Development of a Chemically Reacting Flow Solver on the Graphic Processing Units

A project present to
The Faculty of the Department of Aerospace Engineering
San Jose State University

in partial fulfillment of the requirements for the degree
Master of Science in Aerospace Engineering

By

Hai Phuoc Le

May 2011

approved by

Dr. Periklis Papadopoulos
Faculty Advisor

The Designated Thesis Committee Approves the Thesis Titled

DEVELOPMENT OF A CHEMICALLY REACTING FLOW SOLVER ON THE GRAPHIC PROCESSING UNITS

by

Hai Phuoc Le

APPROVED FOR THE DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2011

Prof. Periklis Papadopoulos	Department of Mechanical and Aerospace Engineering
Prof. Nikos Mourtos	Department of Mechanical and Aerospace Engineering
Dr. Jean-Luc Cambier	Air Force Research Laboratory, Edwards AFB

ABSTRACT

DEVELOPMENT OF A CHEMICALLY REACTING FLOW SOLVER ON THE GRAPHIC PROCESSING UNITS

by

Hai Phuoc Le

The focus of the current research is to develop a numerical framework on the Graphic Processing Units (GPU) capable of modeling chemically reacting flow. The framework incorporates a high-order finite volume method coupled with an implicit solver for the chemical kinetics. Both the fluid solver and the kinetics solver are designed to take advantage of the GPU architecture to achieve high performance. The structure of the numerical framework is shown, detailing different aspects of the optimization implemented on the solver. The mathematical formulation of the core algorithms is presented along with a series of standard test cases, including both non-reactive and reactive flows, in order to validate the capability of the numerical solver. The performance results obtained with the current framework show the parallelization efficiency of the solver and emphasize the capability of the GPU in performing scientific calculations.

ACKNOWLEDGEMENTS

I would like to acknowledge many individuals who have helped me in completing this thesis. This work would have not been accomplished without their support.

First of all, I would like to thank Prof. Periklis Papadopoulos for all his support throughout the course of this work. I am so much indebted to him for all his continued encouragement and advice.

I also want to acknowledge Dr. Jean-Luc Cambier for his direction and guidance in completing this work. He has given me countless support during my time at the AFRL and allowed me to work on the topic of interest. I would also like to thank all the people at the EP Lab who have contributed to this research. I am grateful to Mr. Lord Cole and Mr. David Bilyeu for numerous discussions about CFD. I would like to thank Dr. Justin Koo for all his help during my time at the lab. I would also like to thank Dr. Vladimir Titarev for making the one-dimensional formulation of the ADERWENO scheme available.

I also want to thank Prof. Nikos Mourtos for all his help in the thesis class and serving on this thesis committee. I thank him for being patient, understanding, and helping me through my last year at SJSU.

Foremost, my accomplishment today would not have been successful without my family and friends.

D E DI C A T I O N

Dedicated with love to my parents!

TABLE OF CONTENTS

L IST O F F I G UR ES

LIST OF TABLES

L IST O F SY M B O LS

Roman Symbols

A	flux Jacobian
c_p	specific heat at constant pressure
c_{ps}	specific heat at constant pressure of the s^{th} species
c_v	specific heat at constant volume
c_{vs}	specific heat at constant volume of the s^{th} species
c	speed of sound
e_{0s}	specific formation energy of the s^{th} species
e_i	specific internal energy
e_{is}	specific internal energy of the s^{th} species
e_k	specific kinetic energy
E	total energy per unit volume
$F,\ G,\ H$	flux vectors
G	Gibb¶V energy
H	total enthalpy
K	reaction rate
L	matrix of left eigenvectors
m	momentum
M	Mach number, molecular mass
p	pressure

Q	vector of conserved variables
u, v, w	velocity components
$\tilde{U}, \tilde{V}, \tilde{W}$	normal and tangential components of the velocity
R	gas constant, matrix of right eigenvectors time
t	
$[X]$	molar fraction
T	temperature
V	volume
w	characteristic variable
W	vector of characteristic variable
Y	mass fraction

G reek Symbols

D	third-body efficiency
	vector of source terms
Z	total of production rate and loss rate
J	ratio of specific heat
$/$	diagonal matrix of eigenvalues
O	eigenvalue
Q	stoichiometric coefficient
U	density

CHAPTER 1: INTRODUCTION

1.1 Background and Motivation

Computational Fluid Dynamics (CFD) has been used widely as the main tool for engineering design and analysis in the area of fluid mechanics. CFD refers to the study of fluid flow by numerically solving the fluid dynamical equations such as the Euler and Navier-Stokes equations. Numerical simulation of fluid flow can be performed for a wide range of flow conditions and complex geometry. The fundamental difference which constitutes the modeling process is the physics associated with the flow. One representative example can be found in the modeling of re-entry flow where the combination of low density and high temperature give rise to different physical processes that occurred within the flow such as chemical reaction, ionization and radiation. The governing equations have to be extended to accommodate the non-equilibrium aspect of the flow. The results of these extensions are the two-temperature (2-T), three-temperature (3-T), and multiple-temperature (Multi-T) models (Cambier & Menees, 1989; Candler & MacCormack, 1991) for a gas mixture with multiple species. These models have been used extensively in the hypersonic community to characterize thermo-chemical non-equilibrium flow. In addition, all the physical processes such as chemical and ionization kinetics, internal energies relaxation or radiation must be coupled to the conservative quantities to accurately resolve the flow properties. The coupling between these physical models is critical for an accurate solution.

The complexity of the physics and the fidelity required for these simulations results in intensive computational workload on the computer. Therefore, CFD codes are designed to take advantage of high-performance computing (HPC) capability to speed up the calculation. HPC platforms typically consist of hundreds of processing units connected through a local area network (LAN). The calculation is divided across the number of available processors, making the run time effectively reduced. Unfortunately, traditional HPC platforms are not always readily available due to their cost and storage size. The limitation and restrictions of the traditional HPC platforms have accentuated a need for a compact and low-cost HPC solution where a numerical solver can still be effectively implemented.

1.2 Ove rview of Parallel Computing

Traditional parallel computing architectures come in three standard forms: shared memory, distributed memory, or hybrid distributed-shared memory. Shared memory architecture denotes a system of many processors that share the same memory bank, as illustrated in Figure 1.1. All the processors are accessing the same memory storage, so the data transfer is almost trivial. Parallelization in shared memory architecture is achieved by making use of multi-threading techniques.

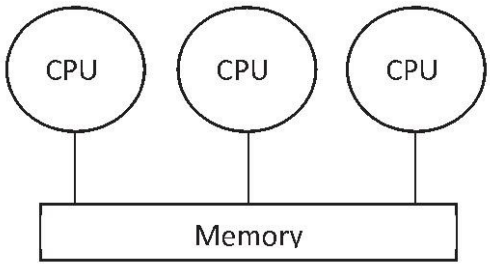


Figure 1.1: Shared Memory Architecture

In contrast, distributed memory architecture (Figure 1.2) is made up of an array of processors (computing nodes) where each processor can have its own memory storage. Parallelization of this kind requires communication between the nodes due to boundary exchange (e.g., ³JKRVW´FHOOV). The overhead due to the boundary exchange needs to be minimized for an effective implementation of a numerical solver. All the nodes can be connected through a standard network protocol.

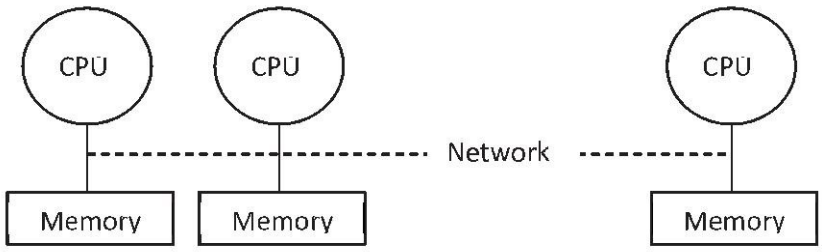


Figure 1.2: Distributed Memory Architecture

Hybrid distributed-shared memory simply refers to the combination of the two former architectures. The schematic diagram for this type is shown in Figure 1.3. Most of the current HPC platforms nowadays fall into this category.

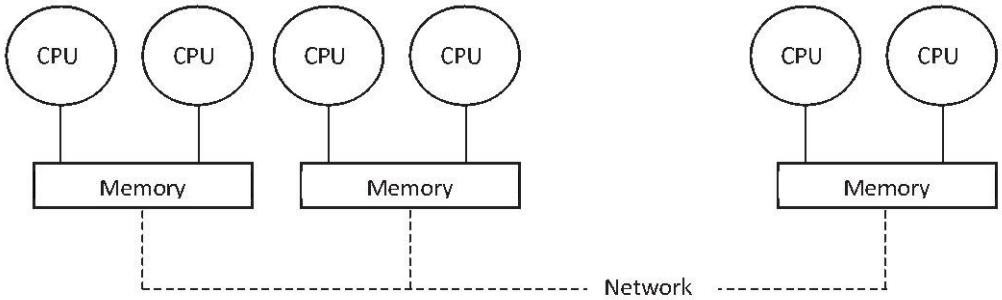


Figure 1.3: Hybrid Shared-Distributed Memory Architecture

Parallelization in shared memory architecture is achieved by constructing multiple threads that can process data simultaneously to reduce computing time. Each thread represents one available processing unit, and the maximum number of parallel threads can be as high as the number of available processors. Multi-threading is readily supported in Java and can be implemented in C/C++ and FORTRAN using OpenMP (Chapman, Jost, van der Pas, & Kuck, 2007). In the context of distributed memory architecture, the work is parallelized by creating multiple processes. One distributed node can be responsible for one or more processes. The communication between the nodes through the network is accomplished using Remote Method Invocation (RMI) in Java (Grosso, 2001) or Message Passing Interface (MPI) in C/C++ and FORTRAN (Gropp, Lusk, & Skjellum, 1999).

1.3 G raphic Processing Unit Computing

During the last seven years, the Graphic Processing Unit (GPU) has been introduced as a promising alternative to high-cost HPC platforms. Within this period, the GPU has evolved into a highly capable and low-cost computing solution for scientific research. Table 1.1 lists the computing power of several models of NVIDIA and AMD GPUs and their memory bandwidth. The theoretical floating-point capability of the GPU is clearly superior to the traditional Central Processing Unit (CPU) due to the fact that GPU is designed for graphic rendering, which is a highly parallel process. Starting from 2008, the GPU began to support double precision calculation, which is demanded for scientific computing. The newest generation of NVIDIA GPUs called ³Fermi´ has been designed to enhance the performance on double precision calculation over the old generation.

Table 1.1: Double precision floating point capability of several models of GPUs.

Model	Double precision Floating-point performance	Memory Bandwidth
NVIDIA Tesla C2050/C2070	515 Gflops	144 GB/sec
NVIDIA Quadro 6000	515 Gflops	144 GB/sec
NVIDIA Quadro 5000	359 Gflops	120 GB/sec
AMD FireStream 9370	528 Gflops	147 GB/sec
AMD FireStream 9350	400 Gflops	128 GB/sec
AMD FireStream 9270	240 Gflops	109 GB/sec

Before the first well-established and user-friendly GPU computing framework called the Compute Unified Device Architecture (CUDA) was introduced in 2007, general purpose GPU computing (GPGPU) had a steep learning curve. The only way to

program on the graphic device was to use specialized application programming interfaces (API) designed for graphic rendering such as OpenGL and Direct3D. ,QWKHHDUO\ ¶V a research group at Stanford University introduced BrookGPU as a programming language for modern graphic hardware in C language syntax; however, the performance of BrookGPU was determined to be poorer than directly using the graphical APIs. Besides CUDA and BrookGPU, there are other languages which were mainly designed for GP*38FRPSXWLQJVXFKDV0LFURVRIW¶V'LUHFW&RPSXWHDQGWKHRSHQ source OpenCL. CUDA is believed to be the most mature programming framework for general purpose computing on the GPU. CUDA has been very attractive to the scientific community due to its capability to perform massive parallel computation with a performance gain of 1-2 orders of magnitude. At the time of writing this thesis, the CUDA programming framework had undergone several development phases and reached a certain level of maturity, which is essential for the design of advanced numerical solvers. All the features are briefly discussed in Chapter 5 of this report.

CUDA has been well received in the areas of scientific and medical research, video processing and financial modeling. One of the early attempts in developing numerical solvers in the field of fluid dynamics on the GPU system had shown very promising performance speed-up. There are also numerous efforts in porting legacy codes into the GPU in order to achieve performance increases. Elsen et al. (2008) re-wrote part of the Navier-Stokes Stanford University solver (NSSUS) to model hypersonic flow on the GPU. The performance of their code ranges from 15 to 40 times speed-up compared to the original solver. At around the same time, Brandvik and Pullan (2008)

also completed porting a two- and three-dimensional Euler code for modeling inviscid flow onto the GPU. The resultant solver ran 30 times faster for a two-dimensional case and 15 times faster for a three-dimensional case. Adaptive mesh refinement (AMR) techniques for finite volume method have also been implemented on the GPU by Schive et al. (2010). The outcome of their work is the GAMER code for astrophysical simulation. Thibault and Senocak (2009) implemented a Navier-Stokes solver for incompressible fluid flow on the GPU. Most of their recent works have concentrated on extending the code to run on a GPU cluster to obtain high scalability. In addition to the finite volume method, the Discontinuous Galerkin (DG) method has also been implemented on the GPU by Klockner et al. (2009). All the past and recent works in writing CFD solvers on the GPU have shown encouraging results. Further performance increases are expected along with the growth of the GPU capability. The remaining challenge in writing a numerical solver on the GPU comes from obtaining the peak-optimized performance. One needs to understand how the data are structured in order to maximize the use of the GPU. Different optimization techniques have been suggested by numerous sources (NVIDIA Corporation, 2010; Kirk & Hwu, 2010), but these techniques are not always pertinent for all the problems. Optimization strategies for GPU computing are discussed in Chapter 5 of this report. It is clear that optimization plays an important role in GPU programming.

1.4 Objectives of the Current Research

The objective of the current research is to implement an advanced CFD framework capable of modeling high-speed fluid flow with high order of accuracy both in the spatial and temporal scale. The solver is designed so that it can easily couple different physical models into the fluid solver. The focus of this work is on the development of an Euler solver for reactive gas coupled with detailed chemical kinetics. The solver takes advantage of the current GPU architecture to achieve high performance and efficient parallelization. Different optimization strategies are considered to improve the performance of the code. In addition, the numerical solver is designed to benefit from the flexibility of Object-Oriented (OO) programming.

CHAPTER 2: COMPUTATIONAL FLUID DYNAMICS

2.1 Introduction

CFD methods assume the flow to be a continuum fluid at various levels of thermo-chemical non-equilibrium. The assumption of a continuum fluid allows the use of conservation laws to model the fluid dynamics. This chapter presents the set of governing equations and the associated physics embedded in the code. The solver utilizes a standard finite volume technique in solving non-linear hyperbolic problems. Several key assumptions have been made in order to simplify the model. The set of model equations discussed herein is applied for an inviscid type of flow field. The flow is also assumed to be in thermal equilibrium such that all the energy modes (translational, rotational, vibrational, and electronic) of all the species are equilibrated with each other. This assumption allows using only one equation for the conservation of energy. In addition, we assume there is no species diffusion, so all the species are convected at the same velocity.

2.2 Governing Equations

The set of Euler equations for a reactive gas flow can be written as

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} + \frac{\partial H}{\partial z} = 0 \qquad (2.1)$$

where Q is the vector of conserved variables and F, G, and H are the flux vectors. The right hand side (RHS) of equation (2.1) denotes the vector of source terms. In general,

the source terms can be composed of exchange terms from different physical processes that occurred in the flow such as chemical reactions, diffusion, ionization and internal energy relaxation. In this work, the source terms represent the exchange process due to chemical reactions. The detailed form of these vectors is given as

$$Q=\begin{bmatrix}\rho\\ \rho_1\\ \vdots\\ \rho_N\\ \rho u\\ \rho v\\ \rho w\\ E\end{bmatrix},\quad F=\begin{bmatrix}\rho u\\ \rho_1 u\\ \vdots\\ \rho_N u\\ \rho u^2+p\\ \rho uv\\ \rho uw\\ (E+p)u\end{bmatrix},\quad G=\begin{bmatrix}\rho v\\ \rho_1 v\\ \vdots\\ \rho_N v\\ \rho uv\\ \rho v^2+p\\ \rho vw\\ (E+p)v\end{bmatrix},\quad H=\begin{bmatrix}\rho w\\ \rho_1 w\\ \vdots\\ \rho_N w\\ \rho uw\\ \rho vw\\ \rho w^2+p\\ (E+p)w\end{bmatrix} \tag{2.2}$$

where N is the number of species used in the simulation and E is the total energy per unit volume.

$$E=E_{int}+\frac{1}{2}\rho\left(u^2+v^2+w^2\right) \tag{2.3}$$

Operator-splitting technique is employed here to treat the source term. This technique allows the convective terms and the source terms to be solved independently of each other. At the end of each time step, the total contribution to the change in the conserved variables due to the two processes is added before moving to the next time step.

$$\frac{\partial Q}{\partial t}=\left.\frac{\partial Q}{\partial t}\right|_{conv}+\left.\frac{\partial Q}{\partial t}\right|_{chem} \tag{2.4}$$

One advantage of using an operator-splitting technique is that each physical process can be decoupled and resolved separately from the others. For example, while the convective term can be solved using an explicit method, the source term is solved

using an implicit method to ensure stability. The time step for the convection equation is restricted based on the Courant-Friedrichs-Levy (CFL) condition. The time step for the chemical kinetics is imposed only to ensure accuracy since the stability is guaranteed by the use of an implicit time integration. For chemically reacting flow, the time step is usually dominated by the chemistry time step ($\Delta t_{chem} \ll \Delta t_{CFL}$), resulting in long computing time for the simulation.

The hyperbolic terms due to convection are solved using a standard finite volume technique where the domain is discretized into small computational cells. Assuming the flow solution is continuous in time, the Euler equations can be written in the integral form. The integration is carried at each volumetric cell inside the domain as follows.

$$\frac{\partial}{\partial t}\int_V Q\,dV + \oint_S \vec{n}\cdot\vec{F}\,dS = \frac{\partial}{\partial t}\int_V \vec{\tau}\,dV \tag{2.5}$$

In equation (2.5), the volume integration of the divergence of the flux has been replaced with the surface integration using Gauss's theorem. The flux integration can be approximated by the summation of all the fluxes around the cell interfaces.

$$\frac{\partial}{\partial t}\int_V Q\,dV + \sum_{n=1}^{N} F_n S_n = \frac{\partial}{\partial t}\int_V \vec{\tau}\,dV \tag{2.6}$$

The integral form of the Euler equations is solved at each discretized cell where the volume-average quantities are introduced.

$$\bar{Q} = \frac{1}{V}\int_V Q\,dV \;; \qquad \bar{\vec{\tau}} = \frac{1}{V}\int_V \vec{\tau}\,dV \tag{2.7}$$

Equation (2.6) now becomes

$$\frac{\partial}{\partial t}\left(\frac{1}{V_{n+1}}\sum_{s=1}^{N} Q_s\right) = \frac{1}{V_{n+1}}\sum_{s=1}^{N} \frac{\partial Q_s}{\partial t}$$

From now on, the volume-average quantities are used for simplification purpose. The convective and source terms can now be simplified as:

$$\left(\frac{\partial Q}{\partial t}\right)_{conv} = \frac{1}{V}\sum_{s=1}^{N} F_s S_s \tag{2.9}$$

$$\left(\frac{\partial Q}{\partial t}\right)_{chem} = \dot{\omega} \tag{2.10}$$

Equation (2.9) and (2.10) are the two governing equations used in the solver. Numerical techniques for approximating the solution of these equations are discussed in chapter 3 and 4 of this report.

2.3 Thermodynamics

Assuming there are several species present in the flow, the total pressure of the gas

PL[WXUHFDQEHFRPSXWHGIURP'DOWRQ¶VODZRISDUWLDOSUHVVXUH The pressure of each individual species is determined from the ideal gas relation.

$$P = \sum_{s=1}^{N} P_s = \sum_{s=1}^{N} \rho_s R_s T \tag{2.11}$$

The total density can be computed as the summation of all the species densities.

$$\rho = \sum_{s=1}^{N} \rho_s \tag{2.12}$$

For convenience, the mass fraction of each species is also defined as

$$Y_s \quad \frac{U_s}{U} \tag{2.13}$$

which results in

$$\sum_{s=1}^{N} Y_s \quad 1 \tag{2.14}$$

Assume the gas is thermally perfect, the internal energy and enthalpy of each species can be expressed as a function of only temperature.

$$e_{is} \quad \int c_{vs}(T)\,dT \tag{2.15}$$

$$h_s \quad \int c_{ps}(T)\,dT \tag{2.16}$$

For simplification purpose, we define the gas constant of the mixture to be

$$R \quad \frac{P}{UT} \quad \frac{\sum_{s=1}^{N} U_s R_s T_s}{UT} \quad \sum_{s=1}^{N} \frac{U_s}{U} R_s \quad \sum_{s=1}^{N} Y_s R_s \tag{2.17}$$

where Y_s is the mass fraction and R_s is the gas constant of each species.

We also define

$$J \quad 1 \quad \frac{\sum_{s=1}^{N} Y_s R_s}{\sum_{s=1}^{N} Y_s c_{vs}} \quad \frac{R}{C_v} \tag{2.18}$$

where J can be defined as the ratio of specific heat of the gas mixture. In order to construct the eigensystem for the Euler equations which is necessary for the flux calculation and linearization, we also define the pressure derivatives with respect to all the conserved variables. The derivation is straight forward starting with the definition of total pressure. By differentiating both sides of equation (2.11),

$$dP \;\; dT \sum_{s\ 1}^{N} R_s U_s \quad T \sum_{s\ 1}^{N} R_s \, dU_s \tag{2.19}$$

an expression for the differential pressure is obtained. The differential temperature can be derived by realizing that energy is the summation of kinetic energy and internal energy.

$$E \quad \sum_{s\ 1}^{N} U_s \, e_{is} \quad \frac{1}{2} \frac{m^2}{U} \tag{2.20}$$

where m represents the momentum of the fluid ($m \quad UV$). Differentiating equation (2.20),

$$dE \quad \sum_{s\ 1}^{N} e_{is} \, dU_s \quad \sum_{s\ 1}^{N} U_s \, de_{is} \quad \frac{1}{2} V^2 \, dU \, V \tilde{} \, dm \tag{2.21}$$

and inserting equation (2.15) for the differential energy of each species,

$$dE \quad \sum_{s\ 1}^{N} e_{is} \, dU_s \quad \sum_{s\ 1}^{N} U_s \, c_{vs} \, dT \quad \frac{1}{2} V^2 \, dU \, V \tilde{} \, dm \tag{2.22}$$

dT now can be expressed as

$$dT \quad \frac{1}{\sum_{s\ 1}^{N} U_s \, c_{vs}} \left[dE \quad \sum_{s\ 1}^{N} e_{is} \, dU_s \quad \frac{1}{2} V^2 \, dU \, V \tilde{} \, dm \right] \tag{2.23}$$

Inserting equation (2.23) to equation (2.19), the final expression for the differential pressure is given as

$$dP = \frac{\sum_{s=1}^{N} U_s R_s \left(dE - \sum_{s=1}^{N} e_{is}\, dU_s \right) + \frac{1}{2} V^2 \left(dU - V \sum_{s=1}^{N} dm_s \right) + T \sum_{s=1}^{N} R_s\, dU_s}{\sum_{s=1}^{N} U_s c_{vs}} \tag{2.24}$$

where

$$dU = \sum_{s=1}^{N} dU_s \tag{2.25}$$

All the pressure derivatives term now can be derived as:

$$P_{U_i} = \left(\frac{\partial P}{\partial U_i} \right)_{U_j, m, E} = \left(J - 1 \right) e_{ii} + \frac{1}{2} V^2 \tag{2.26}$$

$$P_U = \left(\frac{\partial P}{\partial U} \right)_{m, E} = \sum_{s=1}^{N} Y_{U_s} P + \left(J - 1 \right) \sum_{s=1}^{N} \left(-e_{is} + \frac{1}{2} U_s V^2 \right) \frac{\rho}{U} \tag{2.27}$$

$$P_m = \left(\frac{\partial P}{\partial m} \right)_{U_s, E} = \left(J - 1 \right) V \tag{2.28}$$

$$P_E = \left(\frac{\partial P}{\partial E} \right)_{U_s, m} = J - 1 \tag{2.29}$$

We define the total enthalpy H as follows. (2.30)

$$H = \frac{P + E}{U}$$

The speed of sound can also be defined

$$c^2 \left[Y_s \frac{\partial P}{\partial U_s} \Big|_{m,E} + HV^2 \frac{\partial P}{\partial E} \Big|_{U_s} \right]$$

$$\frac{J+1}{U} \sum_{s=1}^{N} \left(U_s e_{is} - \frac{1}{2} U_s V^2 \right) + \frac{p}{U} (J+1) HV^2 \tag{2.31}$$

$$(J+1) \left(\frac{V^2}{2} - \frac{J+1}{U} E_i \right) - \frac{p}{U} (J+1) HV^2$$

$$\frac{P}{U} (J+1) \frac{H E}{U}$$

where

$$J \frac{P}{U}$$

V^2 u^2 v^2

$$w^2 \tag{2.32}$$

2.4 Eigensystem

In order to solve the multi-dimensional Euler equations (convective terms), a dimensional splitting method (Toro E. F., 2009) was used to decompose the system into multiple one-dimensional sweeps. This approach can effectively lower the amount of flux data stored on the machine. Since each sweep is independent of the others, the flux storage can be effectively reduced by one-third. Also, dimensional splitting allows a straight forward implementation of the Riemann solver for the non-linear hyperbolic problems (Toro E. F., 2009). For each sweep, the non-linear set of governing equations takes the form:

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial n} = 0 \tag{2.33}$$

where n is the direction of sweeping. By applying the chain rule on the spatial derivative, one could obtain

$$\frac{\partial Q}{\partial t} + \left[\frac{\partial F}{\partial Q}\right]\left[\frac{\partial Q}{\partial n}\right] = 0 \tag{2.34}$$

The system of equations (2.33) can be classified as a hyperbolic system of partial differential equations (PDE) since the Jacobian matrix A can be diagonalized and all the eigenvalues of the resultant diagonalization are real.

$$A = R \Lambda L \tag{2.35}$$

The eigensystem used in this work are similar to the one given by Cambier et al. (1989) and Gnoffo et al. (1989). The real eigenvalues of the system are

$$\Lambda = \begin{bmatrix} \tilde{U} & & & & \\ & \tilde{U} & & & \\ & & \tilde{U} + \tilde{c} & & \\ & & & \tilde{U} - \tilde{c} & \\ & & & & \tilde{U} \end{bmatrix} \tag{2.36}$$

The set of left eigenvectors is written as

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 & Y_1 & 0 & 0 & Y_1 \\ 0 & 1 & \cdots & 0 & Y_2 & 0 & 0 & Y_2 \\ \vdots & & & & & & & \vdots \\ 0 & 0 & \cdots & 1 & Y_N & 0 & 0 & Y_N \\ u & u & \cdots & u & u - \xi_x c & \eta_x & \zeta_x & u + \xi_x c \\ v & v & \cdots & v & v - \xi_y c & \eta_y & \zeta_y & v + \xi_y c \\ w & w & \cdots & w & w - \xi_z c & \eta_z & \zeta_z & w + \xi_z c \\ e_k - \frac{P_{\rho_1}}{P_\beta} & e_k - \frac{P_{\rho_2}}{P_\beta} & \cdots & e_k - \frac{P_{\rho_N}}{P_\beta} & H - c\tilde{U} & \tilde{V} & \tilde{W} & H + c\tilde{U} \end{bmatrix} \tag{2.37}$$

where e_k is the specific kinetic energy and $\tilde{U}$, $\tilde{V}$, and $\tilde{W}$ are the velocity components normal and tangential to the interface.

$$e_k = \frac{1}{2}\left(u^2 + v^2 + w^2\right) \tag{2.38}$$

$$\tilde{U} = u\xi_x + v\xi_y + w\xi_z \tag{2.39}$$

$$\tilde{V} = u\eta_x + v\eta_y + w\eta_z \tag{2.40}$$

$$\tilde{W} = u\zeta_x + v\zeta_y + w\zeta_z \tag{2.41}$$

The set of right eigenvectors is defined as

$$R = \begin{bmatrix} 1 & \frac{P_{U_1}}{c^2} & Y_1\frac{P_{U_2}}{c^2} & Y_1\frac{P_{U_N}}{c^2} & Y_1 u\frac{P_E}{c^2} & Y_1 v\frac{P_E}{c^2} & Y_1 w\frac{P_E}{c^2} & Y_1\frac{P_E}{c^2} \\ Y_2 & \frac{P_{U_1}}{c^2} & 1+Y_2\frac{P_{U_2}}{c^2} & Y_2\frac{P_{U_N}}{c^2} & Y_2 u\frac{P_E}{c^2} & Y_2 v\frac{P_E}{c^2} & Y_2 w\frac{P_E}{c^2} & Y_2\frac{P_E}{c^2} \\ Y_N & \frac{P_{U_1}}{c^2} & Y_N\frac{P_{U_2}}{c^2} & 1+Y_N\frac{P_{U_N}}{c^2} & Y_N u\frac{P_E}{c^2} & Y_N v\frac{P_E}{c^2} & Y_N w\frac{P_E}{c^2} & Y_N\frac{P_E}{c^2} \\ \frac{P_{U_1}}{2c^2}(\tilde{V}-cU) & \frac{P_{U_2}}{2c^2}(\tilde{V}-cU) & \frac{P_{U_N}}{2c^2}(\tilde{V}-cU) & \frac{P_E}{2c^2}(\tilde{V}+cK_x) & \frac{P_E}{2c^2}(\tilde{V}+cK_y) & \frac{P_E}{2c^2}(\tilde{V}+cK_z) & \frac{P_E}{2c^2}\tilde{V} \\ \tilde{V} & \tilde{V} & \tilde{V} & K_x & K_y & K_z & 0 \\ \tilde{w} & \tilde{W} & \tilde{W} &]_x &]_y &]_z & 0 \\ \frac{P_{U_1}}{2c^2}(\tilde{V}+cU) & \frac{P_{U_2}}{2c^2}(\tilde{V}+cU) & \frac{P_{U_N}}{2c^2}(\tilde{V}+cU) & \frac{P_E}{2c^2}(\tilde{V}-cK_x) & \frac{P_E}{2c^2}(\tilde{V}-cK_y) & \frac{P_E}{2c^2}(\tilde{V}-cK_z) & \frac{P_E}{2c^2} \end{bmatrix} \tag{2.42}$$

The left and right eigenvectors can be used to project equation (2.33) from the physical space to the characteristic space.

$$\frac{\partial LQ}{\partial t} + \frac{\partial LQ}{\partial n} = 0 \tag{2.43}$$

By introducing the characteristic variable,

$$W = LQ \tag{2.44}$$

equation (2.43) now becomes

$$\frac{\partial W}{\partial t} + \frac{\partial W}{\partial n} = 0 \tag{2.45}$$

The system of PDEs now has been linearized and decoupled from the original system resulting in a linear system of ordinary differential equations (ODE). It must be noted that the original set of PDEs is non-linear in nature, so the transformation is only valid

when the Jacobian matrix can be assumed to be constant. The linearization procedures are performed using Roe-average values of the conserved variables. The Roe-average values are defined as follows:

$$\tilde{U} \quad \sqrt{U_L U_R} \tag{2.46}$$

$$\tilde{u} \quad \frac{\sqrt{U_L}\,u_L \sqrt{U_R}\,u_R}{\sqrt{U_L}\ \sqrt{U_R}} \tag{2.47}$$

$$\tilde{h} \quad \frac{\sqrt{U_L}\ h_L \sqrt{U_R} h_R}{\sqrt{U_L}\ \sqrt{U_R}} \tag{2.48}$$

The Euler equations can now be solved using the approximate Riemann Solver. In order to archive high-order spatial accuracy, the interface values are reconstructed using high-order polynomial approximation. The reconstructed values are then limited to prevent non-physical oscillation in the solution.

CHAPTER 3: NUMERICAL FORMULATION

3.1 Introduction

The typical solution procedure for solving the Euler equations includes discretization of the domain into cells and solving the integral form of the Euler equations at each cell. For simplification, consider the Euler equations in one dimension with no source terms; the discretized form of the equations at each cell can be written as

$$Q_i^{n1} \quad Q_i^n \; \frac{\Delta t}{\Delta x} \left(F_{i\,\frac{1}{2}} - F_{i\,\frac{1}{2}} \right) \tag{3.1}$$

where $F_{i\,\frac{1}{2}}$ and $F_{i\,\frac{1}{2}}$ are the flux vectors at the left and right interfaces of cell i. Evaluation of the flux at each interface depends solely on the values of the neighboring cells.

High-order spatial accuracy is obtained by using high-order polynomial to reconstruct the interface data from a stencil of cell-averages. This process typically consists of two steps: reconstruction and limiting. Several approaches have been considered in this work namely the Monotonicity-Preserving (MP) scheme (Suresh & Huynh, 1997) and the weighted essentially non-oscillatory (WENO) scheme (Liu, Osher, & Chan, 1994; Jiang & Shu, 1996). Detailed derivation of these schemes can be found from the cited references and will not be repeated here. This chapter only summarizes the derived formula of these schemes.

3.2 Data Reconstruction

3.2.1 Monotonicity Preserving (MP) Schemes

We present two versions of the MP schemes so-called MP3 and MP5. The former is reconstructed using a quadratic polynomial resulting in a third-order scheme (Kapper, 2009). The numerical approach in reconstructing the high-order term is discussed in Huynh (1993) and will not be repeated here. The MP3 scheme can be written as

$$u_L^{MP3} = \frac{1}{6}\left(2u_{j-1} + 5u_j - u_{j+1}\right) \tag{3.2}$$

where u_L is the left state of cell *j*. The corresponding value of the right state can be determined from symmetry. While the MP3 scheme utilizes a stencil of three cell-averages, the fifth-order MP scheme (MP5) uses a stencil of 5 cell-averages as

$$u_L^{MP5} = \frac{1}{60}\left(2u_{j-2} - 13u_{j-1} + 47u_j + 27u_{j+1} - 3u_{j+2}\right) \tag{3.3}$$

The reconstructed values for these schemes yield high-order accuracy in region where the flow solution is smooth. However, these values cannot be used near a discontinuity in the flow due to the unphysical oscillation in the solution. In order to prevent oscillation near the shock or the contact discontinuity, the reconstructed values are limited following a monotonicity-preserving procedure as discussed in Suresh and Huynh (1997) .

$$u^L_{j+1/2} \leftarrow \text{median}\left(u^L_{j+1/2}, u_j, u^{MP}\right) \tag{3.4}$$

$$u^{MP} = u_j + \text{minmod}\left[u_{j+1} - u_j, \alpha\left(u_j - u_{j-1}\right)\right] \tag{3.5}$$

The diagram illustrating the stencil used for both MP schemes is shown in Figure 3.1.

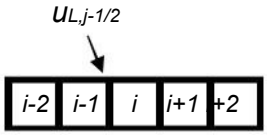


Figure 3.1: Diagram of the stencil used in MP scheme

The value of D in equation (3.15) is typically 2 or 4. The MP schemes have been determined to hold some CFL restriction based on the value of D . It is recommended to use a CFL number close to 0.33 for a stable solution. In addition, the original MP5 scheme of Suresh and Huynh also contains an additional accuracy-preserving constraint to avoid the loss of accuracy near the shock. The detail of the constraint is discussed in their paper (Suresh & Huynh, 1997).

3.2.2 Weighted Essentially Non-Oscillatory Schemes

Weighted Essentially Non-Oscillatory (WENO) schemes, developed by Liu et al. (1994) and Jiang and Shu (1996) are based on the Essentially Non-Oscillatory (ENO) schemes developed by Harten et al. (1987) in the form of cell-averages. The WENO schemes utilize an adaptive-stencil approach as in ENO scheme except that all the contribution of the stencils is taken into account as a convex combination. The WENO schemes preserve the essentially non-oscillatory property of the original ENO scheme, but yield one order higher in the accuracy of the smooth solution of the flow. The fifth-order WENO scheme is given as follows.

$$u_L \quad Z_1 u_L^{(1)} \quad Z_2 u_L^{(2)} \quad Z_3 u_L^{(3)} \tag{3.6}$$

Three stencils are utilized herein with the non-linear weights ω .

$$u_L^{(1)} \quad \frac{1}{3}u_{j-2} \quad \frac{7}{6}u_{j-1} \quad \frac{11}{6}u_j \tag{3.7}$$

$$u_L^{(2)} \quad -\frac{1}{6}u_{j-1} \quad \frac{5}{6}u_j \quad \frac{1}{3}u_{j+1} \tag{3.8}$$

$$u_L^{(3)} \quad \frac{1}{3}u_j \quad \frac{5}{6}u_{j+1} \quad -\frac{1}{6}u_{j+2} \tag{3.9}$$

The non-linear weights in this case are adapted to the smoothness of the stencil to preserve the essentially non-oscillatory properties of the scheme. The weight of a discontinuous stencil is effectively reduced to zero. The formulation of the non-linear weights is given as

$$\omega_{i3} \quad \frac{\alpha_i}{\sum_{n=1}^{3}\alpha_n} \qquad \alpha_i \quad \frac{d_i}{(\varepsilon + IS_i)^r} \tag{3.10}$$

H

In equation (3.10), H is placed in the denominator to avoid it to be zero. Numerical experiments suggest H being in the range of 10^{-5} to 10^{-7}. The optimal weights are given by

$$\begin{aligned} C_1^r &= \frac{1}{10} \\ C_2^r &= \frac{6}{10} \\ C_3^r &= \frac{3}{10} \end{aligned} \tag{3.11}$$

and the smoothness indicators IS are

$$IS_1 = \frac{13}{12}\left(u_{j-2} - 2u_{j-1} + u_j\right)^2 + \frac{1}{4}\left(u_{j-2} - 4u_{j-1} + 3u_j\right)^2 \tag{3.12}$$

$$IS_2 = \frac{13}{12}\left(u_{j-1} - 2u_j + u_{j+1}\right)^2 + \frac{1}{4}\left(u_{j-1} - u_{j+1}\right)^2 \tag{3.13}$$

$$IS_3 = \frac{13}{12}\left(u_j - 2u_{j+1} + u_{j+2}\right)^2 + \frac{1}{4}\left(3u_j - 4u_{j+1} + u_{j+2}\right)^2 \tag{3.14}$$

The WENO schemes have been determined to work well with the total-variation-diminishing (TVD) Runge-Kutta (RK) methods. The TVD RK methods are discussed in section 3.4.2. Recently, Balsara and Shu (2000) have introduced another variation of the WENO schemes called the Monotonicity-Preserving Weighted Non-Oscillatory (MPWENO) schemes. This scheme is different than the WENO version in a sense that the smooth solution following the WENO reconstruction procedure is limited using the MP constraint discussed in section 3.2.1. The resulting scheme yields slightly higher accuracy than the original WENO scheme and more efficient than the MP schemes in terms of the CFL restriction.

3.3 Flux Calculation

The solver utilizes two standard flux splitting techniques: Roe flux-difference splitting and Harten-van Leer-Lax-Einfeldt flux. Both of the fluxes have been tested for several test cases and are also performed well for chemically reacting flow with multiple species. However, an entropy fix is required for the Roe flux-difference splitting when trying to resolve flow with strong rarefaction.

3.3.1 Roe Flux-Difference Splitting

Roe flux-difference splitting is a standard flux splitting technique for the fluid dynamics equations. The idea of Roe flux is to split the flux based on the characteristic wave speed so that the flux is purely upwinding.

$$f_{i\frac{1}{2}}^{Roe} = \frac{1}{2}\left[f_L + f_R - \left| \hat{A}_{i\frac{1}{2}} \right| \left(w_R - w_L \right) \right] \tag{3.15}$$

The flux presented in equation (3.15) is written in form of a characteristic flux. Transformation between the conservative and characteristic variables can be performed via the transformation matrices mentioned in chapter 2 (equation (2.36) and (2.42)). The Roe flux splitting, however, contains issues when trying to resolve the flow with sonic or transonic rarefactions. An entropy fix needs to be applied for such cases.

3.3.2 Harten-Lax-van L eer-Einfeldt (H L L E) Flux

Another flux formulation implemented in this framework is the Harten-Lax-van Leer-Einfeldt Riemann (HLLE) flux. Details of the derivation are given in Harten (1997). The HLLE flux can be summarized as

$$f_{i\frac{1}{2}}^{HLLE} = \frac{b^{+}_{i\frac{1}{2}} f^{r}_{i\frac{1}{2}} - b^{-}_{i\frac{1}{2}} f^{l}_{i\frac{1}{2}}}{b^{+}_{i\frac{1}{2}} - b^{-}_{i\frac{1}{2}}} - \frac{b^{+}_{i\frac{1}{2}} b^{-}_{i\frac{1}{2}}}{b^{+}_{i\frac{1}{2}} - b^{-}_{i\frac{1}{2}}} \Delta w_{i\frac{1}{2}} \tag{3.16}$$

where

$$\begin{aligned} b^{+}_{i\frac{1}{2}} &= \max\left(0, b^{r}_{i\frac{1}{2}} \right) \\ b^{-}_{i\frac{1}{2}} &= \min\left(0, b^{l}_{i\frac{1}{2}} \right) \end{aligned} \tag{3.17}$$

and

$$b^{r}_{i\frac{1}{2}} \quad \max(u_{i\frac{1}{2}} \quad c_{i\frac{1}{2}}, u_{i1} \quad Ec_{i1}) \qquad (3.18)$$

$$b^{l}_{i\frac{1}{2}} \quad \min(u_{i\frac{1}{2}} \quad c_{i\frac{1}{2}}, u_{i} \quad Ec_{i})$$

with

$$E \quad \sqrt{\frac{J\,1}{2J}} \qquad (3.19)$$

The HLLE flux is known to be more diffusive than the other fluxes due to the large bound of the numerical signal velocities: b^{+} , b^{-}.

3.4 Time Marching M ethods

3.4.1 Explicit Euler

The Explicit Euler method serves as the most basic kind of time integration method. It is given as

$$\begin{array}{l} Q^{n}{}_{1} \quad Q^{n} \quad 'tL \\ Q^{n} \end{array} \qquad (3.20)$$

where the spatial operator is

$$\begin{array}{ll} L & \\ Q & \frac{1}{V}\mathrm{I}\, F \quad A \\ & \quad s \quad s \end{array} \qquad (3.21)$$

Although the implementation of the Explicit Euler is straight forward, it is not stable and would result in oscillation in the solution especially when coupled with a high-order scheme for the spatial derivatives. High-order time integration methods are needed to ensure stability and accuracy of the solver.

3.4.2 Total-Variation-Diminishing Runge- K utta

The high-order time integration method used in this work is the total-variation-diminishing (TVD) Runge-Kutta (RK) method. The third-order version of the RK methods (RK3) is implemented for most of the high-order simulation. The formulation of the RK3 method is given as

$$Q^{n} \quad \Delta t L$$
$$Q^{n1/3} \quad Q^{n} \tag{3.22}$$

$$\frac{3}{4} \quad \frac{1}{4} Q^{n1/3} \quad \Delta t L$$
$$Q^{n2/3} \quad Q^{n} \quad Q^{n1/3} \tag{3.23}$$

$$\frac{1}{3} \quad \frac{2}{3} Q^{n2/3} \quad \Delta t L$$
$$Q^{n1} \quad Q^{n} \quad Q^{n2/3} \tag{3.24}$$

Since the RK3 method is a multi-stage integration method, the solution is going through a series of predictor-corrector steps for every iteration. One disadvantage of the RK3 scheme is the overhead caused by storing the old solution of the Q vector at every RK step. In addition, boundary conditions need to be enforced at every time step, which makes the method less efficient for domain decomposition.

3.5 A rbitrary Derivative Riemann Solve r (A D E R)

Recently, a new approach for implementing high-order Riemann solver has been introduced by Titarev and Toro (2005). This new class of Riemann solver is called the Arbitrary Derivative Riemann (ADER) solver. The unique feature of the ADER schemes is that they can accomplish high-order accuracy in time without using multi-stage time

integration methods. This feature is very advantageous for parallel computing because the overhead due to boundary exchange can be reduced to the minimum.

At each interface we seek the solution of the generalized Riemann problem as follows.

$$\begin{aligned} & w_t Q \;\; w_x F(Q) \;\; 0 \\ & Q^{(k)}(x,0) \;\; \begin{matrix} (k) \\ \circledR q_L \end{matrix} \;\; (x) \text{ if } x \;\; 0 \\ & \qquad\qquad \overline{} q_R^{(k)}(x) \text{ if } x \;!\; 0 \end{aligned} \tag{3.25}$$

The approximated solution of equation (3.25) can be given in terms of a Taylor series expansion in time.

$$Q(x_{i1/2}, t \;\; h) \;\; Q(x_{i1/2},0) \;\; \sum_{k\;1}^{r\,1} \frac{h^k}{k!} \frac{w^k}{wt^k} Q(x_{i1/2},0) \tag{3.26}$$

The first term on the right-hand-side of equation (3.26) can be found by solving the classical Riemann problem at the interface. The high-order terms can be determined by using the Cauchy-Kowalewski procedure which relates all the time derivatives to the spatial derivatives.

$$w_t Q \;\; \frac{wF}{wQ} w_x Q \tag{3.27}$$

$$w_{tx} Q \;\; \frac{w^2 F}{wQ^2} w_x Q^2 \;\; \frac{w}{wQ}\frac{F}{} w_{xx} Q \tag{3.28}$$

$$w_{tt} Q \;\; \frac{w^2 F}{wQ^2} w_t Q \;\; w_x Q \;\; \frac{w}{wQ}\frac{F}{} w_{xt} Q \tag{3.29}$$

The solution of the generalized Riemann is then used to compute the numerical flux at the interface. There are two ways of evaluating the flux: state-expansion and flux-expansion. In this work, we use the state-expansion version in which the flux is directly evaluated from the solution of the generalized Riemann problem (equation 3.26). The

flux-expansion approach (Toro & Titarev, 2005), on the other hand, evaluates the flux as the Taylor time expansion of the physical flux.

$$F(x_{i1/2}, t \quad h) \quad F(x_{i1/2},0) \quad \sum_{k\ 1}^{r\ 1} \frac{h^k}{k!} \frac{w^k}{t^k} F(x_{i1/2},0) \tag{3.30}$$

The high-order terms of the fluxes can also be expressed in terms of the time derivatives of the interface state. The solution of the cell can now be updated using a one-step formula similar to the standard Euler explicit method in equation (3.20).

CHAPTER 4: CHEMICAL KINETICS

4.1 Introduction

This chapter introduces the chemistry model used in the solver. When the temperature of the flow is high enough, all the species present in the gas will begin to react at different rates. Each species now has to be tracked because of their fundamental differences in the thermodynamic properties. For example, the internal energy and the heat capacity of a reacting flow change rapidly depending on the temperature of the flow and the mixture composition. In order to capture the chemical reactions and their effects to the flow properties, one could either use a one-step kinetics model or a detailed kinetics model. The detailed kinetics model is implemented in this work to model all the elementary reactions and their reverse processes.

4.2 Chemistry Model

An elementary reaction takes the form

$$\sum_{s=1}^{N} Q_r' > X_s @ \underset{K_{br}}{\overset{K_{fr}}{@}} \sum_{s=1}^{N} Q_r'' > X_s @ \tag{4.1}$$

where Q_r' and Q_r'' are the molar stoichiometric coefficients of the reactants and products of each reaction. $[X_s]$ is defined as the molar concentration of the s^{th} species. K_{fr} and K_{br} are defined as the forward and backward rates of each reaction. The rate constant can be estimated using the empirical Arrhenius law

$$K_{fr} = A_{fr} T^{E_r} \exp\left(\frac{E_r}{RT}\right) \tag{4.2}$$

where A_{fr} is the pre-exponential factor, E_r is the temperature exponent, and E_r is the activation energy. If a reaction is assumed to reach equilibrium, the forward and backward rates are related by the equilibrium constant

$$K_e = \frac{K_f}{K_b} = \left(\frac{P_a}{RT}\right)^{\sum_s Q_s} \exp\left(\frac{\Delta G^0}{RT}\right) \tag{4.3}$$

where P_a is the pressure chosen to be 1 bar and ΔG^0 is the change in Gibbs free energy.

For each reaction, the progression rate can be written as

$$Q_r = K_{fr} \prod_{s=1}^{N} [X_s]^{Q'_{kr}} - K_{br} \prod_{s=1}^{N} [X_s]^{Q''_{kr}} \tag{4.4}$$

In case of a three-body reaction, the progression rate can be modified as

$$Q_r = \sum_s D_{rs} [X_s] \left(K_{fr} \prod_{k=1}^{N} [X_k]^{Q'_{kr}} - K_{br} \prod_{k=1}^{N} [X_k]^{Q''_{kr}} \right) \tag{4.5}$$

where D_{rs} is the third-body efficiency of the s^{th} species. The rate of production for each species can be determined from

$$Z_s = \sum_{r=1}^{N} Z_{rs} = \sum_{r=1}^{N} Q_{rs} Q_r \tag{4.6}$$

where M_s is the mean molecular weight of the s^{th} species.

By conservation of mass, sum of all the species production rates should be equal to zero which yields the following expression.

$$\sum_{s=1}^{N} M_s Z_s \qquad 0 \tag{4.7}$$

In order to solve for the change in the species concentration through production and loss rate, one needs to know all the changes in the thermodynamics for each reaction as well as their rates. In practice, the backward rate can also be computed using curve-fitting technique with the temperature as an input, but to be more rigorous, it is recomputed using the equilibrium constant. From the numerical point of view, all these quantities are read from separated data files which contain all the species information used for the computation along with the elementary reactions.

4.3 Implicit Formulation

The change in the species density due to chemical reaction is solved by using Equation (2.10). Since this is a stiff ODE, an implicit method is chosen to ensure the stability of the solution. The implicit formulation is given as

$$\frac{dQ}{dt} \quad :^{n1} \tag{4.8}$$

Using a Taylor series expansion in time, equation (4.8) can be written as

$$\frac{dQ}{dt} \quad : \; 't \quad \frac{w:}{wt} \quad : \; 't \quad \frac{w:}{wQ} \frac{d}{dt} Q \tag{4.9}$$

where the time derivative has been replaced by applying the chain rule. The $\frac{dQ}{dt}$ term can be computed as:

$$\begin{bmatrix} \frac{\partial \omega_1}{\partial Q_1} & \cdots & \frac{\partial \omega_1}{\partial Q_N} \\ \vdots & & \vdots \\ \frac{\partial \omega_N}{\partial Q_1} & \cdots & \frac{\partial \omega_N}{\partial Q_N} \end{bmatrix} \begin{bmatrix} \frac{dQ_1}{dt} \\ \vdots \\ \frac{dQ_N}{dt} \end{bmatrix} \tag{4.10}$$

For simplification, instead of computing $\frac{dQ}{dt}$ where Q consists of all the conserved variables, we use the change of molar concentration of each species while keeping the change in total energy the same. This is referred as Q_{chem} and can be defined as

$$Q_{chem} = \begin{bmatrix} X_1 \\ \vdots \\ X_N \\ E \end{bmatrix} \tag{4.11}$$

The source terms now express the change in the molar concentration of each species and the change in energy.

$$\omega_{chem} = \begin{bmatrix} Z_1 \\ \vdots \\ Z_N \\ \sum_s M_s Z_s e_{0s} \end{bmatrix} \tag{4.12}$$

In equation (4.12), Z_s represents the total change in molar concentration of the s^{th} species and $e_{0\,s}$ represents the formation energy of the s^{th} species. The change in the conserved variables can be easily recomputed using the following transformation matrix.

$$Q = \begin{bmatrix} M_1 & & \\ & \ddots & \\ & & M_N \\ & & 1 \end{bmatrix} Q_{chem} \tag{4.13}$$

The resultant matrix $\frac{\partial \omega}{\partial Q}$ can be written as

$$\frac{\partial \Omega}{\partial Q} = \begin{bmatrix} \frac{\partial Z_1}{\partial [X_1]} & \frac{\partial Z_1}{\partial [X_2]} & \cdots & \frac{\partial Z_1}{\partial [X_N]} & \frac{\partial Z_1}{\partial E} \\ \frac{\partial Z_2}{\partial [X_1]} & \frac{\partial Z_2}{\partial [X_2]} & \cdots & \frac{\partial Z_2}{\partial [X_N]} & \frac{\partial Z_2}{\partial E} \\ \vdots & & & & \vdots \\ \frac{\partial Z_N}{\partial [X_1]} & \frac{\partial Z_N}{\partial [X_2]} & \cdots & \frac{\partial Z_N}{\partial [X_N]} & \frac{\partial Z_N}{\partial E} \\ \frac{\partial \sum_s M_s Z_s e_{0s}}{\partial [X_1]} & \frac{\partial \sum_s M_s Z_s e_{0s}}{\partial [X_2]} & \cdots & \frac{\partial \sum_s M_s Z_s e_{0s}}{\partial [X_N]} & \frac{\partial \sum_s M_s Z_s e_{0s}}{\partial E} \end{bmatrix} \quad (4.14)$$

with all the derivatives expressed as

$$\frac{\partial Z_i}{\partial X_j} = \sum_r \left[\nu''_{ri} \left(K_{fr} \frac{\nu''_{rj}}{X_j} \prod_s [X_s]^{\nu''_{rs}} - K_{br} \frac{\nu'_{rj}}{X_j} \prod_s [X_s]^{\nu'_{rs}} \right) \right] \tag{4.15}$$

$$\frac{\partial Z_i}{\partial E} = \frac{1}{\rho C_v} \sum_r \left[\nu''_{ri} \left(K_{fr} \left(\frac{\beta_{fr}}{T} + \frac{E_{fr}}{RT^2} \right) \prod_s [X_s]^{\nu''_{rs}} - K_{br} \left(\frac{\beta_{br}}{T} + \frac{E_{br}}{RT^2} \right) \prod_s [X_s]^{\nu'_{rs}} \right) \right] \tag{4.16}$$

$$\frac{\partial \left(\sum_i M_i Z_i e_{0i} \right)}{\partial X_j} = \sum_i M_i e_{0i} \frac{\partial Z_i}{\partial X_j} \tag{4.17}$$

$$\frac{\partial \left(\sum_i M_i Z_i e_{0i} \right)}{\partial E} = \sum_i M_i e_{0i} \frac{\partial Z_i}{\partial E} \tag{4.18}$$

With all the derivatives term computed, equation (4.9) is reduced to a linear system of algebraic equations

$$AX = B \tag{4.19}$$

which can be solved using a direct Gaussian elimination method.

It must be noted that as the number of the species increases, the size of these matrices is also increased by N^2 and the Gaussian Elimination step scales as N^3. Solving the chemical kinetics at every cell is very computationally intensive. The implementation

of the chemical kinetics solver is very efficient by making use of the GPU architecture. The performance of the kinetics solver is discussed in Chapter 7 of this report.

CHAPTER 5: PARALLEL FRAMEWORK

5.1 Introduction

As introduced in chapter 1, the GPU has shown to be very capable of performing scientific computing especially in the area of CFD. CUDA is the programming language of choice for general purpose programming on the GPU. CUDA is an extension from the traditional C language with additional API calls to perform data transfer to the graphic device as well as instructing the device to do work. Recently, CUDA has begun to support C++ language with Object-Oriented features like classes and templates. This capability offers the flexibility in writing code on the GPU. This chapter covers the basics of GPU computing as well as the standard optimization techniques.

5.2 Memory Architecture

The fundamental difference between the GPU and the CPU is that the GPU is designed to maximize the floating-point calculation capability by reducing the control logic for each execution thread. The design philosophy of the GPU is driven by the game industry which aims at the capability to perform massive floating-point operations required for fast graphic rendering. Each graphic device has a set of streaming multi-processors (SM) which also contains an array of streaming processors (SP). These processors can perform massively parallel calculation, and the data can be accessed at different levels of the memory hierarchy. The CUDA memory structure can be

categorized into four different types: global memory, constant memory, shared memory and registers.

Global memory is implemented as dynamic random-access memory (DRAM) which holds the maximum size on the device. Table 5.1 lists several models of the NVIDIA GPUs in terms of the number of CUDA cores, the DRAM size and memory bandwidth.

Table 5.1: Comparison of several models of NVIDIA GPUs

Model	Number of Cores	Memory	Memory Bandwidth
GTX 480	480	1.5 GB GDDR5	177 GB/sec
GTX 580	512	1.5 GB GDDR5	192 GB/sec
Quadro 5000	352	2.5 GB GDDR5	120 GB/sec
Quadro 6000	448	3.0 GB GDDR5	144 GB/sec
Tesla C2050/C2070	448	3GB/6GB GDDR5	144 GB/sec

Data resided on the global memory can be accessed by any processor at any given time. Global memory can also be communicated with the host by calling API functions. Although the size of the DRAM is large, directly accessing the global memory results in high memory latency which can significantly reduce the data parallelism of the program. Constant memory allows read-only access so it is faster than global memory. Constant memory is cached for efficient memory access so its size is very limited. Shared memory is the on-chip memory space for each SM which can provide fast and efficient access pattern (100-150 times faster than global memory). Register is the fastest form of memory on the device but it can only be accessed by each SP. The sizes of shared memory and registers are very small compared to the global memory. One has to be careful not to exceed the size of shared memory and registers. In addition to the four

basic types of memory, there is also another type of memory which is designed for graphic rendering known as texture memory. Texture memory is read-only and also provides fast memory access. Texture memory can also be utilized for calculation on the GPU.

5.3 GPU Programming

Parallel calculation on the GPU is initiated by invoking a kernel function from the host. A kernel function acts as an instruction issued from the host to be executed on the device. Parallelization on the GPU is accomplished by sizing a virtual space on the device which is referred as a grid. A grid consists of multiple blocks and each block contains a number of threads which is handled by the graphic processors. Both the grid and block can be one, two or three-dimensional. The dimensions of the grid and block are independent of the global memory size. The execution order is scheduled based on the available number of SMs available on the device. Table 5.2 indicates all the memory types in CUDA and their scopes. For example, shared memory allocated within a block can only be accessed by the threads of that block.

Table 5.2: CUDA memory hierarchy and their scope

Memory Type	Scope	Life time
Global	Grid, block, thread	Application
Constant	Grid, block, thread	Application
Shared	Block	kernel
Register	Thread	kernel
Texture memory	Grid, block, thread	Application

Once a kernel is launched, each block is typically handled by 2 SMs. All the threads in each EORFNDUHRUJDQL]HGLQWR³ZDUSV´DQG each warp is executed in a single-instruction multiple-data (SIMD) manner. All the warps within a block can be executed in any order to maximize the computational resources. An example of a CUDA program is given below.

```
__global__ void kernel (float* dA) {
        int index = blockIdx.x * blockDim.x + threadIdx.x;;
        dA[index]  *= 2.f;;
}
int main() {
        float       //pointer to host memory
        *hA;;       //pointer to device memory
        float
        *dA;;

        hA = (float*) malloc (100*sizeof(float));; // allocate memory on host
        cudaMalloc((void**)&dA,100*sizeof(float));; // allocate memory on
        device for (int i=0;;i<10;;i++) hA[i] = (float) drand48();; // initialize
        the array

        /  transfer memory to device
        cudaMemcpy(dA,hA,100*sizeof(float),cudaMemcpyHostToDe
        vice);;

        int gridsize = 10;; int blocksize = 10;;

        // invoke CUDA kernel
        kernel <<< gridsize, blocksize >>> (dA);;

        /  transfer memory back to host
        cudaMemcpy(hA,dA,100*sizeof(float),cudaMemcpyDeviceTo
        Host);;

        /  free memory on host and device
        free (hA);; cudaFree (dA);;

        return 0;;
}
```

Figure 5.1: An example CUDA program

The example shown in Figure 5.1 demonstrates how to write a parallel program in CUDA. The program starts with a kernel definition which is very similar to a regular C-function. Each thread is assigned to an element of array A. All the threads and blocks are identified by built-in indices called *blockIdx* and *threadIdx*. In this example, all the threads are instructed to double the current value of the array element. This is very similar to a for-loop in C with all the entries being executed in parallel. The main program highlights all the steps required to allocate memory on the device as well as transferring data to the device. The sizes of the grid and block must be specified before invoking the kernel. In this example, both sizes are specified as 10, and we only consider one-dimensional block and one-dimensional grid. Similarly, in order to construct a two- or three-dimensional block, one must also specify the dimension in other directions. The data is transferred back to the host after exiting the kernel. This is the standard procedure for CUDA programming.

5.4 Optimization Consideration

Optimization plays an important role in CUDA programming. The general approach for maximizing the performance of the GPU is to ensure efficient parallelism in the calculation and fast memory access pattern. It has been shown in the previous chapter that the global memory holds the largest size of memory storage on the device, but accessing this type of memory can result in poor performance due to memory latency. Some optimization techniques have been suggested (NVIDIA Corporation, 2010; Kirk & Hwu, 2010) to maximize the potential of the GPU. Some of these techniques require

performing experimental performance tuning. In general, optimization can consume much more time than writing the code. The programmer needs to be selective when considering these optimization techniques.

5.4.1 Memory Access Efficiency

Global memory is known as the slowest type of memory on the device. Thus, one should try to avoid using global memory whenever possible. However, this is the form of memory with maximum storage size, so for calculation which requires a large amount of data, global memory usage cannot be avoided. In the case where global memory access is required, it is desired to achieve the memory bandwidth close to the theoretical peak. In order to achieve this bandwidth, the memory access pattern needs to be coalesced which means that all the threads in a warp must access consecutive memory locations. It is, therefore, important to understand how the data array is mapped into the memory address space. Since global memory consists of a linear addressed memory space, multi-dimensional array is placed into the global memory following the conventional row-major order. For a two-dimensional array, all the elements of the array are placed into the linear memory such that the column index is the fastest varying index. This is illustrated in Figure 5.2 below.

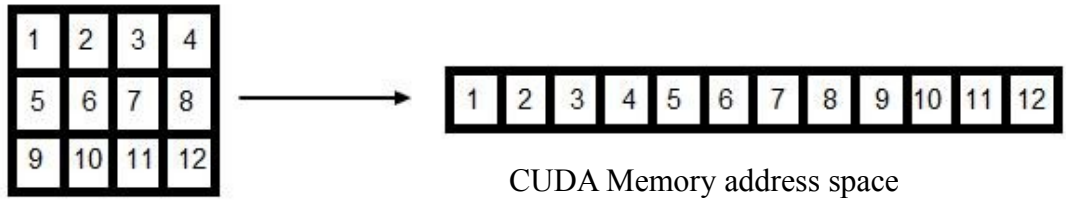


Figure 5.2: Storing multi-dimensional array into linear memory

In order to achieve a coalesced memory access pattern, it is desired to have the thread index associated with the column index and the block index associated with the row index. An example of both memory access patterns is given in Figure 5.3. While the left side of the figure shows a coalesced memory access pattern, the right side shows an uncoalesced access pattern. On the left side of Figure 5.3, since each block handles one row of the matrix, all the threads can access all the elements of that row which are contiguous in the memory.

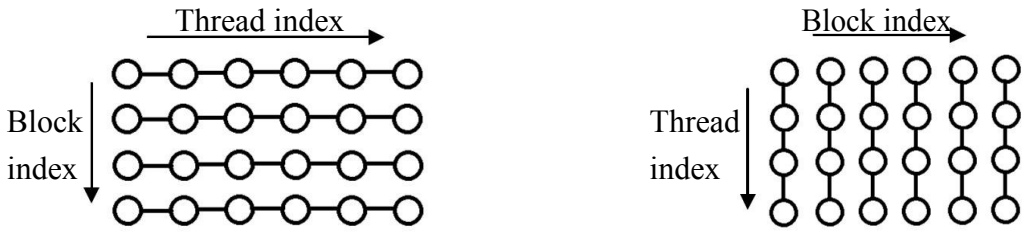


Figure 5.3: Coalesced (left) and uncoalesced (right) memory access pattern.

Memory coalescing allows the DRAM to supply data at high rate close to the theoretical bandwidth. However, this is not necessarily an easy task given that the data

for the calculation can possibly be at random location as in the case of a CFD solver for unstructured grid. In that case, the data of each element can be stored in any location since the grid connectivity is established separately.

In the case where the calculation does not require a sufficiently large amount of data, it is recommended to utilize shared memory in order to avoid global memory access. One effective strategy for using shared memory has been suggested by Kirk and Hwu (2010); the strategy had been tested on a matrix multiplication algorithm with outstanding performance gain. The main idea is to partition the data into tiles which can be fitted into shared memory (This is sometimes referred as memory padding technique). By loading data into shared memory, extra global memory access is eliminated. In addition, accessing data from shared memory is much faster than global memory (100-150 times) resulting in a more efficient parallelization of the calculation. Shared memory in CUDA can be declared inside the kernel as shown below:

__shared__ float A[10][20];

__shared__ double A[10];

One important step in using shared memory is that all the threads within the block need to be synchronized before starting the calculation. The synchronization ensures that all the global memory has been copied into shared memory. This can be done via the *__syncthreads()* call. This call serves as a barrier to make all threads within a block to wait until other threads has completed the same task.

The disadvantage of shared memory is its limitation in size which makes it not useful for computing large amount of data. For example, the problem of interest in this

thesis is a simulation of a multiple-species gas where each vector of conservative variable can be large depending on the reaction mechanism used in the simulation. For the simulation of an ionized gas, one also has to keep track of different excited levels of the ions which results in a very large size array. In addition, characterization of a gas/plasma in thermal non-equilibrium requires the use of multiple temperature models (2-T, 3-T, multi-T) which also increases the size of the vector of the conservative variables. The effects of having to compute a large set of data are the reduction in the tile size used for shared memory and excessive global memory access.

The other fast memory access that could be utilized to reduce global memory traffic is texture cache. Texture memory is a special form of memory designed for graphic rendering. The advantage of using texture memory is that the coalesced memory access can be bypassed since texture memory is cached on the device to achieve high memory bandwidth. Texture memory is extremely useful in the case where un-coalesced memory access cannot be avoided. In addition, accessing data from texture memory can possibly result in exceeding the theoretical bandwidth of the global memory.

5.4.2 Thread Execution

Another important aspect of optimizing CUDA code is based on the thread execution model. Once a kernel is launched, each block will be assigned to 2 SMs which contain a number of SPs. All the threads within the block will be organized into warps and all the SPs are automatically scheduled to perform the calculation. Since the scheduler is designed to maximize the performance of the kernel, each thread in a block

can execute in any order. Thread synchronization is required for the case of transferring data from global memory to shared memory. More importantly, one needs to avoid having all the threads within a warp to execute different instructions. This will cause the issue of thread divergence and those instructions will be executed in a serial manner. One should avoid using an *if* statement based on the thread index unless the condition of that statement still allows all the threads in the warp to follow the same path.

Since memory access is very time consuming, all the threads within a block should be kept busy at all time in order to make up for the memory latency. In order to achieve this goal, all the blocks need to be sized appropriately to maximize the occupancy which is defined as the ratio of the number of the active warps per SM with the actual number of warps. For example, if all the warps of a block are active at all time, the block is determined to have an occupancy factor of 1. The estimated values of the grid and block size can be determined from the CUDA occupancy calculator (NVIDIA Corporation, 2007) provided by NVIDIA. In general, the size of a block should be multiple of the warp size, so all the available SPs can be utilized. Experimental performance tuning can be useful in determining the optimal value of the block size. However, it has been shown by Volkov (2010) that small block can also lead to high performance. This issue will be illustrated further in chapter 7 of this report as part of the optimization study done on the fluid solver.

5.5 Object-O riented Programming

The fluid solver is designed to utilize the concept of Object-Oriented (OO) programming. OO design provides a flexible way of writing scientific codes that can be easily debugged and maintained. Earlier attempts in writing CFD solvers in an OO framework (Kapper, 2009; Cambier, Carroll, & Kapper, 2004) had shown improvement both in terms of flexibility and extensibility. The OO framework implemented here is coupled with the use of the GPU for the purpose of flexibility and robustness. Although CUDA (version 3.0 and above) has begun to supported several OO features (in C++) such as templates, classes and inheritance, the level of maturity of the compiler is still questionable. However, OO design still gives more flexibility in writing code and allows more complex software architecture as compared to a procedural-based framework.

The fluid solver holds a very basic architecture and is ready to be expanded to incorporate more complex features. All the data of the fluid solver are grouped into objects which can be transferred to the device for processing. For example, all the geometric data such as the cell, node and face are contained within a class called mesh. This provides a quick and easy way of accessing the data inside a kernel without having to pass each individual pointer. Making use of class in packing data does not necessarily affect the memory access pattern since the class only holds a pointer to the data. The actual data can still be allocated in a linear fashion to coalesce the memory access pattern. In addition, some solver modules can also be packed inside a class as a method. This is very convenient due to the fact that the same method can be used both on the host and the

device. This also allows a quicker way of comparing the performance between the CPU and GPU since both are calling the same version of the function.

5.6 Visualization Capability

One of the most unique features of CUDA is the graphic interoperability which can ultimately lead to the so-called "real-time" visualization. Since most of the computation is performed on the graphic device, it is not necessary to transfer data back and forth between host and device for visualization. CUDA provides a unique feature to allow programmer to directly access the graphic resource on the device both in pixel or vertex format.

CUDA supports visualization in OpenGL and Direct3D. The visualization attempted in this work is done via OpenGL (Shreiner, Woo, Neider, & Davis, 2008). The overall process of accessing and manipulating graphic data (either pixel or vertex buffer) is highlighted in Figure 5.4.

```
// Register buffer

FXGD*UDSKLFV*/5HJLVWHU%XIIHU
«
```

// Mapbuffer

FXGD*UDSKLFV0DS5HVRXUFHV
«

// run kernel to edit buffer

0DSBWH[WXUHBNHUQHO«!!!
«

-

// Unmapbuffer

FXGD*UDSKLFV8QPDS5HVRXUFHV
«

Figure 5.4: Graphic interoperability in CUDA programming.

The *cudaGraphicsMapResources* call retrieves the pointer to the graphic resource which can be edited by a kernel. Upon completion of the kernel, the buffer needs to be unmapped. The data mapped to the graphic resource gets updated quickly allowing a fast and efficient renderization pipeline. It must be noted that since the data transfer between host and device is reduced, the run time is also kept at the minimum.

CHAPTER 6: BENCHMARK

6.1 Introduction

This chapter presents a series of test cases for the solver including non-reactive and reactive flows both in one and two-dimension. The solver utilizes all the numerical schemes mentioned in Chapter 3. The variety of solutions is presented here to demonstrate the capability of the solver and to determine which scheme is the most efficient and capable in terms of being able to reproduce the correct flow features.

6.2 Non-reactive Flows

6.2.1 One-Dimensional Flows

6.2.1.1 Sod Shock Tube Problem

One of the most basic test problems for CFD benchmarking is the Sod shock tube problem. The problem is described as a standard Riemann problem with the initial conditions given as

$$\begin{pmatrix} U_L \\ P_L \\ u_L \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 0.0 \end{pmatrix}, \qquad \begin{pmatrix} U_R \\ P_R \\ u_R \end{pmatrix} = \begin{pmatrix} 0.125 \\ 0.1 \\ 0.0 \end{pmatrix} \tag{6.1}$$

Figures 6.1-6.3 show the solution of the problem at $t = 0.2$. The contact discontinuity and the shock are well-resolved by all schemes. Both MP schemes seem to perform better in terms of resolving the discontinuity of the flow. It must be noted that the solution of the WENO scheme matches almost exactly point-by-point with the solution of the ADERWENO scheme. This is expected since both schemes utilize the

same reconstruction procedure and are at the same order of accuracy both in space and time.

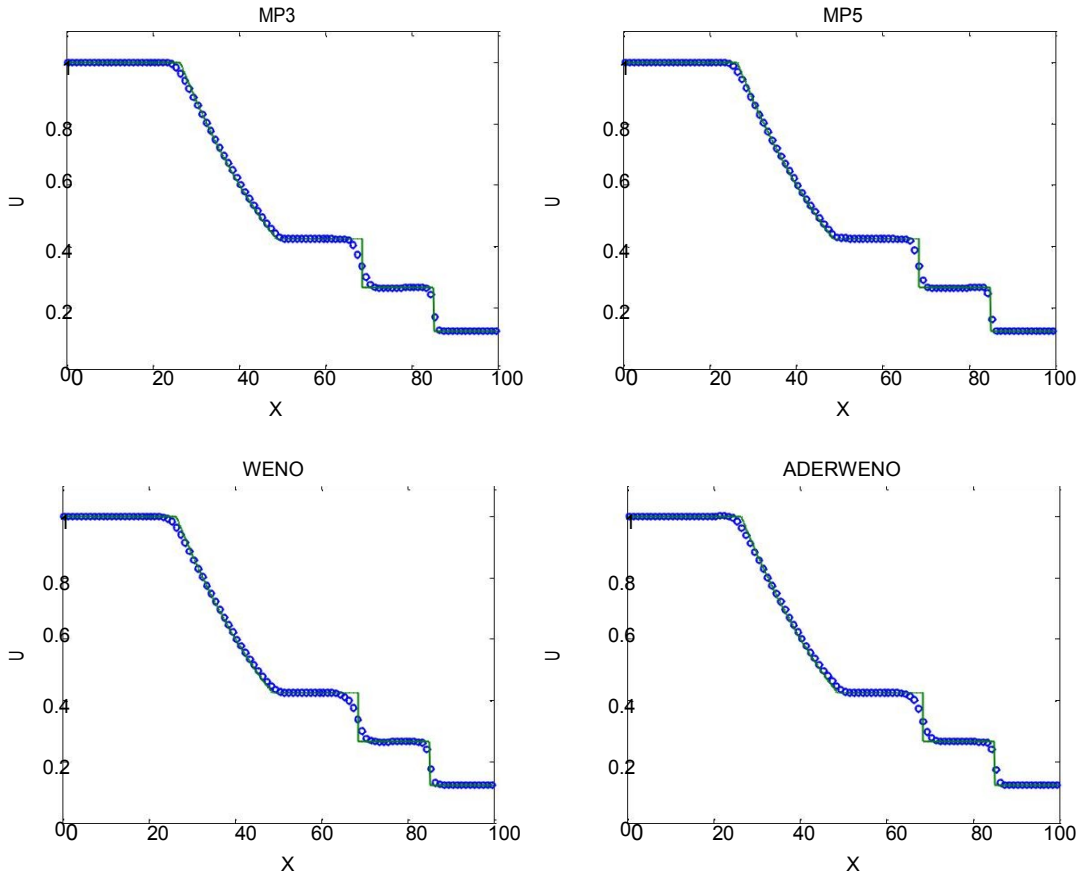


Figure 6.1: Density plot of the Sod shock tube problem with 100 points.

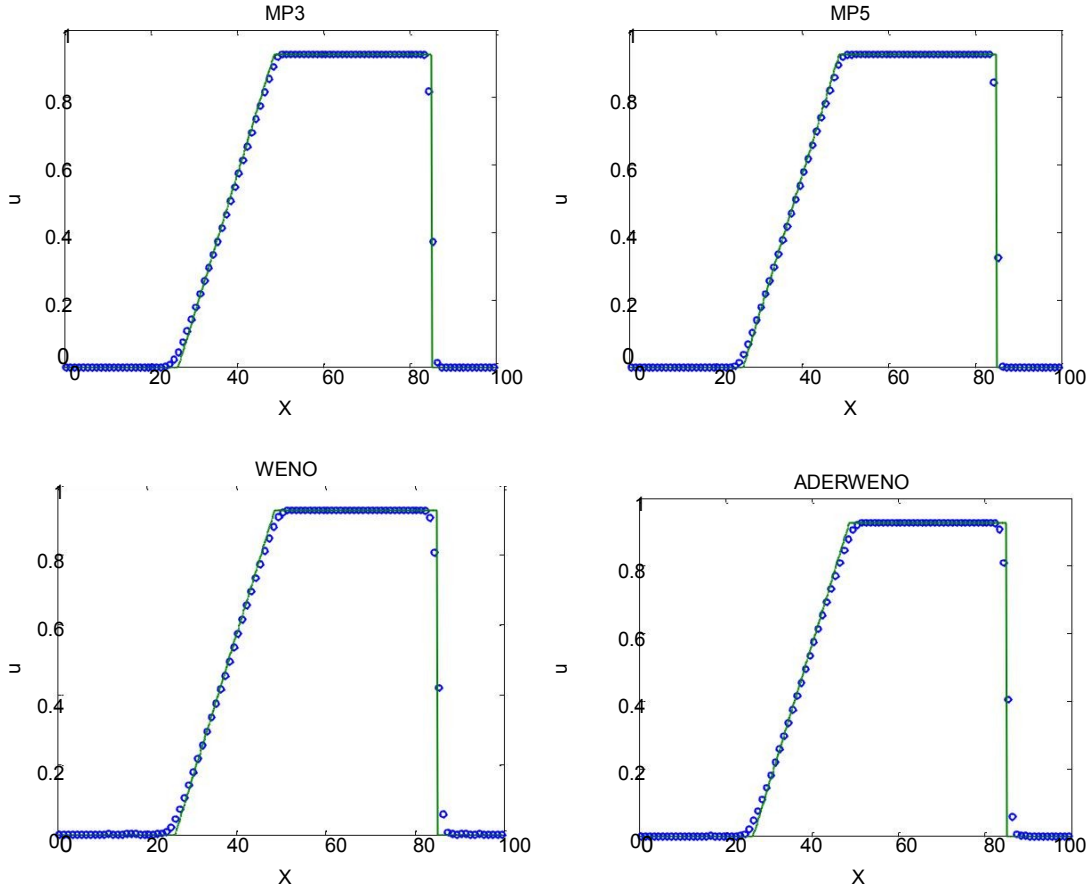


Figure 6.2: Velocity plot of the Sod shock tube problem with 100 points.

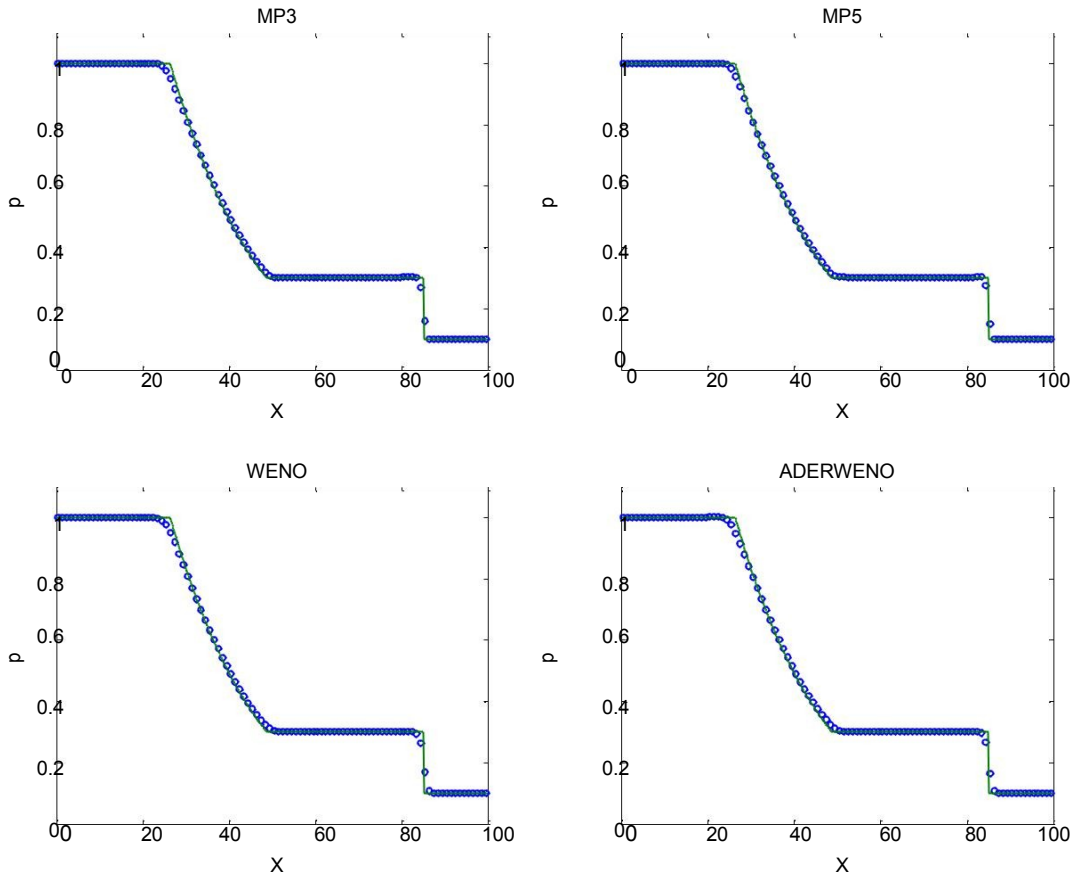


Figure 6.3: Pressure plot of the Sod shock tube problem with 100 points.

6.2.1.2 Lax Problem

The next problem in the 1-D test cases is the Lax problem. The Lax problem is initiated similar to the Sod problem. The difference between the two problems is that the density of the right state is now slightly higher than the left state, and the left state is initiated with some positive velocity. The initial conditions for the Lax problem are given as bellow.

$$\begin{matrix} \begin{pmatrix} U \\ P \\ u \end{pmatrix}_L = \begin{pmatrix} 0.445 \\ 3.528 \\ 0.698 \end{pmatrix}, & \begin{pmatrix} U \\ P \\ u \end{pmatrix}_R = \begin{pmatrix} 0.5 \\ 0.571 \\ 0.0 \end{pmatrix} \end{matrix} \tag{6.2}$$

The numerical solution of the Lax problem is shown in Figures 6.4-6.6. The MP5 scheme is outperforming the others with excellent capability in resolving the contact discontinuity. This is clearly shown in the density plot.

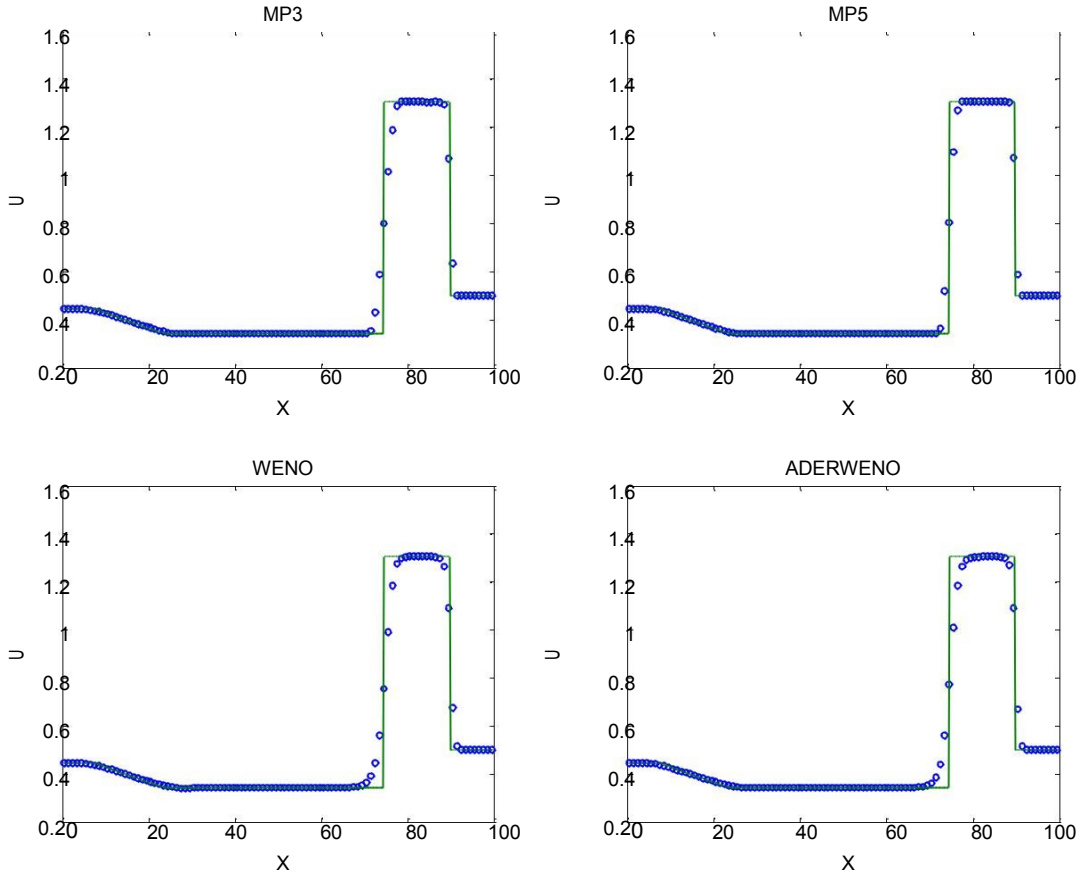


Figure 6.4: Density plot of the Lax problem with 100 points.

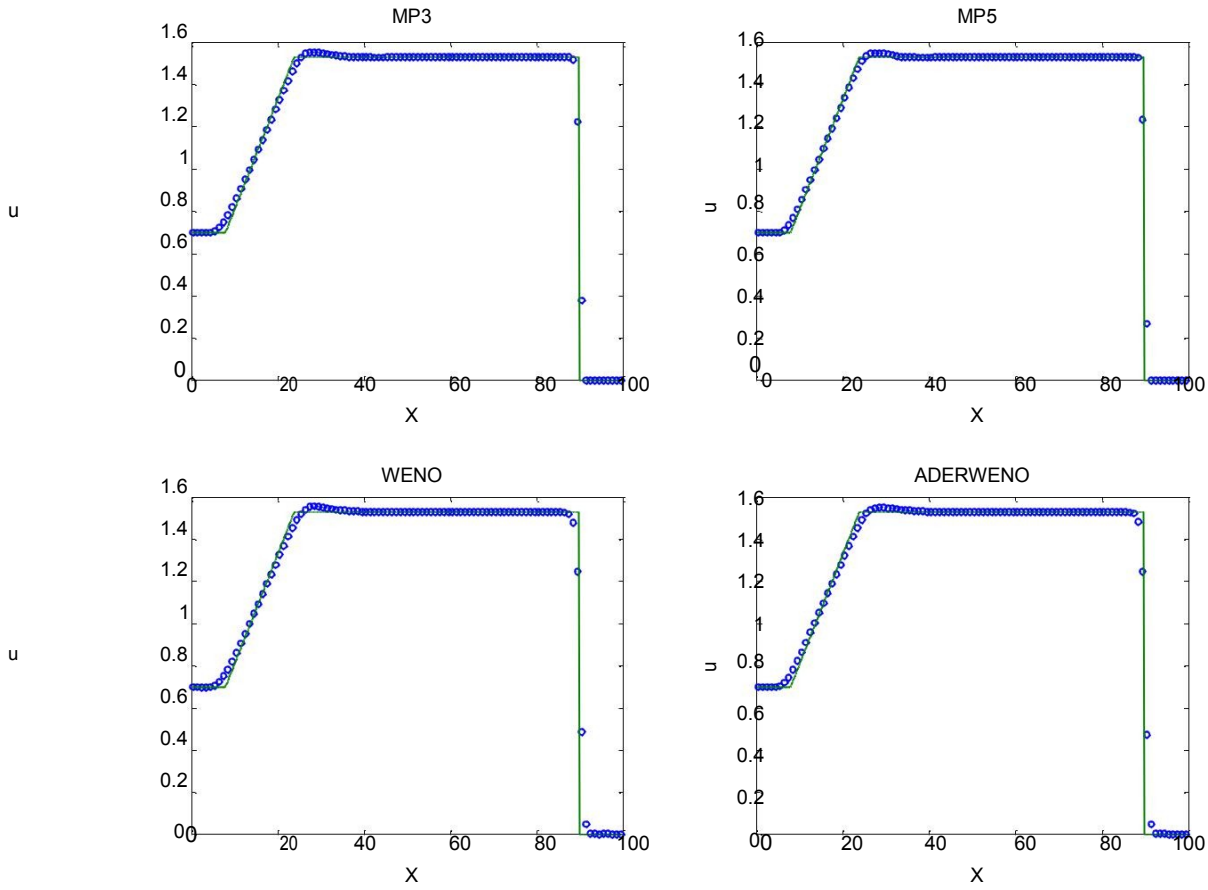


Figure 6.5: Velocity plot of the Lax problem with 100 points.

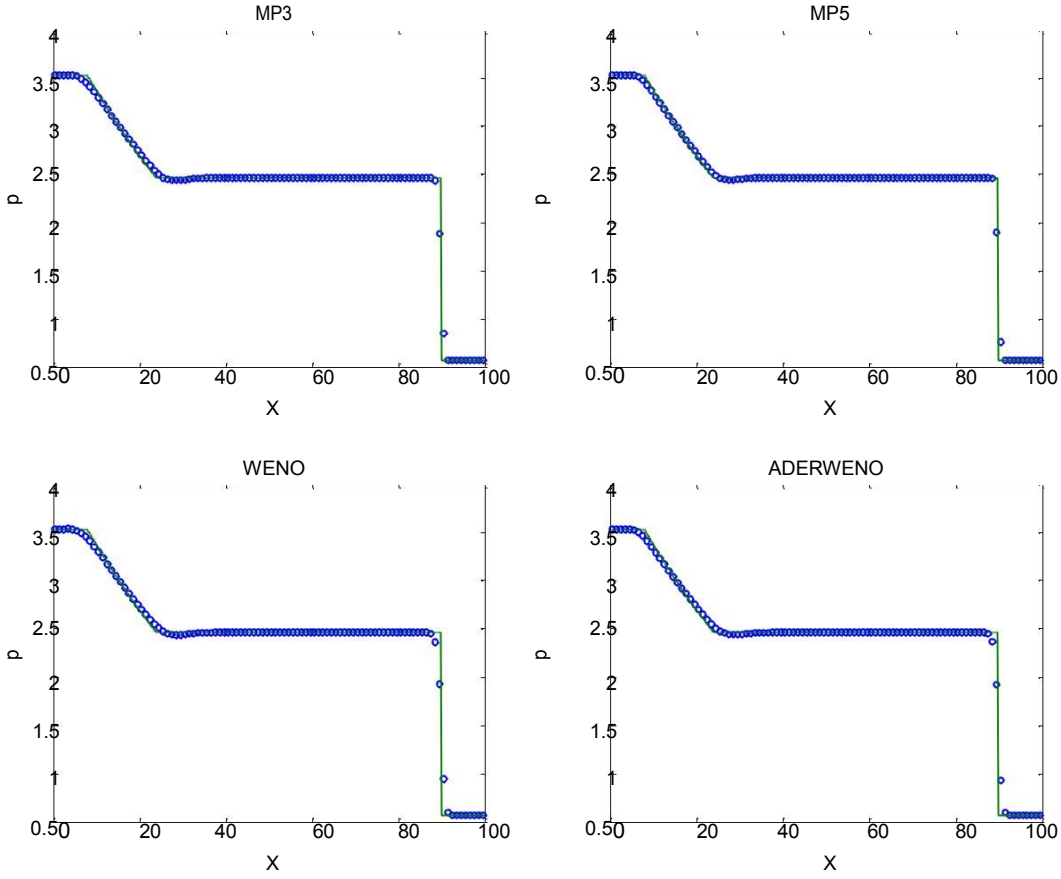


Figure 6.6: Pressure plot of the Lax problem with 100 points.

6.2.1.3 Shu-Osher Problem

The Shu-Osher problem is presented here to model the interaction of a moving shock wave with an entropy disturbance given in a sinusoidal form. Since there is no exact solution to this problem, the reference solution is computed using the MP5 scheme with 1600 points. The initial conditions are given as follows.

$$\begin{pmatrix} U_L \\ P_L \\ u_L \end{pmatrix} = \begin{pmatrix} 3.857 \\ 10.333 \\ 2.629 \end{pmatrix}, \qquad \begin{pmatrix} U_R \\ P_R \\ u_R \end{pmatrix} = \begin{pmatrix} 1 + 0.2\sin(5x) \\ 1 \\ 0 \end{pmatrix} \tag{6.3}$$

The results of the Shu-Osher problem utilizing all four schemes are shown in Figures 6.7-6.9 using 300 points. The MP5 scheme results in the best solution with all the local minimum and maximum well-resolved.

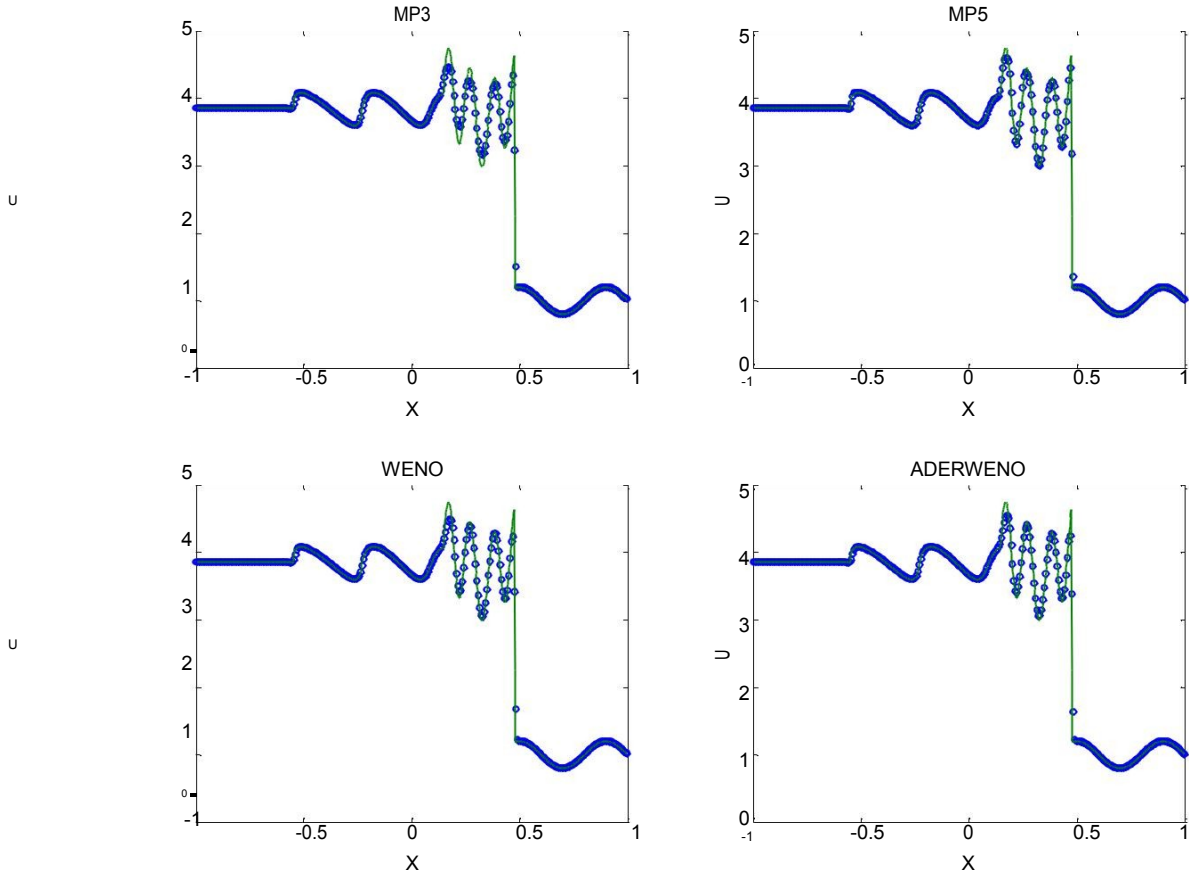


Figure 6.7: Density plot of the Shu-Osher problem with 300 points.

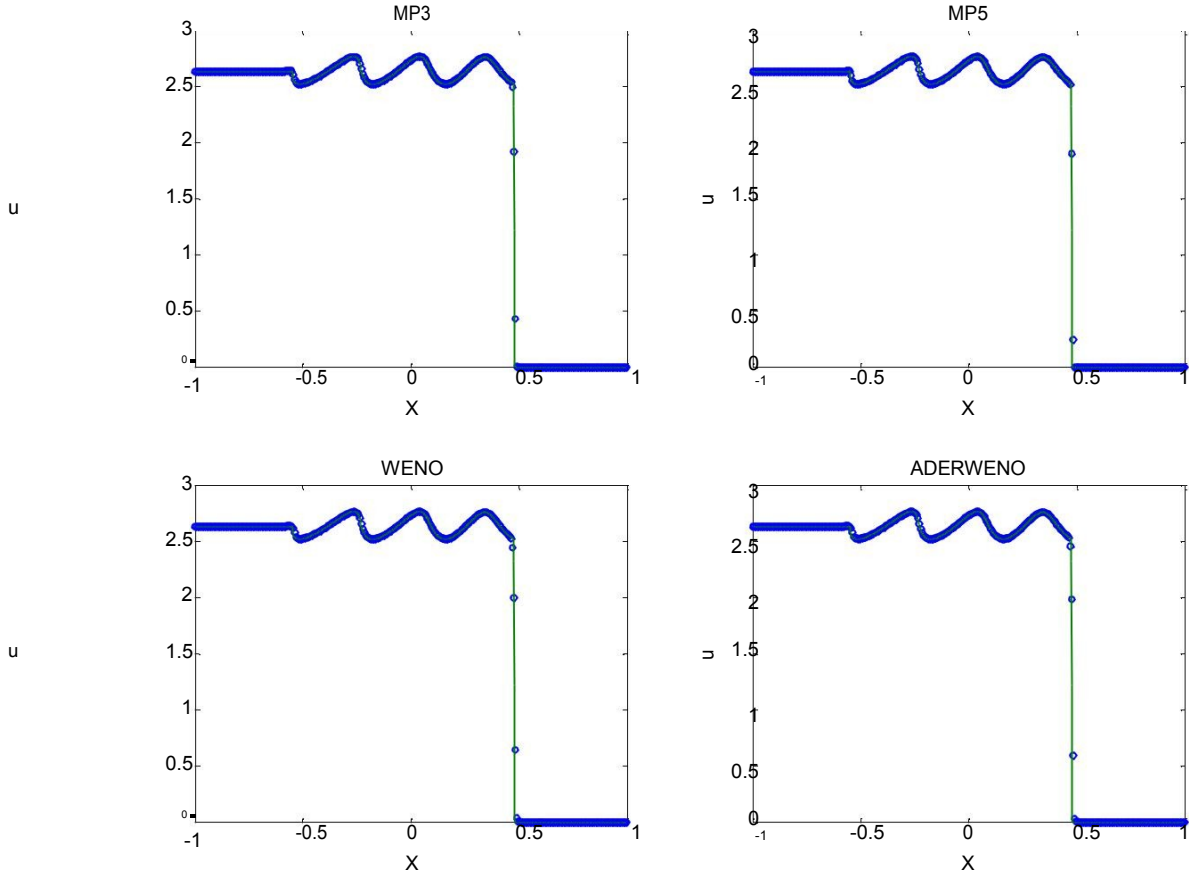


Figure 6.8: Velocity plot of the Shu-Osher problem with 300 points.

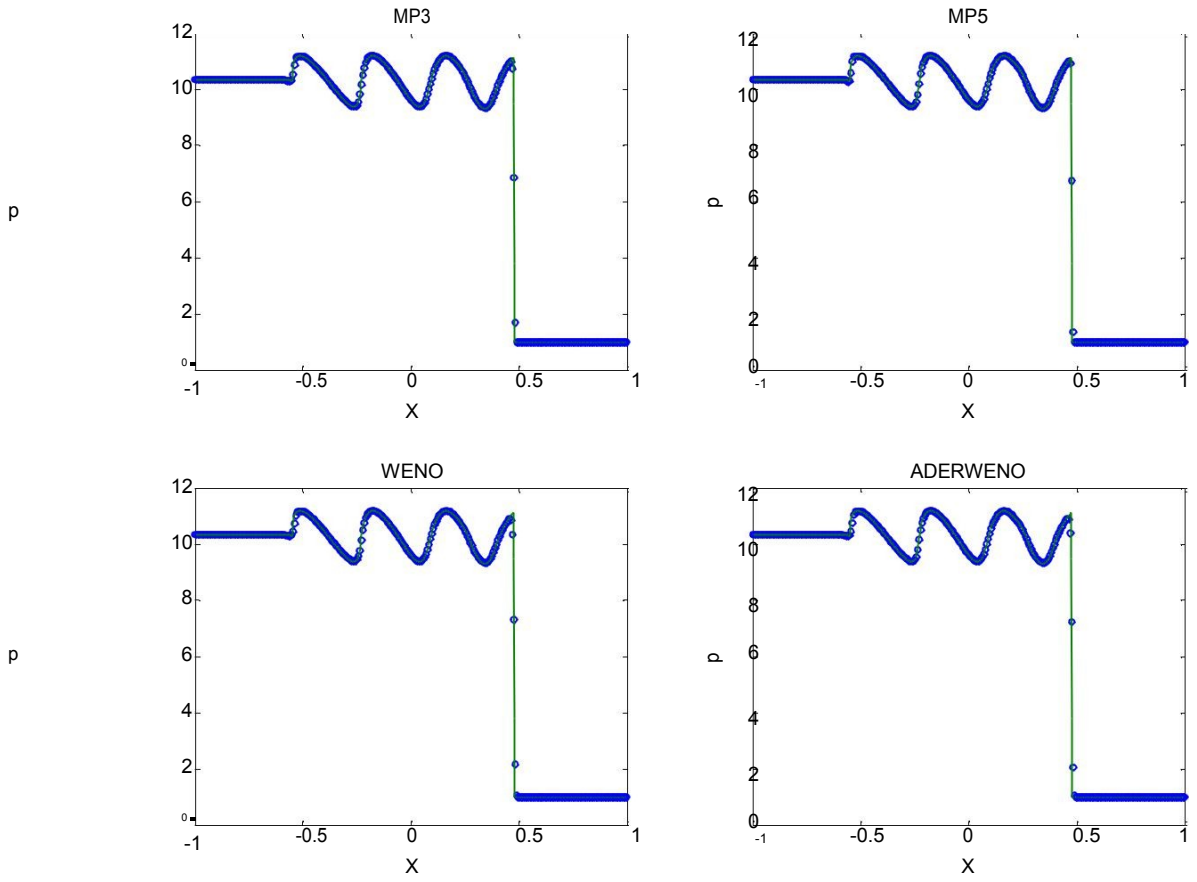


Figure 6.9: Pressure plot of the Shu-Osher problem with 300 points.

6.2.1.4 Blast Waves Problem

This problem represents two blast waves interacting with each other. The problem had been used by Woodward and Colella (1984) to conduct a comprehensive test of various numerical schemes in modeling strong shock interaction. The problem is initiated by having two blast waves traveling towards each other. Reflective wall boundary conditions are enforced at both ends to expedite multiple interactions of the waves. The interaction results in a complex flow structure. Several contact discontinuities are present and well-captured in the simulation. Since there is no exact

solution for the blast waves problem, the reference solution is computed using the MP5 scheme with 10,000 points. The initial conditions of the problem are given below. The domain is separated by three different regions in which the flow variables are specified.

$$\begin{pmatrix} \rho_L \\ u_L \\ P_L \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 10^3 \end{pmatrix}, \quad \begin{pmatrix} \rho_M \\ u_M \\ P_M \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 10^2 \end{pmatrix}, \quad \begin{pmatrix} \rho_R \\ u_R \\ P_R \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 10^2 \end{pmatrix} \tag{6.4}$$

Figures 6.10-6.12 show the numerical solution of the blast waves problem with 600 points. All schemes are performing well in terms of resolving the contact discontinuity. However, the contact discontinuity computed from the MP3 scheme is not as sharp as the others. This is due to the fact that the MP3 scheme is only third order. It has been shown again that both the WENO and ADERWENO schemes provide identical results. It must be noted that the small pressure region in the middle region may yield negative values for pressure and density during the reconstruction. In order to remedy this, we have utilized a flattening algorithm similar to the one given by Balsara et al. (2009). The flattening algorithm is only effective in the vicinity of a strong shock or rarefaction.

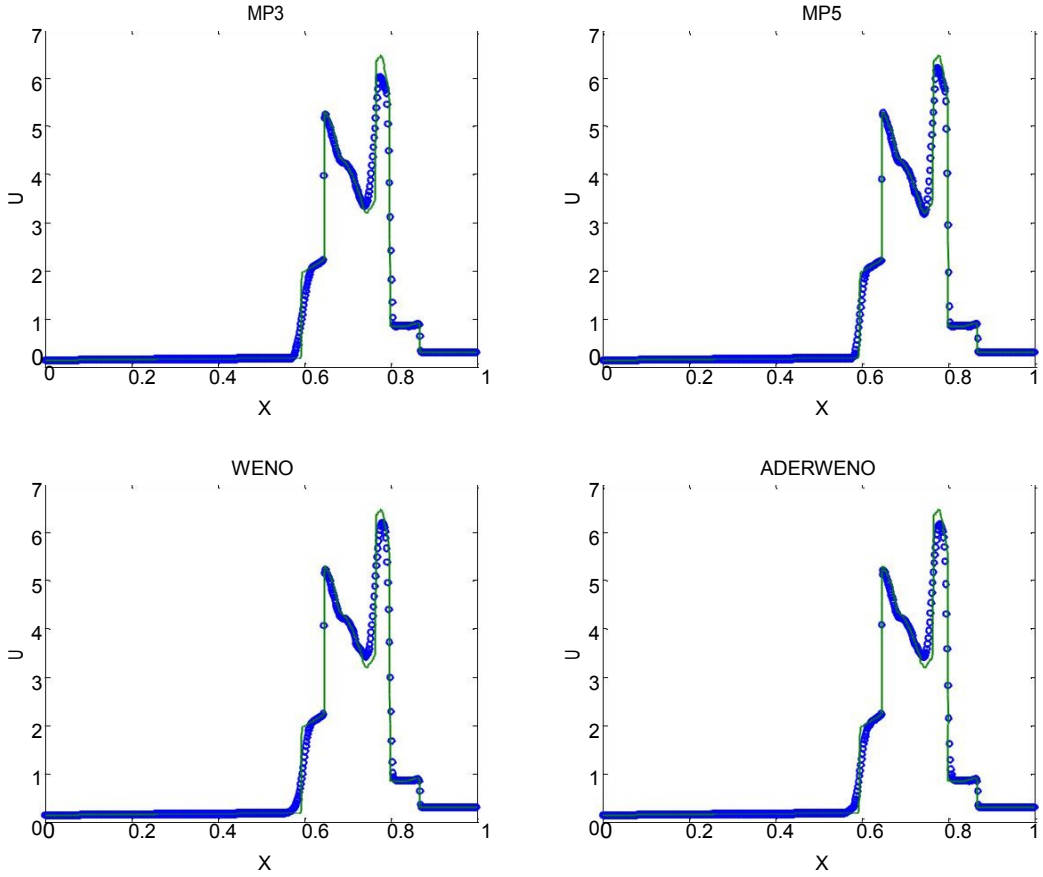


Figure 6.10: Density plot of the blast waves problem with 600 points.

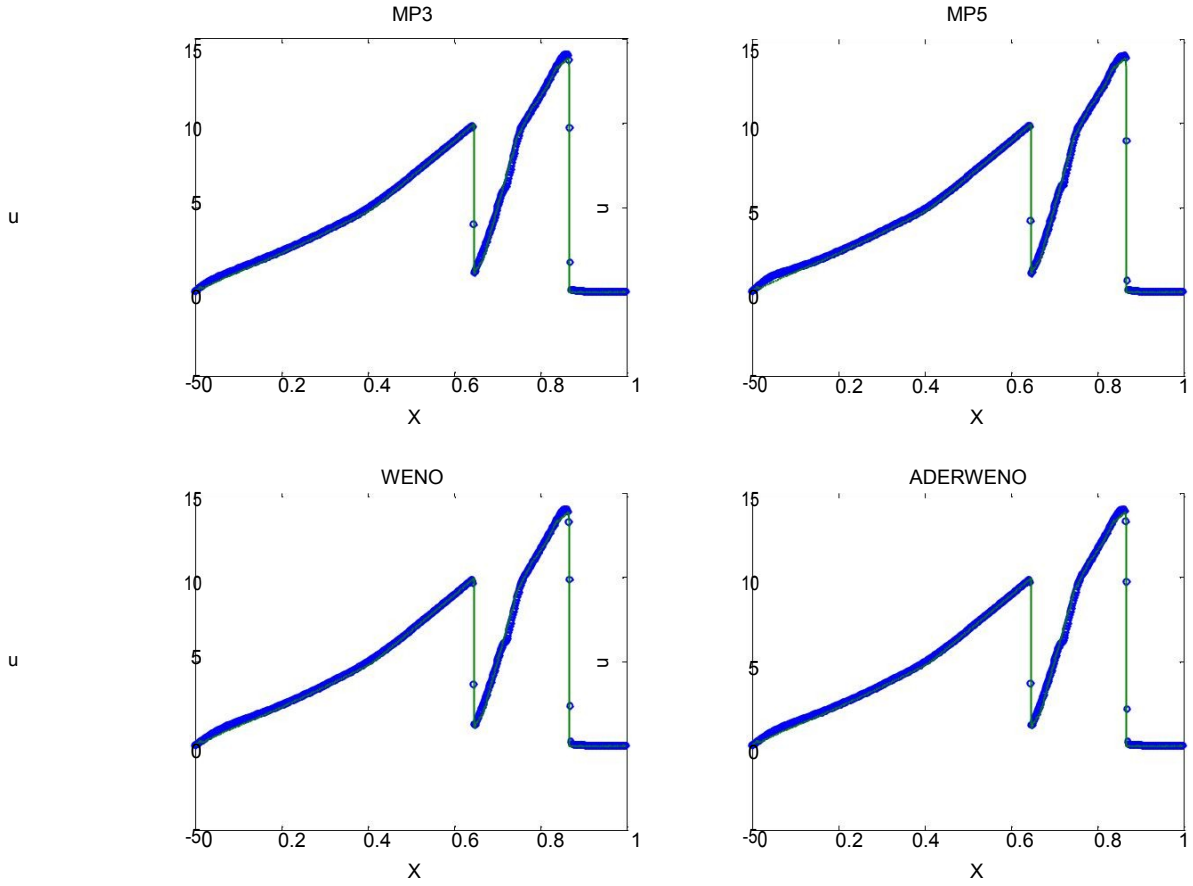


Figure 6.11: Velocity plot of the blast waves problem with 600 points.

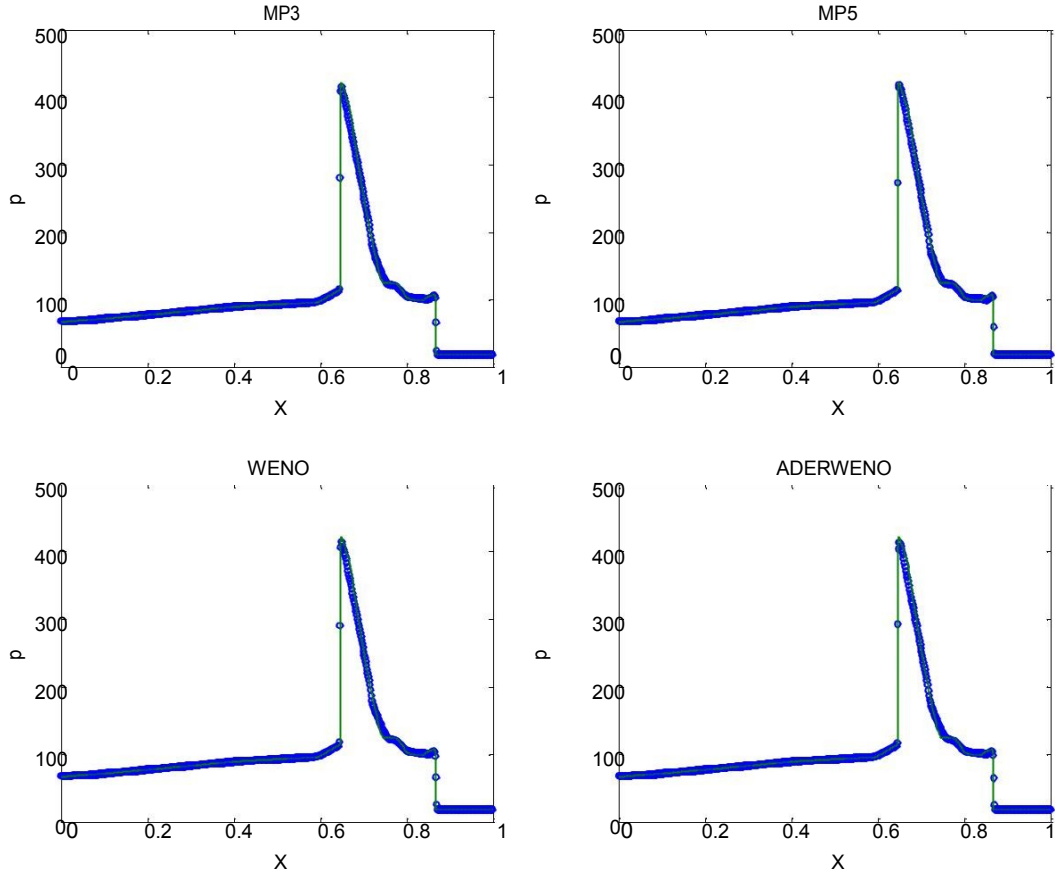


Figure 6.12: Pressure plot of the blast waves problem with 600 points.

6.2.1.5 Einfeldt¶V Problem

7KH(LQIHOGW¶VSUREOHP, introduced by Einfeldt et al. (Einfeldt, Munz, Roe, & B, 1991), is used to test the capability of the solver to model low density flow. The initial conditions for this problem are given as

$$\begin{pmatrix} \rho_L \\ P_L \\ u_L \end{pmatrix} = \begin{pmatrix} 1 \\ 0.4 \\ -2 \end{pmatrix}, \qquad \begin{pmatrix} \rho_R \\ P_R \\ u_R \end{pmatrix} = \begin{pmatrix} 1 \\ 0.4 \\ 2 \end{pmatrix} \tag{6.5}$$

The numerical solution RIWKH(LQIHOGW¶VSUREOHPis shown in Figures 6.13-15. The solution to this problem makes use of the HLLE flux instead of the standard Roe Flux-Difference Splitting (FDS). The Roe FDS requires an artificial entropy fix to ensure that there is no non-physical solution of the density and energy during the computation.

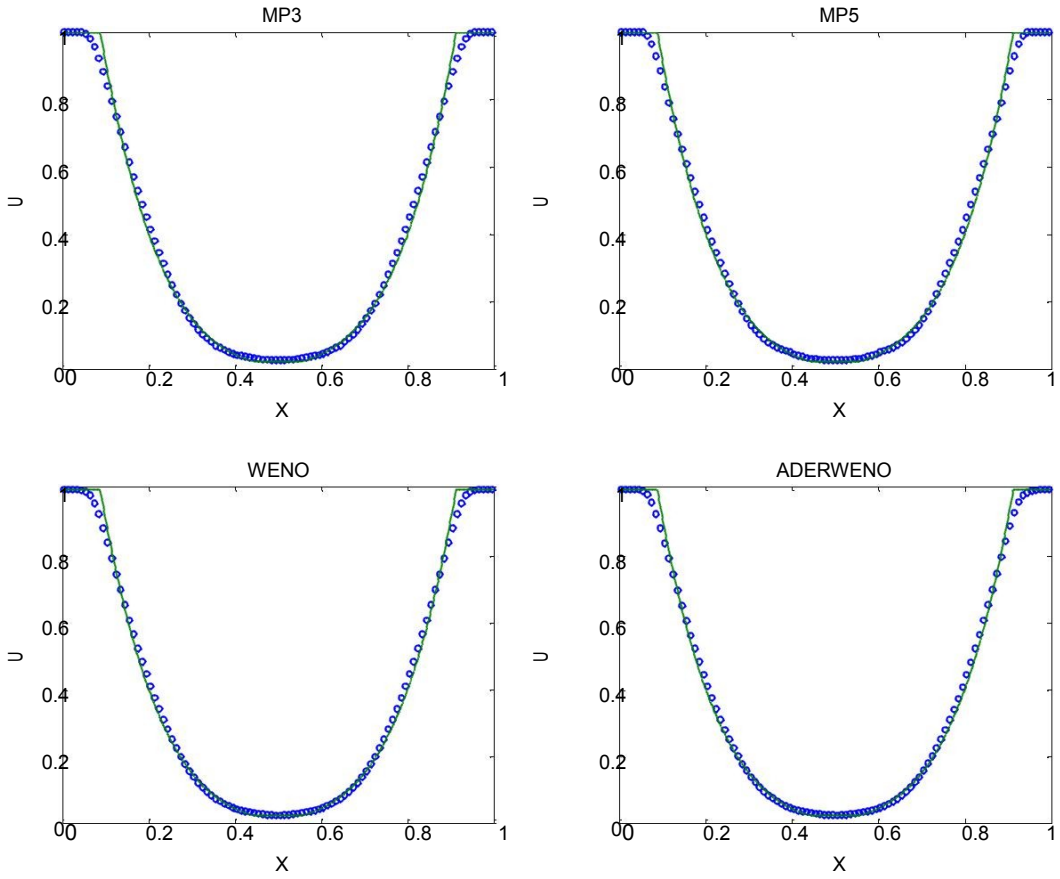


Figure 6.13: 'HQVLW\SORWRIWKH(LQIHOGW¶VSUREOHPZLWK SRLQWV

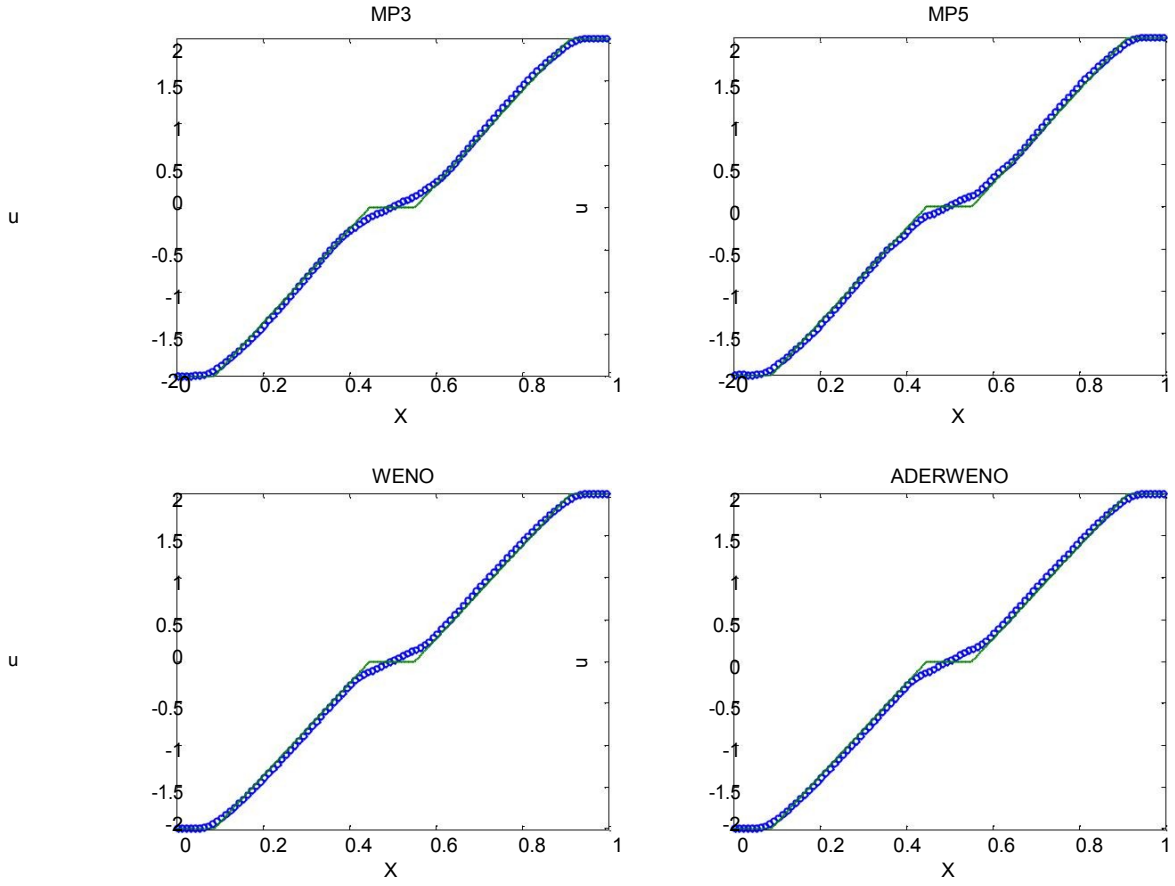


Figure 6.149HORFLW\SORWRIWKH(LQIHOGW¶VSUEOHPZLWK SRLQWV

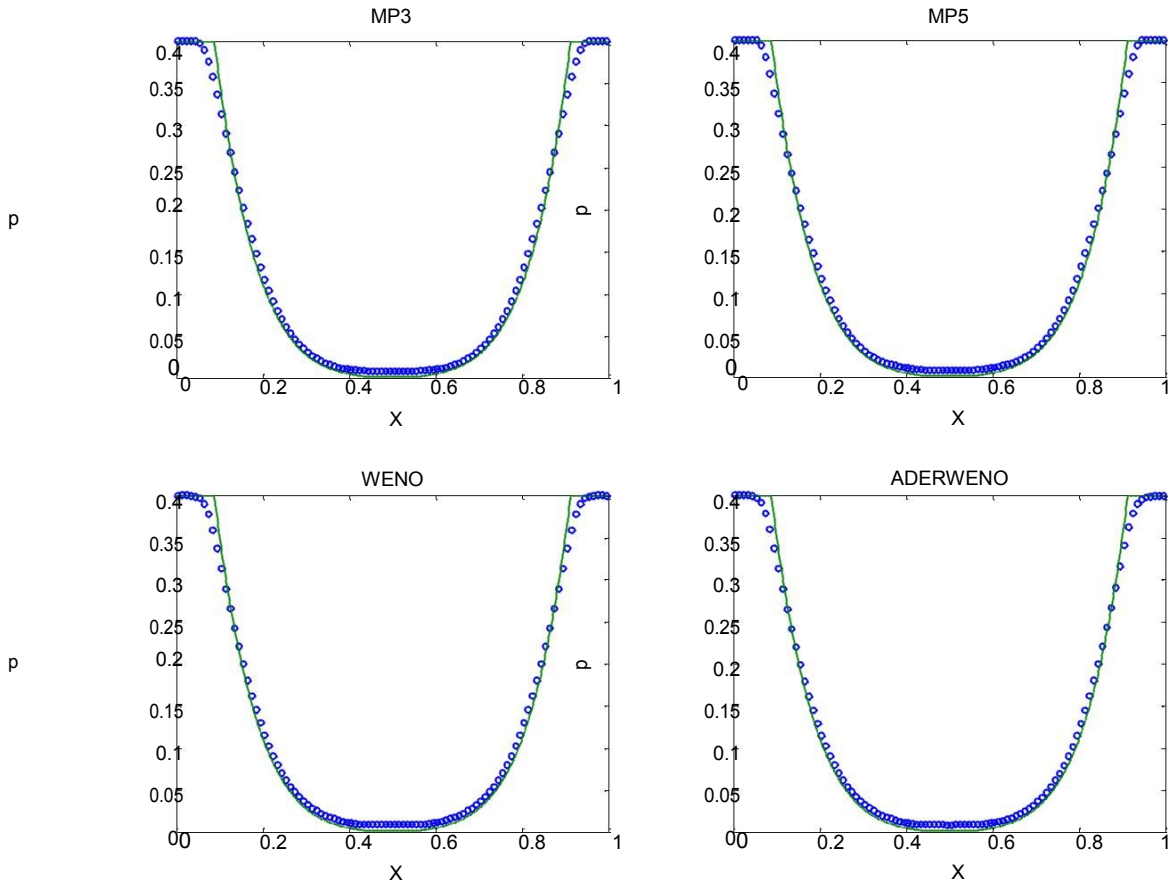
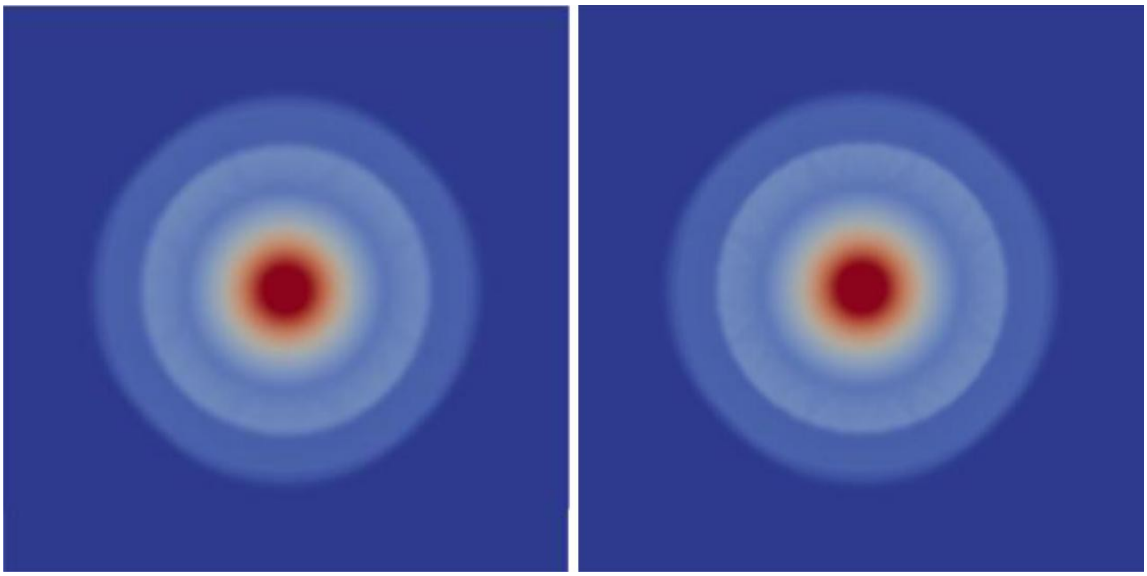


Figure 6.15: 3UHVVXUHSORWRIWKH(LQIHOGW¶VSUREOHPZLWK SRLQWV

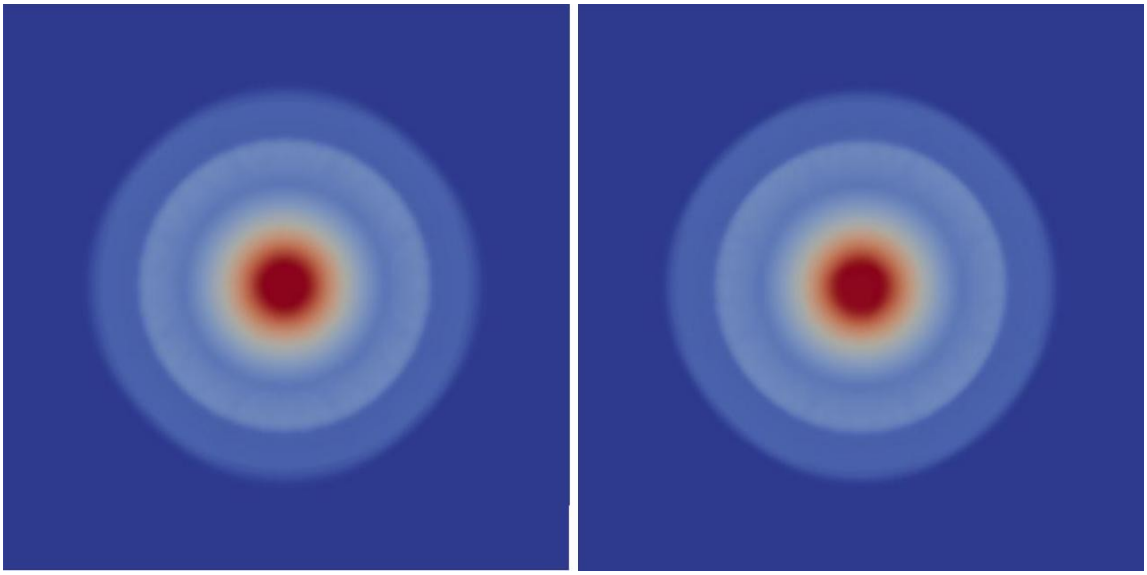
6.2.2 Two-Dimensional Flows

6.2.2.1 Two-Dimensional Sod Problem

The first two-dimensional test case is the extension of the one-dimensional Sod problem to two-dimensional. The problem can be described as an explosion initiated by a high pressure region in the middle. The initial conditions are the same as the one-dimensional test case. The shock is allowed to propagate radially away from the center. As shown in Figure 6.16, the symmetry of the solution is preserved although some gridding effect is evident in some schemes.



(a) MP3 (b) MP5

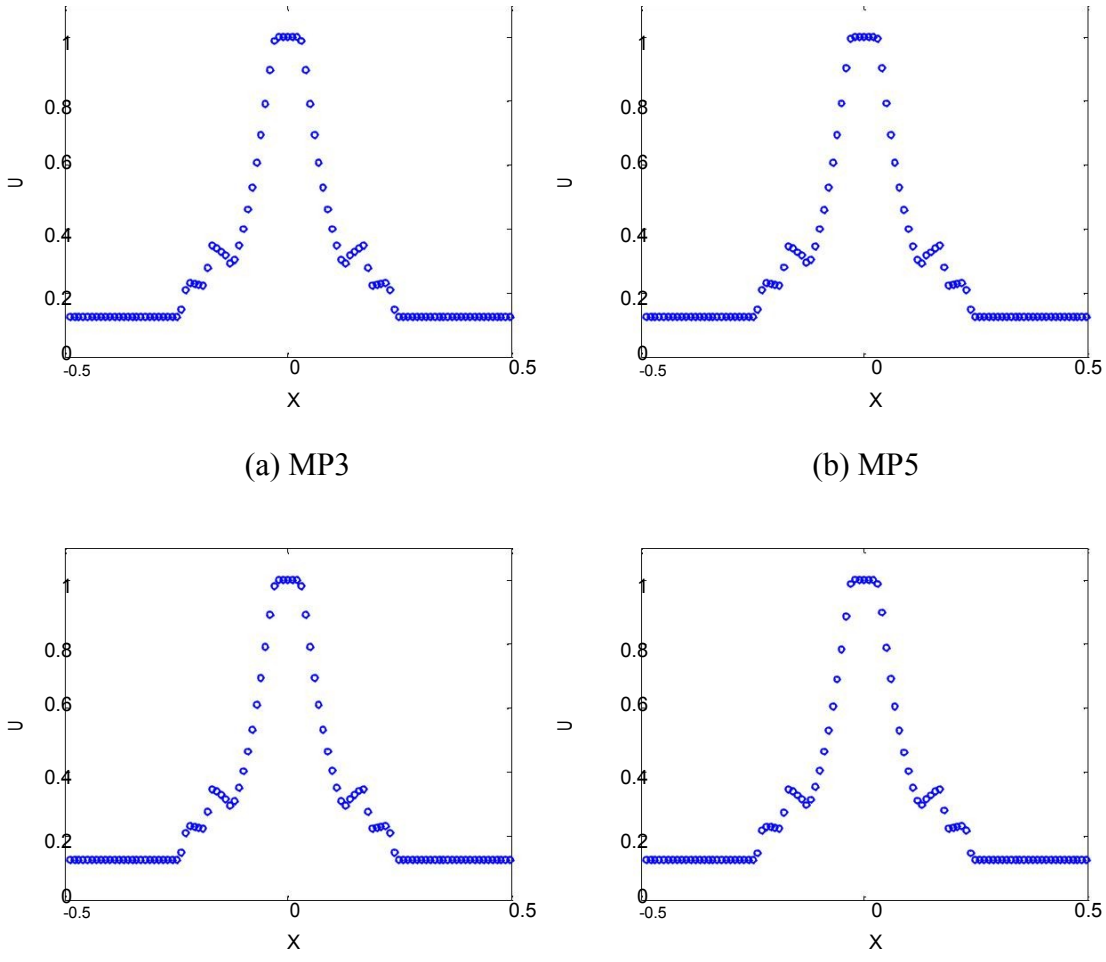


(c) WENO (d) ADERWENO

Color map:

Figure 6.16: Density for of the 2-D Sod problem computed on a 256 x 256 grid.

The density of the center line is extracted and plotted in Figure 6.17. The contact discontinuity and the shock are well-resolved, and there is no significant difference between all the schemes.



(a) MP3 (b) MP5

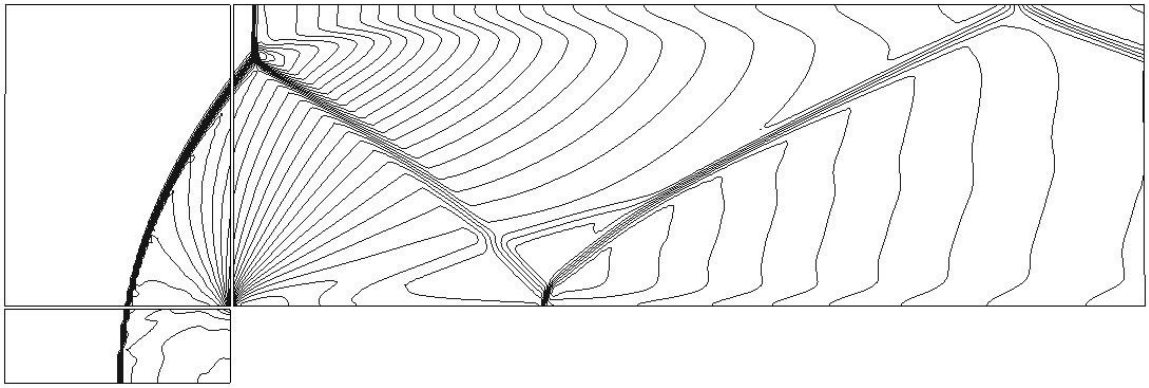
(c) WENO (d) ADERWENO

Figure 6.17: Density plot of the 2-D Sod problem computed on a 256 x 256 grid.

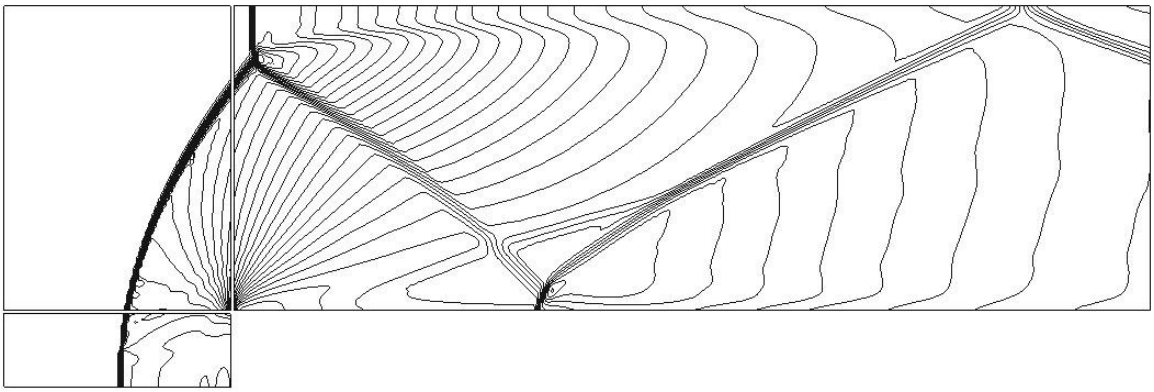
6.2.2.2 Mach-3 Wind Tunnel Problem

The next test problem is the Mach 3 wind tunnel with a step also known as the Emery problem. This problem had been utilized by Woodward and Colella (1984) to test a variety of numerical schemes. The whole domain is initialized with Mach-3 flow. Reflective boundary condition is enforced on the step. We also set the upper part of the domain to be reflective. The left and the right boundary conditions are set as in-flow and out-flow, respectively. Special attention is required at the corner of the step since this is a singular point of the flow. The numerical error generated in this region creates a numerical boundary layer which can affect the structure of the flow. Treatment to the problem was given by Woodward and Colella by assuming the flow near the corner is nearly steady. However, this fix was not used in this simulation since we want to test the robustness of the solver in the case of strong shock and how it handles the singularity.

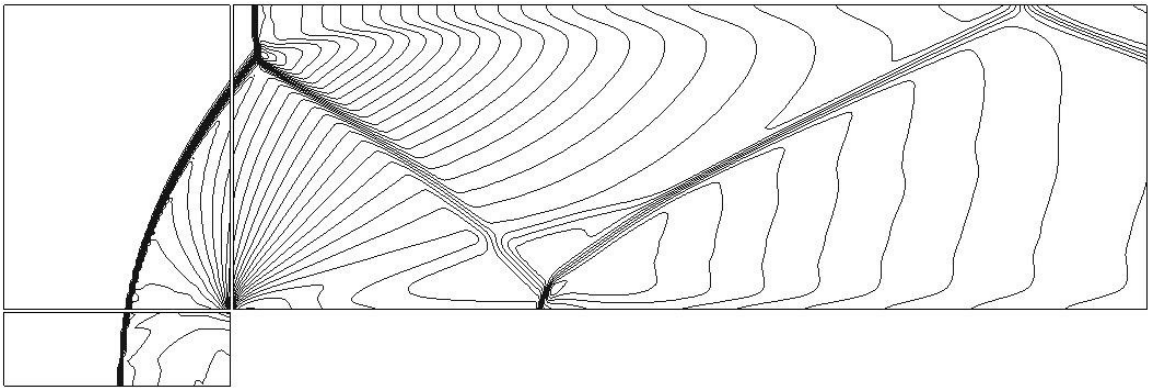
Results of the simulation are shown in Figure 6.18 using a 125 x 375 grid. The density contours obtained by four schemes are consistent with each other. The solution computed by ADERWENO scheme shows some oscillation in the contour because the ADER scheme used in this work is not TVD. It can be seen in all the contours that the boundary layer generated at the corner is a direct consequence of the Mach stem on the upper surface of the step. However, this problem does not affect the overall structure of the flow and can be eliminated simply by increasing the resolution of the grid.



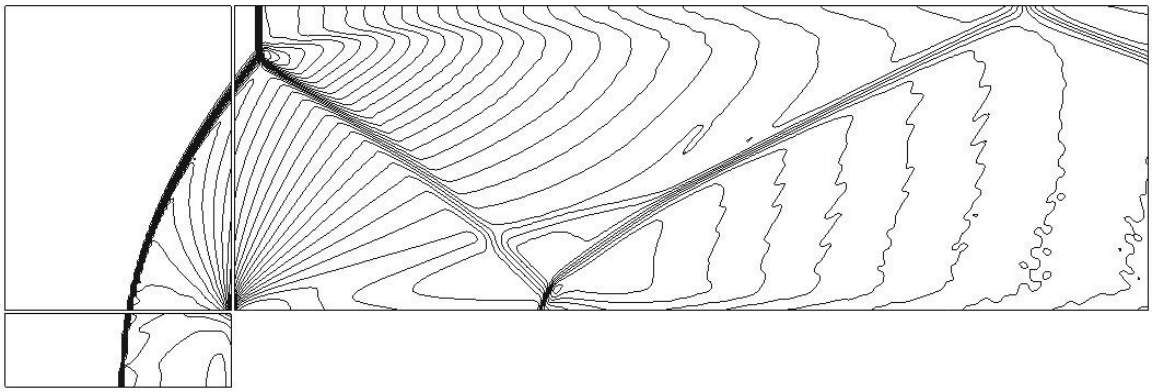
(a) MP3



(b) MP5



(c) WENO



(d) ADERWENO

Figure 6.18: Density contour of the Mach 3 wind tunnel problem

6.2.2.3 Shock Diffraction Down a Step

This test problem is described as the diffraction of a shock wave ($M = 2.4$) down a step (Van Dyke, 1982). The diffraction results in a strong rarefaction generated at the corner of the step which can cause problem of having negative density when performing the reconstruction. The problem is modeled using 27,000 cells with the MP5 scheme. The numerical simulation is shown in pair with the experimental images in Figure 6.19. The numerical solution is presented as numerical schlieren images which are ideal for comparison with experimental images. It has been shown that the solver was able to reproduce the correct flow features in the region of the rarefaction fan.

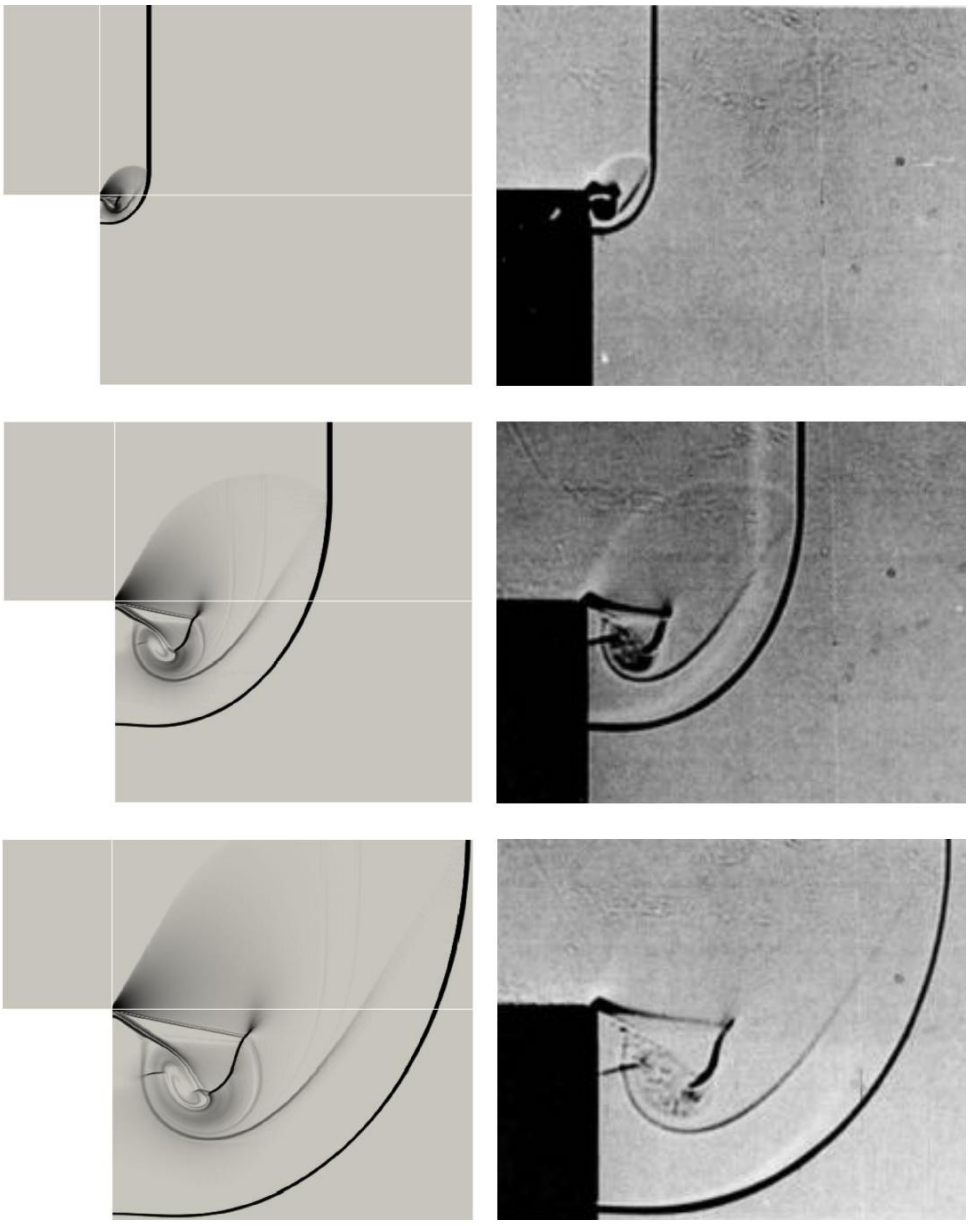
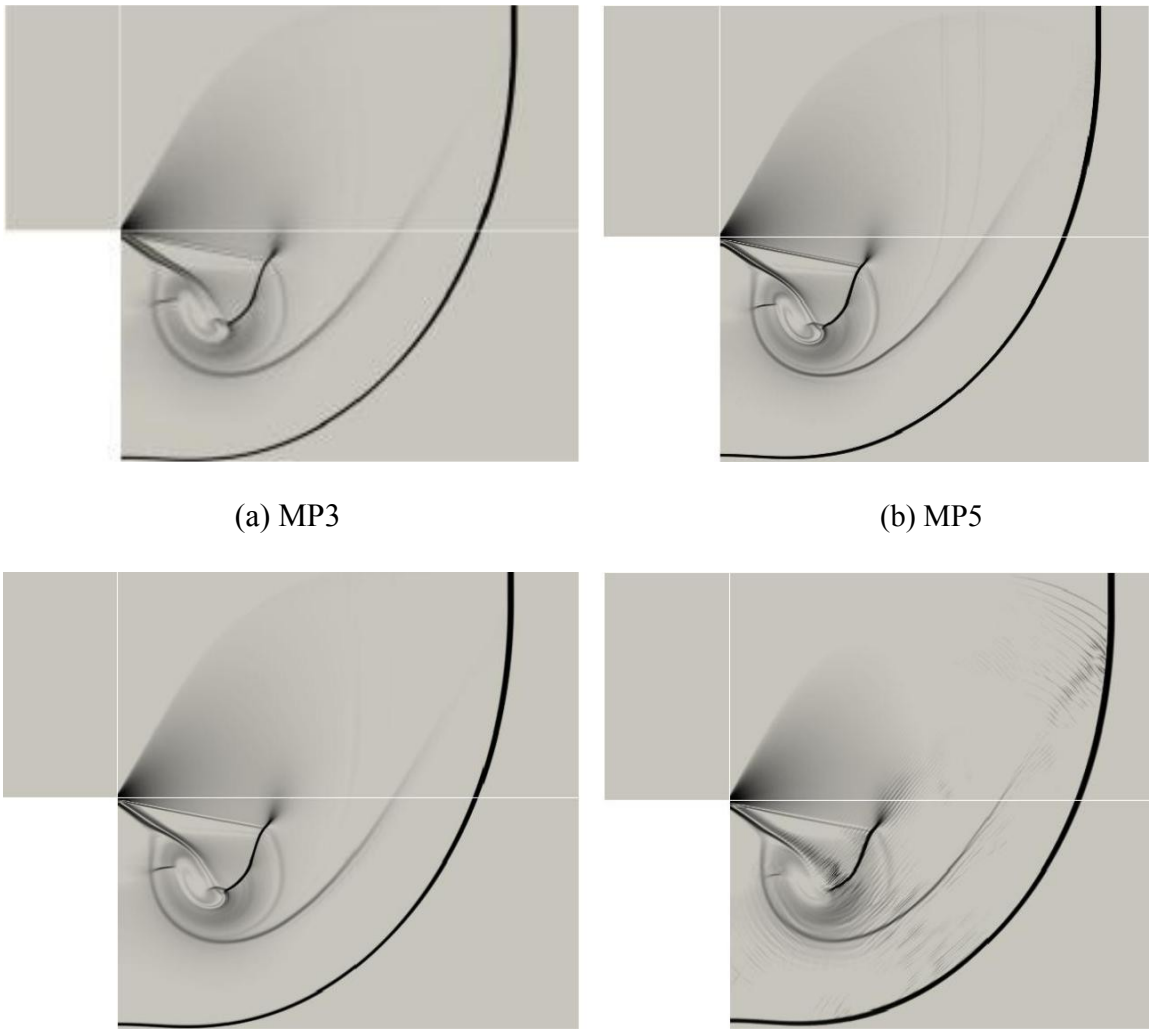


Figure 6.19: Diffraction of a Mach 2.4 shock wave down a step. Comparison between numerical schlieren and experimental images

Figure 6.20 demonstrates the numerical solutions obtained from all schemes. It can be shown that the contact discontinuity and several flow features are well-resolved by the MP5 scheme resulting in a sharp rarefaction fan at an angle of 60^0 from the corner of

the step. The solution computed using the ADERWENO scheme also shows instabilities in the solution near the region behind the shock which is likely due to the non-TVD behavior of the scheme.



(a) MP3 (b) MP5

(c) WENO (d) ADERWENO

Figure 6.20: Numerical solution of the shock diffraction problem.

6.3 Reactive Flows

All the test cases presented in the previous section indicate the superior performance of the MP5 scheme over the other schemes. However, when using the MP5 scheme, RK time integration is required to obtain a TVD solution. The performance difference between the RK time integration versus ADER method is discussed in chapter 7. From now on, all the simulations are performed using the MP5 scheme. The flow solver now is coupled with the kinetics solver in order to model reactive flow.

6.3.1 One-dimensional Detonation Wave

The first test case for simulation of reactive flows is the 1-D simulation of a detonation wave. The detonation is started by applying a spark ignition which can be described as a high pressure and high temperature region spanning several cells from the left side of the domain. As shown by He (2004), the temperature, pressure and input energy of the spark must be sufficient in order to initiate the detonation. The parameter of the spark is taken from He (2004), and the mixture used in this simulation is composed of nine species: H_2, O_2, H, O, OH, HO_2, H_2O_2, H_2O and N_2 with thirty-eight elementary reactions. The reaction mechanism used for this simulation is given in Appendix A.

The numerical set-up of the problem is fairly simple. A 1-D domain of length 30 cm was used with a simulated spark ignition source of 0.5 cm in length. The spark ignition source has the pressure and temperature of 40 atm and 1500 K, respectively. Figure 6.21 shows the pressure profile at five different times: 50, 75, 100, 125 and 150 micro-seconds. It can be seen that the peak pressure varies as the detonation wave travels downstream as a result of the instability. The time history of the peak pressure, shown in

Figure 6.22, demonstrates the evolution of the oscillatory galloping instability which starts at about 20 micro-seconds after the ignition. Investigation of the instability is an on-going research effort (Cole, 2010), and the high-order schemes such as the MP schemes presented in this work has proven to be capable of resolving such complex phenomenon.

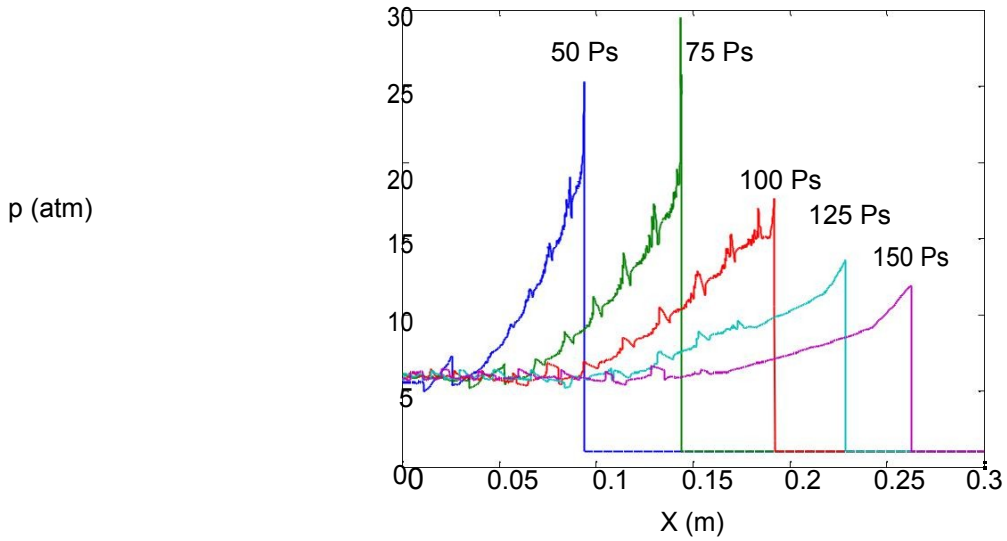


Figure 6.21: Pressure profile at five different times of a one-dimensional detonation wave computed using MP5 scheme ($'x\ 10Pm$).

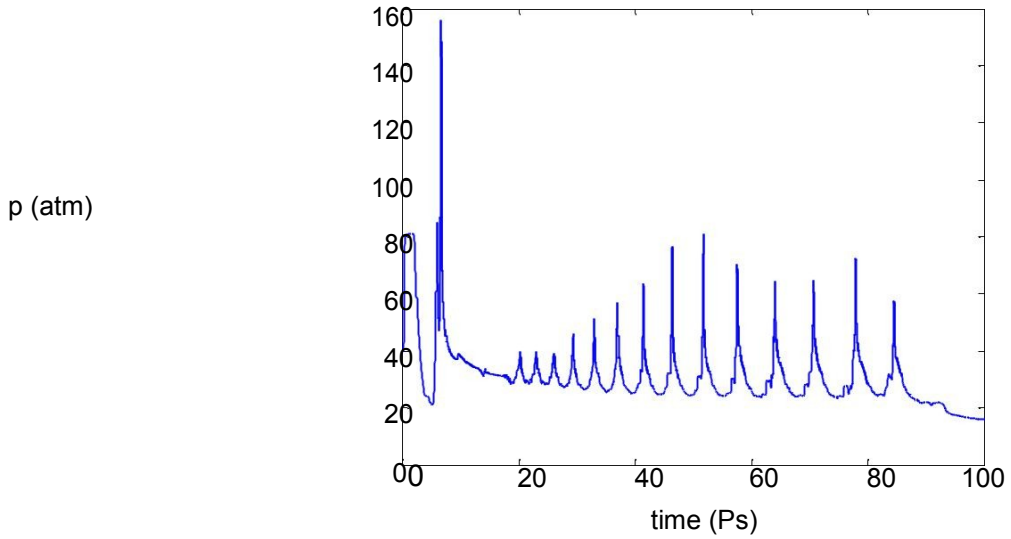


Figure 6.22: Time history of the peak pressure of a one-dimensional detonation wave.

6.3.2 Two-Dimensional Detonation Wave

The simulation of the detonation wave is extended to two-dimension to test the capability of the solver. One interesting feature observed in this simulation is the development of the cellular structure emanated from the detonation wave as a result of the chemical reactions. The 2-D problem is initiated similarly to the 1-D case. However, the spark ignition source is now arranged to introduce a small perturbation to the flow field at the initiation of the detonation wave.

Figure 6.23 shows the numerical schlieren images of the detonation wave at five different times. It can be seen that the cellular structure composed of multiple triple points connecting the Mach stem emanated from the shock starts to develop as the shock propagates further downstream. Similar structure has been observed both in experiments and numerical simulations and is referred as detonation cells.

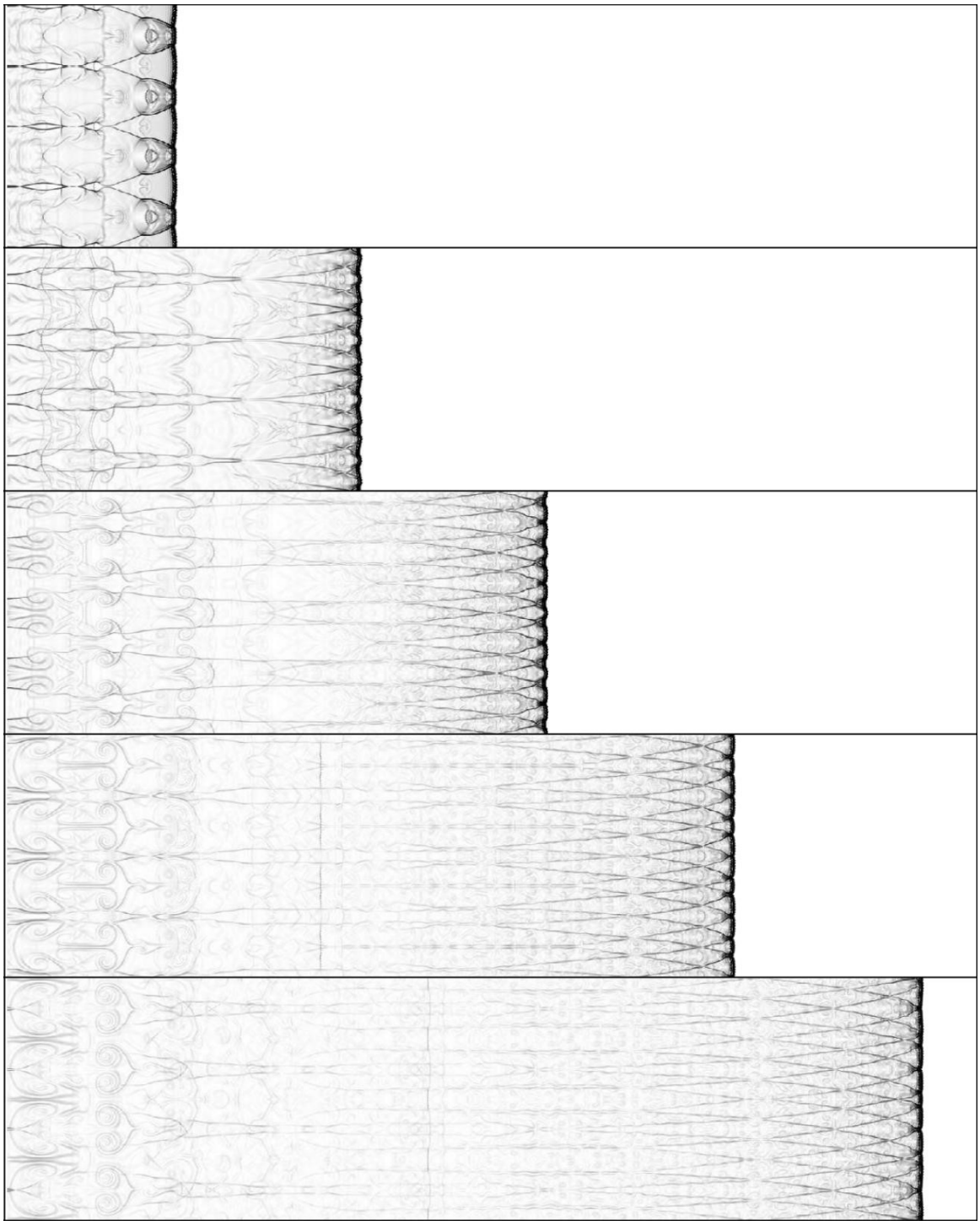


Figure 6.23: Two dimensional simulation of a detonation wave using MP5 schemes.

CHAPTER 7: PARALLEL PERFORMANCE AND OPTIMIZATION STUDY

7.1 Introduction

The fluid solver and the kinetics solver have gone through several optimization stages. The optimization strategies implemented in the fluid solver and the kinetics solver are detailed with the achieved parallel performance. Since the fluid solver and the kinetics solver are decoupled as a result of operator splitting, different optimization strategies have been employed for each solver in order to maximize the performance. The performance of the fluid solver and the kinetics solver are presented separately to illustrate the efficiency of the optimization. The overall performance of the solver which includes coupling of both the fluid solver and the kinetics solver is also reported. All the results presented in this section are based on double precision calculation. The comparison is made between a NVIDIA Tesla C2050 with one core of an Intel Xeon X5650 CPU.

7.2 Optimization the Fluid Solver

7.2.1 Kernel Types

In general, there are two types of kernels required for the CFD calculation: cell-based and face-based. The cell-based kernels mostly involve solving the equation of state (EOS) at each cell as well as updating the flow variables at each cell from the surface fluxes. The face-based kernels are responsible for the reconstruction process and flux calculation. Most of the computationally intensive calculations are placed on the face-

based kernels. For example, the reconstruction process requires projecting the conservative variables into characteristic space which is done by computing the left and the right eigenvectors of the Euler equations.

The parallelization is done by directly mapping the computational domain to the CUDA grid as illustrated in Figure 7.1. The face values can be mapped the same way with a larger grid since the number of faces in each direction is always 1 greater than the number of cells in that direction. Each CUDA thread can be associated with one cell/face inside the computational domain.

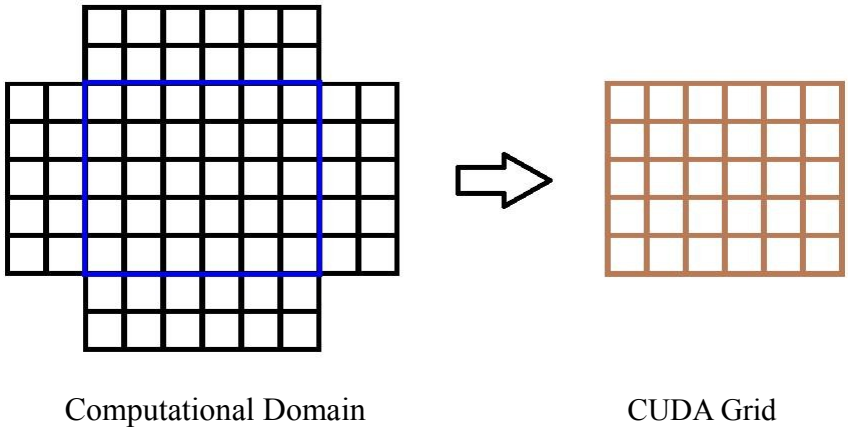


Figure 7.1: Mapping of the computational domain to the CUDA Memory

7.2.2 Domain Decomposition

In order to maximize the memory access efficiency, the domain is decomposed into one-dimensional stencils of cells/faces where each stencil can be assigned to one CUDA block. Since the computational domain can be up to three-dimensional, one can

split the stencil along different directions. However, since the storage array is always placed into linear addressed memory, it is desired to split the stencil so that all the components of a stencil are located in contiguous memory space. For example, if *x* is the fastest varying index of a two-dimensional data array, the stencil is created by splitting the domain along the *y* direction. As illustrated in Figure 7.2, each stencil is fitted into a block and each component of the stencil is associated with a thread. Since all the threads within a block are accessing consecutive memory address, the access pattern is coalesced resulting in high memory bandwidth. The calculation inside the kernel requires a certain amount of registers especially for high-order schemes, so the size of the stencil is only constrained by the size of the available registers in each warp. However, within that constraint, the size of the block can have an impact on the performance of the kernel.

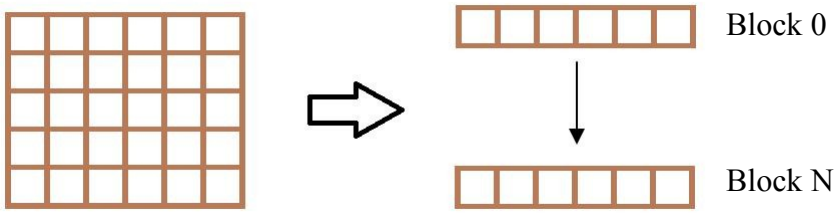


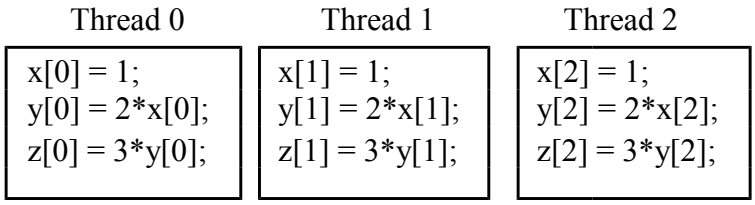
Figure 7.2: Decomposing of the CUDA memory into one-dimensional stencils

7.2.3 Thread-level Parallelism (T LP) and Instruction-level Parallelism (I LP)

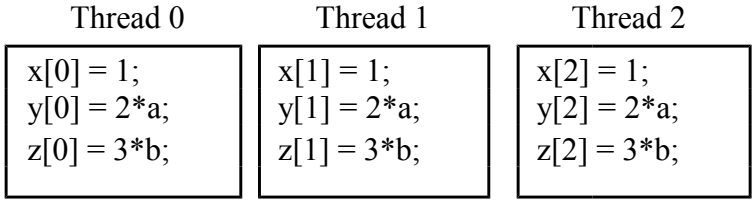
Although coalesced access pattern is very efficient in terms of maximizing the memory bandwidth, global memory still remains to be a bottleneck due to high DRAM latency. The most effective way of reducing DRAM latency is to make use of the

available shared memory on the SM. Although this approach is highly recommended (Kirk & Hwu, 2010; NVIDIA Corporation, 2010), there are several drawbacks to this approach for the problem of modeling a gas/plasma with multiple species/components. The number of the components can be quite large for the case of a plasma which results in small size stencils due to the constraint of the registers. If shared memory is utilized, the size of the stencil must be further reduced to accommodate the shared memory constraint. This is not an ideal solution for the current problem since the software framework is designed to be able to incorporate more complex physics and high-order schemes which ultimately increase the size of the problem. For the current optimization study, we attempt to reduce the DRAM latency by exploring two different types of parallelisms so-called thread-level parallelism (TLP) and instruction-level parallelism (ILP). Their effects on the performance of the kernel are also examined.

TLP is obtained by making using of multi-threading to execute an instruction in parallel. This is the core of any HPC platform including CUDA. ILP, on the other hand, is measured by the number of independent operations performed within one single thread. An example is given below to illustrate the difference between the two.



(a)



(b)

Figure 7.3: Examples of TLP and ILP: (a) TLP with no ILP, (b) TLP with ILP

In Figure 7.3a, all the instructions within one thread is dependent, because the completion of the previous operation is required before starting the next operation. This is not the case in Figure 7.3b where all the operations can be performed independently of each other. It is usually recommended to maximize the block occupancy in order to hide memory latency. Maximizing the block occupancy also increases the level of TLP within a block. However, as shown by Volkov (2010), ILP can also be used in conjunction with TLP to hide memory latency. Several test cases done by Volkov have confirmed that it is sometimes preferred to maximize the ILP instead of the TLP in order to achieve high performance. The general strategy for approaching the optimized block size for each problem is based on the balance of both of the ILP and TLP. For kernels with low ILP instructions, the occupancy should be increased to maximize the TLP. In contrast,

kernels with high ILP should have a low occupancy in order for the thread to maximize the use of the registers to cover for the memory latency. Figure 7.4 below further illustrates this point by showing the two representative kernels of the fluid solver in terms of their performance and the block size. The normalized kernel time is measured using the CUDA profiling tool.

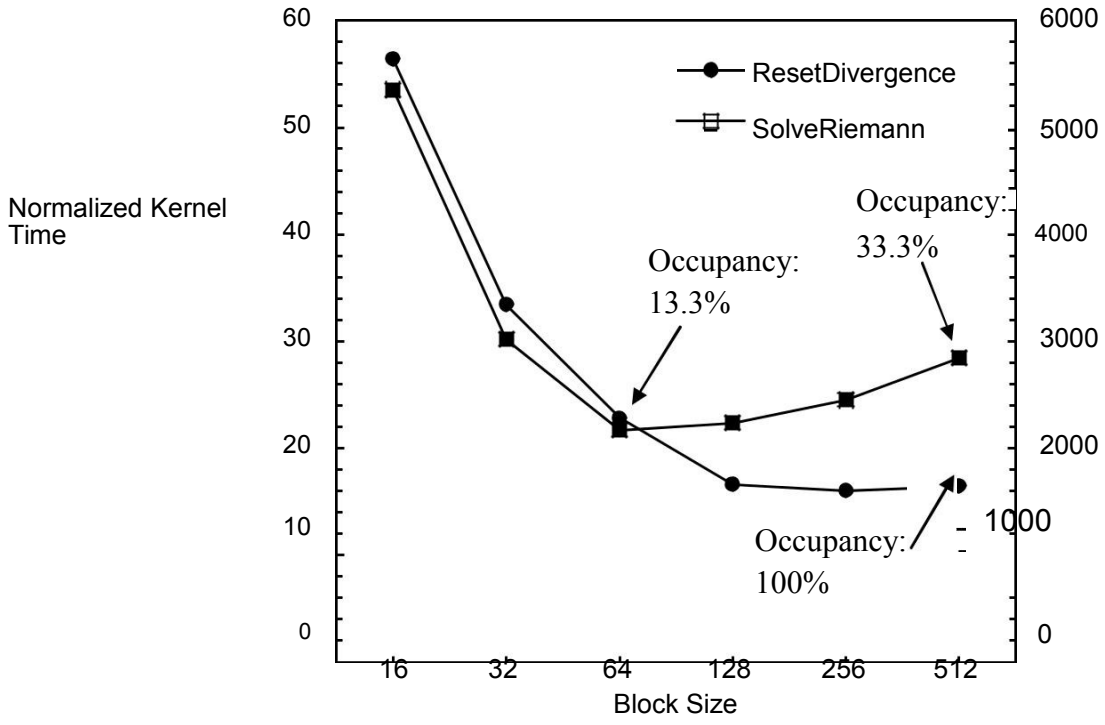


Figure 7.4: Normalized kernel time for two representative kernels.

The *SolveRiemann* kernel performs the reconstruction of the face values using high-order schemes. The reconstructing procedure is done on the characteristic variables instead of the conserved variables. Since the characteristic equations are decoupled, the

reconstruction procedure can be performed independently for each characteristic variable. This reflects a high amount of ILP which explains why this kernel is performing better at low occupancy. The *ResetDivergence* kernel, on the other hand, determines the divergence factor of each face and reduces the face values to first-order solution if the flow is in a vicinity of a strong shock or rarefaction. This can also be seen as an ILP type; however, the number of independent instructions in this case is low, so maximizing the TLP will improve the performance of the kernel. This is shown in Figure 7.4 where the best performance of this kernel corresponds to the maximum occupancy factor.

To further demonstrate the importance of selecting the block size, Figure 7.5 below shows the computational speed-up achieved using different configurations of block size for a 2-D simulation. It can be shown that neither low nor high occupancy level would result in the peak performance. The optimized performance comes about the balance of the TLP and ILP in each kernel to yield the optimal block size. It must be noted that for most of the cases presented here, the low occupancy set of block size performs slightly better than the high occupancy one, because most of the kernels used in the solver have more ILP than TLP types of operations.

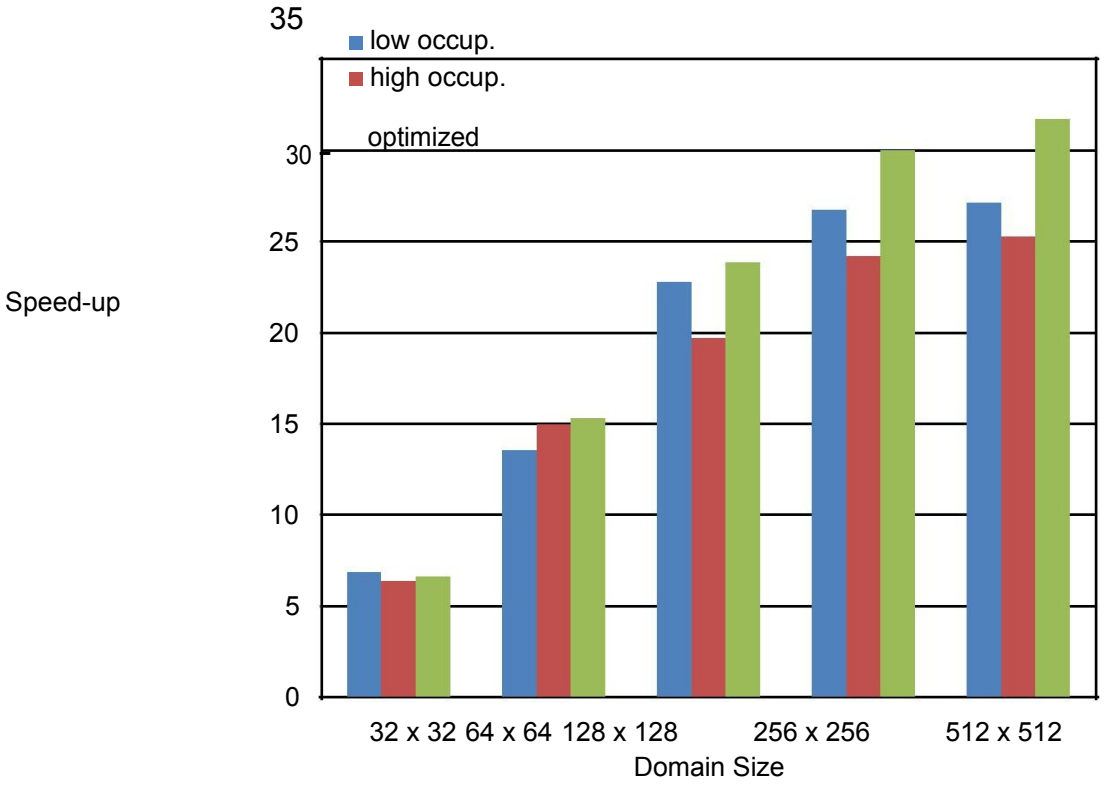


Figure 7.5: Performance of a 2-D fluid solver utilizing different sets of block size.

Figure 7.6 shows the performance comparison of a 2-D simulation of an ideal gas utilizing two different time integration methods. The first method is to use the TVD RK time integration, and the second one is to utilize the state expansion version of the ADER method to achieve the same order of accuracy. Both schemes are 5^{th} order in space and 3^{rd} order in time. The maximum speed-up obtained for the fluid solver using RK method is about 30 times faster than the CPU version. It is clearly shown that the ADER method outperforms the RK method (56 times faster than CPU). The reason for the difference is based on the fact that the RK integration has a sufficient amount of overhead due to the memory transfer and application of the boundary conditions at every RK step.

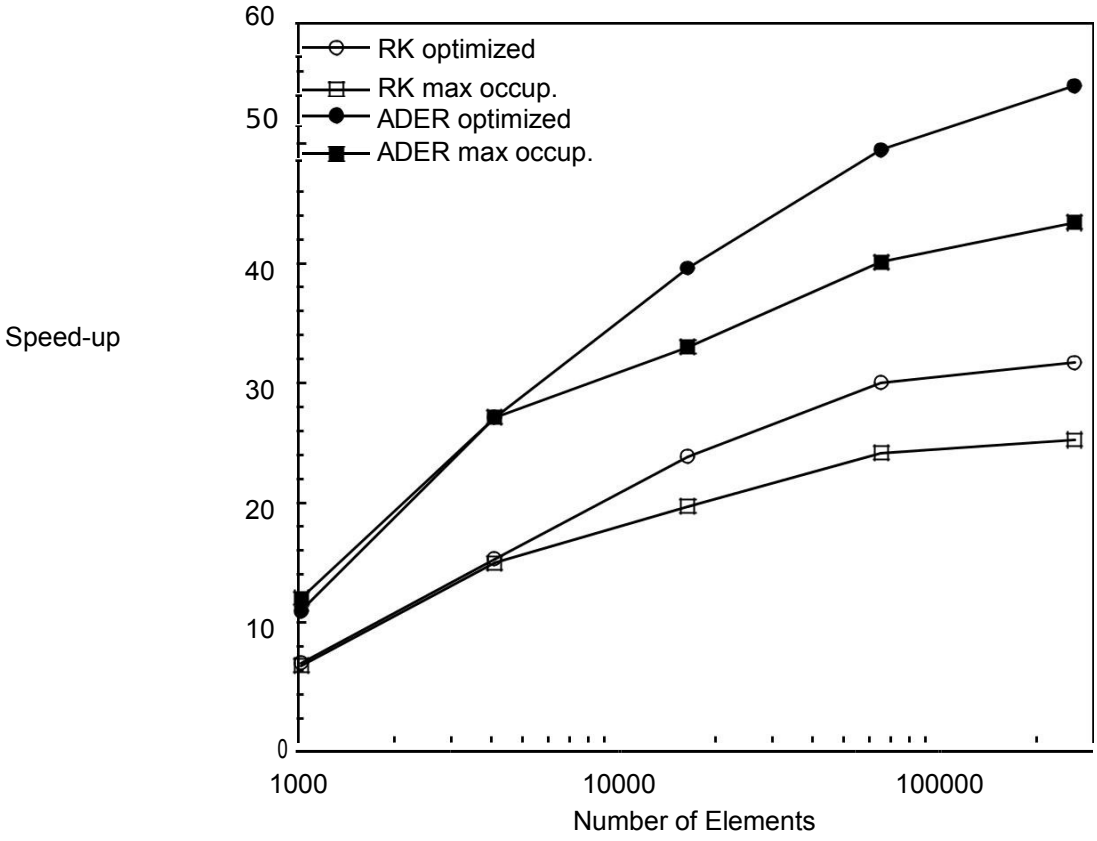


Figure 7.6: Performance of the fluid solver utilizing the RK time integration method and ADER method.

7.3 Optimization of the K inetics Solve r

Following the formulation discussed in chapter 4, the kinetics problem can be defined as a linear system of algebraic equations. The solution of the system represents the change in mass of the species and the change in the total energy. We employ a Gaussian elimination algorithm to solve for the kinetics problem. Gaussian elimination is typically done in two steps: forward elimination and backward substitution. First, the Jacobian matrix is reduced to row echelon form by performing row operations on the

Jacobian matrix and the right-hand-side vector. After the reduction, the solution can be obtained from backward substitution.

The solution of the kinetics problem must be computed at each cell for each time iteration associated with the flow solver. This is clearly a computational intensive calculation which can be benefited from the GPU. The kinetics solver is implemented to take advantage of the shared memory to avoid global memory access. There are several advantages in utilizing the shared memory in this case. The reduction and substitution are considered serial operations since they can only be done at one row of the Jacobian matrix at a time. In addition to the arithmetic operations, the reduction process also requires multiple *read* and *write* operations to modify all the entries of the Jacobian matrix. Computing the Jacobian matrix directly on global memory would significantly affect the parallelism of the kernel due to the high amount of DRAM access especially in the case of a large size matrix. In order to avoid this situation, we store all the entries of the Jacobian matrix and the right-hand-side (RHS) vector in shared memory. The memory traffic can be effectively reduced in this case; however, the size of the matrix is now restricted by the size of shared memory. In order to maximize the use of shared memory, the domain is now mapped to the CUDA grid such that each block is responsible for one system. This is similar to the mapping procedure shown in Figure 7.1 except that each entry of the CUDA grid now represents a block. The performance of the kinetics solver is shown in Figure 7.7.

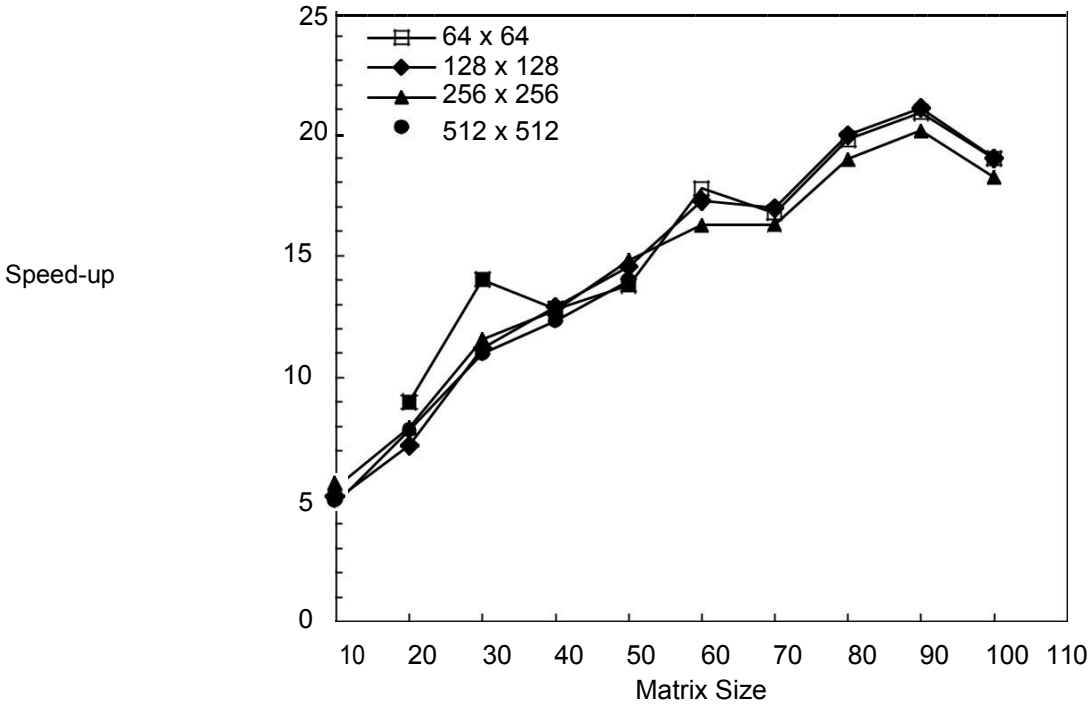


Figure 7.7: Performance of the kinetics solver utilizing shared memory.

The kinetics solver is tested with different domain sizes. It is shown in the figure that the domain size has little effect on the speed-up factor. This is due to the fact that increasing the size of the domain only affects the number of blocks. The efficiency of the parallelization depends solely on the matrix operations in the kernel. As the system size increases, the performance increases almost linearly. Since the size of shared memory is limited, the matrix size used in the test problems is also restricted. It must be noted that as the size of the system increases, the Gaussian elimination is no longer effective due to the growth in machine error.

7.4 Ove rall Performance of the Solver

The overall performance of the fluid solver when coupling to the kinetics solver is very promising as demonstrated in Figure 7.8. It must be noted that the computation time in this simulation is dominated by the kinetics solver, so the overall performance of the solver should be close to the performance of the kinetics solver. The simulation done in this case is for a thermally perfect gas consisting of 9 species. The reaction mechanism used for the chemical kinetics includes a total of 19 elementary reactions and their reverse processes. Detail of the mechanism is listed in Appendix A. The simulation was done using both the RK time integration and ADER methods. The performance of the kinetics solver is also shown in the graph for comparison purpose.

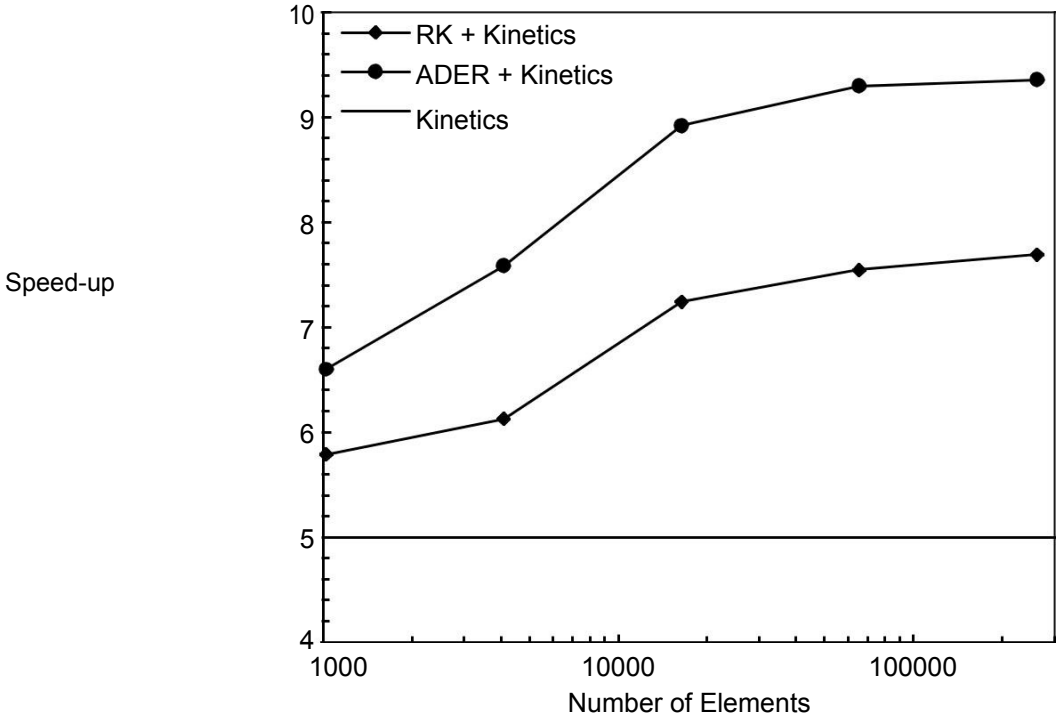


Figure 7.8: Performance of a 2-D simulation of chemically reacting flow on the GPU.

The result of the reactive flow simulation is consistent with the ideal gas case. Since ADER scheme only requires single-stage time integration, the overhead associated with the memory transfer between host and device is eliminated. The speed-ups obtained in both methods are consistent with each other. Result obtained from the simulation using ADER scheme is about 20% faster than RK method. The performance of the reactive flow simulation is lower than in the case of an ideal gas since the computation time is dominated by the chemical kinetics. It must be noted that the simulation performed here only consists of 9 species, so the size of the matrix in the kinetics calculation is only 10. Better performance of the solver can be expected for simulation with a larger reaction mechanism (e.g., large number of species and reactions).

CHAPTER 8: CONCLUSION

8.1 Conclusion and Accomplishments

A numerical framework for modeling reactive flow phenomena has been developed utilizing modern GPU architecture. The solver incorporates several high-order numerical schemes for finite volume method and is coupled with an implicit solver for the chemical kinetics. The fluid solver and the kinetics solver are optimized for parallel performance and efficiency. Performance tests show that the current solver is 10 times faster than the CPU for the simulation of a 9-species gas mixture, and could possibly be higher for larger test problems.

The solver is benchmarked with a variety of standard test cases and has shown to be very capable of simulating both reactive and non-reactive fluid flows. The design of the solver is based on an object-oriented framework, which provides certain advantages in flexibility and extensibility. The solver can be easily extended to incorporate more physical processes as well as simulating problems with large data structure (e.g., ionized gas/plasma).

8.2 Recommendation for Future Work

The solver can be improved both in terms of capability and performance. The object-oriented design offers an easy way of incorporating new modules to the solver. For instance, since the basic structure for the kinetics solver has been established, extension from chemical kinetics to ionization kinetics should be achievable. In addition,

it is also possible to couple the current fluid solver with a particle solver (Particle-in-Cell, Direct Simulation Monte-Carlo) to perform hybrid modeling.

The performance of the solver can also be improved by extending the solver to support multiple GPU which requires the use of MPI to perform boundary exchange between the GPUs. The extension should be straight forward since most optimization issues have already been resolved.

REFERENCES

Balsara, D. S., & Shu, C.-W. (2000). Monotonicity Preserving Weighted Essentially Non-oscillatory Schemes with Increasingly High Order of Accuracy. *Journal of Computational Physics, 160*(2), 405-452.

Balsara, D. S., Rumpf, T., Dumbser, M., & Munz, C.-D. (2009). Efficient, High Accuracy ADER-WENO Schemes for Hydrodynamics and Divergence-Free Magnetohydrodynamics. *Journal of Computational Physics, 228*(7), 2480-2516.

Brandvik, T., & Pullan, G. (2008). Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware. AIAA 2008-607.

Cambier, J.-L., & Menees, G. (1989). A Multi-Temperature TVD Algorithm for Relaxing Hypersonic Flows. AIAA 1989-1971.

Cambier, J.-L., Carroll, M., & Kapper, M. (2004). Development of a Hybrid Model for Non-Equilibrium High-Energy Plasmas. *35th AIAA Plasmadynamics and Lasers Conference.* AIAA-2004-2166.

Candler, G., & MacCormack, R. (1991). The Computation of Hypersonic Ionized Flows in Chemical and Thermal Nonequilibrium. *Journal of Thermophysics and Heat Transfer, 5*(3), 266-273.

Chapman, B., Jost, G., van der Pas, R., & Kuck, D. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press.

Cole, L. K. (2010). *Combustion and Magnetohydrodynamic Processes in Advanced Pulse Detonation Rocket Engines.* Ph.D. prospectus, Department of Mechanical and Aerospace Engineering, UCLA.

Einfeldt, B., Munz, C. D., Roe, P. L., & B, S. (1991). On Godunov-Type Methods Near Low Densities. *Journal of Computational Physics, 92*, 273-295.

Elsen, E., LeGresley, P., & Darve, E. (2008). Large Calculation of the Flow Over a Hypersonic Vehicle Using a GPU. *Journal of Computational Physics, 227*(24), 10148±10161.

Gnoffo, P. A., Gupta, R. N., & Shinn, J. L. (1989). *Conse rvation Equations and Physical Models for Hypersonic Air Flows in Thermal and Chemical Nonequilibrium.* NASA-TP-2867, NASA Langley, Hampton, Virginia.

Gropp, W., Lusk, E., & Skjellum, A. (1999). *Us ing MPI: Portable Parallel Programming with the Message Passing Interface* (2nd ed.). The MIT Press.

Grosso, W. (2001). *Java RMI.* O'Reilly Media.

Harten, A. (1997). High Resolution Schemes for Hyperbolic Conservation Laws. *Journal of Computational Physics, 135*(2), 260-278.

Harten, A., Engquist, B., Osher, S., & Chakravarthy, S. R. (1987). Uniformly High Order Accurate Essentially Non-Oscillatory Schemes, III. *Journal of Computational Physics, 71*(2), 231-303.

He, X. (2004). *Numerical Simulation of Pulse Detonation Engine Phenomena.* PhD dissertation, University of California, Los Angeles, Department of Mechanical Engineering.

Huynh, H. T. (1993). Accurate Monotone Cubic Interpolation. *SIAM Journal Numerical Analysis, 30*, 57-100.

Jiang, G.-S., & Shu, C.-W. (1996). Efficient Implementation of Weighted ENO Schemes. *Journal of Computational Physics, 126*(1), 202-228.

Kapper, M. (2009). *A High-Order Transport Scheme for Collisional-Radiative and Nonequilibrium Plasma.* Ph.D. Dissertation, Ohio State University, Department of Mechanical Engineering.

Kirk, D. B., & Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-On Approach.* Morgan Kaufmann.

Klockner, A., Warburton, T., Bridge, J., & Hesthaven, J. S. (2009). Nodal Discontinuous Galerkin Methods on Graphics Processors. *Journal of Computational Physics, 228*(21), 7863-7882.

Liu, X.-D., Osher, S., & Chan, T. (1994). Weighted Essentially Non-Oscillatory Schemes. *Journal of Computational Physics, 115*, 200-212.

NVIDIA Corporation. (2007). *CUDA Occupancy Calculator.* Retrieved from http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

NVIDIA Corporation. (2010). *Compute Unified Device Architecture Programming Guide v3.2.*

Schive, H.-Y., Tsai, Y.-C., & Chiueh, T. (2010). GAMER: a GPU-Accelerated Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal Supplement Series, 186*(2), 457-484.

Shreiner, D., Woo, M., Neider, J., & Davis, T. (2008). *OpenGL Programming Guide: The Official Guide to Learning OpenGL* (Sixth Edition ed.). Addison Wesley.

Suresh, A., & Huynh, H. T. (1997). Accurate Monotonicity-Preserving Schemes with Runge±Kutta Time Stepping. *Journal of Computational Physics, 136*(1), 83-99.

Thibault, J., & Senocak, I. (2009). CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. AIAA Paper 2009-758.

Titarev, V. A., & Toro, E. (2005). ADER Schemes for Three-Dimensional Non-Linear Hyperbolic Systems. *Journal of Computational Physics, 204*(2), 715-736.

Toro, E. F. (2009). *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction* (3rd ed.). Springer.

Toro, E. F., & Titarev, V. A. (2005). TVD Fluxes for the High-Order ADER Schemes. *Journal of Scientific Computing, 24*(3), 285-309.

Van Dyke, M. (1982). *An Album of Fluid Motion.* Parabolic Press, Inc.

Volkov, V. (2010). Better Performance at Lower Occupancy. *NVIDIA GPU Technology Conference.* San Jose, CA.

Woodward, P., & Colella, P. (1984). The Numerical Simulation of Two-Dimensional Fluid Flow with Strong Shocks. *Journal of Computational Physics, 54*, 115-173.

APPENDIX A – REACTION MECHANISM FOR THE REACTIVE FLOW TEST

Reaction number	Reaction
1	$H + O_2 \Rightarrow O + OH$
2	$O + H2 \Rightarrow H + OH$
3	$H_2 + OH \Rightarrow H + H_2O$
4	$OH + OH \Rightarrow HO + O$
5	$H + OH + M \Rightarrow H_2O + M$
6	$H + H + M \Rightarrow H_2 + M$
7	$H + O + M \Rightarrow OH + M$
8	$2O + M \Rightarrow O_2 + M$
9	$H_2 + O_2 \Rightarrow HO_2 + H$
10	$H + O_2 + M \Rightarrow HO_2 + M$
11	$H + HO_2 \Rightarrow OH + OH$
12	$H + HO_2 \Rightarrow O + H_2O$
13	$O + HO_2 \Rightarrow O_2 + OH$
14	$OH + HO_2 \Rightarrow O_2 + H_2O$
15	$H_2O_2 + M \Rightarrow OH + OH + M$
16	$HO_2 + HO_2 \Rightarrow H_2O_2 + O_2$
17	$H + H_2O_2 \Rightarrow H_2 + HO_2$
18	$O + H_2O_2 \Rightarrow OH + HO_2$
19	$OH + H_2O_2 \Rightarrow H_2O + HO_2$