along $y = 0$ and characteristic outflow conditions with grid stretching along $y = 15$ and in both x directions. Grid stretching is employed to dampen all perturbations and introduce numerical dissipation [28], thus absorbing boundary conditions are not needed. The domain contains 2074 cells and the case was run for second, third, fourth, and fifth order accurate ($P^1 - P^4$), yielding 16,592 and 259,250 degrees of freedom per equation for P^1 and P^4 respectfully. The case only requires solving the Euler equations (inviscid terms). The radius of the cylinder, r, is 1 and data was recorded at two locations, both at radius $r = 5$ and angle $\theta = 90$ and 180 degrees (points A and B respectfully), where the exact solution is known. Figure 5.1 shows the pressure contours of the case at four different times for P^3, while figure 5.2 shows the pressure disturbance (p') histories, where $p = p_\infty + p'$. From the figures it is clear that second order

(a) t = 3 seconds

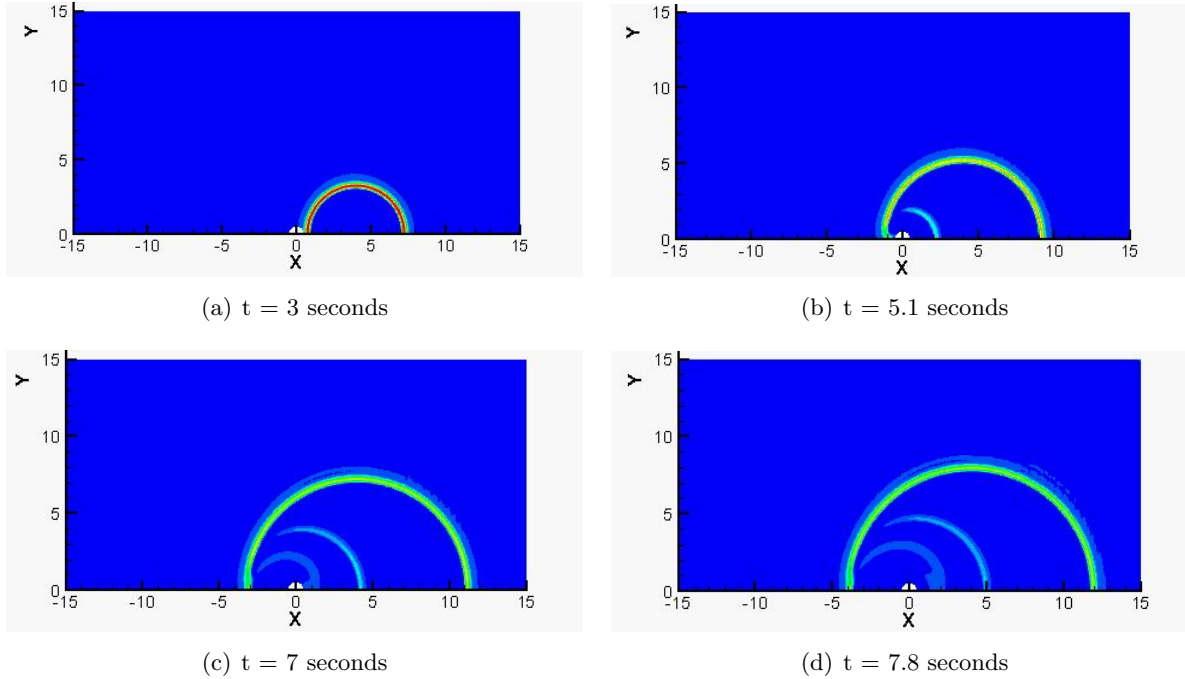(b) t = 5.1 seconds

(c) t = 7 seconds

(d) t = 7.8 seconds

Figure 5.1   Pressure contours for acoustic cylinder case

methods cannot capture the effects completely, and even third order has difficulties. Fourth and fifth orders, however, exhibit very good results, matching well with the exact solution at both points and demonstrating the ability of the CPR CUDA code to capture the small disturbances present in aeroacoustic type problems.

Larger and more complicated problems are desired to be solved as well. Consider a Selig SD7003 airfoil at a low angle of attack and low Reynolds number. The flow at the airfoils surface will separate, transition to turbulence, and reattach downstream. A fine grid is required at the surface boundary with high-order solutions to capture all aspects of the flow. Figure 5.3 illustrates the computational mesh used at the surface. The far-field was placed significantly far away (100 chord lengths) to eliminate any interaction from the boundary. The total number of elements used in the calculations was 68,040, resulting in a total number of degrees-of-freedom per equation of 1,837,080 and 4,354,560 for 3rd and 4th order runs respectfully. Simulations were carried out at a free-stream Mach number $M_\infty = 0.2$ with a chord based Reynolds number of $Re_c = 60,000$ at an angle of attack $\alpha = 4$ degrees. The simulation was ran for 17.5 non-dimensional time ($t = t^*/(c/U_\infty)$) to eliminate any spurious transient data from start-up effects.

(a) Point A   (b) Point B
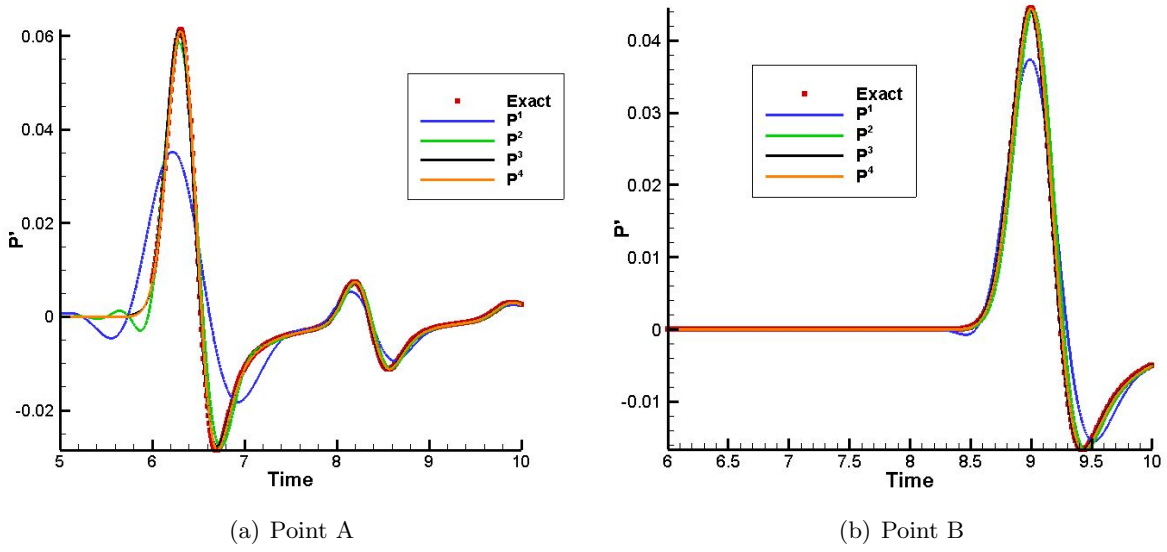
Figure 5.2 Pressure fluctuations (p')

The solution was then time-averaged for a non-dimensional time of 8 to capture the mean flow field. With a time-step of 0.000125 and 0.0001 for P^2 and P^3 respectfully, total iterations numbered 1,000,000 to 1,250,000 to reach the appropriate averaged solution.

Figure 5.4 shows the Q-criterion colored by U-velocity for the SD7003 airfoil at different orders of accuracy. The figure demonstrates the flow detaching itself from the airfoils surface and transitioning to turbulence. Reconstruction of P^3 captures more fluid structures than that of P^2, which is expected. Averaging the flow field gives more insight into the flow characteristics, including a laminar separation bubble. In figure 5.5, the laminar separation region is viewed on the upper surface, given by the blue coloring. Verification of bubble detachment and reattachment is seen in table 5.3. Results from two sources [33, 10] are shown with the results from the CPR code and good agreement is seen from all cases. For final verification of this case, the mean coefficient of pressure $(C_p = (p - p_\infty)/\frac{1}{2}\rho_\infty U_\infty^2)$ distribution on the wing surface are shown in figure 5.6 between the CPR code and the results in [33]. Once again, good agreement is shown.
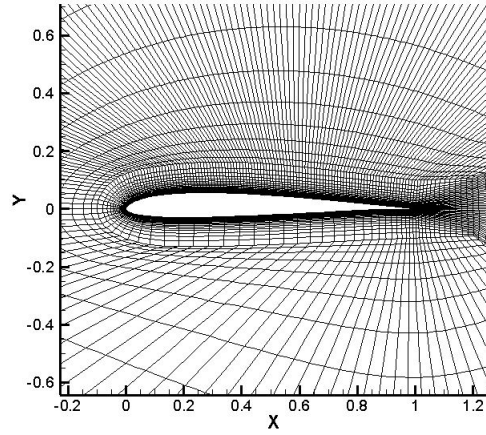
Figure 5.3   Computational grid
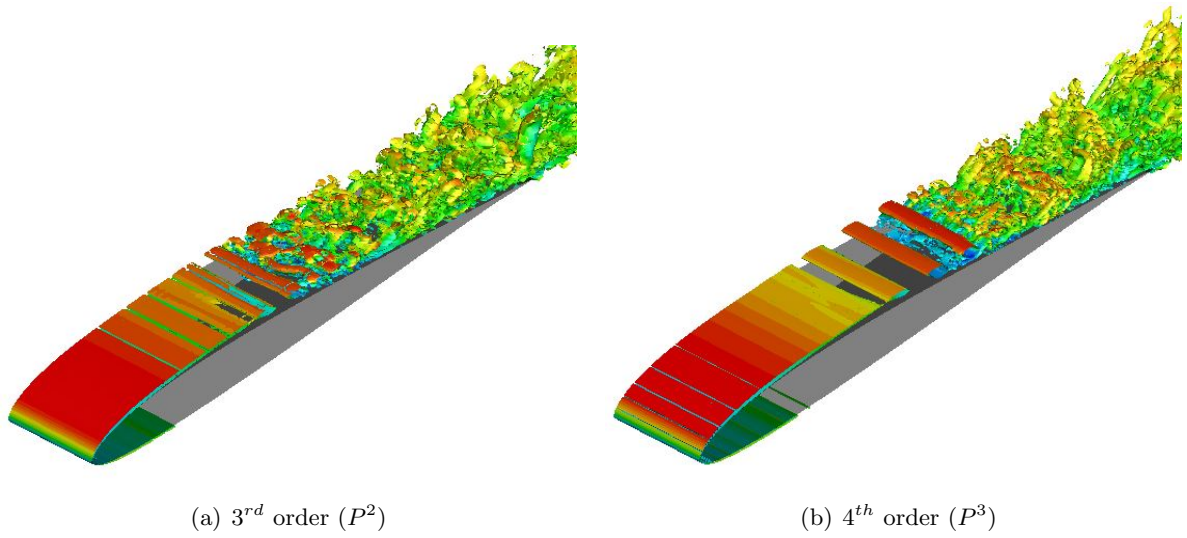
(a) 3rd order (P^2)

(b) 4th order (P^3)

Figure 5.4   Q-criterion colored by U-velocity
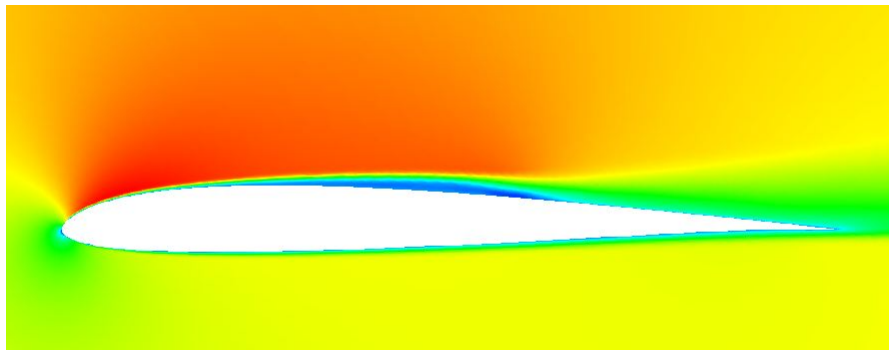

Figure 5.5   Mean u-velocity field
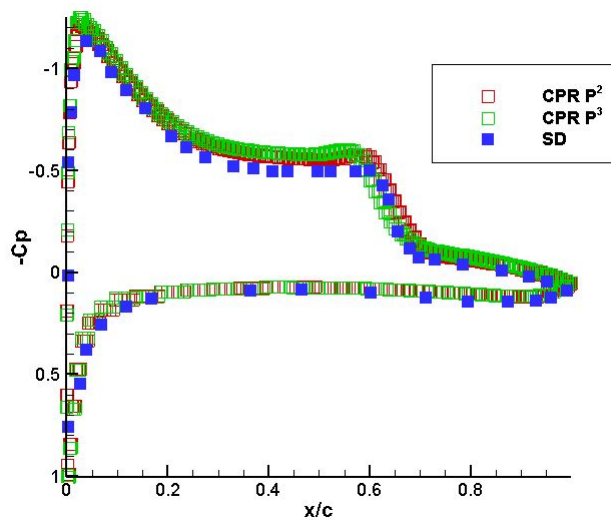
Figure 5.6   Mean coefficient of pressure $(\overline{C}_p)$

Table 5.3   Separation and reattachment locations

Case	Separation	Reattachment
Galbraith et. al. [10]	0.223	0.65
Ying et. al. [33]	0.227	0.685
CPR P^2	0.22	0.645
CPR P^3	0.221	0.683

5.2 CUDA performance

The results presented in this thesis compared one core of a CPU to that of a GPU. The CPU tested was an Intel Xeon X5650 at 2.67 GHz and the GPUs tested were a Tesla C2070 and Tesla K20x (or Kepler). The Kepler is a new generation card with six times the number of cores than the C2070. All cases presented used double precision computing with proper optimization flags employed for both CPU and GPU code compilation. Optimization of the GPU code required three stages, and two of the three optimization stages are shown in figure 5.7. The kernel, **GPU_INV_Flux** as discussed in chapter 4, was originally completed with three separate kernels: **PROJECTION**, **CORRECTION**, and **RES** to compute the projection of the flux $\left(\Pi\left[\vec{\nabla}\cdot\vec{F}(Q_i^h)\right]\right)$, the correction at the faces (δ_i), and update the residual. The majority of computational time was spent in the kernel **RES** from profiling results, hence optimizations were completed within this kernel (see **GPU_INV_Flux** part 6 and **CPU_RES_Update** in chapter 4 for residual optimization). A noticeable performance increase of the **RES** kernel was demonstrated and is shown in figure 5.7 (b). Without optimizations, 83% of the calculation was spent in calculating the residual, which was reduced to 57% after optimizing the CUDA code. However, the three individual kernels yields three separate writes into global memory, and if this memory is necessary for computations in a later kernel, it requires re-loading the memory. In addition, if the same memory is required for computations in each kernel, then the memory is loaded three times, creating a redundancy in memory access. The third optimization sought to merge the three kernels: **PROJECTION**, **CORRECTION**, and **RES**, into one kernel (**FLUX**). Joining the three kernels could potentially decrease computational time by limiting memory access and transfers. The end result of the merge for inviscid code performance is shown in figure 5.8, where the kernel **FLUX** is the kernel **GPU_INV_Flux** from chapter 4. It should be noted that combining the percentages for **PROJECTION**, **CORRECTION**, and **RES** from figure 5.7 (b) yields a value of 87%, while the kernel **FLUX** from figure 5.8 is 82%. Hence the combination of the kernels is more efficient than computing the three individual kernels. Similar optimizations of profiling kernel computations were employed for the viscous flux until an optimal configuration was found.

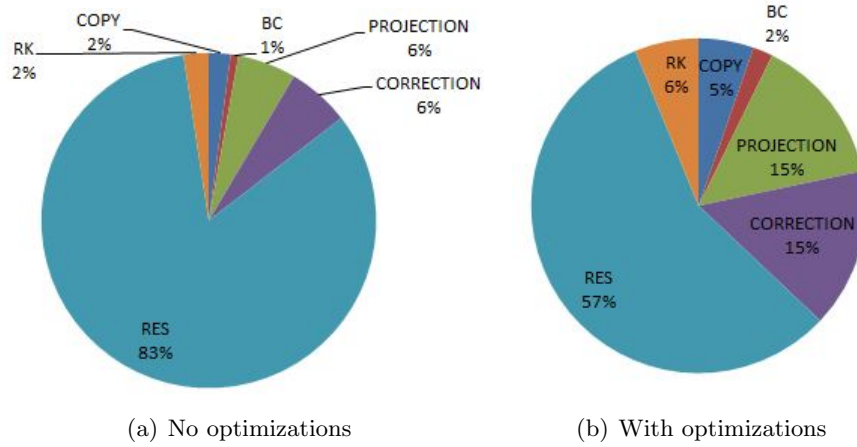(a) No optimizations

(b) With optimizations

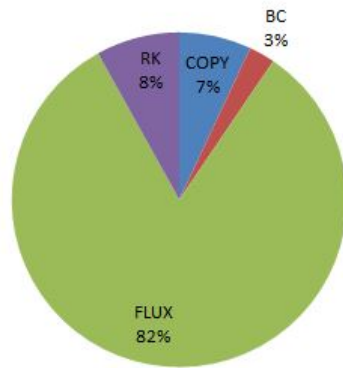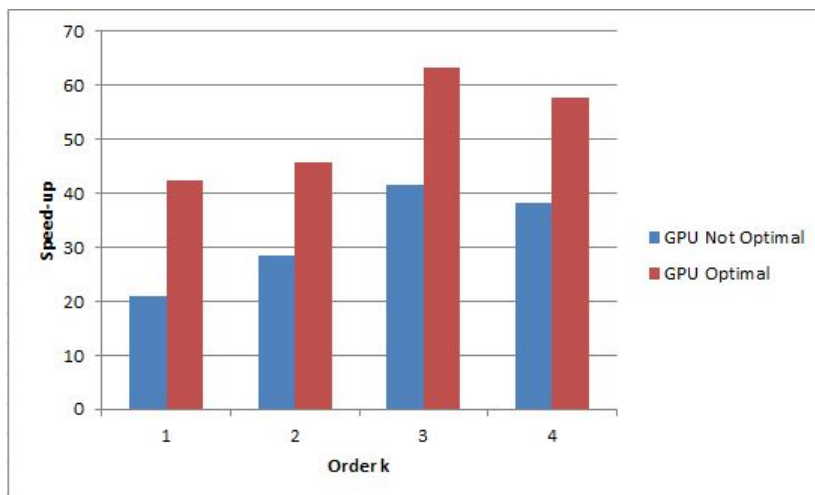Figure 5.7   Profiling for CUDA optimizations

Figure 5.8   Complete optimization profile

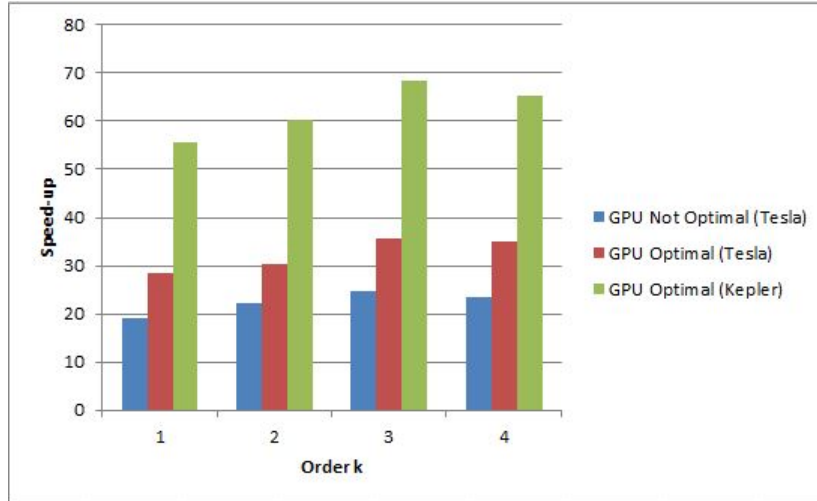Figure 5.9   Performance of GPU code compared to CPU code (inviscid) at P^1 to P^4

Figure 5.10 Performance of GPU code compared to CPU code (viscous) at P^1 to P^4

The performance of the GPU CUDA code compared to the CPU code for the inviscid flux is shown in figure 5.9. Peak speed performance is viewed at P^3 with 42 times faster at no optimizations and 63 times faster with optimizations, while P^1 demonstrated the best increase in performance when comparing the optimized to the non-optimized CUDA code (just over 2 times performance gain). The speed results for the viscous flux are shown in figure 5.10 for the initial write and an optimized code, which was run with two different GPUs (Tesla C2070 and Kepler). Peak performance is again viewed at P^3 with just over 35 times faster at the optimized Tesla card. Utilization of the new Kepler card demonstrates additional performance increases, achieving nearly 70 times faster than the CPU at P^3. Inviscid performance is superior to the viscous performance due to higher memory usage, memory reads and access, and shared memory allocation required by the viscous portion of the code. With current GPU architecture, this issue cannot be avoided, and explains why similar performance results are not seen from inviscid and viscous runs.

CHAPTER 6.   CONCLUSIONS AND FUTURE WORK

This thesis has demonstrated the efficient implementation of GPU CUDA computing with the high-order CPR method. Applying GPUs to a high-order CFD method yields huge saves in computational cost when compared to the CPU implementation, gaining orders of magnitude in computational speeds (twenty to seventy times as shown), allowing solutions to be generated quickly and computed more frequently. Similar CFD methods with local cell reconstruction, like that of CPR, could benefit further from GPU computing, depending on the efficiency of the method. Hence, the application of GPUs to current CFD solvers is a viable solution to reduce computational costs, especially solvers which use high-order methods.

Running large simulations on desktop machines in the past was unrealistic, as significant computational power was needed, which was supplied by CPU servers. However, as demands for computational power increase, CPU servers continue to expand, increasing the required space to house them, power to supply them, and money to buy them. GPUs have made desktop computations of large simulations feasible with high computational power, which is appealing to small teams with high performance computing requirements without the support for CPU servers. Additionally, GPU workstations can support a single GPU card to eight total, furthering the workstations computational power. Finally, next generations GPUs illustrate more computational power when compared to current generation cards (see figure 5.10), leading to even faster computing speeds. The gap in computing speeds from current to next generation cards is noticeable, and demonstrates a promising future for computing with CUDA.

Full utilization of GPU computing is not without flaws. The results presented in this thesis required a re-write of existing C++ code into CUDA C++, which numbered nearly 4,000 lines of new code. CPU data was restructured for efficient use with GPU computing and algorithms were manipulated and developed further for GPUs, since algorithms that excel with

CPU implementations can yield performance degradation for GPU computing. Furthermore, optimal implementation of GPU CUDA requires complete knowledge of GPU architecture and memory types, and as discussed in Chapter 5, multiple optimization steps were completed to achieve the optimal results, increasing development time. In addition, memory usage must be monitored, as new GPU cards only contain 6 giga-bytes of memory, limiting the problem size. Considerations for number of blocks, threads, and shared memory size is also needed (see Chapter 4). However, the performance benefits illustrated in this thesis demonstrate that optimized GPU implementation is significant due to computational time saved per simulation.

As GPUs continue to enhance, the existing GPU CUDA CPR code can be optimized further and continue to improve on performance. The developed GPU code only applies hexahedral cells in the domain, limiting the computational mesh. Development of tetrahedral and prism type cells for the GPU will generalize the required mesh, and allow computation across mixed-cell grids. In addition, grid adaptation and order adaptation would present a difficult challenge for GPU CUDA. Future work and development of GPUs with all Aerospace sciences (not only CFD) is strongly recommended. Solving problems orders of magnitude faster than existing applications should appeal to all researchers and developers, especially those without the support of CPU servers. Application of GPUs provides an interesting and exciting problem, which presents potentially huge performance pay-offs in all aspects of the Aerospace sciences.

APPENDIX A.   SAMPLE CUDA CODE

This appendix contains an example of a GPU CUDA program. The code will be discussed step by step, with the intent to explain every aspect, so readers can further understand the CUDA implementation as discussed in Chapter 4.

One-dimensional Lagrange interpolation

This GPU code completes a simple one-dimensional polynomial interpolation. The threads on the GPU will act on each data point for interpolation, while each block will conform to each cell in the domain. As an example, if the number of cells, $n_c = 10$, and there exist 3 points per cell for interpolation, $n_{sp} = 3$, then the GPU will run with 10 blocks, each containing 3 threads, for a total of 30 threads in the domain. The threads and blocks will be $t = [3]$ and $b = [10]$.

Algorithm GPU_Sample_Kernal

  $n = \text{threadIdx.x}$
  $k = \text{blockIdx.x}$
  __shared__ **double** $\text{tmp}_s[3]$
  ▷ First load in data points into all shared memory
  $tmp_s[n] = Q_t[n + k * n_{sp}]$
  $Q_l = 0$
  __syncthreads
  for $m = 0$ to $(n_{sp} - 1)$ **do**
    ▷ Load in Lagrange coefficient
    $c_l = c_t[n + m * n_{sp}]$
    ▷ Form the polynomial on each thread
    $Q_l = Q_l + c_l * tmp_s[m + k * n_{sp}]$
  end for
  ▷ Store the result in global memory
  $Q_g[n + k * n_{sp}] = Q_l$

In the algorithm, n will run through all solution points in each of the k, cells at the same time. The shared memory has the lifetime of each block, so it only needs to be allocated for the amount of memory it will hold in each block, and since each block is a cell which has 3 data points within it, the allocation size only needs to be n_{sp}, or 3 in this case. The data points reside in textured memory in the Q_t array which has size $n_{sp} * n_c = 30$. Each thread in each block reads the corresponding data from the array and stores it into shared memory. Thread synchronization is required to ensure all the data is loaded properly into the shared memory array before operations are carried on it. Next, the loop through n_{sp} is needed to construct the polynomial. Each thread loads in the proper Lagrange coefficient from textured memory c_t (with size $n_{sp} * n_{sp} = 9$) and stores it into the local memory of each thread. Then, the polynomial is built from the Lagrange coefficient in local memory and the data points in shared memory. Finally, the loop is finished, and the results are written to global memory. The write to global memory is coalesced since there is one write per thread and access is aligned properly.

APPENDIX B.   DERIVATION OF CORRECTION COEFFICIENTS

This appendix derives the correction coefficients as shown in table 3.1. Recall equation (3.1.12),

$$\int_{\partial V_i} W\left[F^n\right] dS = \int_{V_i} W \delta_i dV,$$

where the integration is completed over an element i with volume V_i, $[F^n]$ is the normal flux difference, δ_i is the correction function, and W is the weighting function (chosen to be Lagrange interpolation polynomials). Consider a one-dimensional cell for P^2 reconstruction as illustrated in figure B.1. The cell contains a left and right boundary with three solution points located at $x_1 = -1$, $x_2 = 0$, and $x_3 = 1$. A Lagrange interpolation polynomial is formulated at each point,

$$L_1 = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)},$$
$$L_2 = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)},$$
$$L_3 = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)},$$

where x is the solution point position. Lagrange polynomials have the property that $L_i(x_j) = 1$ when $i = j$, but when $i \neq j$ then $L_i(x_j) = 0$. The correction function is formulated by a linear combination of the correction coefficients and the Lagrange polynomials,

$$\delta_i = \alpha_1 L_1 + \alpha_2 L_2 + \alpha_3 L_3,$$

where α_j is the correction value at point j. To formulate the correction values, the correction function is employed with equation (3.1.12) and set equal to the value of the weighting function at the left or right boundary [12]. Due to symmetry, $\alpha_{L,j} = \alpha_{R,k+2-j}$ where k is the order of
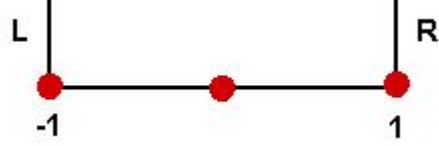
Figure B.1   One-dimensional element for P^2

accuracy [12]. The formulation for solving the correction coefficients takes the following form,

$$
\int_{V_i} W \delta_i dV =
\begin{bmatrix}
\int_{-1}^{1} L_1 L_1 & \int_{-1}^{1} L_1 L_2 & \int_{-1}^{1} L_1 L_3 \\
\int_{-1}^{1} L_2 L_1 & \int_{-1}^{1} L_2 L_2 & \int_{-1}^{1} L_2 L_3 \\
\int_{-1}^{1} L_3 L_1 & \int_{-1}^{1} L_3 L_2 & \int_{-1}^{1} L_3 L_3
\end{bmatrix}
\begin{Bmatrix}
\alpha_{L,1} \\
\alpha_{L,2} \\
\alpha_{L,3}
\end{Bmatrix}
=
\begin{bmatrix}
L_1(-1) \\
L_2(-1) \\
L_3(-1)
\end{bmatrix} .
$$

Solving the above system will result in the values at P^2 in table 3.1. Similar operations were completed to derive the coefficients for additional orders of accuracy. The coefficients found are employed to correct the flux difference through the flux points at the cell faces.

BIBLIOGRAPHY

[1] J. Andren, H. Gao, M. Yano, D. Darmofal, C. Ollivier-Gooch, & Z. J. Wang. A comparison of higher-order methods on a set of canonical aerodynamics applications. *AIAA*, 2011–3230, 2011.

[2] J. Barth & P. O. Frederickson. High-order solution of the Euler equations on unstructured grids using quadratic reconstruction. *AIAA*, 1990–0013, 1990.

[3] F. Bassi & S. Rebay. GMRES discontinuous Galerkin solution of the compressible Navier-Stokes equations. *Lecture Notes in Computational Sciences and Engineering 11*, 197–208, 2000.

[4] F. Bassi & S. Rebay. High-order accurate discontinuous finite element solution of the 2D Euler equations. *Journal of Computational Physics 138*, 251–285, 1997.

[5] C. E. Baumann & T. J. Oden. A discontinuous hp finite element method for the Euler and Navier-Stokes equations. *International Journal for Numerical Methods in Fluids. 31*(1), 79–95, 1999.

[6] R. Biswas & R. C. Strawn. A dynamic mesh adaptation procedure for unstructured hexahedral grids. *AIAA*, 1996–0027, 1996.

[7] B. Cockburn & C. W. Shu. The Runge-Kutta discontinuous Galerkin method for conservation laws V: multidimensional systems. *Journal of Computational Physics 141*, 199–224, 1998.

[8] A. Corrigan, F. Camelli, & Löhner. Running unstructured grid based CFD solvers on modern graphics hardware. *AIAA*, 2009–4001, 2009.

[9] M. Delanaye, & Y. Liu. Quadratic reconstruction finite volume schemes on 3D arbitrary unstructured polyhedral grids. *AIAA*, 1999–3529-CP, 1999.

[10] M. Galbraith, & M. Visbal. Implicit large Eddy simulation of low-Reynolds number flows past the SD7003 airfoil. *AIAA*, 2008–225, 2008.

[11] M. Hoffmann, C-D. Munz, & Z. J. Wang. Efficient implementation of the CPR formulation for the Navier-Stokes equations on GPUs. *ICCFD*, 7-2603, 2012.

[12] H. T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods. *AIAA*, 2007-4079, 2007.

[13] D. A. Jacobsen, J. C. Thibault, & I. Senocak. MPI-CUDA implementation for massively parallel incompressible flow computations on Multi-GPU clusters. *AIAA*, 2010–522, 2010.

[14] D. A. Kopriva. A staggered-grid multidomain spectral method for the compressible navier-stokes equations. *Journal of Computational Physics 143*, 125–158, 1998.

[15] D. A. Kopriva & J. H. Kolias. A conservative staggered-grid Chebyshev multidomain method for compressible flows. *Journal of Computational Physics 125*, 244, 1996.

[16] Y. Liu, M. Vinokur, & Z. J. Wang. Discontinuous spectral difference method for conservation laws on unstructured grids *Journal of Computational Physics 216*, 780–801, 2006.

[17] Y. Liu, M. Vinokur, & Z. J. Wang. Three-dimensional high-order spectral finite volume method for unstructured grids, *AIAA*, 2003-3837, 2003.

[18] NVIDIA. *CUDA C Best Practices Guide*. Ver. 4.1.

[19] NVIDIA. *NVIDIA CUDA C Programming Guide*. Ver. 5.0.

[20] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics 43*, 357–372, 1981.

[21] V. V. Rusanov. Calculation of interaction of non-steady shock waves with obstacles. *Journal of Computational Mathematical Physics USSR 1*, 267–279, 1961.

[22] C. W. Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws, in advanced numerical approximation of nonlinear hyperbolic equations. *Lecture Notes in Mathematics*, *1697*, P. 325, 1998.

[23] C. W. Shu. Total-variation-diminishing time discretizations. *SIAM Journal on Scientific and Statistical Computing. 9*, 1073–1084, 1988.

[24] Y. Sun & Z. J. Wang. High-order multidomain spectral difference method for the navier stokes equations. *AIAA*, 2006–301, 2006.

[25] Y. Sun & Z. J. Wang. High-order multidomain spectral difference method for the Navier-Stokes equations on unstructured hexahedral grids. *Journal of Computational Physics 2*(2), 301–333, 2007.

[26] J. C. Tannehill, A. A. Anderson, & R. H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Vol. 2, 1997.

[27] J. C. Thibault & I. Senocak. CUDA implementation of a Navier-Stokes solver on MultiGPU desktop platforms for incompressible flows. *AIAA*, 2009–758, 2009.

[28] M. R. Visbal & D. V. Gaitonde. Very high-order spatially implicit schemes for computational acoustics on curvilinear meshes. *Journal of Computational Acoustics*, *9*43), 1259–1286, 2001.

[29] Z. J. Wang. *Adaptive high-order methods in computational fluid dynamics* Ch.15, P. 424, 2011.

[30] Z. J. Wang. Spectral (finite) volume method for conservation laws on unstructured grids: basic formulation. *Journal of Computational Physics 178*, 210–251, 2002.

[31] Z. J. Wang & H. Gao. A residual-based procedure for hp-adaptation on 2d hybrid meshes. *AIAA*, 2011–492, 2011.

[32] Z. J. Wang, T. Haga, & H. Gao. A high-order unifying discontinuous formulation for the Navier-Stokes equations on 3D mixed grids. *Mathematical Modeling of Natural Phenomena*, *6*(3), 28–56, 2011.

[33] Y. Zhou & Z. J. Wang. Effects of surface roughness on laminar separation bubble over a wing at a low-Reynolds number. *AIAA*, 2011–736, 2011.

[34] O. C. Zienkiewicz & R. C. Tayler. *The Finite Element Method The Basics*. Vol. 1, 2000.