GPU Accelerated Kinetic Meshfree Solver for Inviscid Compressible Flows

Undergraduate Thesis

Submitted in fulfillment of the requirements of
BITS F423T Thesis

By

Harivallabha Rangarajan
ID No. 2016B4A70519H

Under the supervision of:

Dr. Anil Nemili

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, HYDERABAD CAMPUS

Declaration of Authorship

I, Harivallabha Rangarajan, declare that this Undergraduate Thesis titled, 'GPU Accelerated Kinetic Meshfree Solver for Inviscid Compressible Flows' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Harivallabha Rangarajan

Date: 26.11.2020

Certificate

This is to certify that the thesis entitled, "*GPU Accelerated Kinetic Meshfree Solver for Inviscid Compressible Flows*" and submitted by Harivallabha RANGARAJAN ID No. 2016B4A70519H in partial fulfillment of the requirements of BITS F423T Thesis embodies the work done by him under my supervision.

Supervisor
Dr. Anil NEMILI
Assistant Professor,
BITS-Pilani Hyderabad Campus
Date:

"One small step at a time. One task, one problem, one challenge. Over and over again."

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, HYDERABAD CAMPUS

Abstract

Master of Science (Hons.), Mathematics

GPU Accelerated Kinetic Meshfree Solver for Inviscid Compressible Flows

by Harivallabha RANGARAJAN

A Least Squares Kinetic Upwind Method ($\boldsymbol{q}$-LSKUM) based Meshfree Solver has been developed, and implemented in C++. This scheme has been extensively researched for over 20 years and is used for modeling the flow physics in aerodynamic shape optimization problems. In this thesis, we implement the C++ version of the serial and parallel meshfree solvers, and benchmark it against Fortran.

First, the serial version of the primal solver is implemented and validated. Second, the CUDA parallel version of the primal solver is implemented and benchmarked. Numerical simulations are performed with a NVIDIA M5000 GPU. The $\boldsymbol{q}$-LSKUM algorithm lends itself beautifully to data parallelism, which we make use of to accelerate it on the GPU. The C++ CUDA version offers a speedup of **15x** on the coarsest grid, and **23x** on the finest grid.

We present extensive comparisons against the Fortran serial and parallel versions, to both of which C++ performs comparably well. We evaluate the two codes on five different grids, with the coarsest grid containing 40K points, and the finest one containing 2.5M points. These grids have been generated using a Quadtree based point refinement method. With scope for even further optimizations of the C++ serial and CUDA versions, this thesis presents us with very promising directions for future research.

Acknowledgements

I would like to thank my supervisor Dr. Anil Nemili, for providing constant support, guidance and direction; for introducing me to Scientific Computing, and for being a great friend. This thesis would not have been possible without Anil.

Additionally, I am grateful to Nischay, Kumar, and Rupanshu for providing valuable feedback and support in collecting different code benchmarks. And, I am lucky to have found such good friends in them. Nischay has been a breath of fresh air.

I would like to express my gratitude to Mahesh Doijade, NVIDIA for his inputs on optimizing the GPU code; and Professor Suresh M. Deshpande, Jawaharlal Nehru Centre for Advanced Scientific Research, for guiding our group, and pioneering much of the research around kinetic numerical meshfree methods used in the Indian aerospace programs.

Finally, I would be remiss if I were not to mention Ritika Ramchandani, Shreyas Ravishankar, Mukundhan Jayaraman, Abhijith Ganapathiraman, and Bharathwaj Suresh for being the amazing friends that they are, and for being an integral part of my journey at BITS Pilani.

Contents

List of Figures

Abbreviations

LSKUM	**L**east **S**quares **K**inetic **U**pwind **M**ethod
GPU	**G**raphical **P**rocessing **U**nit
AD	**A**utomatic **D**ifferentiation
R-H	**R**ankine-**H**ugoniot
RANS	**R**eynolds-**A**veraged **N**avier-**S**tokes
KFVS	**K**inetic **F**lux **V**ector **S**plitting
SM	**S**treaming **M**ultiprocessor

To Krishnamachari, my grandpa. For all that I am, and shall ever be.

Chapter 1

Introduction

Meshless methods have gained increasing adoption over the last couple of decades for computing flows over complex geometries. Advancements in computational capabilities have enabled researchers to perform increasingly complex simulations and experiments. The generation of meshes over the complex geometries for for modelling aerodynamic flows, however, still poses large computational difficulties. On the other hand, improvements in meshfree methods have offered rich and promising alternative directions. In this thesis, we implement a Least Squares Kinetic Upwind Method ($\boldsymbol{q}$-LSKUM) based Meshfree Solver in C++, and accelerate it on the GPU. We show that the GPU parallel solver offers a significant speedup over the serial solver. We also present extensive benchmarks with the Fortran version of the solver, and show that the C++ solver performs comparably well with Fortran. In future, we wish to leverage multiple GPUs to further enhance the time efficiency of the solver. From a software perspective, the implementation aims to exploit the richness and efficiency of the C++ language, and avails the promise of continued support and enhancements from both the C++ as well as the CUDA C++ community thus motivating the desire for a C++ implementation of the solver.

The rest of the thesis is organized as follows: Chapter 2 offers insight into Related Work; Chapter 3 explains the theory of the LSKUM solver; Chapter 4 details implementation specifics of the serial solver; Chapter 5 offers an introduction to the NVIDIA CUDA programming model and explains the parallel implementation of the solver; In Chapter 6 we present extensive benchmarks and comparisons with Fortran; Finally, Chapter 7 presents the conclusions, and mentions directions of Future Work.

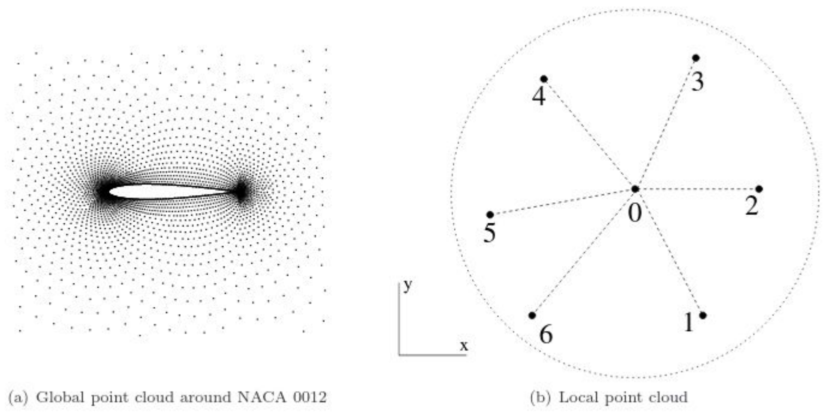
Chapter 2

Related Work

2.1 Background

The Least Squares Meshfree Method has been extensively researched by the likes of Deshpande [3, 5, 6, 10] for over two decades. The two main classes of Meshless schemes for fluid flows are based on (1) Least Squares, and (2) Radial Basis Functions [8]. In this section, we provide a brief introduction to the developments leading to the ***q***-LSKUM solver.

FIGURE 2.1: Point clouds for meshless schemes. [8]



2.1.1 A gist of the problem statement

The goal is to develop a robust algorithm to numerically solve the Euler conservation laws for inviscid compressible flows. This involves finding solutions to the Euler equation - a partial differential with respect to time, and space. And, we wish to do this in the meshless/unstructured grid regime. We consider the 2-D case here (which can later be extended to three dimensions as well). Finding the solution to the 2-D RANS equation involves two key steps: Spatial discretization, and then integration of the semi-discrete law.

2.1.2 Overview of the proposed solution, and algorithm properties

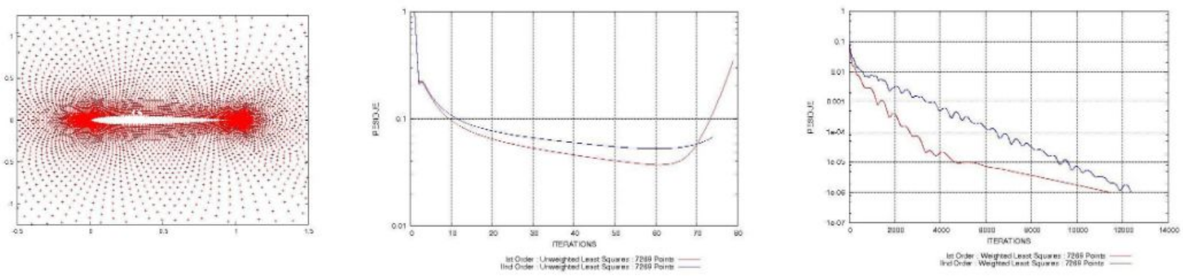
For the integration, we make use of a four-stage Runge-Kutta (SSP-RK3) time marching algorithm, with local time stepping - to get the steady state solution. This is essentially a fixed point iterative solver.

At the heart of the $\boldsymbol{q}$-LSKUM solver is a Least Squares based spatial discretization technique. The least squares method has proven to be quite powerful in the meshless regime, and captures the pressure contours, and shocks accurately [6]. Although theoretically the least squares method has not been proven to uphold conservation, empirically it is shown to satisfy the Rankine-Hugoniot (R-H) shock jump conditions thus giving accurate location of shocks and jumps. The Least Squares method is also robust to perturbations in the input point cloud distribution.

The weighted least squares method minimizes a weighted sum of squared deviations post truncation of the Taylor Series around the target point, whose spatial derivative we want to approximate numerically after discretization. Post discretization, we need to solve an over-determined system of linear equations to get the required spatial derivative. We get a closed-form expression for the approximate spatial derivatives in terms of the neighbouring points (obtained by minimizing the sum of squared deviations).

There are several points to note here. One, Since the process involves finding the solution to an over-determined linear system, an appropriate selection of the weights above has an impact on the properties of the solution. This impact could be in the form of the condition number of the Least Squares (LS) matrix that has an effect on the stability of the solution; It could be in the form of spectral resolution of the solution - one could select an optimal configuration of weights such that the resulting solution has better spectral properties, and captures the lower wavelengths as it does for the higher wavelengths. Likewise, it could affect the robustness of the solver: By avoiding rank-deficiency in the LS matrix, one could make the solver more robust. This is precisely what Deshpande and Arora show in [1]: By choosing the weights along the eigen directions of the spatial co-ordinates, they ensure that the LS matrix is of high rank. Perturbed input point distributions that lead to smeared pressure contours in the solution often have rank deficient LS matrices as compared to the so called good point distributions.

FIGURE 2.2: Robustness of the WLSKUM-ED over conventional LSKUM on tampered point distribution [1]

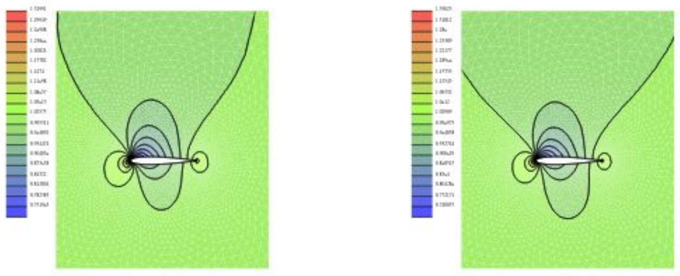


The effect of choosing such weights is illustrated in the figure above: Tampered point distribution (left); Results with conventional LSKUM, solution diverges (centre); Results with conventional LSKUM, solution diverges (centre); Results with WLSKUM-ED, solution converges (right).

Coming to the second point: the formal accuracy of the LS solution depends on the location of the truncation in the Taylor Series expansion. For instance, truncation beyond the linear terms leads to first-order accurate spatial derivatives. Deshpande and Ghosh [7] show that it is possible to achieve second order accuracy by employing a method known as defection correction. Defect correction is used to obtain a closed form solution for the spatial derivative that is similar in form to the first order solution but actually has properties of the second order solution. In other words, we obtain second order accuracy while retaining a form similar to the first order solution - this equips the LS matrix with the properties of the first order solution - its robustness, and conditioning; and, enables the second order accurate solution to have crisp flow features whilst giving the smooth, wiggle-free pressure contours just like in the first order accurate numerical scheme (which, on the other hand, smears the flow features). The pressure contours captured by the second order accurate LS solution is sharper, and does not smear close to the boundary. It should be noted that this defect correction procedure, in practice, is to be implemented with inner iterations to achieve this second order accuracy (in practice). Defect correction, therefore enables second order accuracy by using perturbed or modified Maxwellian distributions of the velocity.

We refer the reader to [7], [6] and [1] for a more thorough discussion of the above. The entire pipeline is formulated and implemented in the microscopic regime described by Boltzmann's 2-D equations for the Kinetic theory of gases. This is referred to as Kinetic Flux Vector Splitting (KFVS). Following this, the solution is propagated upward to the macroscopic regime in the Eulerian limit through the moment method strategy. By taking the inner product of the moments and Maxwellian derivatives, and integrating it in the Eulerian limit we arrive at a

FIGURE 2.3: Pressure Contours for Subsonic flow test case over NACA 0012 Aerofoil using $\boldsymbol{q}$-LSKUM. Without inner iterations (left) and with inner iterations (right) [1]

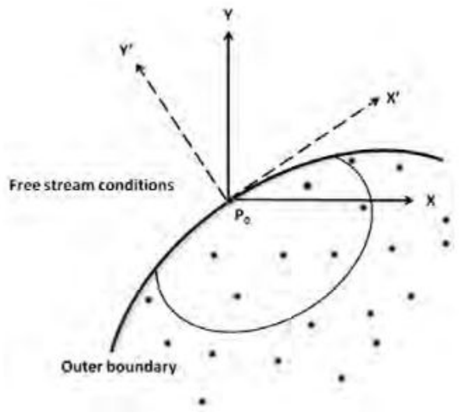


scheme at the macroscopic level [7]. In this manner, we develop a spatial discretization algorithm to solve the differential equation at the Boltzmann level and then upwind it to the Euler equations.

Now, two issues remain: One, we need to formulate an update rule for the time derivative of the conserved vector. Two, we need to solve a potential problem in the perturbation procedure used in defect correction.

For the first, we (again) derive update rules for the interior, wall and outer (boundary) points at the Boltzmann level and then use the moment method strategy to upwind it to the Euler level. The interior point updates are easy to perform because of the availability of suitable neighbouring points. At the wall and boundary points, we need to be cognizant of the incoming and outgoing fluxes and derive an appropriate update rule. This has been described in Deshpande et. al. [6]

FIGURE 2.4: Neighbourhood stencil around a far-field boundary point [6]



Now for the second issue: Defect correction essentially employs perturbed Maxwellians, which, as it turns out, could result in a non-Maxwellian distribution. Therefore, Deshpande et. al. [3] formulate a strategy involving entropy variables or $\boldsymbol{q}$-variables to cast the Euler equations in symmetric hyperbolic form. The $\boldsymbol{q}$-variables are uniquely mapped to the Maxwellian F, and the conserved vector, $\boldsymbol{U}$. Therefore, we can make perturbations in the $\boldsymbol{q}$ domain and uniquely map them back to F and $\boldsymbol{U}$. Since the $\boldsymbol{q}$-variables have a logarithmic dependence on $[\rho, u, \beta]$, it can be shown that a linear combination of $\boldsymbol{q}$-variables is mapped back to a valid Maxwellian distribution. Hence, the defect correction procedure (which, now, involves a linear combination of $\boldsymbol{q}$-variables) can be successfully employed in the $\boldsymbol{q}$ domain whilst ensuring that the resultant velocity distribution is still a Maxwellian.

Circling back to the issue of the lack of a theoretical proof for the conservation property of LSKUM, one should note that this is because the least squares method decomposes the space into connectivity sets that form an overlapping cover of the point cloud distribution. This is in contrast to the finite volume method, where a triangulation of the space results in non-overlapping mutually disjoint sets. In the latter case, it is easy to prove conservation by pairwise cancellation of the fluxes at the common edges of the triangulated space. However, as has been empirically shown, the LSKUM method accurately captures solutions yielding all flow features prompting Deshpande and Ramesh et. al [6] to hypothesize that proven conservation is a sufficient but not necessary condition for obtaining accurate R-H jump conditions across shocks.

This section was meant to give the reader an intuitive understanding of the $\boldsymbol{q}$-LSKUM solver, and its properties. The following chapter formalizes all of the discussion above, and lays down the mathematics of the solver.

Chapter 3

Least Squares Kinetic Upwind Meshfree Solver

3.1 The Kinetic Meshfree Method: *q*-LSKUM

3.1.1 Basic theory of LSKUM

The Least Squares Kinetic Upwind Method (LSKUM) belongs to the family of kinetic theory based upwind schemes for the numerical solution of Euler or Navier-Stokes equations that govern the compressible fluid flows. These schemes are based on the moment-method-strategy [10], where an upwind scheme is first developed at the Boltzmann level and after taking suitable moments, we arrive at an upwind scheme for the governing conservation laws. In this section, we briefly present the basic theory of LSKUM for $2-D$ Euler equations that govern the inviscid compressible fluid flows.

In the differential form, the Euler equations in two-dimensions are given by

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{G}}{\partial x} + \frac{\partial \boldsymbol{H}}{\partial y} = 0 \tag{3.1}$$

Here, $\boldsymbol{U}$ is the conserved vector, $\boldsymbol{G}$ and $\boldsymbol{H}$ are the flux vectors along the coordinate directions x and y respectively. These vectors are given by

$$\boldsymbol{U} = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho e \end{bmatrix}, \boldsymbol{G} = \begin{bmatrix} \rho u_1 \\ p + \rho u_1^2 \\ \rho u_1 u_2 \\ (p + \rho e)\, u_1 \end{bmatrix}, \boldsymbol{H} = \begin{bmatrix} \rho u_2 \\ \rho u_1 u_2 \\ p + \rho u_2^2 \\ (p + \rho e)\, u_2 \end{bmatrix} \tag{3.2}$$

Here ρ is the mass density, u_1 and u_2 are the cartesian components of the fluid velocity along the coordinate directions x and y respectively and p is the pressure. e is the specific total energy per unit mass, given by

$$e = \frac{p}{\rho\left(\gamma - 1\right)} + \frac{1}{2}\left(u_1^2 + u_2^2\right) \tag{3.3}$$

where, γ is the ratio of specific heats. The conservation laws in eq. (3.1) can be obtained by taking Ψ - moments of the $2-D$ Boltzmann equation in the Euler limit. In the inner product form, this can be written as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{G}}{\partial x} + \frac{\partial \boldsymbol{H}}{\partial y} = \left\langle \Psi, \frac{\partial F}{\partial t} + v_1 \frac{\partial F}{\partial x} + v_2 \frac{\partial F}{\partial y} \right\rangle = 0 \tag{3.4}$$

Here, F is the Maxwellian velocity distribution function, given by

$$F = \frac{\rho}{I_0}\frac{\beta}{\pi} exp\left[-\beta\left\{\left(v_1 - u_1\right)^2 + \left(v_2 - u_2\right)^2\right\} - \frac{I}{I_0}\right] \tag{3.5}$$

where, v_1 and v_2 are the molecular velocities along the coordinate directions x and y respectively. $\beta = 1/\left(2RT\right)$, R is the gas constant per unit mass, T is the absolute temperature, I is the internal energy variable and I_0 is the internal energy due to non-translational degrees of freedom. The moment function vector Ψ is defined by $\Psi = \begin{bmatrix} 1 & v_1 & v_2 & I + \frac{v_1^2 + v_2^2}{2} \end{bmatrix}^T$. The inner product $\langle \Psi, f \rangle$ is defined by

$$\langle \Psi, f \rangle = \int\limits_{\mathbb{R}^+ \times \mathbb{R}^2} \Psi f\left(\boldsymbol{v}\right) d\boldsymbol{v} dI \tag{3.6}$$

Using Courant-Issacson-Rees (CIR) splitting [2] of molecular velocities, an upwind scheme for the Boltzmann equation in eq. (3.4) can be constructed as

$$\frac{\partial F}{\partial t} + v_1^+ \frac{\partial F}{\partial x} + v_1^- \frac{\partial F}{\partial x} + v_2^+ \frac{\partial F}{\partial y} + v_2^- \frac{\partial F}{\partial y} = 0 \tag{3.7}$$

where, $v_1^{\pm}$ and $v_2^{\pm}$ are defined as

$$v_1^{\pm} = \frac{v_1 \pm |v_1|}{2}, \quad v_2^{\pm} = \frac{v_2 \pm |v_2|}{2} \tag{3.8}$$

The basic idea of LSKUM is to obtain discrete approximations to the spatial derivatives using least squares principle. We illustrate this approach to determine F_x and F_y at a point P_0 using the data at its neighbours. The set of neighbours, also known as the stencil of P_0 is defined by

$$N\left(P_0\right) = \left\{P_i : d\left(P_0, P_i\right) < \epsilon\right\} \tag{3.9}$$

where $d\left(P_i, P_0\right)$ is the Euclidean distance between the points P_0 and P_i. ϵ is the user defined characteristic linear dimension of $N\left(P_0\right)$.

Consider the Taylor series expansion of F up to linear terms at a neighbour point P_i around P_0

$$\Delta F_i = \Delta x_i F_{x0} + \Delta y_i F_{y0} + O\left(\Delta x, \Delta y\right)^2, i = 1, \ldots, n \tag{3.10}$$

where $\Delta x_i = x_i - x_0$, $\Delta y_i = y_i - y_0$, $\Delta F_i = F_i - F_0$ and n represents the number of neighbours of the point P_0. For $n \geq 3$, eq. (3.10) leads to an over-determined linear system, which can be solved using the least squares principle. The first-order accurate least squares approximations to the partial derivatives F_x and F_y at the point P_0 are then given by

$$\begin{bmatrix} F_x^{1} \\ F_y^{1} \end{bmatrix} = \begin{bmatrix} \sum \Delta x_i^2 & \sum \Delta x_i \Delta y_i \\ \sum \Delta x_i \Delta y_i & \sum \Delta y_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum \Delta x_i \Delta F_i \\ \sum \Delta y_i \Delta F_i \end{bmatrix} \tag{3.11}$$

In the above formulae, the subscript *1* on F_x and F_y denotes first-order accuracy. Taking Ψ - moments of eq. (3.7) along with the formulae in eq. (3.11), we obtain the semi-discrete form of the first-order least squares kinetic upwind scheme for $2D$ Euler equations,

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{G}^+}{\partial x} + \frac{\partial \boldsymbol{G}^-}{\partial x} + \frac{\partial \boldsymbol{H}^+}{\partial y} + \frac{\partial \boldsymbol{H}^-}{\partial y} = 0 \tag{3.12}$$

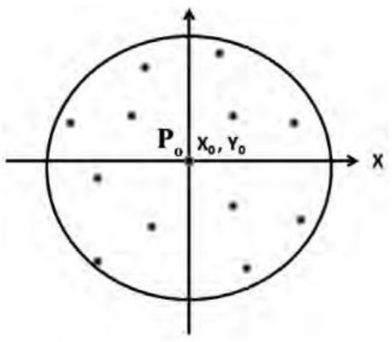
Here, $\boldsymbol{G}^{\pm}$ and $\boldsymbol{H}^{\pm}$ are respectively the kinetic split fluxes [4] along x and y directions. The least squares formulae for the split flux derivatives $\boldsymbol{G}_x^{\pm}$ and $\boldsymbol{H}_y^{\pm}$ are given by

$$\begin{aligned} \boldsymbol{G}_x^{\pm} &= \left[\frac{\sum \Delta y_i^2 \sum \Delta x_i \Delta \boldsymbol{G}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta y_i \Delta \boldsymbol{G}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \sum \Delta x_i \Delta y_i}\right] \\ \boldsymbol{H}_y^{\pm} &= \left[\frac{\sum \Delta x_i^2 \sum \Delta y_i \Delta \boldsymbol{H}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta x_i \Delta \boldsymbol{H}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \sum \Delta x_i \Delta y_i}\right] \end{aligned} \tag{3.13}$$

Note that $\boldsymbol{G}_x^{\pm}$ and $\boldsymbol{H}_y^{\pm}$ are evaluated using the split stencils $N_x^{\pm}\left(P_0\right)$ and $N_y^{\pm}\left(P_0\right)$ respectively. These subsets are defined by

$$\begin{aligned} N_x^{\pm}\left(P_0\right) &= \left\{P_i \mid P_i \in N\left(P_0\right), \Delta x_i = x_i - x_0 \lessgtr 0\right\} \\ N_y^{\pm}\left(P_0\right) &= \left\{P_i \mid P_i \in N\left(P_0\right), \Delta y_i = y_i - y_0 \lessgtr 0\right\} \end{aligned} \tag{3.14}$$

FIGURE 3.1: Connectivity of node P_0 in two dimensions. [6]



3.1.2 Second-order accuracy using q-variables

One way of obtaining second-order accurate approximations to the spatial derivatives F_x and F_y is by considering the Taylor series expansion of F up to quadratic terms

$$\begin{aligned}\Delta F_i =& \Delta x_i F_{x0} + \Delta y_i F_{y0} + \frac{\Delta x_i^2}{2} F_{xx0} + \Delta x_i \Delta y_i F_{xy0} \\ &+ \frac{\Delta y_i^2}{2} F_{yy0} + O\left(\Delta x_i, \Delta y_i\right)^3, \quad i = 1, \dots, n\end{aligned} \tag{3.15}$$

For $n \geq 6$, we get an over-determined linear system of the form

$$\begin{bmatrix} \Delta x_1 & \cdots & \frac{\Delta y_1^2}{2} \\ \Delta x_2 & \cdots & \frac{\Delta y_2^2}{2} \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ \Delta x_n & \cdots & \frac{\Delta y_n^2}{2} \end{bmatrix} \begin{bmatrix} F_{x0} \\ F_{y0} \\ \vdots \\ \vdots \\ F_{yy0} \end{bmatrix} = \begin{bmatrix} \Delta F_1 \\ \Delta F_2 \\ \vdots \\ \vdots \\ \Delta F_n \end{bmatrix} \tag{3.16}$$

If we denote the coefficient matrix as A, the unknown vector as $\boldsymbol{dF}$ and the right hand side vector as $\boldsymbol{\Delta F}$, then the solution of the linear system using least squares is given by

$$\boldsymbol{dF} = \left(A^T A\right)^{-1} \left(A^T \boldsymbol{\Delta F}\right) \tag{3.17}$$

The first two elements of the vector $\boldsymbol{dF}$ give the desired approximations to F_x and F_y. It can be observed that these formulae involve the inverse of a 5×5 least squares matrix $A^T A$. Central to the success of this formulation is that this matrix should be well-conditioned. In the case of first-order approximation, the 2×2 least squares matrix corresponding to a point becomes singular if and only if the points in its stencil lie on a straight line. On the other hand, the least squares matrix in the second-order formulae can become singular if the alignment of the stencil is such that at least two rows of the matrix $A^T A$ are linearly dependent. Furthermore, it lacks robustness as the least squares matrix corresponding to the boundary points can be poorly conditioned, which results in loss of accuracy.

Alternatively, second-order accuracy can be achieved by employing the defect correction method. An advantage of this approach is that the dimension of the least squares matrix remains the same as in the first-order scheme. To derive the desired formulae, we rearrange the eq. (3.15) as

$$\begin{aligned}\Delta F_i =& \Delta x_i F_{x0} + \Delta y_i F_{y0} + \frac{\Delta x_i}{2}\left(\Delta x_i F_{xx0} + \Delta y_i F_{xy0}\right) \\ &+ \frac{\Delta y_i}{2}\left(\Delta x_i F_{xy0} + \Delta y_i F_{yy0}\right) \\ &+ O\left(\Delta x_i, \Delta y_i\right)^3, \quad i = 1, \dots, n\end{aligned} \tag{3.18}$$

The basic idea of the defect correction procedure is to cancel the second-order derivative terms in the above equation by defining a modified ΔF_i so that the leading terms in the truncation errors for the formuale for F_x and F_y are of the order of $O\left(\Delta x_i, \Delta y_i\right)^2$. Towards this objective, consider the Taylor series expansions of F_x and F_y up to linear terms

$$\begin{aligned} \Delta F_{x_i} =& \Delta x_i F_{xx_0} + \Delta y_i F_{xy_0} + O\left(\Delta x_i, \Delta y_i\right)^2 \\ \Delta F_{y_i} =& \Delta x_i F_{xy_0} + \Delta y_i F_{yy_0} + O\left(\Delta x_i, \Delta y_i\right)^2 \end{aligned} \tag{3.19}$$

where $\Delta F_{x_i} = F_{x_i} - F_{x_0}$ and $\Delta F_{y_i} = F_{y_i} - F_{y_0}$. Using these expressions in eq. (3.18), we obtain

$$\begin{aligned} \Delta F_i =& \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + \frac{1}{2}\Delta x_i \Delta F_{x_i} + \frac{1}{2}\Delta y_i \Delta F_{y_i} \\ &+ O\left(\Delta x_i, \Delta y_i\right)^3, \quad i = 1, \ldots, n \end{aligned} \tag{3.20}$$

We now introduce the modified perturbation in Maxwellians, $\Delta\widetilde{F}_i$ and define it as

$$\begin{aligned} \Delta\widetilde{F}_i =& \Delta F_i - \frac{1}{2}\Delta x_i \Delta F_{x_i} - \frac{1}{2}\Delta y_i \Delta F_{y_i} \\ =& \Delta F_i - \frac{1}{2}\Delta x_i \left(F_{x_i} - F_{x_0}\right) - \frac{1}{2}\Delta y_i \left(F_{y_i} - F_{y_0}\right) \end{aligned} \tag{3.21}$$

Using $\Delta\widetilde{F}_i$, eq. (3.20) reduces to

$$\Delta\widetilde{F}_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + O\left(\Delta x_i, \Delta y_i\right)^3, \quad i = 1, \ldots, n \tag{3.22}$$

Solving the modified over-determined system using least squares, the second-order accurate approximations to F_x and F_y at the point P_0 are given by

$$\begin{bmatrix} F_x^{2} \\ F_y^{2} \end{bmatrix} = \begin{bmatrix} \sum \Delta x_i^2 & \sum \Delta x_i \Delta y_i \\ \sum \Delta x_i \Delta y_i & \sum \Delta y_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum \Delta x_i \Delta\widetilde{F}_i \\ \sum \Delta y_i \Delta\widetilde{F}_i \end{bmatrix} \tag{3.23}$$

Note that the subscript *2* on F_x and F_y denotes second-order accuracy. The above formulae satisfy the test of k-exactness as they yield exact derivatives for polynomials of degree ≤ 2. Furthermore, these formulae have the same structure as the first-order formulae in eq. (3.11), except that the second-order approximations use modified Maxwellians. In contrast to first-order formulae that are explicit in nature, the second-order approximations have implicit dependence as the evaluation of F_x and F_y at the point P_0 requires the values of F_x and F_y at P_0 and its neighbours apriori, so that $\Delta\widetilde{F}_i$ in eq. (3.21) can be estimated.

Taking Ψ - moments of the spatial terms in eq. (3.7) along with the formulae in eq. (3.23), we get the second-order accurate discrete approximations for the kinetic split flux derivatives as

$$
\begin{aligned}
\frac{\partial \boldsymbol{G}^{\pm}}{\partial x} &= \frac{\sum \Delta y_i^2 \sum \Delta x_i \Delta \widetilde{\boldsymbol{G}}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta y_i \Delta \widetilde{\boldsymbol{G}}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \sum \Delta x_i \Delta y_i} \\
\frac{\partial \boldsymbol{H}^{\pm}}{\partial y} &= \frac{\sum \Delta x_i^2 \sum \Delta y_i \Delta \widetilde{\boldsymbol{H}}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta x_i \Delta \widetilde{\boldsymbol{H}}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \sum \Delta x_i \Delta y_i}
\end{aligned}
\tag{3.24}
$$

where, $\Delta \widetilde{\boldsymbol{G}}_i^{\pm}$ and $\Delta \widetilde{\boldsymbol{H}}_i^{\pm}$ are defined by

$$
\begin{aligned}
\Delta \widetilde{\boldsymbol{G}}_i^{\pm} &= \Delta \boldsymbol{G}_i^{\pm} - \frac{1}{2}\left\{ \Delta x_i \frac{\partial}{\partial x} \Delta \boldsymbol{G}_i^{\pm} + \Delta y_i \frac{\partial}{\partial y} \Delta \boldsymbol{G}_i^{\pm} \right\} \\
\Delta \widetilde{\boldsymbol{H}}_i^{\pm} &= \Delta \boldsymbol{H}_i^{\pm} - \frac{1}{2}\left\{ \Delta x_i \frac{\partial}{\partial x} \Delta \boldsymbol{H}_i^{\pm} + \Delta y_i \frac{\partial}{\partial y} \Delta \boldsymbol{H}_i^{\pm} \right\}
\end{aligned}
\tag{3.25}
$$

A drawback of this formulation is that the second-order scheme thus obtained reduces to first-order at the boundaries as the stencils to compute the split flux derivatives may not have enough neighbours. Furthermore, $\Delta \widetilde{F}_i$ is not the difference between two Maxwellians. Instead, it is the difference between two perturbed Maxwellians, given by

$$
\begin{aligned}
\Delta \widetilde{F}_i = \widetilde{F}_i - \widetilde{F}_0 = &\left\{ F_i - \frac{1}{2}\left(\Delta x_i F_{x_i} + \Delta y_i F_{y_i}\right) \right\} \\
&- \left\{ F_0 - \frac{1}{2}\left(\Delta x_i F_{x_0} + \Delta y_i F_{y_0}\right) \right\}
\end{aligned}
\tag{3.26}
$$

Unlike F_i and F_0, the distribution functions $\widetilde{F}_i$ and $\widetilde{F}_0$ may not be non-negative and therefore need not be Maxwellians.

In order to preserve positivity, instead of Maxwellians, we employ the $\boldsymbol{q}$-variables [3, 5] in the defect correction procedure to obtain second-order accuracy. In terms of the primitive variables $(\rho,\ u_1,\ u_2,\ \beta)$, the $\boldsymbol{q}$-variables in two-dimensions are defined by

$$
\boldsymbol{q} = \left[\ln \rho + \frac{\ln \beta}{\gamma - 1} - \beta\left(u_1^2 + u_2^2\right),\ \ 2\beta u_1,\ \ 2\beta u_2,\ \ -2\beta \right]^T \tag{3.27}
$$

Note that the transformations $F \longleftrightarrow \boldsymbol{q}$ and $\boldsymbol{U} \longleftrightarrow \boldsymbol{q}$ are unique and therefore the $\boldsymbol{q}$-variables can be used to represent the fluid flow at the macroscopic level. The second-order LSKUM based on $\boldsymbol{q}$-variables is then obtained by replacing $\Delta \widetilde{\boldsymbol{G}}_i^{\pm}$ and $\Delta \widetilde{\boldsymbol{H}}_i^{\pm}$ in eq. (3.25) with $\Delta \boldsymbol{G}_i^{\pm}(\widetilde{\boldsymbol{q}})$ and $\Delta \boldsymbol{H}_i^{\pm}(\widetilde{\boldsymbol{q}})$ respectively. The new perturbations in split fluxes are defined by

$$
\begin{aligned}
\Delta \boldsymbol{G}_i^{\pm}(\widetilde{\boldsymbol{q}}) &= \boldsymbol{G}^{\pm}(\widetilde{\boldsymbol{q}}_i) - \boldsymbol{G}^{\pm}(\widetilde{\boldsymbol{q}}_0) \\
\Delta \boldsymbol{H}_i^{\pm}(\widetilde{\boldsymbol{q}}) &= \boldsymbol{H}^{\pm}(\widetilde{\boldsymbol{q}}_i) - \boldsymbol{H}^{\pm}(\widetilde{\boldsymbol{q}}_0)
\end{aligned}
\tag{3.28}
$$

Here, $\widetilde{\boldsymbol{q}}_i$ and $\widetilde{\boldsymbol{q}}_0$ are the modified $\boldsymbol{q}$-variables, given by

$$\begin{aligned}\widetilde{\boldsymbol{q}}_i &= \boldsymbol{q}_i - \frac{1}{2}\left(\Delta x_i \boldsymbol{q}_{xi} + \Delta y_i \boldsymbol{q}_{y_i}\right)\\ \widetilde{\boldsymbol{q}}_0 &= \boldsymbol{q}_0 - \frac{1}{2}\left(\Delta x_i \boldsymbol{q}_{x0} + \Delta y_i \boldsymbol{q}_{y_0}\right)\end{aligned} \tag{3.29}$$

The necessary condition for obtaining second-order accurate split flux derivatives is that $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ in eq. (3.29) are evaluated with second-order accuracy. Note that these components are approximated using full stencil in a way similar to that of F_x and F_y in eq. (3.23). The resulting least squares formulae for $\boldsymbol{q}_x^2$ and $\boldsymbol{q}_y^2$ are implicit in nature and need to be solved iteratively. These sub-iterations are called inner iterations. Once evaluated, the $\widetilde{\boldsymbol{q}}$-variables are used to compute $\Delta\boldsymbol{G}^{\pm}\left(\widetilde{\boldsymbol{q}}\right)$ and $\Delta\boldsymbol{H}^{\pm}\left(\widetilde{\boldsymbol{q}}\right)$ at the first step and then the split flux derivatives in *eq.* (3.24).

It is well-known that the second-order schemes use limiters to prevent the generation of spurious oscillations and to preserve monotonicity of the solution in regions with large gradients. In the current scheme, monotonicity at any point P_0 is enforced by introducing a slope limiter $\boldsymbol{\phi}_0$ for $\widetilde{\boldsymbol{q}}_0$ such that the condition

$$\boldsymbol{q}_{\min} \leq \boldsymbol{q}_0 - \frac{\boldsymbol{\phi}_0}{2}\left(\Delta x_i \boldsymbol{q}_{x0} + \Delta y_i \boldsymbol{q}_{y_0}\right) \leq \boldsymbol{q}_{\max} \tag{3.30}$$

holds for all points in the neighbourhood of P_0. Here, $\boldsymbol{q}_{\min}$ and $\boldsymbol{q}_{\max}$ are defined by

$$\begin{aligned}\boldsymbol{q}_{\min} &= \min\left\{\boldsymbol{q}_i, i \in N\left(P_0\right)\right\}\\ \boldsymbol{q}_{\max} &= \max\left\{\boldsymbol{q}_i, i \in N\left(P_0\right)\right\}\end{aligned} \tag{3.31}$$

An advantage of this approach is that higher-order accuracy can be achieved even at boundary points as $\boldsymbol{q}$-variables can be combined with the kinetic wall [10] and outer boundary [12] conditions. Furthermore, the distribution functions $F\left(\widetilde{\boldsymbol{q}}_i\right)$ and $F\left(\widetilde{\boldsymbol{q}}_0\right)$ corresponding to $\widetilde{\boldsymbol{q}}_i$ and $\widetilde{\boldsymbol{q}}_0$ are always Maxwellians and therefore preserves the positivity of numerical solution.

Finally, the state-update formula for steady problems can be constructed by replacing the pseudo-time derivative in *eq.* (3.12) with a suitable discrete approximation and local time stepping. In the present work, the solution is updated using a four-stage Runge-Kutta (SSP-RK3) [9] time marching algorithm.

Chapter 4

GPU Accelerated Meshfree Solver in C++

In this chapter, we elaborate on the specifics of the implementation of the meshfree solver in C++. First, we describe the structure of the serial solver, the objects and functions involved. This is followed by an introduction to the CUDA framework in then next chapter, alongside details of the implementation of the parallel solver.

Aerodynamic shape optimization problems involving the numerical scheme discussed in the previous chapter require thousand of iterations. Hence, time is a very valuable commodity in these applications. Hence, time is a very valuable commodity in these applications. As a result, Fortran and C++ are the natural choices of programming languages for these scientific computing applications. C++ offers efficiency, flexible expression templates, is richly supported and is also amenable for parallelism using CUDA C++. Moreover, C++ has support for automatic differentiation through libraries such as CoDiPack, which is based on the operator overloading method. For these reasons, we decided to develop a C++ version of the $\boldsymbol{q}$-LSKUM solver.

Before we get started with the specifics, it is important to note that:

- Our primary objective is to develop a meshfree solver that is time-efficient. We benchmark it against the Fortran version; Speed is our first criterion.
- We validate the numerical solution's accuracy to the order of **13** digits beyond the decimal. Crucially this means that most of our variables need to be have double precision, and that the residue values should match with the Fortran version upto the 13th decimal place.

4.1 Data Structures

Essentially the solver makes use of two major data structures:

- A *Point Struct* that stores the spatial co-ordinates, the connectivity (neighbours), prim values, flux residual values, the q variable and its derivative values, the delta value and flags to indicate the point type (wall, interior, or boundary).
- And, a global array of the above *Point Structs*, named *globaldata* that is passed to each of the five major functions.

FIGURE 4.1: Definition of the *Point Struct*

```cpp
struct Point
{
        int localID;
        double x, y;
        int left, right;
        int flag_1, flag_2; // Int8 in the Julia code
        double short_distance;
        int nbhs;
        int conn[20];
        double nx, ny;
        // Size 4 (Pressure, vx, vy, density) x numberpts
        double prim[4];
        double flux_res[4];
        double q[4];
        // Size 2(x,y) 4(Pressure, vx, vy, density) numberpts
        double dq1[4];
        double dq2[4];
        double entropy;
        int xpos_nbhs, xneg_nbhs, ypos_nbhs, yneg_nbhs;
        int xpos_conn[20];
        int xneg_conn[20];
        int ypos_conn[20];
        int yneg_conn[20];
        double delta;
        double max_q[4];
        double min_q[4];
        double prim_old[4];

        //Point Constructor

        Point() {}

};
```

4.2 Codeflow, Functions and Sub-functions

The first step involves reading the input grid points generated by a Quadtree based point generation method, from a file onto the *globaldata* array of *Point structs*.

We pass the *globaldata* pointer as an argument into the subsequent functions, and perform in-place mutations. At the heart of the solver, is a fixed-point-iterative method that runs for a number of iterations (typically in the thousands), with each iteration seeking to reduce the value of the residues. The residual value is computed and displayed at the end of the `state_update` function in each iteration. The lower the residue values, the better is the numerical solution obtained.

Each iteration calls the following functions in the listed order: (1) `func_delta`, which computes and sets `delta` for each point in `globaldata`.

```
double delta_t = dist/(mod_u + 3*sqrt(globaldata[conn].prim[3]/globaldata[conn].prim[0]));
delta_t *= cfl;
```

where `conn` parses through the connectivity of the current point, looking at each of its neighbors.

Following this, the functions
(2) `q_variables`,
(3) `q_var_derivatives`,
(4) `q_var_derivatives_innerloop`,
(5) `cal_flux_residual`, and
(6) `state_update`
are called `rks` number of times where `rks` = 4 for a 4-stage Runge-Kutta (SSP-RK3) time marching algorithm.

The subroutine `q_variables` evaluates the $\boldsymbol{q}$-variables outlined in the previous chapter, while `q_derivatives` and `q_derivatives_innerloop` compute the second order accurate approximations of $\boldsymbol{q_x}$ and $\boldsymbol{q_y}$ along with inner iterations. This is followed by the calculation of the flux residuals through the least squares approximation of the split fluxes, mentioned in the $\boldsymbol{q}$-LSKUM algorithm detailed in the previous chapter. This `cal_flux_residual` function is the most time consuming function (takes about **80%** of the wall clock time) of the solver, hence identified as the primary bottleneck. Finally, the `state_update` function updates the flow solution at each Runge-Kutta step. [1]

[1] Link to the accompanying code: Meshfree-cpp-github

Algorithm 1: Serial meshfree solver based on q-LSKUM

```
function q-LSKUM
    call preprocessor()
    for $n \leftarrow 1$ to $n \leq N$ do
        call timestep()
        for $rk \leftarrow 1$ to $rks$ do
            call q_variables()
            call q_var_derivatives()
            call q_var_derivatives_innerloop()
            call cal_flux_residual()
            call state_update(rk)
        end
    end
    call postprocessor()
end function
```

To recap,

• A Quadtree based point refinement algorithm is used for generating the input point cloud distribution. Specifics of the quadtree algorithm is relegated to the appendix. In short, the quadtree data structure represents a hierarchical decomposition of the 2-D space. Each point, and its properties are stored in a quadtree node. Each node has four children corresponding to the four directions: North (N), West(W), South (S) and East (E). The algorithm seeks to refine the point distribution based on the notion of a Mahalanobis distance between the Maxwellians; The following refined tree is balanced, and smoothed, and the process is repeated.
The input format is as follows: (point index, X, Y, left index, right index, flag_1, flag_2, neighbours, connectivity).

• The input point distributions take up increasingly large amounts of memory on the secondary storage as they become finer, and finer. It is an encouraged practice to use the HDF5 file format for efficient file-reads and file-writes [**hdf5**]. Therefore, we have used the HDF5 format to store the Quadtree generated grids.

• Upon completion of input file-read, the first line of business is to compute and set the normals through `placeNormals`. This is followed by connectivity generation in `calculateConnectivity`. Together this ends the input file pre-processing. We are now ready to enter `fpi_solver`.

• `fpi_solver` is run for the specified number of iterations (1000, 5000, 8000, etc.), and is where all the magic happens. First, `func_delta` computes and updates the delta for each point in `globaldata`. [Mention where these delta values are used]. This is followed by the computation of `q_variables`; the derivatives of $\boldsymbol{q}$-variables in `q_var_derivatives`; the inner iterations necessary for second order accuracy in `q_var_derivatives_innerloop`. All of these functions are relatively inexpensive compared to the next.

• The major bottleneck in the $\boldsymbol{q}$-LSKUM solver code is the `cal_flux_residual` function. This function performs spatial discretization, and LS approximation of the fluxes as explained in the previous chapter. The upwind scheme following the kinetic flux vector split, and the state updates are performed in the `state_update` function. First, the perturbed $\boldsymbol{q}$-variables (or) $\widetilde{\boldsymbol{q}}$-variables are mapped to the primitive variables through `qtilde_to_primitive`, and then the upwind scheme is completed with `primitive_to_conserved`. `state_update` performs the Runge-Kutta time stepping.

• We now have a scientific software framework. How do we optimize it from a software perspective? How do we squeeze every last bit of efficiency out of it?

• Observation #1: `cal_flux_residual` is our major bottleneck.

• Observation #2: All the computations happen over a huge number of points (ranging from thousands to millions to billions). And, the computations corresponding to each point in each of the six major functions are *Independent of each other!*

• Observation #3: GPUs are great at performing compute intensive operations but not so good at context switching. And, our application does not require much context switching in its present form. Hence, GPUs are ideal for accelerating our code.

“ Premature optimization is the root of all evil ” - Donald Knuth

Chapter 5

GPU Acceleration with CUDA

In this chapter, we give a brief introduction to the CUDA framework, the prospects of parallelization that it offers and the implementation of the CUDA parallel version of the Meshfree solver. Without getting too much into the architectural and hardware details, we review concepts from the perspective of a programmer looking to speed up computations by making use of the data parallelism that our algorithm lends itself to. For further details the reader may refer [11], which is an excellent reference for CUDA C++.

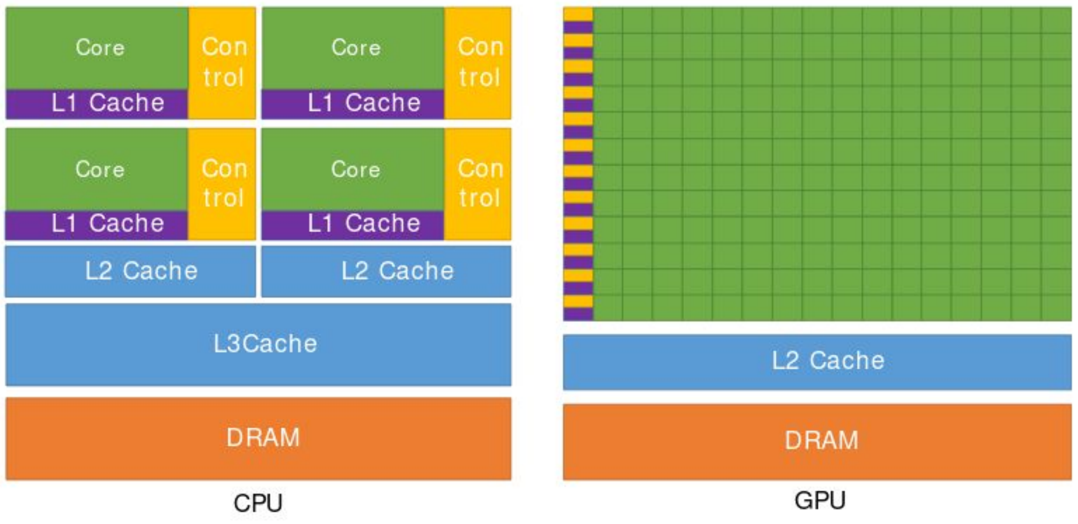
5.1 The CUDA Programming Model

Traditionally, programmers have sought MPI based task parallelism for speeding up their applications on the CPUs. The Scientific Computing literature is rich with such parallel implementations, which have been existent for decades. In recent years, there is a growing trend of programmers moving parallelizing their applications on Graphical Processing Units (GPUs).
The GPUs have brought about a paradigm shift in parallel programming, by offering much higher instruction throughput and memory bandwidth. They aren't nearly as good as the CPU for context-switching and logic involving the use of extensive control flows; Rather they have been architected with a fundamentally different goal in mind - one that enables them to efficiently perform data intensive computations. The CUDA framework is a middlework that abstracts out the hardware details and provides a convenient extension of the high-level language as an interface for the programmer to work with.

Naturally we sought to speed up our meshfree solver by parallelizing it on the GPU. And the result was an immediate and tangible speedup of **15x** on the coarsest grid (40K points), and **23x** on the finest one (2.5 M points).

The following diagram illustrates the distribution of transistors on a GPU chip as compared to that in a GPU chip. Designed for highly parallel operations, the GPU chip has a much higher number of transistors devoted to data processing rather than flow control and data caching.

FIGURE 5.1: Distribution of transistors on the NVIDIA GPU vs. the CPU [11]



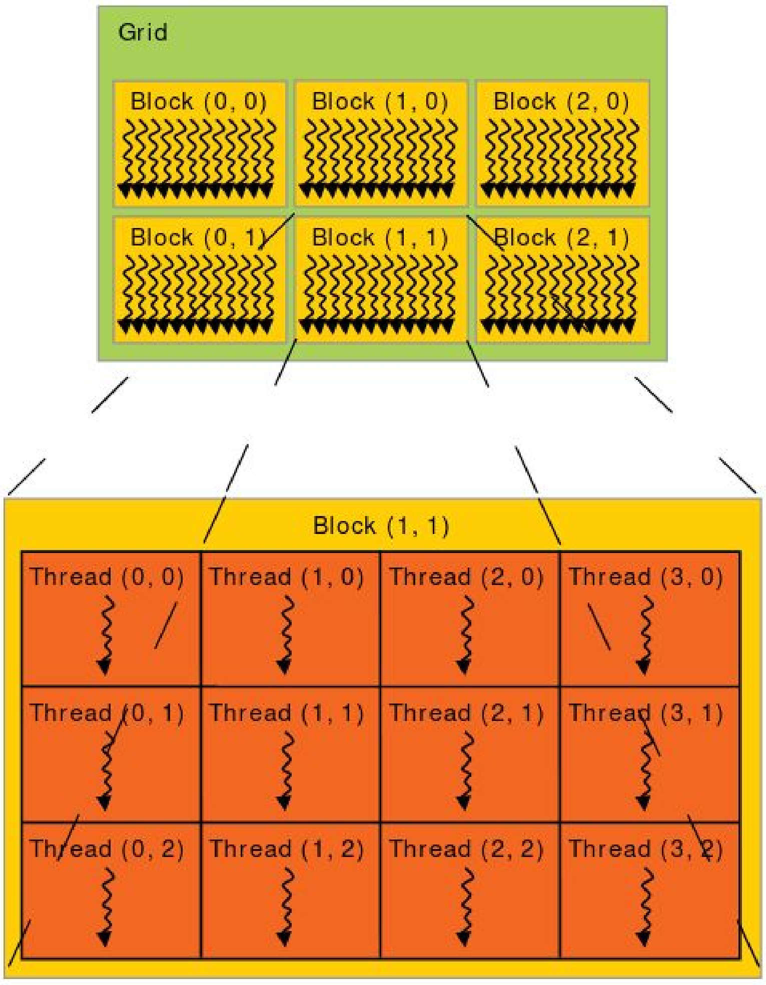
The CUDA programming model is quite simple, and convenient. At the core are three main abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization. [11] The execution model follows SIMT (Single Instruction Multiple Threads), much similar to SIMD (Single Instruction Mutiple Data). Essentially, multiple threads execute the same instruction on the streaming processor. Threads are grouped together into blocks; and blocks are grouped together into grids. Threads, Blocks and Grids are uniquely identified by their respective IDs, exposed to the user through an API call.

Each GPU device contains a number of CUDA cores or stream processors. These are packed within a Streaming Multiprocessor (SM). A set of threads, called a warp is scheduled on the SM by a warp scheduler. SMs can concurrently execute one or more warps. It should be noted that although the thread is the lowest unit of the hierarchy, the execution model executes one warp at a time (which entails concurrent execution of a set of threads, 32 to be precise). This is done as per the SIMT model, and to maximize SM utilization one should try to minimize thread divergence within a warp. Divergence occurs when different threads try to execute different statements (for instance, one thread may have evaluated the boolean expression in a conditional branch to false whereas the expression might have evaluated to true in another thread). Unlike a CPU, no branch prediction occurs in a GPU. Rather, threads execute NOP (No-operation) whenever a thread divergence occurs leading to lower SM utilization. It should be noted that computational resources for all the threads of a block are to be allocated on the same stream processor. Therefore, there are physical constraints on the number of threads assigned to a block.

In current NVIDIA GPUs, the limit is 1024 threads per block.

Interthread communication is possible through the use of shared memory. This memory is shared among all the threads in a block. The NVIDIA GPU has the following types of memory blocks: *Global memory* that is accessible globally within a device, but which has the highest latency; *Constant memory*, again globally accessible and latency same as that of global memory; *Local memory* available per thread level; *Thread-level registers* are the least in size and provide lowest latency; Finally, *shared memory* is accessible at the block level. The latency, in increasing order is as follows: *Shared* < *Local* < *Global*. Lesser the latency, greater is the throughput. One should, therefore, try to maximize the use of shared memory.

FIGURE 5.2: Memory Hierarchy [11]

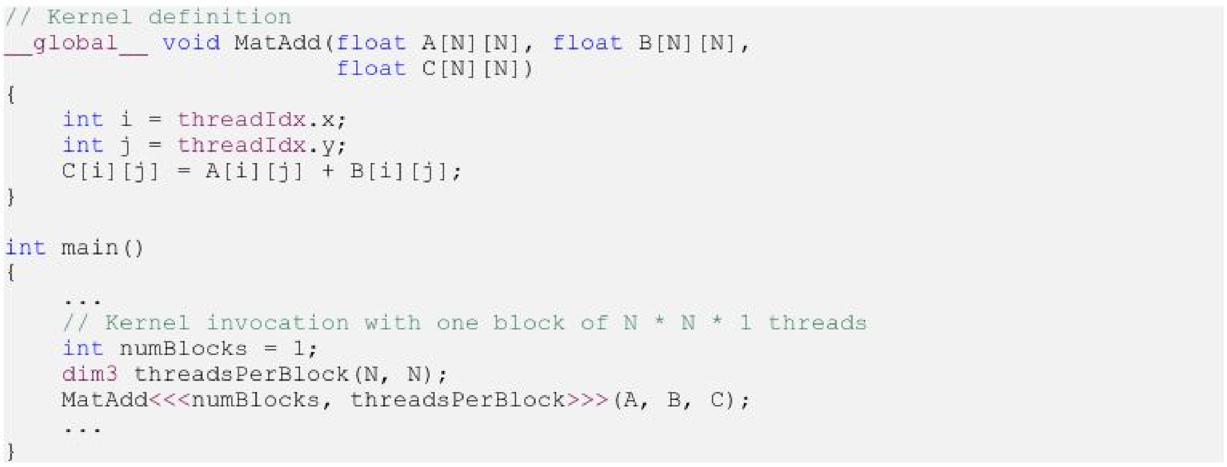


Recently, *Unified Memory* was made available - which allows the memory locations to be accessed by both the device and the host. Prior to this, device functions or kernels couldn't access host or CPU memory, and vice versa. Host memory had to be explicitly offloaded to the device, and copied back after performing the necessary computations. These *MemCpy* calls are synchronous in nature, by default and are often expensive. Therefore, care must be taken to keep such memory transfers to a minimum lest the communication overhead dominate the speedup gained in computation.

SideNote: As with any parallel programming paradigm, conscious effort must be taken to ensure that race conditions don't occur, that the threads are synchronized appropriately, and that atomic operations suitably to prevent such thread race conditions. The issue with these logical errors is that they don't fail **spectacularly**; they fail silently and dormant errors are hard to debug.

Functions not preceded by any specifier are treated de-facto as `__host__` functions by the `nvcc` compiler, and can be executed only on the CPU. Using the `__global__` specifier before a function allows for execution by both the CPU and the GPU; And, the `__device__` informs the compiler that the function is to be called only on the GPU. These `__device__` or `__global__` functions, in a CUDA Aware environment, when called with a number of threads, require the invocation to specify block and grid dimension in the form of a triple chevron syntax. The following diagram illustrates a basic parallel Matrix Addition example:

FIGURE 5.3: Example Matrix Addition Kernel [11]



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

5.2 Parallel Meshfree Solver

5.2.1 The design of the parallel solver in a nutshell

The ***q***-LSKUM meshfree solver perfectly presents itself for data parallelism. We break down all the loops that run from 1 $\rightarrow$ N; Individual threads perform the computations associated with each of the N grid points. The mapping is as follows: `idx = threadIdx.x + blockSize` $\times$

`blockIdx.x`. We make optimal use of *Shared Memory* and *Block Reductions.* The optimizations performed and the improved performance of the solver with the addition of each of these optimizations is detailed in the next chapter.

5.2.2 Implementation Specifics of the Parallel Solver

Once we understand the CUDA programming model, it becomes easy to conceive of the GPU parallel ***q***-LSKUM solver. The first design decision is regarding whether we want to parallelize the entire solver, or just the bottleneck function `cal_flux_residual`. We decide to parallelize the entire solver keeping in mind that the speedup gained due to parallelism will be evident for other functions as well as we move on to much finer grids. Moreover, as a natural extension of the current work for solving aerodynamic shape optimization problems, we shall be implementing the adjoint meshfree solver as well. This would be implemented with the help of the operator overloading based automatic differentiation package, CoDiPack. CoDiPack maintains a register tape of the operations and variable values from the forward pass, which it uses to perform the backward pass in a constant factor times the time taken to perform the primal calculations. Support for this taping process has not been extended to automatically traverse both the CPU and the GPU within a single program evaluation. Hence, it makes sense to port over the entire solver to the GPU.

We parallelize each of the functions detailed in the previous section, starting from `func_delta` right upto `state_update` converting them to GPU kernels. `cudaMemcpy` and `cudaDevice-synchronize` calls are heavy, and impose significant communication overhead. Hence, the major storage data structure `globaldata` is pushed to the GPU before the iterations of the `fpi_solver` begin, and copied back to the CPU at the very end.

Now for the modifications inside each function: The first change, to port over the solver to CUDA C++ is to break up the `for` loops in each of the kernels and assign a thread for the corresponding computations. Since `globaldata` is a single dimensional array of structs, and since most of the array structures we use are 1-D, we use the following mapping between array index and thread index: `idx = threadIdx.x + blockSize` × `blockIdx.x`. A `if` conditional checks and ensure no array access out-of-bounds is made.

Since the solver calls the kernels in succession, and since CUDA launches the kernels within a single stream one after another, there is no need for additional syncing calls in-between the kernels. We, however, need a `cudaDeviceSynchronize` call at the end of each state update, before the computation of the `sig_res_sqr` values (sum of squared residues). This standalone summation is performed following the `state_update` as shown:

```
double sig_res_sqr = thrust::reduce(thrust::cuda::par.on(stream), res_sqr_d,
res_sqr_d + numPoints, (double) 0.0, thrust::plus<double>());
```

While implementing the parallel solver, we make sure that thread divergence is reduced by minimizing the use of conditional branches. We make use of two CUDA concepts: *Shared Memory* and *Block Reductions* as subsequent optimizations. As explained in the previous section, shared memory offers fast memory accesses due to lower latency. We therefore, store `G_xp`, `G_xn`, `G_yp`, `G_yn`, `result`, and other variables in `cal_flux_residual_cuda` in the shared memory. In practice, shared memory did offer a tangible improvement in performance, as expected. Block reductions are used to optimize for block commutative operations over the GPU. It should be noted that block reductions, while time-efficient ($\mathcal{O}(logN)$ time for addition of a 'N' element array) are not cost-efficient($\mathcal{O}(NlogN)$ cost). We go ahead and implement block reductions because time-efficiency is our concern, and we do not optimize for cost efficiency.

In the GPU version, we strip the `globaldata` array of structs into individual, separate arrays of 'N' elements each. This is done to ensure better memory alignment, and avoid dead memory due to padding. This leads to an increase in performance through better memory accesses.

Following this, we implemented CUDA Graphs to batch up the kernel calls. Normally the kernels are visible to the `nvcc` compiler only when it comes into scope, and the compiler is not cognizant of all the kernel invocations to be made up front. Batching up the kernel invocations, and providing it as a graph allows CUDA to perform smart compiler optimizations. However, for our use-case implementing CUDA graphs yielded no significant improvement in performance.

In the next chapter, we detail the extensive experiments and benchmarks against Fortran. We see that the C++ solver performs comparably well with Fortran.

Chapter 6

Results and Discussion

In this chapter, we present a comparative analysis with Fortran, and show that the C++ solver performs comparably well. Hence, the ***q***-LSKUM meshfree solver has been successfully ported over to C++ from Fortran, with scope for future optimizations. We use the RDP as a cost metric, to compare the performance of the solvers. The lower the RDP value, the better is the performance.

Rate of Data Processing (RDP): *Defined as the time taken, in seconds by the program to perform the computations divided by (the number of iterations $\times$ the number of grid points) i.e., time taken for the execution of the solver per grid point per iteration.*

All experiments were performed on a linux workstation and NVIDIA M5000 GPU with configuration details as given below:

	CPU	GPU
Model	Intel Xeon E5-2698 v4	Nvidia Quadro M5000
Cores	40 (20 $\times$ 2)	2048
Core Frequency	2.20 GHz	1.038 GHz
Global Memory	128 GB	8 GB
L2 Cache	5 MB	2 MB

	C++	Fortran
Compiler	GCC	PGI
Compiler Version	9.3.0	18.10
Compiler Flags	O3	O3 -Mcuda = rdc
NVIDIA Version	450.51.06	450.51.06
CUDA Version	11.0	11.0

6.1 Performance Analysis of the Meshfree Solvers: C++ and Fortran

6.1.1 Serial version

Performance of the C++ and Fortran **serial** codes on different grids (The 40K grid is the coarsest). RDP values and wall-clock execution times are reported for a single core CPU computation.

Number of Points	Number of Iterations	C++ (secs)	Fortran (secs)	C++ RDP	Fortran RDP
48738	1000	1297.38	1453.96	2.6619E-05	2.9832E-05
192632	1000	5239.76	4938.51	2.7201E-05	2.5637E-05
345140	1000	9235.87	7809.73	2.6760E-05	2.2628E-05
804824	1000	21047.58	15325.27	2.6152E-05	1.9042E-05
2642264	100	6874.75	4650.15	2.6018E-05	1.7599E-05

Table 6.1: RDP values and wall-clock times for single core CPU computation.

6.1.2 GPU version

We perform several ablation experiments to show the effect of each GPU optimization. First we implement the GPU parallel version of the C++ solver sans block reductions and shared memory. Then we show the performance improvement due to the usage of block reductions and shared memory. Finally we optimize for padding and alignment on the GPU by splitting the `globaldata` array of structs into individual arrays thereby reducing the latency for memory accesses by effectively reducing the amount of dead memory. An important hyperparameter in these experiments is the **number of threads per block** (`threads_per_block`). We set `threads_per_block` to 32, 64, 128, and 256 and choose the optimal value. For the C++ version the optimal `threads_per_block` is found to be **32**, whereas for Fortran, it is seen to be **64**. The values reported below are for the optimal number of threads per block.

Note that throughout these comparisons, we make use of the most optimal configuration for Fortran with regards to `threads_per_block`, block reductions, shared memory and padding/alignment optimizations. The ablation experiments are performed on the C++ version alone. We find that the C++ version making use of shared memory v1, block reductions, with the stripped point struct, and `threads_per_block` set to **32** shows the best performance. In shared memory v1, the arrays `Gxp`, `Gxn`, `Gyp`, `Gyn`, `result`, `sig_del_x_del_f`, `sig_del_y_del_f`, `qtilde_i`, `qtilde_k` are allocated in the shared memory space; In shared memory v2, we further allocate `phi_i`, `phi_k`, `G_i`, and `G_k` on the shared memory space. The former version is seen to be faster than the latter version, which is an interesting result. **All comparions are made against the most optimal configuration of the Fortran GPU code.**

Number of Points	Number of Iterations	C++ (secs)	Fortran (secs)	C++ RDP	Fortran RDP
48738	1000	106.38	76.55	2.1827E-06	1.5706E-06
192632	1000	386.66	260.15	2.0073E-06	1.3505E-06
345140	1000	640.17	437.88	1.8548E-06	1.2687E-06
804824	1000	1473.44	934.16	1.8308E-06	1.1607E-06
2642264	1000	4815.58	2900.62	1.8225E-06	1.0978E-06

Table 6.2: C++ version **without block reductions and shared memory**.

Number of Points	Number of Iterations	C++ (secs)	Fortran (secs)	C++ RDP	Fortran RDP
48738	1000	80.82	76.55	1.6584E-06	1.5706E-06
192632	1000	282.86	260.15	1.4684E-06	1.3505E-06
345140	1000	480.13	437.88	1.3911E-06	1.2687E-06
804824	1000	1101.72	934.16	1.3689E-06	1.1607E-06
2642264	1000	3604.23	2900.62	1.3641E-06	1.0978E-06

Table 6.3: C++ version **with block reductions and shared memory, v1**.

Number of Points	Number of Iterations	C++ (secs)	Fortran (secs)	C++ RDP	Fortran RDP
48738	1000	88.54	76.55	1.8167E-06	1.5706E-06
192632	1000	305.70	260.15	1.5870E-06	1.3505E-06
345140	1000	504.95	437.88	1.4630E-06	1.2687E-06
804824	1000	1121.21	934.16	1.3931E-06	1.1607E-06
2642264	1000	3628.38	2900.62	1.3732E-06	1.0978E-06

Table 6.4: C++ version **with block reductions and shared memory, v2**.

It is to be noted that we implemented CUDA Graphs for the C++ version, to batch up the kernel invocations but that yielded no consistently noticeable improvement. Clearly the C++ version demonstrates comparable performance to the Fortran version as seen from both the wall-clock times as well as the RDP values. For the coarser grids, in fact, the performance of the C++ version is seen to be better than the Fortran version. However, the RDP values of Fortran for the 800K grid as well as the 2.5M grid are marginally higher than the C++ version. We

Number of Points	Number of Iterations	C++ (secs)	Fortran (secs)	C++ RDP	Fortran RDP
48738	1000	**74.41**	76.55	**1.5268E-06**	1.5706E-06
192632	1000	**257.09**	260.15	**1.3346E-06**	1.3505E-06
345140	1000	**431.09**	437.88	**1.2490E-06**	1.2687E-06
804824	1000	945.10	934.16	1.1743E-06	1.1607E-06
2642264	1000	2993.56	2900.62	1.1330E-06	1.0978E-06

Table 6.5: C++ version with **stripped point struct, block reductions and shared memory, v1**.

hypothesize that this is due to the difference in the type and kind of optimizations performed by the GCC compiler as opposed to those performed by the PGI compiler. The grid sizes in typical CFD applications are expected to be quite large; hence we intend to further optimize the C++ implementation by vectorizing the operations. It can be seen that the GPU parallelization of the C++ code offers a speedup of upto **23x**.

Furthermore, it is observed that as the point distributions become finer, the relative speedup achieved by the C++ solver is higher as compared to that of the Fortran solver. However, one should note that the RDP value of the GPU accelerated Fortran solver for these point distibutions is lesser than that of the C++ solver indicating that the Fortran solver is more efficient. The higher relative speedup achieved, therefore, is a result of the performance of the corresponding serial counterparts. In other words, the serial version of the C++ solver, on finer point distributions, is seen to have a higher RDP value than that of the Fortran solver, and therefore the speedup achieved by the GPU parallel C++ solver is seen to be higher.

6.1.3 Kernel Performance Analysis Tables

• Performance metrics of the `flux_residual` kernel for optimal number of `threads_per_block`.

Code	SM Utilisation	Memory Utilisation	Achieved Occupancy	Theoretical Occupancy
			# of points = 48738	
C++	70.97%	42.80%	15.03%	15.62%
Fortran	72.12%	19.17%	10.57%	12.5%
			# of points = 192632	
C++	96.90%	69.40%	48.83%	50%
Fortran	86.49%	21.49%	11.77%	12.5%
			# of points = 345140	
C++	98.08%	71.24%	49.33%	50%
Fortran	90.49%	25.37%	12.28%	12.5%
			# of points = 804824	
C++	98.85%	73.38%	49.60%	50%
Fortran	72.12%	19.17%	10.57%	12.5%
			# of points = 2642264	
C++	99.67%	75.17%	49.81%	50%
Fortran	72.12%	19.17%	10.57%	12.5%

• Performance metrics of the `flux_residual` kernel on the finest point distribution, with varying number of `threads_per_block`.

Code	SM Utilisation	Memory Utilisation	Achieved Occupancy	Theoretical Occupancy
			# of `threads_per_block`= 32	
C++	96.90%	69.40%	48.83%	50%
Fortran	86.49%	21.49%	11.77%	12.5%
			# of `threads_per_block` = 64	
C++	98.08%	71.24%	49.33%	50%
Fortran	90.49%	25.37%	12.28%	12.5%
			# of `threads_per_block` = 128	
C++	98.85%	73.38%	49.60%	50%
Fortran	72.12%	19.17%	10.57%	12.5%

• Relative run-time of the kernels across the point distributions, for the optimal number of `threads_per_block` (**32** for C++, and **64** for Fortran).

Code	q_variables	q_derivatives	q_der_innerloop	flux_res
			# of points = 48738	
C++	70.97%	42.80%	15.03%	15.62%
Fortran	72.12%	19.17%	10.57%	12.5%
			# of points = 192632	
C++	96.90%	69.40%	48.83%	50%
Fortran	86.49%	21.49%	11.77%	12.5%
			# of points = 345140	
C++	98.08%	71.24%	49.33%	50%
Fortran	90.49%	25.37%	12.28%	12.5%
			# of points = 804824	
C++	98.85%	73.38%	49.60%	50%
Fortran	72.12%	19.17%	10.57%	12.5%
			# of points = 2642264	
C++	99.67%	75.17%	49.81%	50%
Fortran	72.12%	19.17%	10.57%	12.5%

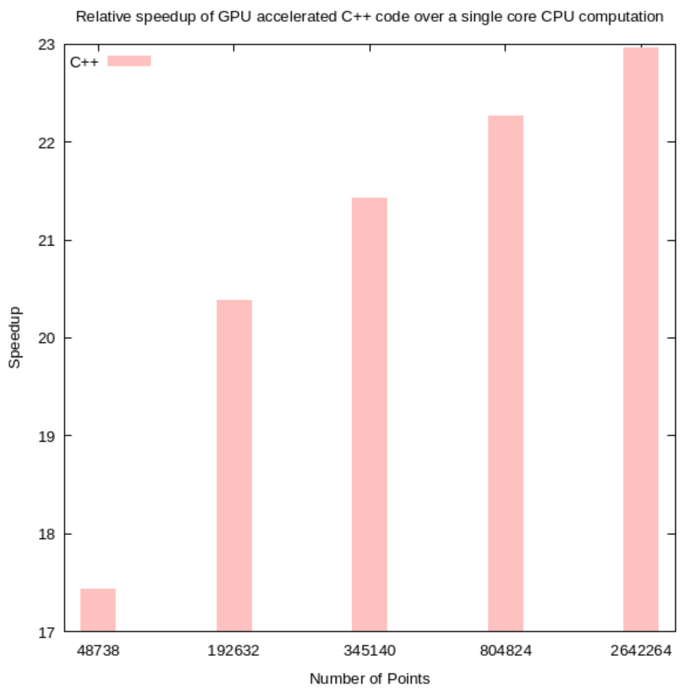


FIGURE 6.1: Relative speedup of the C++ code.

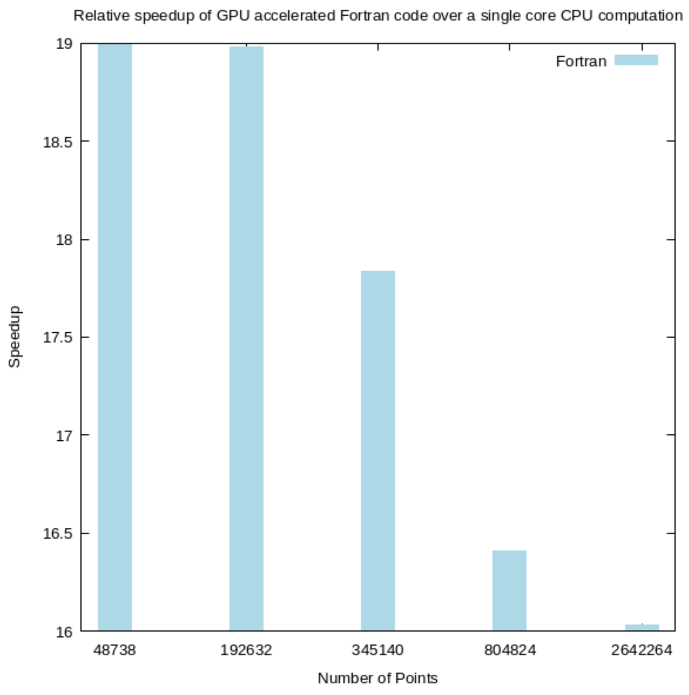


FIGURE 6.2: Relative speedup of the Fortran code.

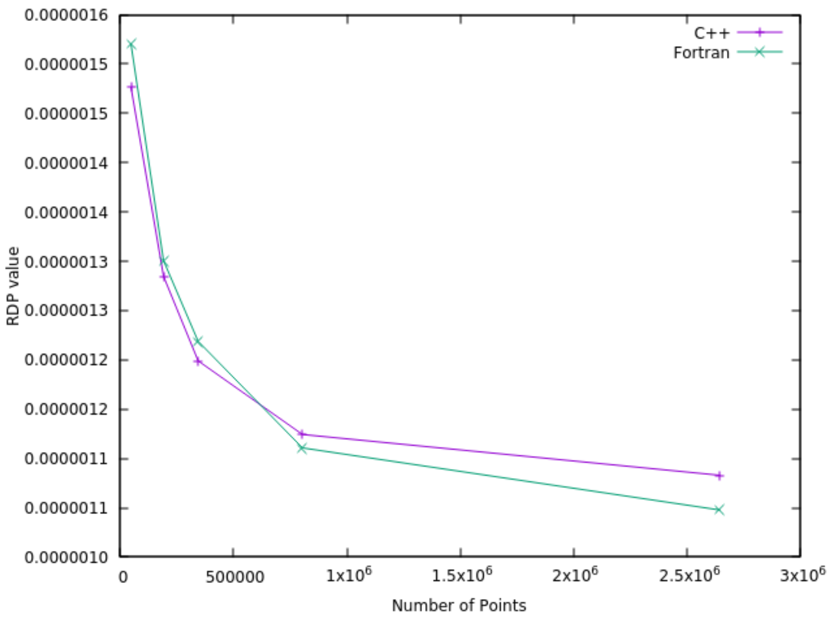


FIGURE 6.3: RDP Comparison across grids, with optimal number of threads per block.

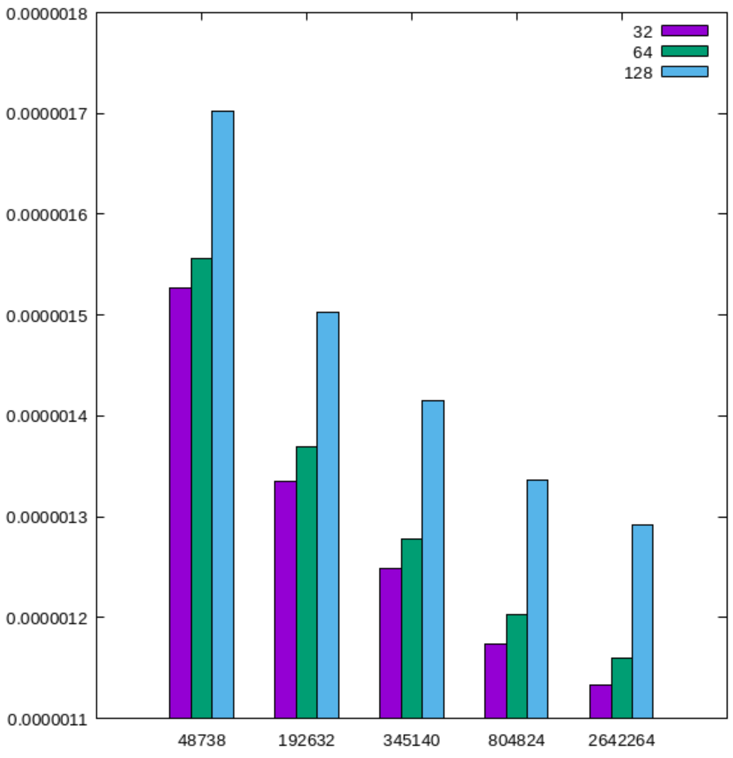


FIGURE 6.4: C++ RDP variation with threads per block.

Chapter 7

Concluding Remarks and Future Work

This thesis develops and implements a Least Squares Kinetic Upwind Method (***q***-LSKUM) based meshfree solver in C++, for modelling aerodynamic flows around complex configurations. Both the serial as well as the GPU parallel versions of the solver are implemented. Numerical results on five levels of point distribution are shown, and it is seen that the GPU codes achieved impressive speedups as compared to their corresponding serial codes. We present numerous kernel metrics, and show that a speedup of upto **23x** is achieved through GPU parallelization of the solver. The solver is extensively benchmarked against the Fortran version, and performs comparably well demonstrating the efficacy of the solver. On coarser distributions, the GPU accelerated C++ solver is seen to have a lower RDP value than the equivalent Fortran solver. However, on finer distributions the Fortran solver is seen to have lower RDP values. While the ***q***-LSKUM algorithm implemented has been proven to effectively handle large unstructured grids, and has seen increasing adoption notably in the Indian Aerospace programs, certain algorithmic enhancements remain open such as improving the condition number of the least squares matrices, and optimal selection of weights for better spectral resolution - all valuable directions for future research.

Further investigations are underway to increase the computational efficiency of the GPU solver based on C++. In future, we want to extend the code to three dimensional flows, and accelerate it on multiple GPUs.

Appendix A

Input File Format; Pseudocode; Terms and Definitions

A.1 Input Point Distribution

A.1.1 HDF5

The input grids are stored as HDF5 files to facilitate efficient reads and writes. This is essential especially as the grids become finer: HDF5 offers reduced file operation times for these large input files. The HDF5 is a hierarchical data file format that support large and complex heterogenous data. It makes use of a directory-like structure to store the data. The data is hierarchically partitioned into datasets, groups and attributes. Table objects are indexed using B-trees.

In our implementation we make use of h5cpp, an open source library that abstracts away many of the HDF5 function calls in a parsimonious manner, and exposes the functionality in a programmer friendly manner.

A.1.2 Quadtree point refinement algorithm

The input grids have been refined, and generated using a Quadtree based point refinement algorithm. A quadtree is a data structure that can be used to efficiently store, and operate on spatial data. It performs a hierarchical decomposition of the 2-D input space. Each internal node of the quadtree contains four children: North (N), South (S), East (E) and West (W). A quadtree is said to be balanced if the difference in depth or level between any pair of neighbouring nodes is atmost one. The level of a node is computed as the level of the parent plus one. The root node is taken to be at level 0. All the information necessary for the $\boldsymbol{q}$-LSKUM computations is stored in the leaf nodes.

The quadtree is first built through a recursive decomposition of the initial input grid space, and stores the initial input point distribution. This is further refined, and the refined point distribution is taken as the input for the ***q***-LSKUM solver. The refinement process consists of three steps: First, we identify certain quadrants or the corresponding points for refinement. These quadrants are split into four disjoint sub-quadrants. The centroids of these sub-quadrants represent the spatial co-ordinates of the newly spawned children. Second, the resultant quadtree is balanced. And third, the refined quadtree is smoothed.

To identify the points for refinement, we use employ a metric based on the Mahalanobis Distance between the Maxwellian distributions of the velocities for each pair of points. In practise, we impose a limit on the minimum size of the quadrant. Post refinement, points inside the wall boundary are culled. It should be noted that the meshfree solver is capable of operating on unstructured point distributions coming from any such generation algorithm be it quadtree based refinement or the advancing front method to name a couple.

A.2 Algorithm Pseudocode

Algorithm 2: q-LSKUM Meshfree Solver

```
function q-LSKUM
    call readQuadtreeInput()
    call calculateConnectivity()
    for $n \leftarrow 1$ to $n \leq N$ do
        call func_delta()
        for $rk \leftarrow 1$ to $rks$ do
            call q_variables()
            call q_var_derivatives()
            call q_var_derivatives_innerloop()
            call calc_flux_residual()
            call state_update()
        end
    end
end function
```

Algorithm 3: q_var_derivatives_innerloop

```
function q_var_derivatives_innerloop
    for $n \leftarrow 1$ to $n \leq N$ do
        Compute $\Sigma\Delta x^2$
        Compute $\Sigma\Delta y^2$
        Compute $\Sigma\Delta x\Delta y$
        for $i \leftarrow 1$ to $num(nbhs)$ do
            call q_var_derivatives_get_sum_delq_innerloop()
            call update_temp_dq()
        end
    end
    for $n \leftarrow 1$ to $n \leq N$ do
        call q_var_derivatives_update_innerloop()
    end
end function
```

Algorithm 4: calc_flux_residual

```
function calc_flux_residual
    Initialize Gxp[4], Gyp[4], Gxn[4], Gyn[4] $\leftarrow 0$
    for $n \leftarrow 1$ to $n \leq N$ do
        call wallindices_flux_residual()
        call outerindices_flux_residual()
        call interiorindices_flux_residual()
    end
end function
```

Algorithm 5: wallindices_flux_residual

```
function wallindices_flux_residual
    call wall_dGx_pos()
    call wall_dGx_neg()
    call wall_dGy_neg()
    Update Globaldata.flux_res
end function
```

Algorithm 6: outerindices_flux_residual

```
function outerindices_flux_residual
    call outer_dGx_pos()
    call outer_dGx_neg()
    call outer_dGy_pos()
    Update Globaldata.flux_res
end function
```

Algorithm 7: innerindices_flux_residual

```
function innerindices_flux_residual
    call inner_dGx_pos()
    call inner_dGx_neg()
    call inner_dGy_pos()
    call inner_dGy_neg()
    Update Globaldata.flux_res
end function
```

Algorithm 8: wall_dGx_pos

```
function wall_dGx_pos
    Initialize phi, G, qtilde, $\Sigma\Delta x\Delta f$, $\Sigma\Delta y\Delta f$ $\leftarrow 0$
    for $n \leftarrow 1$ to $n \leq num(nbhs)$ do
        call calculate_qtilde()
        call qtilde_to_primitive()
        call flux_quad_GxII()
        call qtilde_to_primitive()
        call flux_quad_GxII()
        call update_delf()
    end
    Compute det = $\Sigma\Delta x^2\Sigma\Delta y^2 - (\Sigma\Delta x\Delta y)^2$
    Update Gxp
end function
```

Algorithm 9: calculate_qtilde

```
function calculate_qtilde
    call venkat_limiter()
    call update_q_tildes()
end function
```

Algorithm 10: qtilde_to_primitive

```
function qtilde_to_primitive
    $\beta$ = -qtilde[3]*0.5
    u1 = qtilde[1]*temp; u2 = qtilde[2]*temp;
    temp1 = qtilde[0] + $\beta$*(u1*u1 + u2*u2);
    temp2 = temp1 - (log($\beta$)/($\gamma$-1));
    $\rho$ = exp(temp2);
    pr = $\rho$*temp;
    result[0] = u1;
    result[1] = u2;
    result[2] = $\rho$;
    result[3] = pr;
end function
```

The pseudocode for `wall_dGx_neg`, `wall_dGy_neg` and the corresponding subroutines for `outerindices`, and `innerindices` follow a similar structure. We refer the reader to the implementation at Hari-Meshfree-CPP-Github for further implementational details, and for the `split_flux`, `wall_flux`, and `quad_flux` routines.

Algorithm 11: state_update

```
function state_update
    Initialize U[4], Uold[4] $\leftarrow 0$
    for $n \leftarrow 1$ to $n \leq N$ do
        call state_update_wall()
        call state_update_outer()
        call state_update_inner()
    end
end function
```

Algorithm 12: state_update_wall

```
function state_update_wall
    call primitive_to_conserved()
    Compute res_sqr
    Update Globaldata.prim
end function
```

Algorithm 13: state_update_outer

```
function state_update_outer
    call conserved_vector_Ubar()
    Compute res_sqr
    Update Globaldata.prim
end function
```

Algorithm 14: state_update_interior

```
function state_update_interior
    call primitive_to_conserved()
    Compute res_sqr
    Update Globaldata.prim
end function
```

A.3 Terms and Definitions

- **RDP**: Rate of Data Processing, RDP is defined as the time taken by the program to perform the computations per grid point per iteration.
- **SM utilization**: SM utilization is the percentage of the time for which one or more kernels were executing on the GPU streaming multiprocessors over the past sample.
- **Memory utilisation**: Memory utilisation is defined as the percentage of time for which global (device) memory was accessed for read/write operations over the past sample.
- **Warp**: A warp is a group of 32 CUDA threads running in a lock-step manner.
- **Achieved Occupancy**: The ratio of active warps on a streaming multiprocessor to the maximum number of active warps that can be theoretically supported by the SM.
- **Branch Efficiency**: The ratio of executed uniform flow control decisions over all the executed conditionals.

- **Arithmetic Intensity**: Arithmetic Intensity is defined as the ratio of work done by the streaming multiprocessor to total data movement.
- **Maxwellian Distribution**: The Maxwell-Boltzmann distribution is a chi distribution used to characterize the velocities of particles in idealized gases assuming free flow interspersed with very brief collisions.

Bibliography

[1] Konark Arora, Rajan N K S, and Suresh Deshpande. "ON THE ROBUSTNESS AND ACCURACY OF LEAST SQUARES KINETIC UPWIND METHOD (LSKUM)". In: Aug. 2008.

[2] R. Courant, E. Issacson, and M. Rees. "On the solution of Nonlinear Hyperbolic Differential Equations by Finite Differences". In: *Comm. Pure Appl. Math.* 5 (1952), pp. 243–255.

[3] S.M. Deshpande. "On the Maxwellian distribution, symmetric form, and entropy conservation for the Euler equations". In: *NASA-TP-2583* (1986).

[4] S.M. Deshpande, P.S. Kulkarni, and A.K. Ghosh. "New developments in kinetic schemes". In: *Computers Math. Applic.* 35.1 (1998), pp. 75–93.

[5] S.M. Deshpande et al. "Theory and application of 3-D LSKUM based on entropy variables". In: *Int. J. Numer. Meth. Fluids* 40 (2002), pp. 47–62.

[6] Suresh Deshpande et al. "Least squares Kinetic Upwind Mesh-free Method". In: *Defence Science Journal* 60.6 (2010), pp. 583–597. DOI: `10.14429/dsj.60.579`. URL: `https://publications.drdo.gov.in/ojs/index.php/dsj/article/view/579`.

[7] A.K. Ghosh and S.M. Deshpande. "Least squares kinetic upwind method for inviscid compressible flows". In: *AIAA paper 1995-1735* (1995).

[8] Aaron Katz and Antony Jameson. *A Comparison of Various Meshless Schemes Within a Unified Algorithm.* 2009.

[9] J. F. B. M. Kraaijevanger. "Contractivity of Runge-Kutta methods". In: *BIT Numerical Mathematics* 31.3 (1991), pp. 482–528.

[10] J.C. Mandal and S.M. Deshpande. "Kinetic flux vector splitting for Euler equations". In: *Comp. & Fluids* 23.2 (1994), pp. 447–478.

[11] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide.* Version 3.2. 2010.

[12] V. Ramesh and S.M. Deshpande. "Least squares kinetic upwind method on moving grids for unsteady Euler computations". In: *Comp. & Fluids* 30.5 (2001), pp. 621–641.