PARALLEL IMPLEMENTATION OF THE FINITE ELEMENT METHOD ON GRAPHICS PROCESSORS FOR THE SOLUTION OF INCOMPRESSIBLE FLOWS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MAHMUT MURAT GÖÇMEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

DECEMBER 2014

Approval of the thesis :

PARALLEL IMPLEMENTATION OF THE FINITE ELEMENT METHOD ON GRAPHICS PROCESSORS FOR THE SOLUTION OF INCOMPRESSIBLE FLOWS

submitted by **MAHMUT MURAT GÖÇMEN** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver ______________________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. R. Tuna Balkan ______________________
Head of Department, **Mechanical Engineering**

Assist. Prof. Dr. Cüneyt Sert ______________________
Supervisor, **Mechanical Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. İlker Tarı ______________________
Mechanical Engineering Dept., METU

Assist. Prof. Dr. Cüneyt Sert ______________________
Mechanical Engineering Dept., METU

Assoc. Prof. Dr. Mehmet Metin Yavuz ______________________
Mechanical Engineering Dept., METU

Assist. Prof. Dr. Merve Erdal ______________________
Mechanical Engineering Dept., METU

Assist. Prof. Dr. Barbaros Çetin ______________________
Mechanical Engineering Dept., Bilkent University

Date: 04.12.2014

ABSTRACT

PARALLEL IMPLEMENTATION OF THE FINITE ELEMENT METHOD ON GRAPHICS PROCESSORS FOR THE SOLUTION OF INCOMPRESSIBLE FLOWS

Göçmen, Mahmut Murat
M.S., Department of Mechanical Engineering
Supervisor : Assist. Prof. Dr. Cüneyt Sert
December 2014, 96 pages

In recent years clock speeds and memory bandwidths of Graphics Processing Units (GPUs) increased dramatically compared to CPUs. Also GPU vendors developed and freely released new programming tools to make scientific computing on GPUs easier. With these recent developments the use of GPUs for general purpose computing becomes a popular research field. Researchers previously demonstrated that use of GPUs may provide tens of times of speeds-ups compared to CPU solvers for CFD methods such as Smoothed Particle Hydrodynamics, Lattice Boltzmann and Discontinuous Galerkin, which are known to offer very high parallelization potential. However, studies for the utilization of GPUs for classical finite volume and especially for finite element based CFD codes are rare in the literature.

This study involves the development of a flow solver based on the Finite Element Method (FEM) working parallel on GPUs. CUDA (Compute Unified Device Architecture) programming toolkit developed by NVIDIA is used for GPU programming. Three-dimensional, laminar, incompressible, flows with possible heat transfer effects are considered. Governing equations are discretized using 2 different fractional step algorithms. Accuracy of the developed solver is tested using 5 benchmark problems, including a microchannel flow and flow inside a tube with conjugate heat transfer.

Each step of the fractional step algorithm is investigated in detail on the CPU and GPU for run time performance. Speed-up tests are performed on a series of meshes with total number of unknowns between 700,000 and 6.7 million. Parallelization on the CPU is achieved by using Intel’s MKL library and OpenMP and on the GPU mostly CUBLAS, CUSPARSE and CUSP libraries are used with some scratch-built GPU kernels whenever necessary. For the largest mesh tried, GPU usage resulted in 5.79 and 1.86 times speed-ups compared to single-thread and 8-thread CPU solutions, respectively. The use of single precision arithmetic is investigated from accuracy and efficient points of view and it is seen that it does not degrade accuracy, while providing almost 2 times speed-up both on the CPU and the GPU. Compared to the explicit version, implicit fractional step algorithm turned out to be advantageous in terms of run time for steady state problems. On the other hand, explicit method uses less memory as expected.

Keywords: General Purpose GPU Computing, GPGPU, CUDA, Parallel Computing, Finite Element Method, Fractional Step Method, Computational Fluid Dynamics, Incompressible Flows

ÖZ

GRAFİK KARTLARINDA PARALEL BİR BİÇİMDE ÇALIŞACAK SONLU ELEMANLAR YÖNTEMİ TABANLI SIKIŞTIRILAMAZ AKIŞ ÇÖZÜCÜ GELİŞTİRİLMESİ

Göçmen, Mahmut Murat
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. Cüneyt Sert
Aralık 2014, 96 sayfa

Son yıllarda grafik kartlarının (GPU) performanslarının ana işlemci (CPU) performanslarına göre çok daha hızlı artması ve bu kartların bilimsel hesaplama için kullanılmasını kolaylaştıran programlama araçlarının geliştirilmesi ile birlikte GPU'lar yüksek başarımlı bilimsel hesaplama ihtiyaçları için önemli bir alternatif olmuş ve popüler araştırma konuları arasına girmiştir. Akışkanlar mekaniği alanında çalışan araştırmacılar, paralelleştirme potansiyeli çok yüksek olan Lattice Boltzmann ve Sürekli Olmayan Galerkin gibi metotların GPU üzerinde programlanması ile CPU'larda çalışan kodlara göre 10'larca kata varan hız artışları elde edebilmişlerdir. Ancak sonlu hacim ve özellikle sonlu eleman metodu tabanlı akış çözücülerinin GPU üzerinde paralel çalıştırılması ile ilgili çalışmalar literatürde çok az sayıdadır.

Bu proje çalışması Sonlu Eleman Metodu temelli bir akış çözücüsünün GPU üzerinde paralel çalışacak biçimde geliştirilmesini kapsamaktadır. GPU üzerinde paralel programlama için NVIDIA firmasının 2007 yılında geliştirdiği CUDA (Compute Unified Device Architecture) programlama aracı kullanılmış ve üç boyutlu, laminer, sıkıştırılamayan, ısı transferi içeren akışlar çalışılmıştır. Denklemlerin ayrıştırılmasında 2 farklı kademeli adım tekniği kullanılmıştır.

Geliştirilen çözücünün doğruluğu mikro kanal akışı ve eşlenik ısı transferli boru akışı da dahil olmak üzere 5 farklı test problemiyle denenmiştir.

Kademeli adım tekniğinin her bir aşamasının CPU ve GPU'da aldığı süreler detaylıca incelenmiştir. Karşılaştırmalı hız testleri 700 bin ile 6.7 milyon arasında bilinmeyen içeren ağlarda yapılmıştır. CPU'daki paralelleştirmeler Intel'in MKL kütüphanesi ve OpenMP ile GPU'daki paralelleştirmeler ise çoğunlukla CUBLAS, CUSPARSE ve CUSP kütüphaneleri gerektiğinde ise GPU üzerinde çalışacak kerneller yazılarak yapılmıştır. Denenen en büyük ağ için GPU, CPU'nun 1 izleği (thread) ve 8 izleği (8 threads) karşısında sırasıyla 5.79 kat ve 1.86 hızlı çalışmaktadır. Kayan noktalı sayıların tek hassasiyetli depolanması durumu özel olarak incelenmiş ve bu durumda çözüm doğruluğunda bir kötüleşme tespit edilmemekle birlikte, hem CPU'da hem GPU'da 2 kat hızlanma kaydedilmiştir. Açık (explicit) kademeli adım tekniği ile karşılaştırıldığında kapalı (implicit) yöntemin zamana bağlı olmayan problemleri daha hızlı çözdüğü görülmüş, buna karşın açık kademeli adım tekniğinin daha az hafıza kullandığı ortaya çıkmıştır.

Anahtar Kelimeler: Genel Amaçlı GPU Hesaplama, GPGPU, CUDA, Paralel Hesaplama, Sonlu Eleman Metodu, Kademeli Adım Tekniği, Hesaplamalı Akışkanlar Dinamiği, Sıkıştırılamaz Akışlar

To my parents

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor, Dr. Cüneyt Sert, for his guidance, encouragement, criticism, support and insight throughout the research. Without him, this thesis could not be as improved as it is now.

I would like to express my gratitude to my parents for their unconditional love. Awareness of being loved no matter what you do must be the most comforting feeling in the world.

I want to thank my company, Punto Mühendislik, for supporting me in obtaining a Master’s degree. I also want to thank my employee at Punto Mühendislik, Dr. Ertuğrul Başeşme, and colleagues at Punto Mühendislik, especially Hüsnü Muhammed Enis Dönmez and Gökay Gökçe, for their contributions and encouragements.

This thesis would not have been possible without the help and support of my dearest friends and colleagues Kadir Ali Gürsoy and Ali Karakuş. I am also thankful to my friends Yiğitcan Güden, Emin Oğuz İnci, Sercenk Hızal, Safa Kabakcı, Sinan Karamahmutoğlu, Erinç Erol, Alper Çelik, Gökay Günaçar, Tufan Akba and Merve Özdemir for their supply in various ways.

This study was supported by the TÜBİTAK. Grant No: 111M740.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

$[A]$	Advection Matrix
$[K]$	Viscous-Stiffness Matrix
$[M]$	Mass Matrix
$[M_d]$	Lumped Diagonal Mass Matrix
$\{F\}$	Force Vector
$\{G\}$	Discrete Gradient Operator
$\{G^T\}$	Discrete Divergence
$\{P\}$	Nodal Pressure Unknowns
$\{U\}$	Nodal Velocity Unknowns
BLAS	Basic Linear Algebra Subprogram
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CPU	Central Processing Unit
CUBLAS	CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
CUSPARSE	CUDA Sparse Matrix Library
c_p	Specific Heat
D_h	Hydraulic Diameter
f	Force Vector in FEM Formulation
$\vec{f}$	External Force Vector
FEM	Finite Element Method
FLOPs	Floating-Point Operations per Second
ENO	Essentially Non-Oscillatory
GFEM	Galerkin Finite Element Method
GPU	Graphics Processing Unit
h	Convective Heat Transfer Coefficient
$\bar{h}$	Average Convective Heat Transfer Coefficient
HPC	High Performance Computing
i	Iteration Number
k	Thermal Conductivity
k_{fluid}	Thermal Conductivity of Fluid
k_{solid}	Thermal Conductivity of Solid
k_{sf}	Solid to Fluid Thermal Conductivity Ratio

LAPACK	Linear Algebra Package
LBB	Ladyzhenskaya-Babuska-Brezzi Condition
LHS	Left Hand Side
$\dot{m}$	Mass Flow Rate
MAGMA	Matrix Algebra on GPU and Multicore Architecture
MIMD	Multiple Instruction, Multiple Data
MKL	Math Kernel Library
NBC	Natural Boundary Condition
n	Time Level
NE	Number of Elements
NENp	Number of Pressure Nodes
NENv	Number of Velocity Nodes
NN	Total Number of Unkowns
Nu	Nusselt Number
OpenMP	Open Multi-Processing
p	Pressure
PCG	Preconditioned Conjugate Gradient
Pr	Prandtl Number
q	Heat Flux
Q	Flow Rate
R_i	Residual Function / Resulting RHS Vector
Re	Reynolds Number
RHS	Right Hand Side
S_j	Shape Function
SIMD	Single Instruction, Multiple Data
t	Time
T	Temperature
T_L	Thermal Entrance Length
T_m	Bulk Fluid Temperature
T_{wi}	Inner Wall Temperature
u	Flow Velocity in x-Direction
u^h	Approximate Unknown Distribution on Each Element
u_{avg}	Average Velocity
u_{inlet}	Inlet Velocity
u_j	Unknown u at Elemental Nodes
u_m	Mean Velocity
u_{max}	Maximum Velocity
$\vec{V}$	Velocity Vector
V_θ	Streamwise Velocity

v	Flow Velocity in y-Direction
w	Flow Velocity in z-Direction
w_i	Weight Function
∇	Gradient Operator
∇^2	Laplacian Operator
$\|X\|_2$	Euclidean Norm of a Vector X
Γ	Boundary Domain
δ_{ij}	Kronecker-Delta
Δp	Pressure Drop
Δt	Time Step
ϵ	Convergence Criteria
μ	Dynamic Viscosity
ν	Kinematic Viscosity
ρ	Density
φ	Viscous Dissipation

CHAPTER 1

INTRODUCTION

Computational Fluid Dynamics (CFD) is a fluid mechanics practice that uses computers for the numerical solution of conservation equations that govern fluid flow and heat transfer problems. Navier-Stokes equations are the fundamental basis of almost all flow problems that can be treated as continuum. Depending on the dominance of different effects, certain simplifications can be done on the Navier-Stokes equations to obtain parabolized Navier-Stokes equations, Stokes equations, Euler equations, full potential equations, linearized potential equations, etc. Historically, linearized potential equations were the ones solved initially. Two-dimensional (2D) methods based on conformal transformations of the flow around an airfoil to the flow around a cylinder were developed in the 1930s [1]. Over the years, with increasing computational power and availability of robust numerical methods, three-dimensional, transient and turbulent flows became solvable. As the problems that can be simulated becomes more complicated serial solutions on a single computer turn out to be insufficient. For many years researchers parallelized their CFD codes on shared and distributed memory architectures using OpenMP, MPI, PVM and similar libraries. On the other hand, in recent years clock speeds and memory bandwidths of Graphics Processing Units (GPUs) have increased dramatically compared to Central Processing Units (CPUs), making them a promising alternative. Some of the most commonly used terms throughout this thesis in the context of programming and hardware are explained briefly at Appendix.

In 2007 NVIDIA freely released Compute Unified Device Architecture (CUDA) programming toolkit, which makes general purpose computing on GPUs almost as simple as writing codes for CPUs. With the availability of CUDA, utilizing GPUs for very demanding scientific tasks, including CFD, becomes a popular research field.

Researchers previously demonstrated that, for methods that are known to offer very high parallelization potential such as Lattice Boltzmann Method (LBM) or Discontinuous Galerkin Method (DGM), the use of GPUs may provide tens of times of speeds-ups compared to CPUs. However, studies for the utilization of GPUs for classical finite volume and especially finite element based CFD codes are rare in the literature. This thesis study involves the development and performance tests of a flow solver based on the Finite Element Method (FEM) working parallel on GPUs. Present works in the literature on the use of FEM with GPU's follow the idea of porting codes that are initially designed for CPUs to GPUs. This is obviously not the best approach for the full utilization of the parallel performance potential of GPUs. Instead, in this study a CFD code is written directly for the GPU from scratch.

1.1 Finite Element Method (FEM)

Since 1960's FEM is the most commonly used numerical technique for solving structural mechanics problems. Actually it is better to consider FEM as a mathematical tool for solving Partial Differential Equations, so its use should not be limited to structural mechanics. In order to solve a differential equation by FEM, first it is put into an equivalent weak form [2]. This is based on the weighted residual statement ($\int_{\Omega} w(x)\, R(x)\, dx = 0$) of the problem which forces the integral of the residual of the differential equation multiplied by a weight function to zero. The residual R makes use of approximate solution over each element ($u^e(x) = \sum_j u_j^e S_j\,(x)$) is defined as a combination of nodal unknown values (u_j^e) and shape functions (S_j). After the selection of proper weight functions, the task reduces to the solution of a linear algebraic system of equations for the nodal unknowns.

Finite Volume Method (FVM) is used more widely for fluid flow problems. Main reason for this is the suitability of FVM to the Eulerian based conservation equations of fluid mechanics written in integral form for a control volume. Another reason behind the popularity of the use of FVM for CFD is its simplicity, which appeals practicing engineers. Contrary to FEM's complex mathematical details, FVM's discretization of conservation equations on control volumes is more natural and easier to grasp for an engineer. However, from a numerical point of view it is not easy to state a clear advantage of FVM over FEM. Due to its exceptional success at structural mechanics problems, and particularly for elliptic equations, FEM had a hard time to meet the expectations of fluid flow problems. However, this should not be considered as a weakness of FEM. Numerical solution of fluid flow problems are generally more demanding than structural mechanics ones, mainly due to the existence of the advection phenomena. Nearly all of the difficulties that FEM faces while solving fluid flows are also seen with FVM. FEM has also advantages over FVM. For example, it is naturally applicable to unstructured meshes. Unlike FVM, increasing the approximation order on unstructured meshes is easy. Related to this, p-type mesh refinement that can be used to increase regional accuracy can easily be done with FEM. Neumann type boundary conditions are naturally supported by FEM. Because almost all structural mechanics solvers are FEM based, fluid structure interaction problems can be solved in a single framework, when the fluid flow part is also formulated using FEM. In summary it is possible to say that both FEM and FVM have their own advantages and disadvantages and choosing between them is mostly related with personal experience and level of knowledge. Literature review for the use of FEM for fluid flow problems will be given in the next chapter.

1.2 Graphics Processing Units (GPU)

Originally GPUs were designed as pure fixed-function devices to specifically process stages of a graphics pipeline such as vertex and pixel shaders [3]. But over the years, they have evolved into increasingly flexible programmable processors. In a simple way, modern GPU's can be defined as many core chips built around an array of streaming multiprocessors (SM). This architecture of a GPU makes it suitable for high performance computing (HPC). As seen in Table 1.1 Tesla C2075 GPU that was specifically produced for HPC by NVIDIA and also used in this thesis study has 448 CUDA cores and 14 SMs. Each SM has 32 CUDA cores and all CUDA cores in a SM execute the same instruction. On the other hand Intel Xeon E5-2670 CPU which has 8 classic CPU cores used in this study.

Not only the number of cores, but also GPU's core clock speeds are increasing hence their theoretical computational power is increasing even faster as shown at Fig. 1.1. Memory access characteristics of CPUs and GPUs are also different. Because of the requirement for very fast execution of one thread on CPUs, their memory access

Table 1.1 Xeon E5-2670 and Tesla C2075 Specifications [4, 5, 6]. (SP: Single Precision, DP: Double Precision, flops: Floating Point Operations Per Second)

Device	Number of Cores	Frequency of Cores	Peak DP Performance	Peak SP Performance	Bandwidth
Xeon E5-2670	8	2.6 GHz	177 Gflops	333 Gflops	51.2 GB/s
Tesla C2075	448	1.15 GHz	515 Gflops	1030 Gflops	144 GB/s

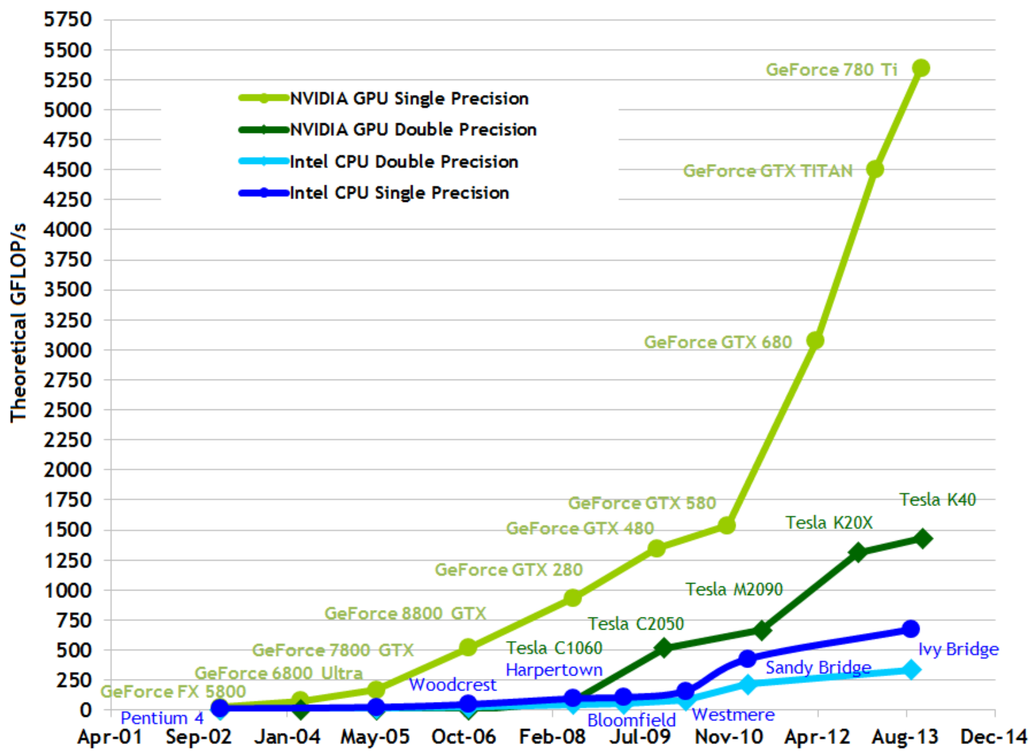


Figure 1.1 Increasing Computational Power of CPUs and GPUs over the Years [7].

speed is very high. On the other hand, GPU is processing too many data in a highly parallel way, where all threads are executing the same sequential code. In other words CPUs are developed for low latency for a single throughput while GPUs are developed for higher throughput while sacrificing latency. At the end overall memory bandwidth values heavily favor GPUs and the bandwidth gap between CPU and GPU is continuously increasing as it can be seen at Fig. 1.2. The following analogy with car manufacturing [8] can be very explanatory for latency and throughput:

A factory with ten assembly lines is manufacturing cars. It takes 6 hours to manufacture a car on an assembly line. In this case, the latency is 6 hours and the throughput is 40 cars/day (An assembly line produces 4 cars per day and there are 10 assembly lines.). If this specific factory is accepted as CPU, GPU is a factory with 400 assembly lines. But this time each of these assembly lines can produce a car in 24 hours. This second factory’s latency is 24 hours and throughput is 240 cars/day (An assembly line produces 1 car per day and there are 240 assembly lines.).

Even though these advantages make GPUs very suitable for scientific calculations, there were very few such studies until recently. The main reason behind this was the necessity of using graphic processing libraries like OpenGL or Direct3D for coding on the GPU. In other words programmers were forced to express their scientific calculation algorithms in terms of graphical calculation algorithms, which requires a high level of programming skill. With the help of CUDA (Compute Unified Device Architecture), which is parallel programming toolkit developed by NVIDIA in 2007,

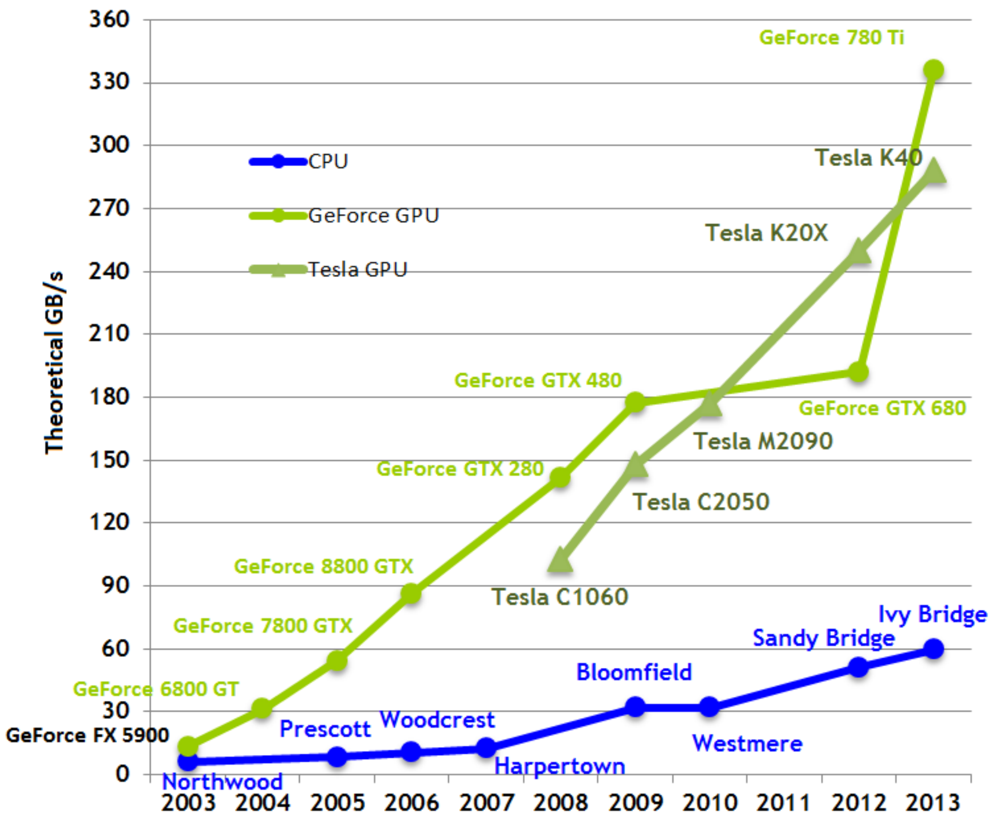


Figure 1.2 Increasing Memory Bandwidth of CPUs and GPUs over the Years [7].

today it is much easier to write general purpose codes on GPUs. After the release of CUDA, a huge increase is observed in the number of studies utilizing GPUs for purpose scientific computing, including Computational Fluid Dynamics. Following the success of CUDA, similar GPU programming toolkits like OpenCL (Open Computing Language) [9] and APP SDK (AMD OpenCL™ Accelerated Parallel Processing) [10] appeared. OpenCL is a framework for writing programs for both CPUs and GPUs in parallel. It supports a variety of products by different manufacturers (AMD, NVIDIA, Intel, ARM, etc.). On the other hand, APP SDK is a code development platform particularly for AMD products.

CUDA Toolkit comes with a C compiler (nvcc) with extra functionality compared to a standard one. These add-ons enable functions such as data transfer between CPU and the GPU, synchronization after asynchronous parallel processes, etc. A CUDA code basically has two parts, standard C/C++ codes and CUDA kernels. CUDA's C compiler sends the standard C codes to the standard C compiler and handles the kernels by itself. It then combines everything into a single executable. In order to write a code that will work on a GPU, the first thing to do is to determine the parallel programmable parts of the task at hand. After determining these parts, suitable kernels to operate on them in parallel should be written and the data transfer between the CPU and GPU should be established.

Not only a compiler but also other programming tools, such as a GPU debugger and a performance analyzer comes with the CUDA Toolkit. Also it includes CUBLAS library [11], which is GPU counterpart of BLAS (Basic Linear Algebra Subprograms) and CUSPARSE library [12], which is used for sparse matrix operations. As CUDA usage spreads, some third party free and commercial libraries for scientific computations are developed as well. Examples are Tennessee University's MAGMA library [13], which can be used to work with dense matrices, CUSP library [14], which provides iterative solvers for sparse matrices, CULA library [15], which provides GPU versions of a number of standard LAPACK functions. At different stages of the thesis study, all of these libraries were tried and CUBLAS, CUSPARSE and CUSP libraries were used in the final version of the developed solver.

According to Flynn's taxonomy processor architectures can be classified into four based upon their number of concurrent instruction (or control) and data streams [16]. These four different types are Single Instruction Single Data stream (SISD), Single Instruction Multiple Data streams (SIMD), Multiple Instruction Single Data stream (MISD) and Multiple Instruction Multiple Data streams (MIMD). GPUs are mainly suitable for SIMD type parallelization, in which each of the concurrently executing processors applies the same instruction on a different part of a big data set, as can be seen from Fig. 1.3. Adding two large vectors and writing the sum on a third one is an example of this type of work. In MIMD, processors work asynchronously on lots of different small data sets with same instruction. Calculating the inverse of a large number of small matrices is this kind of work. Although GPUs can be used for MIMD type calculations in theory, libraries developed and studies conducted so far are generally based on SIMD. For example flow solvers based on Smoothed Particle Hydrodynamics (SPH) and Lattice Boltzmann Method (LBM) have very parallelizable algorithms with SIMD type calculations, and researchers reported very large speed-ups when they implemented these techniques on GPUs.

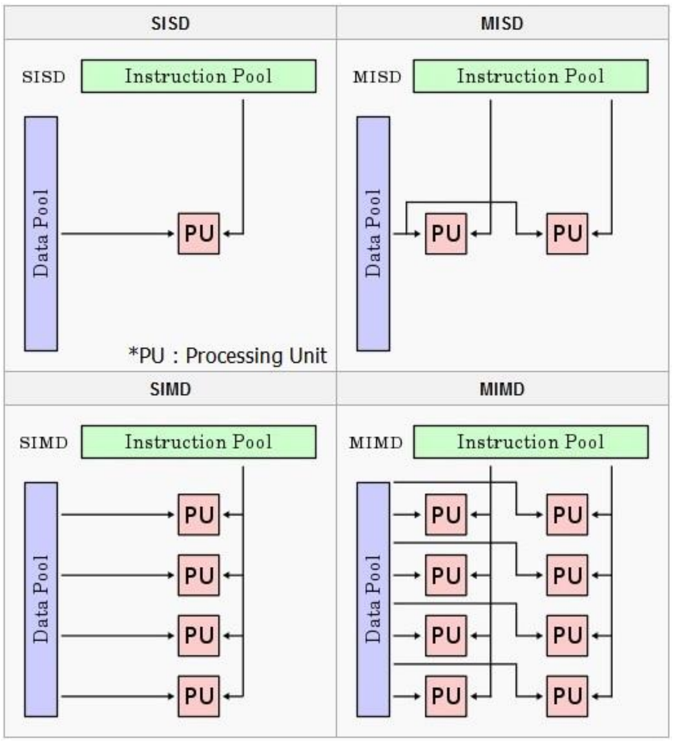


Figure 1.3 Classification due to Flynn's Taxonomy [17].

As mentioned above certain algorithms are very suitable to GPU parallelization. For example, compared to CPUs, it was possible to obtain 40, 50, 114 times speed-ups when GPUs are used for Fast Fourier Transform (FFT) [18], Monte Carlo Method [19], Lattice Boltzmann Method (LBM) [20], respectively. However, one must be cautious about these speed-up values and be aware of the problems for which they are reported. As demonstrated by Lee et al. [21] CPU and GPU performances can be quite different for different tasks. As seen in Fig. 1.4, they studied 14 different problems such as LBM, sparse matrix vector multiplication (SpMV), sorting, etc., and they obtained CPU and GPU performances that are closer to each other than the orders of magnitude differences reported in literature. Whether the problem is compute bounded or bandwidth bounded changes the performance of the GPU, most of the time. They reached as much as 14.9 times speed-up for Gilbert–Johnson–Keerthi (GJK) distance algorithm (It is used for real time collision detection used in physics engines of video games.) and mean speed-up value for all algorithms is 2.5 times. For example SpMV is a very commonly encountered task in FEM and nearly 2 times speed-up is obtained for it. The study was conducted using Intel Core i7-960 CPU and NVIDIA GTX280 GPU, both utilized fully as much as possible. It is worth to note that in this study one year old NVIDIA GPU was compared with a brand new Intel CPU of the time.

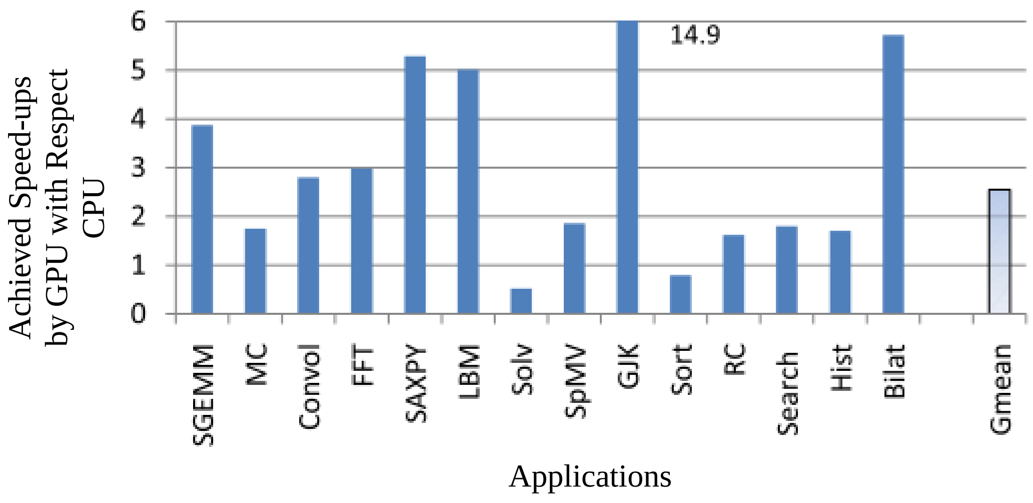


Figure 1.4 Comparison of Intel Core i7-960 CPU and NVIDIA GTX280 GPU Performances [21]. (SGEMM: SP General Matrix Multiply, MC: Monte Carlo, Convol: Convolution, Solv: Constraint Solver, RC: Ray Casting, Hist: Histogram Computation, Bilat: Bilateral Filter)

1.3 GPU Programming with CUDA

Even though the CUDA compiler enables a very similar coding experience with the standard C/C++ programming, there are some differences between the codes that work on the CPU and on the GPU. In order to understand the similarities and differences, a simple example that adds two vectors and stores the result in a third one is explained below.

CPU version of vector addition (no parallelization) [22]

```
#include <stdio.h>
#include <stdlib.h>

#define N 256

void add(int *vec1, int *vec2, int *sum) {
   int threadID = 0;
   while (threadID < N) {
      sum[threadID] = vec1[threadID] + vec2[threadID];
      threadID += 1; // There is single CPU core, so increment is one
   }
}

int main(void) {
   int *vec1, *vec2, *sum;

   // Allocate the memory on the CPU
   vec1 = (int*)malloc(N * sizeof(int));
   vec2 = (int*)malloc(N * sizeof(int));
   sum  = (int*)malloc(N * sizeof(int));

   // Fill the vectors 'vec1' and 'vec2' on the CPU
   for (int i=0; i<N; i++) {
      vec1[i] = -i;
      vec2[i] = i * i;
   }

   // Add vec1 to vec2 and write the results to sum
   add(vec1, vec2, sum);

   // Free the memory we allocated on the CPU
   free(vec1);
   free(vec2);
   free(sum);

   return 0;
}
```

The `add()` function is consciously written in that complex form in order to be able to compare it with its GPU version. The usual simple way of writing it will result in the following code;

```
void add(int *vec1, int *vec2, int *sum) {
   for (int i=0; i<N; i++) {
      sum[i] = vec1[i] + vec2[i];
   }
}
```

Using CUDA the same vector addition can be performed on a GPU in parallel by mainly writing the GPU version of the function `add()` and transferring the vectors between the GPU and the CPU.

<u>GPU version of vector addition</u> [22]

```
#include <stdio.h>
#include <stdlib.h>

#define N 256

__global__ void add(int *vec1, int *vec2, int *sum) {
   int threadID = threadIdx.x; // This thread handles the data
                               // at its thread id
   if (threadID < N) {
      sum[threadID] = vec1[threadID] + vec2[threadID];
   }
}

int main(void) {
   int vec1[N], vec2[N], sum[N];    // Allocate the memory on the CPU
   int *dev_vec1, *dev_vec2, *dev_sum;

   // Allocate the memory on the GPU
   cudaMalloc((void**)&dev_vec1, N * sizeof(int));
   cudaMalloc((void**)&dev_vec2, N * sizeof(int));
   cudaMalloc((void**)&dev_sum,  N * sizeof(int));

   // Fill the vectors 'vec1' and 'vec2' on the CPU
   for (int i=0; i<N; i++) {
      vec1[i] = -i;
      vec2[i] = i * i;
   }

   // Copy the vectors 'vec1' and 'vec2' to the GPU
   cudaMemcpy(dev_vec1, vec1, N * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(dev_vec2, vec2, N * sizeof(int), cudaMemcpyHostToDevice);

   // Add vec1 to vec2 and write the results to sum
   add<<<1,N>>>(dev_vec1, dev_vec2, dev_sum);

   // Free the memory allocated on the GPU
   cudaFree(dev_vec1);
   cudaFree(dev_vec2);
   cudaFree(dev_sum);

   return 0;
}
```

Implementation of the `main()` function is very similar in both versions. As the main difference `cudaMalloc()` is used in the GPU version to allocate memory instead of `malloc()` and at the end `cudaFree()` is used to free memory instead of `free()`. Using `cudaMemcpy()`, input vectors are copied to the device (GPU). Input vector could also be filled directly in the GPU, which is not preferred here because the main intent is to show how a particular operation, namely, the addition of two vectors, can be performed on a graphics processor. GPU version of the main function calls the `add()` function that works on the GPU using the special triple angle bracket syntax (<<< … >>>).

GPU version of the `add()` function is defined using the `__global__` qualifier, making it a device kernel. Inside it parallel vector addition is performed. Although CPU and GPU versions of the `add()` function look very similar, CPU code is sequential, i.e. working with a single thread, whereas GPU code is working with *N* threads simultaneously. Number of threads, *N*, is defined when function is called as `add<<<1,N>>>`. `threadIdx.x` is one of the built-in variables that the CUDA runtime defines for the user. It contains the value of the thread index for the thread that is currently running the device code. In this sample program *N* threads that have `threadIdx.x` values changing from 0 to *N*-1 are working simultaneously.

Unfortunately, it is not that simple to port every algorithm to GPU. Reduction is a simple yet useful example to show it. Summation of the values of a vector is selected as a reduction operation. In order to keep the discussion short only the summation function is considered. Size of `vector1` is taken as 256 for the particular example.

<u>CPU version of vector reduction</u>

```
void sumVector(float *vector1, float *sumTotal) {
   sumTotal[0] = 0.0; // Note: sumTotal[0] is used instead of sumTotal
                      // in order to be compatible with GPU code
   for (int i=0; i<256; i++) {
      sumTotal[0] += vector1[i];
   }
}
```

<u>Parallelized GPU version of vector reduction</u> [23]

```
__global__ void sumVector(float *vector1, float *sumTotal)
{
   __shared__ float sum[256]; // Declare array in shared memory

   int threadID = threadIdx.x;
   sum[threadID] = vector1[threadID]; // Copy array to shared memory

   __syncthreads();
```

```
    int nTotalThreads = blockDim.x; // Total number of active threads

    while(nTotalThreads > 1)
    {
       int halfPoint = (nTotalThreads / 2);  // Divide by two to obtain
                                             // the # of active threads

       if (threadID < halfPoint) // Only the first half of the threads
                                 // will be active
       {
          sum[threadID] += sum[threadID + halfPoint]; // Calculate the
                                                      // sum
       }
       __syncthreads();

       nTotalThreads = (nTotalThreads / 2);  // Divide by two to be ready
                                             // for next reduction step
    }

    // At this point, thread zero has the sum.
    if (threadID == 0)
    {
       sumTotal[0] = sum[0];
    }
 }
```

The idea is that each thread adds two of the values in `sum` and stores the result back to `sum`. Since each thread combines two entries into one, every step of the while loop is completed with half as many entries as it started with (see Fig. 1.5). In the next step, the same operation is done on the remaining half. When every entry in `sum` is summed, program exits the while loop.

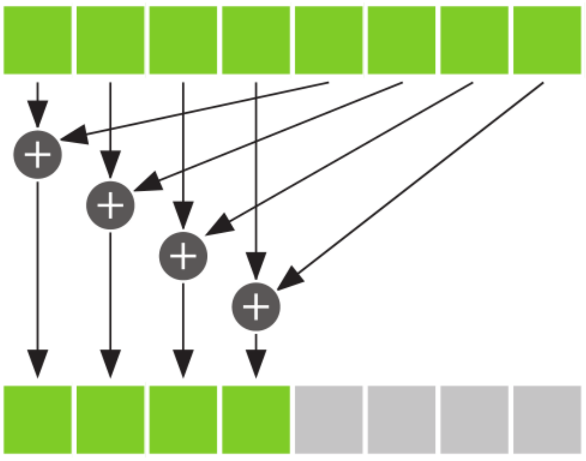


Figure 1.5 One Step of Summation Reduction [22].

For this example, when 256 threads per block are used, it takes 8 iterations of this process to reduce the 256 entries in `sum` to a single value. One can realize that a new function called `__syncthreads()` is used. This call guarantees that every thread in the block has completed its instructions before the hardware executes the next instruction on any thread. With the help of this synchronization, it is guaranteed that all of the writes to the shared array `sum` are completed before anyone tries to read from it.

CHAPTER 2

LITERATURE

Existing studies in the literature will be presented in two parts, the ones related with the use of the Finite Element Method (FEM) for flow problems and the use of GPUs for scientific computing.

2.1 Literature on the Use of FEM for Incompressible Flows

Use of FEM for the solution of flow problems goes back to 1960s when Zienkiewicz and Gheung [24] used the technique to study potential flows. Since early 1970's it has been used for the solution of Navier-Stokes equations. One of the pioneering studies is the thesis work of Hood [25], in which Navier-Stokes equations was solved for incompressible flows. In Hood's study the conservation of mass and momentum equations were solved separately, known as the segregated approach. First, the velocity was calculated by the use of assumed pressure values, followed by the calculation of pressure by solving a Poisson equation, which was obtained by combining conservation of mass and momentum equations. Velocities and pressures are estimated like this iteratively until convergence occurs. In this manner, Hood's study was the first of its kind.

Pressure and velocity components are known to be the primate variables of the incompressible Navier-Stokes equations. For a three-dimensional, isothermal flow with constant fluid properties, there are a total of four scalar unknowns (pressure and velocity components) and four scalar equations (continuity and momentum conservation). The solution of all primitive variables by the use of a single set of linear algebraic equation system is known as the mixed (coupled) formulation. In the early days of CFD, mixed formulation was too costly due to its high memory requirements. Despite this, it found use in the literature (Huyakorn et al. [26]). With constant development of computer hardware mixed formulation became more and more accessible (Zahedi et al. [27]).

In literature, it is possible to find various different versions of FEM applied to flow problems. For incompressible flows, in order to overcome difficulties of satisfying the continuity equation accurately, i.e. finding the correct pressure field that yields a divergence free velocity field, people tried using alternative unknown sets, such as the vorticity and the stream function. Although this approach resulted in successful simulations of two-dimensional problems, it turned out to be costly in three-dimensional problems due to the increased number of scalar unknowns (Taylor and Hood [28], Barragy and Carey [29]). Also specification of boundary conditions turned out to be cumbersome.

One of the methods that eliminates the problematic pressure unknown and the challenging continuity equation is the penalty technique. It works by removing the pressure term from the momentum equation by the help of the continuity equation (Hughes et al. [30], Reddy [31]). This requires the definition of a penalty parameter, which is unfortunately problem dependent. Proper selection of the penalty parameter and the increased condition number of the resulting linear algebraic system are the disadvantages of this method.

Similar to the use of central differencing in the Finite Difference and Finite Volume Methods, the use of the standard Galerkin formulation in the Finite Element Method is known to yield unphysical wiggles for the simulation of highly convective flows. Although these wiggles can be considered as the warning of a not fine enough mesh, researches worked a lot to find formulations that provide smooth solutions on relatively coarse meshes. The most popular of such stabilizations is known as the Streamline Upwind Petrov Galerkin (SUPG), which is frequently used to solve high Reynolds number flows in acceptable tolerances without using excessively fine meshes (Brooks and Hughes [32]). Galerkin Least Squares (GLS) and Taylor Galerkin are among alternative stabilization techniques (Hannai et al. [33], Donea [34]).

In the literature of the use of FEM for incompressible flows, a great number of studies were done to overcome the Ladyzhenskaya-Babuska-Brezzi (LBB) condition, which requires the storage of pressure and velocity components at different sets of points over the elements. This is not desired because it greatly limits the elements that can be used and makes the programming difficult. To overcome this, Rice and Schipke [35] developed a modified Galerkin FEM that can work with velocity and pressure stored at the same nodes. Another important similar study that combines Finite Volume Method and FEM was done by Prakash and Patankar [36]. SUPG and similar stabilization techniques mentioned in the previous paragraph are also known to circumvent the LBB condition (Hughes et al. [37]).

Patankar's SIMPLE algorithm is the most commonly used segregated solution technique used with the Finite Volume Method. Haroutunian's [38] remarkable work is a successful finite element adaptation of this approach. The studies by Shaw [39] and Du Toit [40] also used the same approach with FEM.

Unlike compressible flows, solving transient incompressible flows are not straight forward due to the missing time derivative in the continuity equation. Fractional step (aka splitting or projection) is a solution algorithm commonly used for solving transient incompressible flows (Donea et al. [41], Blasco et al. [42], Guermond et al. [43]). In this technique, solution is achieved in two stages. At the first step, approximate velocities are calculated by the solution of the momentum equation, without using the continuity equation. At the second step, calculated velocities are corrected according to the mass conservation. The solver developed in this thesis is of fractional step type.

For structural mechanics problems, low order elements are generally used in a finite element analysis, however in literature it is possible to find studies that make use of high-order elements for the simulation of flow problems (Volker [44]). Hierarchical shape functions, which make mesh coarsening and refining easier compared to the use of Lagrange type shape functions was preferred by Whitling and Jansen [45]. Characteristic Based Split (CBS) method is a relatively new FEM version conducted by a group of scientist led by Zienkiewicz (Nithiarasu et al. [46]). Yet another popular FEM technique in recent years is the Discontinuous Galerkin technique which stands out especially with its nature very suitable for parallelization (Cockburn [47]).

In this thesis an incompressible flow solver is developed. Therefore the above literature summary focused on incompressible flow studies. However, FEM was also utilized for compressible flows with success (Löhner et al. [48]). There are also works which managed to solve both compressible and incompressible flows with a single formulation (Hauke and Hughes [49]).

2.2 Literature on the Use of GPUs for Flow Problems

Scientific calculations with GPUs and particularly GPU usage for CFD are fairly new topics. Nearly all studies were performed after 2000. Moreover, studies which used CUDA programming Toolkit started after 2007, which was the year that CUDA was introduced. Because there are very few CFD studies that combine FEM and GPU, the ones that use GPU with different methods will also be mentioned in this section.

First CFD applications running parallel on GPUs are real time, particle based solutions for movie and advertisement industries (Liu et al. [50]). Creating a realistic visualization was the main aim of these studies and accurate satisfaction of the physical laws were of secondary importance. These particle based Lagrangian simulations were very suitable for parallelization on the GPU. Smoothed Particle Hydrodynamics (SPH) is another particle based method, with high parallel performance potential and gives better results in physical regards than real time solvers. SPH is the most frequently used CFD technique on the GPUs. Herault et al. [51] reported that SPH runs which lasted weeks can be solved in days or hours when GPUs are utilized.

It is possible to notice that, as far as GPU usage is concerned, compressible flow studies are more common than incompressible flow ones. Hagen et al. [52] reported 12 times speed up when GPUs are used instead of CPUs in their finite volume based two- and three-dimensional compressible flow solver. Brandwik and Pullan [53] achieved 29 and 16 times speed ups for two- and three-dimensional inviscid flows, respectively, around turbine blades. They used CUDA programing toolkit for GPU coding. Elsen et al. [54] used CUDA to parallelize their finite difference solver for hypersonic flows and they reached 15 to 40 times speed up compared to their code working serial on the CPU.

Lattice Boltzmann Method (LBM) has an algorithm that is highly parallelizable and used with GPUs frequently. Tölke and Krafczyk [55] used CUDA programming toolkit with success to solve Lattice Boltzmann equations and they reached a performance in the order of teraFLOPS. In a similar study, Riegel and Indiger [56] solved flow around a motorcycle with LBM on a NVIDIA Tesla C1060 GPU and reported 23 times speed-ups.

Sparse matrices and sparse matrix solvers are an important part of many CFD codes. In literature it is possible to find studies focusing on them, such as the one performed by Bolz et al. [57]. Sparse matrix-vector multiplication (SpMV) is an important operation for many CFD codes. Bell and Garland [58] studied the parallel implementation of SpMV on a GPU.

One of the few FEM based flow solver implementations on GPU was done by Göddeke et al. [59]. They ported their flow solver named FEAST to a small GPU cluster. According to their findings, in contrast to their earlier structural mechanics studies, incompressible flow solver could reach lower speed up values (only 2 times). At the time they conducted this study, double precision performance of GPUs were rather poor, so they used single precision and discussed its effect on the accuracy. In a similar study, Euler equations were solved using FEM on GPUs (Phillips et al. [60]). A distinct feature of this study is the use of FORTRAN language for GPU coding, which is not a common choice. There are also successful implementations of Discontinuous Galerkin type FEM on GPUs (Klöckner et al. [61]).

Constructing the elemental system is a time consuming part of a FEM solver and the process is highly parallelizable. With the help of GPUs working in single precision, Cecka et al. [62] reached 30 times speed up compared to single core of CPU in constructing elemental systems. In another study, global stiffness matrices weren't created at all and the solution was reached directly through elemental stiffness matrices (Refsnæs [63]). Poisson equations were solved on GPUs with more than 30 million unknowns and up to 24 times speed up was observed compared to the use of a single core of a CPU.

In another noteworthy study, Overflow's (a code developed and used by NASA for space research) small but time consuming SSOR solver was transferred to GPU and

2.5 to 3 times speed up values were reported (Jespersen [64]). As an interesting aspect of this study, with the experience obtained while transferring the code to GPU, CPU version of the code was also modified and experienced an increase in speed.

Corrigan et al.'s [65] study presented many interesting details about transferring a code designed for CPU to GPU. In another important study, Malecha et al. [66] added GPU support to OpenFOAM, which is popular freely available, open source CFD software, and used the code to solve biological flows.

Looking at the studies between 2011 and 2014, which is the period of the current thesis study, one can notice the dominance of FVM, LBM and SPH techniques. Asouti et al. [67] reported 45 times speed up while solving steady and unsteady turbulent flows on GPU compared to single core of CPU with a vertex centered finite volume code. Lefebvre et al. [68] solved Euler equations with FVM and reported 3.5 and 2.5 times speed ups for single and double precision usage, respectively. Niemeyer and Sung [69] used GPUs in their finite volume and finite difference codes and reported 10 times and 8.1 times speed ups, respectively. Habich et al. [70] developed optimization strategies for LBM on GPUs using CUDA in double precision. They created their own performance criterion as the number of lattice cells updated per second (FluidMLUPS/s). CPU and GPU codes reached 75 FluidMLUPS/s and 100 FluidMLUPS/s, respectively. Another LBM implementation was done by Stumbauer et al. [71], who reported 20 times speed up while solving The Couette-Taylor photo-bioreactor problem. Dominguez et al. [72] studied CPU and GPU implementations of the SPH method. They compared the optimized codes for CPU (4 cores) and two different GPUs, and reported 12.5 times and 6.1 times speed ups.

As mentioned before general purpose scientific computations using GPUs became a very popular research field in recent years. Not only the constantly improving hardware, but also the software tools that made GPU programming easier contributed to this. On the software side, the CUDA Toolkit itself renewed itself periodically. Third party libraries like CUSP [14], MAGMA [13], CULA [15], ArrayFire [73] and Paralution [74] also had an important role. On the hardware side, better double precision performance and increased memory of GPU architectures like Tesla, Fermi and Kepler that are specifically designed for scientific computing made the switch from CPU to GPU easier. Realizing the potential of many-core usage for general purpose computing, Intel recently joined the competition and started to manufacture scientific computing oriented accelerator chips called Xeon Phi Coprocessors (Reinder and Jeffers [75]).

As seen from the above review, studies that utilize GPUs for incompressible flows are very rare. Also in only a few studies that use GPU, the discretization was done using the finite volume method. The current thesis study is conducted to fill this gap by writing a FEM based incompressible flow solver that can take advantage of GPU parallelization as much as possible.

CHAPTER 3

FORMULATIONS AND CODE DEVELOPMENT

In this thesis study, three-dimensional, incompressible, laminar flows are solved in a time dependent setting. For constant viscosity, these flows require the solution of the following mass and momentum conservation equations;

$$\frac{\partial \vec{V}}{\partial t} + (\vec{V} \cdot \nabla)\vec{V} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \vec{V} + \vec{f} \qquad 3.1$$

$$\nabla \cdot \vec{V} = 0 \qquad 3.2$$

where ρ and μ are the constant and known density and dynamic viscosity, $\vec{f}$ is a known body force per unit mass, $\vec{V}$ and p are the unknown velocity and pressure fields. For a three-dimensional flow these equations constitute a set of four scalar equations with 4 scalar unknowns (3 velocity component and pressure). These unknowns are functions of time and the three-dimensional space.

For incompressible flows, Eqn. (3.2), known as the continuity equation, is always independent of time, even for unsteady flows. This makes it impossible to perform an explicit time discretization, which is a common procedure for compressible flow solutions. Also the continuity equation does not contain the pressure unknown. Not only that but also the missing equation of state, makes the bonding between the pressure and velocity unknowns weak for incompressible flows. Due to these, it can be said that numerical simulation of incompressible flows are more challenging compared to compressible ones.

In literature researchers tried many different approaches for the solution of incompressible flows. The ones that are most frequently used with a finite element formulation are the penalty method [30], artificial incompressibility method [76], pressure correction method [77] and the fractional step method. The last one is used in this study and its details will be given in the next section.

3.1 Explicit Fractional Step Method

This method works with time dependent equations and the solution is obtained in two main steps. In the first step, approximate values for the velocity field are calculated without using the mass conservation equation. In the second step, this velocity field is updated such that conservation of mass is ensured. Due to the decoupled solution of velocity and pressure fields, fractional step method turns out to be an efficient technique for simulating large scale problems [78]. In literature it is possible to find a number of different versions of this method with small differences between each. Two such versions are considered in this study, explicit and implicit ones. The explicit one used as the main solver is based on the work of Blasco et al. [42] and its details will be explained below.

First Step: In this step, Eqn. (3.1) is considered to find an intermediate velocity ($\vec{V}^{\,n+\frac{1}{2}}$), which is obtained using the velocity ($\vec{V}^{\,n}$) and pressure (p^n) fields of the previous time level n. Equation that needs to be solved is given below.

$$\frac{\vec{V}^{\,n+\frac{1}{2}} - \vec{V}^{\,n}}{\Delta t} + \left(\vec{V}^n \cdot \nabla\right)\vec{V}^{\,n+\frac{1}{2}} - \frac{\mu}{\rho}\nabla^2\vec{V}^{\,n+\frac{1}{2}} = -\frac{1}{\rho}\nabla p^n + \vec{f}^{\,n} \qquad 3.3$$

Time derivative is discretized using a first order explicit Euler scheme. The convective term, which is the second one on the left hand side of the equation is linearized by using velocity values of time levels n and $n+\frac{1}{2}$ together. Viscous term is included implicitly by using the intermediate velocity value in this term. On the other hand, pressure term is treated explicitly. The unknown of Eqn. (3.3) is intermediate velocity vector ($\vec{V}^{\,n+1/2}$).

Second Step: With the help of the intermediate velocity field calculated in the first step, velocity of the new time step ($\vec{V}^{\,n+1}$) can be obtained using

$$\frac{\vec{V}^{\,n+1} - \vec{V}^{\,n+\frac{1}{2}}}{\Delta t} - \frac{\mu}{\rho}\nabla^2\left(\vec{V}^{\,n+1} - \vec{V}^{\,n+\frac{1}{2}}\right) + \frac{1}{\rho}\nabla\left(p^{n+1} - p^n\right) = 0 \qquad 3.4$$

Velocity of the new time step ($\vec{V}^{\,n+1}$) have also ensure the conservation of mass.

$$\nabla \cdot \vec{V}^{\,n+1} = 0 \qquad 3.5$$

As seen, nonlinear term is totally missing from Eqn. (3.4). If Eqn. (3.3) and Eqn. (3.4) are put together we get;

$$\frac{\vec{V}^{\,n+1} - \vec{V}^{\,n}}{\Delta t} + \left(\vec{V}^{n} \cdot \nabla\right)\vec{V}^{\,n+\frac{1}{2}} - \frac{\mu}{\rho}\nabla^2\vec{V}^{\,n+1} = -\frac{1}{\rho}\nabla p^{n+1} + \vec{f}^{\,n} \qquad 3.6$$

which is a discretization of the momentum conservation Eqn. (3.1). Dividing the solution into 2 steps allows alleviating the numerical difficulties related to the saddle-point nature of the variational formulation of incompressible flows [79]. The basic idea is to separate the nonlinear convective term and the conservation of mass, thus decomposing the initially difficult problem into relatively easier sub problems. Convective term is only present in the first step, where mass conservation is not considered.

When discretized using the standard Galerkin FEM, details of which will be given in the Section 3.4, Eqns. (3.3), (3.4) and (3.5) become

$$[M]\frac{\{U^{\,n+\frac{1}{2}}\} - \{U^{\,n}\}}{\Delta t} + [A(U^n)]\{U^{\,n+\frac{1}{2}}\} + [K]\{U^{\,n+\frac{1}{2}}\} = -[G][P^n] + \{F^{\,n}\} \qquad 3.7$$

$$[M]\frac{\{U^{\,n+1}\} - \{U^{\,n+\frac{1}{2}}\}}{\Delta t} + [K]\left(\{U^{\,n+1}\} - \{U^{\,n+\frac{1}{2}}\}\right) + [G](\{P^{n+1}\} - \{P^n\}) = 0 \qquad 3.8$$

$$[G]^T\{U^{n+1}\} = 0 \qquad 3.9$$

where the vectors $\{U^{\,n+1}\}$ and $\{P^{\,n+1}\}$ store the velocity components and the pressures at the nodes of the finite element mesh, at the new time level $n+1$. $[M]$ is the mass matrix, $[A]$ is the advection matrix, $[K]$ is the viscous-stiffness matrix, $[G]$ represents the discrete gradient operator, $[G]^T$ represents the discrete divergence, and $\{F\}$ is the forcing term.

Unfortunately, there are difficulties in the solution of this equation set. The advection matrix, $[A]$, of the Eqn. (3.7) depends on the velocity field and it has to be calculated once every time step. Moreover, a new linear algebraic system needs to be solved at each time step to get the intermediate velocity field, and the coefficient matrix of Eqn. (3.7) is not symmetrical due to the convective term. In order to overcome this difficulty Blasco et al. [42] suggested the use of an extra iterative loop in each time step. With this approach, Eqns. (3.7), (3.8) and (3.9) can be revised as follows

$$[M]\frac{\{U_{i+1}^{n+\frac{1}{2}}\}-\{U^{n}\}}{\Delta t}+[A(U^n)]\{U_i^{n+\frac{1}{2}}\}+[K]\{U_i^{n+\frac{1}{2}}\}=-[G]\{P^n\}\ +\{F^{n}\} \tag{3.10}$$

$$[M]\frac{\{U_{i+1}^{n+1}\}-\{U_{i+1}^{n+\frac{1}{2}}\}}{\Delta t}+[K]\left(\{U_i^{n+1}\}-\{U_i^{n+\frac{1}{2}}\}\right)+[G](\{P_{i+1}^{n+1}\}-\{P^n\})=0 \tag{3.11}$$

$$[G]^T\{U_{i+1}^{n+1}\}=0 \tag{3.12}$$

The difference between Eqns. (3.7), (3.8), (3.9) and Eqns. (3.10), (3.11), (3.12) is the usage of i and $i+1$ indices, which are the iteration counters of each time step. By using the intermediate velocity values of the previous iteration in the convective and viscous terms, the only unknown intermediate velocity of Eqn. (3.10) appears in the first term as $\{U_{i+1}^{n+1/2}\}$. Convergence in each time step can be reached in a few (usually in the range of one to four) iterations at each time step.

Another important challenge is to solve for the velocity and pressure fields of the new time level by using Eqn. (3.11) and (3.12) together. The reason behind the difficulty is the lack of pressure in Eqn. (3.12). To overcome this problem, a new equation that can be obtained by eliminating $\{U_{i+1}^{n+1}\}$ from Eqn. (3.11) and (3.12) can be used. First Eqn. (3.11) can be used to write $\{U_{i+1}^{n+1}\}$ as follows

$$\{U_{i+1}^{n+1}\}\ =$$
$$[M]^{-1}\left([M]\frac{\left\{U_{i+1}^{n+\frac{1}{2}}\right\}}{\Delta t}-[K]\left(\{U_i^{n+1}\}-\{U_i^{n+\frac{1}{2}}\}\right)-[G](\{P_{i+1}^{n+1}\}-\{P^n\})\right) \tag{3.13}$$

Substituting this into Eqn. (3.12) one gets

$$[G]^T\left(\frac{\left\{U_{i+1}^{n+\frac{1}{2}}\right\}}{\Delta t}-[M]^{-1}[K]\left(\{U_i^{n+1}\}-\{U_i^{n+\frac{1}{2}}\}\right)-[M]^{-1}[G](\{P_{i+1}^{n+1}\}-\{P^n\})\right)=0 \quad 3.14$$

Finally Eqn. (3.14) can be rearranged as follows, which can be used to solve for $\{P_{i+1}^{n+1}\}$.

$$([G]^T[M]^{-1}[G])(\{P_{i+1}^{n+1}\}-\{P^n\})=$$
$$\frac{1}{\Delta t}[G]^T\left(\{U_{i+1}^{n+\frac{1}{2}}\}-\Delta t[M]^{-1}[K]\left(\{U_i^{n+1}\}-\{U_i^{n+\frac{1}{2}}\}\right)\right) \quad 3.15$$

After calculating $\{P_{i+1}^{n+1}\}$ using Eqn. (3.15), $\{U_{i+1}^{n+1}\}$ is calculated with using Eqn. (3.13). This completes one iteration of a time step. Eqn. (3.16) is used to check for the convergence of these iterations. $|X|_2$ is the Euclidean norm of a vector X. Typically maximum 4 iterations is enough for convergence, and the number drops to 1 as the solution approaches to its steady state, if it exists.

$$max\left(\frac{\left|\{U_{i+1}^{n+1}\}-\{U_i^{n+1}\}\right|_2}{\left|\{U_{i+1}^{n+1}\}\right|_2},\frac{\left|\{P_{i+1}^{n+1}\}-\{P_i^{n+1}\}\right|_2}{\left|\{P_{i+1}^{n+1}\}\right|_2}\right)\leq\epsilon \quad 3.16$$

The final difficulty in solving Eqns. (3.10), (3.13) and (3.15) is taking the inverse of the mass matrix $[M]$. Fortunately the lumped diagonal ($[M_d]$) version of $[M]$ that is easy to work with can be used. This simplification worked well for the steady state problems that are solved in this study, but its effect on the accuracy of transient problems should be investigated carefully. With the use of lumped mass matrix, fractional step method can be summarized as the solutions of the following three systems

System 1 : $$[M_d]\frac{\{U_{i+1}^{n+\frac{1}{2}}\}-\{U^n\}}{\Delta t}=\{R_1\} \quad 3.17$$

System 2 : $$([G]^T[M_d]^{-1}[G])(\{P_{i+1}^{n+1}\} - \{P^n\}) = \{R_2\} \tag{3.18}$$

System 3 : $$[M_d]\frac{\{U_{i+1}^{n+1}\} - \{U_{i+1}^{n+\frac{1}{2}}\}}{\Delta t} = \{R_3\} \tag{3.19}$$

where the right hand side vectors of these systems are given as

$$\{R_1\} = -[A(U^n)]\{U_i^{n+\frac{1}{2}}\} - [K]\{U_i^{n+\frac{1}{2}}\} - [G]\{P^n\} + \{F^n\} \tag{3.20}$$

$$\{R_2\} = \frac{1}{\Delta t}[G]^T\left(\{U_{i+1}^{n+\frac{1}{2}}\} - \Delta t[M_d]^{-1}[K]\left(\{U_i^{n+1}\} - \{U_i^{n+\frac{1}{2}}\}\right)\right) \tag{3.21}$$

$$\{R_3\} = -[K]\left(\{U_i^{n+1}\} - \{U_i^{n+\frac{1}{2}}\}\right) - [G](\{P_{i+1}^{n+1}\} - \{P^n\}) \tag{3.22}$$

The first system calculates $\{U_{i+1}^{n+\frac{1}{2}}\}$, i.e. the intermediate velocity at the new iteration. Calculation of $\{R_1\}$ is costly in the sense that it requires the evaluation of a new $[A]$ matrix for each time step (not every iteration of each time step). The actual solution of the system is computationally cheap due to the use of the lumped mass matrix.

The second system calculates $\{P_{i+1}^{n+1}\}$ by using $\{U_{i+1}^{n+1/2}\}$ that was just calculated in the previous step. Constructing of $\{R_2\}$ is not costly. $([G]^T[M_d]^{-1}[G])$ matrix on the left hand side of System 2 is independent of time. It can be calculated at the beginning of the solution once and because of its symmetric and positive-definite nature, Cholesky decomposition can be used for the solution of this system. With Cholesky decomposition, System 2 can be efficiently solved with two triangular matrix solutions. However, increase in the size of the resulting triangular matrices with increasing problem size becomes a problem for the limited memory of GPUs. To overcome this issue Conjugate Gradient (CG) type iterative techniques can be employed. Both Cholesky decomposition and CG is used in this study.

Lastly, the third system calculates $\{U_{i+1}^{n+1}\}$ by using $\{P_{i+1}^{n+1}\}$ that was just calculated in the previous step. Both the calculation of the right hand side vector and the solution of the system is cheap in this case. After the solution of System 3, convergence check is performed and either one more iteration is done or the calculations of the next time step are started.

As mentioned previously, there are various different versions of the fractional step method. The one explained above is selected mainly due to its simple algorithm. An alternative implicit one is also investigated. Its details will be given in next section and the advantages and disadvantages of these two formulations will be compared in Section 5.5.

3.2 Implicit Fractional Step Method

The fractional step method described in detail in the previous section has severe limitations on the allowable time step, due to its explicit treatment of certain terms. This resulted in quite long total run times to complete a solution. To improve this, an implicit fractional step formulation, which has no time step restriction due to stability concerns, is also tried. This second formulation is based on the work of Guermond and Quartapelle [80].

In the implicit formulation there is no intermediate velocity calculation. First the advection-diffusion equation given in Eqn. (3.23) is solved. As seen, the velocity at the new time step ($\vec{V}^{\,n+1}$) is obtained using the velocity of the previous time step ($\vec{V}^{\,n}$) and pressures from earlier two time steps (p^n and p^{n-1}).

$$\frac{\vec{V}^{\,n+1}-\vec{V}^{\,n}}{\Delta t}+\left((\vec{V}^n\cdot\nabla)\vec{V}^{\,n+1}+\frac{1}{2}(\nabla\cdot\vec{V}^n)\vec{V}^{n+1}\right)-\frac{\mu}{\rho}\nabla^2\vec{V}^{\,n+1}= -\frac{1}{\rho}\nabla(2p^n-p^{n-1})+\vec{f}^{\,n} \qquad 3.23$$

Unlike the explicit method that uses convective form for the advection term (see Eqn. (3.3)), Eqn. (3.23) uses skew-symmetric form as suggested by Guermond and Quartapelle [80]. In the second step, the following Poisson equation is solved for the pressure increment ($p^{n+1}-p^n$)

$$-\nabla^2(p^{n+1}-p^n)=-\frac{1}{\Delta t}\nabla\cdot\vec{V}^{n+1} \qquad 3.24$$

After introducing the Galerkin finite element discretization, Eqns. (3.23) and (3.24) take the following forms;

$$[M]\frac{\{U^{n+1}\}-\{U^{n}\}}{\Delta t}+[A(U^n)]\{U^{n+1}\}+[K]\{U^{n+1}\}= -[G](2\{P^n\}-\{P^{n-1}\})+\{F^{n}\} \qquad 3.25$$

$$[\widehat{K}](\{P^{n+1}\}-\{P^n\})=-\frac{1}{\Delta t}[G]^T\{U^{n+1}\} \qquad 3.26$$

where $\{U\}$,$\{P\}$, $[M]$, $[A]$, $[K]$, $[G]$ and $\{F\}$ were defined in Section 3.1. The new $[\widehat{K}]$ is the global stiffness matrix associated with pressure interpolation, which is a Laplacian operator. After proper adjustments, the implicit formulation solves the following systems at each time step. It is worth to note that, unlike the explicit formulation, implicit one does not use any iterations in a time step.

System 1 : $$\left[\frac{1}{\Delta t}[M]+[A(U^n)]+[K]\right]\{U^{n+1}\}=\{R_1\} \qquad 3.27$$

System 2 : $$[\widehat{K}](\{P^{n+1}\}-\{P^n\})=\{R_2\} \qquad 3.28$$

where the right hand sides vectors are

$$\{R_1\}=\frac{1}{\Delta t}[M]\{U^{n}\}-[G](2\{P^n\}-\{P^{n-1}\})+\{F^{n}\} \qquad 3.29$$

$$\{R_2\}=-\frac{1}{\Delta t}[G]^T\{U^{n+1}\} \qquad 3.30$$

Sparse linear systems need to be solved at each step to get new velocity and pressure values. System 1 has a non-symmetric left hand side matrix, which changes at each time level. System 2 has a symmetric left hand side matrix that does not depend on time. Because of the non-symmetric nature of the first system matrix, Biconjugate Gradient Stabilized (BiCGStab) solver is used to solve it. Intel MKL and CUSP libraries are used for BiCGStab on the CPU and GPU, respectively. $[\widehat{K}]$ is a symmetric matrix so Preconditioned Conjugate Gradient (PCG) is used to solve this pressure Poisson problem. For both BiCGStab and PCG methods Jacobi preconditioner is employed.

3.3 Time Consuming Parts of the Explicit Fractional Step Solution and Their Parallelization on the CPU and the GPU

The explicit fractional step method, explained in the Section 3.1, have a number of time consuming calculations. They can be classified under four main categories; sparse matrix-vector operations, vector-vector operations, creating global matrices and solving sparse systems. Sparse matrix-vector and vector-vector operations are SIMD type operations. Creating a stiffness matrix is MIMD type work due to the calculation of independent elemental systems and their assembly. Solving a sparse system mostly contains SIMD type operations particularly for a symmetric, positive-definite system. Details of these operations and how they are parallelized on the CPU and the GPU are explained below

- *Calculation of convective stiffness matrix,*$[A(U^n)]$*:* This is performed once at each time step for the construction of $\{R_1\}$ (see Eqn. (3.20)). This process needs the creation of a large number of small elemental stiffness matrices and their assembly. There is no readily available GPU library that can perform this operation as a black box. Therefore a new GPU kernel is written for it. Parallelization of this task is not straightforward and in literature a number of different alternatives were suggested [81, 82, 83]. These techniques were developed with the purpose of preventing the race condition arising during the assembly and effective utilization of GPU hardware. They differ from each other in terms of their usage of various levels of GPU memories (registers, shared memory and global memory), the responsible computing unit (thread, block) for creating the elemental stiffness matrices and the way the assembly operation is done. Some of them favor fast calculations and the others favor the use of less memory.

  In the current study the mesh coloring method, which groups the elements that do not share a common node, is utilized for eliminating the race condition. Fig. 3.1 demonstrates an example of mesh coloring of a two-dimensional mesh consisting of 6 quadrilateral elements. Elements of the same color have no common nodes and therefore their stiffness matrices are created and assembled together, while the calculations on the elements of a different color can be done in parallel simultaneously. Because the GPU works best with blocks consisting of multiples of 32 threads [22], blocks consisting of 32 threads are used for 27 node hexahedral elements. By this way, calculations associated with each velocity node of an element can be calculated parallel on a different thread.

As seen in Eqn. (3.20) what really is needed is the $[A(U^n)]\{U_i^{n+1/2}\}$ matrix-vector multiplication, but not the $[A]$ matrix itself. This suggests an alternative solution, which totally eliminates the assembly of the global $[A]$ matrix. Instead matrix-vector multiplications can be done at the element level and the resulting vectors can be assembled. Again the mesh coloring method is utilized in order to prevent race conditions. This alternative method is used in the final version of the code because it has better performance and it needs less memory. Performance of these two approaches will be compared in the Chapter 5.

CPU parallelization of this part is completed using OpenMP [84]. Mesh coloring method is also utilized on the CPU side.

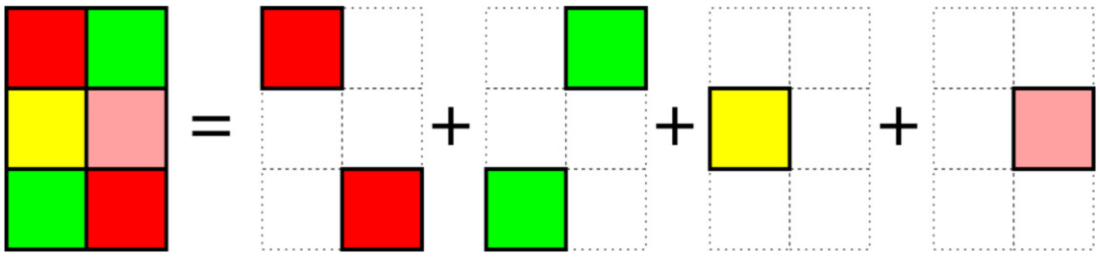


Figure 3.1 Illustration of Mesh Coloring in 2D [83].

- *BLAS operations to calculate $\{R_1\}$, $\{R_2\}$ and $\{R_3\}$:* As seen in Eqns. (3.20), (3.21) and (3.22) calculation of the right hand side vectors require sparse matrix-vector calculations and vector-vector operations. These are all simple SIMD type operations and they are very suitable for parallelization. GPU implementation of these operations is mainly done with CUSPARSE and CUBLAS libraries that come with the CUDA Toolkit, and new small GPU kernels are written when a readily available solution cannot be found. On the CPU side these are parallelized using Intel's Math Kernel Library (MKL) library [85].

- *Solution of Systems 1 and 3:* As seen in Eqns. (3.17) and (3.19) due to the use of the diagonalized mass matrix, these system solutions are simple and performed using CUBLAS. Intel's MKL library is used for CPU parallelization.
- *Solution of System 2:* As discussed in the previous section either Cholesky factorization or CG technique is used for the solution of Eqn. (3.18). When Cholesky factorization is used, solution of the system requires two triangular system solutions, which are performed using Timothy Davis' CSparse library [86] on the CPU, whereas CUSPARSE is used on the GPU. When the problem size, hence the size of the triangular systems exceed a certain limit, GPU's memory becomes insufficient for Cholesky factorization. For such cases Preconditioned Conjugate Gradient (PCG) solver is preferred, for which Intel's MKL is used on the CPU and CUSP library is used on the GPU.

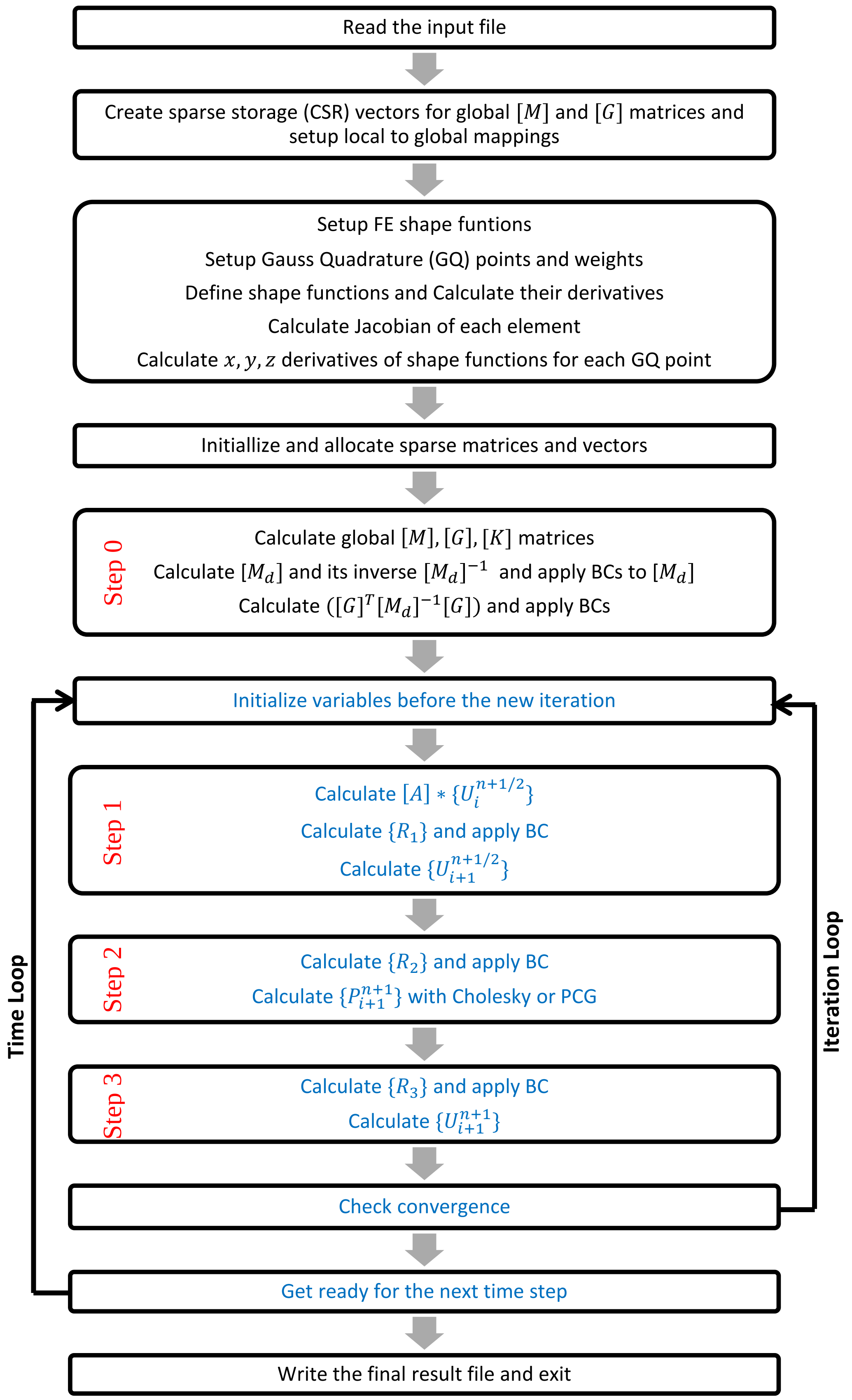
Other than these calculations, there are some that have to be performed only once outside the time loop, such as the calculation of time independent global systems $[M], [M_d], [M_d]^{-1}, [K], [G], [G]^T, ([G]^T[M_d]^{-1}[G])$ and calculating the Cholesky factorization of $([G]^T[M_d]^{-1}[G])$ matrix. These calculations are not parallelized.

Fig. 3.2 shows the flow chart of the developed flow solver. As seen all the time consuming operations that lie inside the time loop are parallelized on the GPU. Step numbers 0, 1, 2 and 3 will be used in analyzing the performance of the solver in the Chapter 5.

3.4 Finite Element Formulation

In this study the discretized Eqns. (3.17), (3.18) and (3.19) are obtained using the standard Galerkin Finite Element Method (GFEM). As with almost all numerical techniques, first the problem domain is discretized into small parts, called elements. After creating the numerical mesh by defining the elements and their nodes where the unknowns are stored at, FEM formulation makes use of approximate solutions over each element as given below;

$$\phi^h(x,y,z) = \sum_{j=1}^{NEU} \phi_j \, S_j(x,y,z) \qquad 3.31$$



Read the input file
Create sparse storage (CSR) vectors for global $[M]$ and $[G]$ matrices and setup local to global mappings
Setup FE shape funtions
Setup Gauss Quadrature (GQ) points and weights
Define shape functions and Calculate their derivatives
Calculate Jacobian of each element
Calculate x, y, z derivatives of shape functions for each GQ point
Initiallize and allocate sparse matrices and vectors
Step 0
Calculate global $[M]$, $[G]$, $[K]$ matrices
Calculate $[M_d]$ and its inverse $[M_d]^{-1}$ and apply BCs to $[M_d]$
Calculate $([G]^T[M_d]^{-1}[G])$ and apply BCs
Initialize variables before the new iteration
Step 1
Calculate $[A] * \{U_i^{n+1/2}\}$
Calculate $\{R_1\}$ and apply BC
Calculate $\{U_{i+1}^{n+1/2}\}$
Step 2
Calculate $\{R_2\}$ and apply BC
Calculate $\{P_{i+1}^{n+1}\}$ with Cholesky or PCG
Step 3
Calculate $\{R_3\}$ and apply BC
Calculate $\{U_{i+1}^{n+1}\}$
Check convergence
Get ready for the next time step
Write the final result file and exit
Time Loop
Iteration Loop

Figure 3.2 Flow Chart of Solver. (Blue Parts are the Ones Parallelized on the GPU)

where ϕ^h represents the approximate solution over an element for any scalar unknown ϕ, which can be a velocity component or pressure. ϕ_j are the values of the unknown at the nodes of an element and S_j are polynomial shape functions that are defined separately at each elemental node. NEU is number of nodal ϕ unknowns defined over the element. The developed solver makes use of hexahedral elements seen in Fig. 3.3. Instead of using a set of different shape functions over each actual element, it is preferred to work with a master element (see Fig. 3.3) and only define a single set of shape functions, which requires the construction of a geometric mapping between the actual coordinates (x, y, z) and master coordinates (ξ, η, ζ).

In order to satisfy the Ladyzenskaja-Babuška-Brezzi (LBB) condition, elements that have more velocity nodes than pressure nodes are used. Two element types are used. The high order element with 27 velocity nodes and 8 pressure nodes, i.e. quadratic in velocity and linear in pressure, is shown in Fig. 3.4. In total this element has $3 * 27 + 8 = 89$ scalar unknowns over it. The low order one with 8 velocity nodes and 1 pressure node, i.e. linear in velocity and constant in pressure, is shown in Fig. 3.5. It has $3 * 8 + 1 = 25$ scalar unknowns on it. For each element type two different shape function sets are used for velocity and pressure approximation.

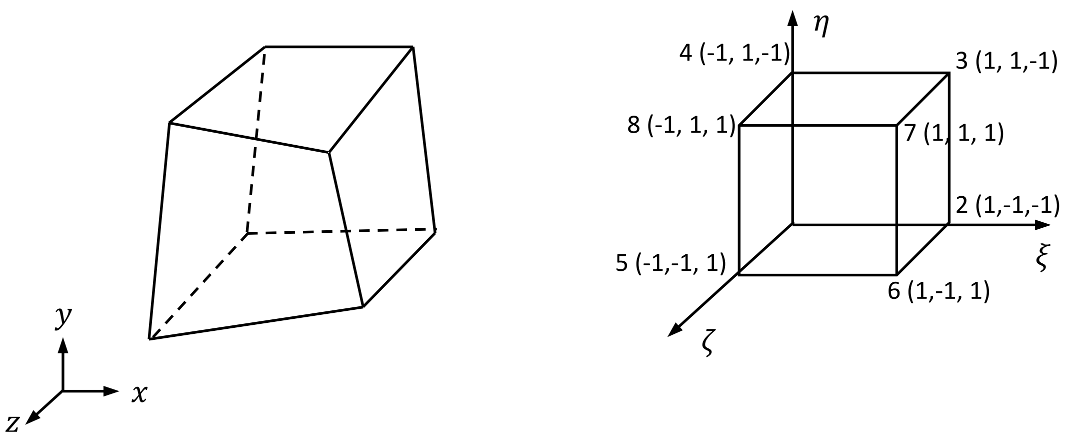


Figure 3.3 An Actual, Arbitrarily Shaped Hexahedral Element and Its Corresponding Master Element.

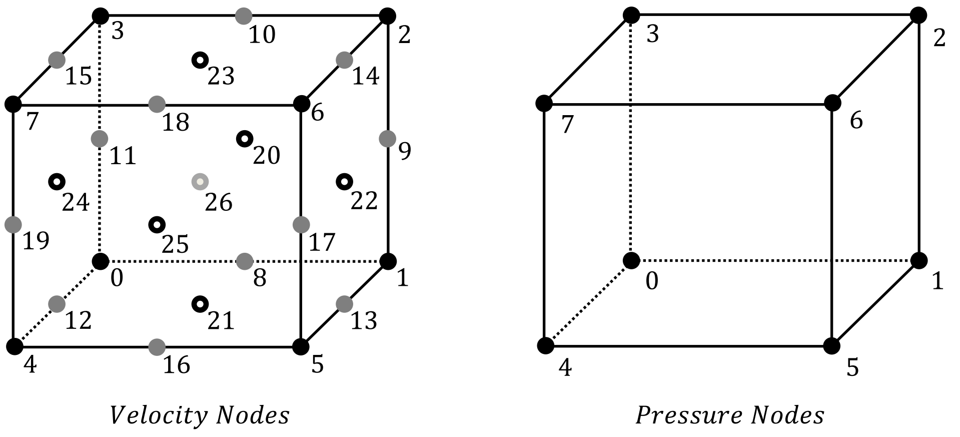


Figure 3.4 Hexahedral Element with 27 Velocity Nodes and 8 Pressure Nodes (Element Type 1).

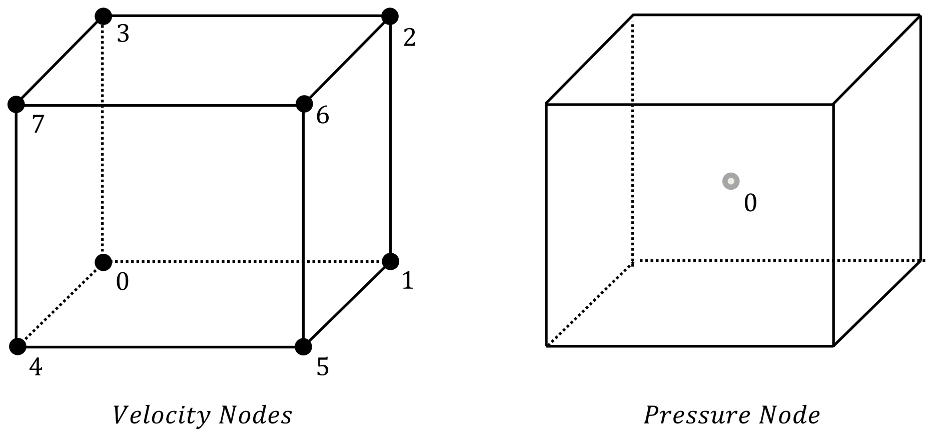


Figure 3.5 Hexahedral Element with 8 Velocity Nodes and 1 Pressure Node (Element Type 2).

After defining element types and the corresponding shape functions, discretization procedure continues by obtaining the residual of each differential equation, multiplying them with proper weight functions and equating their integrals over each element to zero. Starting with writing Eqn. (3.3) in the Cartesian coordinate system;

$$\text{x} - \text{Momentum}: \quad \begin{gathered} \frac{u^{n+1/2} - u^n}{\Delta t} + \left(u_0 \frac{\partial u}{\partial x} + v_0 \frac{\partial u}{\partial y} + w_0 \frac{\partial u}{\partial z} \right) \\ - \frac{\mu}{\rho} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial p}{\partial x} - f_x = 0 \end{gathered} \qquad 3.32$$

$$\text{y} - \text{Momentum}: \quad \frac{v^{n+1/2} - v^n}{\Delta t} + \left(u_0 \frac{\partial v}{\partial x} + v_0 \frac{\partial v}{\partial y} + w_0 \frac{\partial v}{\partial z}\right) - \frac{\mu}{\rho}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + \frac{1}{\rho}\frac{\partial p}{\partial y} - f_y = 0 \qquad 3.33$$

$$\text{z} - \text{Momentum}: \quad \frac{w^{n+1/2} - w^n}{\Delta t} + \left(u_0 \frac{\partial w}{\partial x} + v_0 \frac{\partial w}{\partial y} + w_0 \frac{\partial w}{\partial z}\right) - \frac{\mu}{\rho}\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) + \frac{1}{\rho}\frac{\partial p}{\partial z} - f_z = 0 \qquad 3.34$$

where variables u_0, v_0, w_0 are known velocity components from previous time step. They are used in order to linearize the non-linear advection term. Residuals of the aforementioned DEs are;

$$R(x) = \frac{\dot{u}}{\Delta t} + \left(u_0 \frac{\partial u}{\partial x} + v_0 \frac{\partial u}{\partial y} + w_0 \frac{\partial u}{\partial z}\right) - \frac{\mu}{\rho}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + \frac{1}{\rho}\frac{\partial p}{\partial x} - f_x \qquad 3.35$$

$$R(y) = \frac{\dot{v}}{\Delta t} + \left(u_0 \frac{\partial v}{\partial x} + v_0 \frac{\partial v}{\partial y} + w_0 \frac{\partial v}{\partial z}\right) - \frac{\mu}{\rho}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + \frac{1}{\rho}\frac{\partial p}{\partial y} - f_y \qquad 3.36$$

$$R(z) = \frac{\dot{w}}{\Delta t} + \left(u_0 \frac{\partial w}{\partial x} + v_0 \frac{\partial w}{\partial y} + w_0 \frac{\partial w}{\partial z}\right) - \frac{\mu}{\rho}\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) + \frac{1}{\rho}\frac{\partial p}{\partial z} - f_z \qquad 3.37$$

The $(u^{n+1/2} - u^n)$, $(v^{n+1/2} - v^n)$, $(w^{n+1/2} - w^n)$ term are replaced with $\dot{u}$, $\dot{v}$, $\dot{w}$ for the sake of simplicity.

Weighted residual methods' main principle is to minimize the residual in a weighted integral logic as demonstrated below;

$$\int_{\Omega} w(x)\, R(x)\, dx = 0 \qquad 3.38$$

$$\int_{\Omega} w(y)\, R(y)\, dy = 0 \qquad 3.39$$

$$\int_\Omega w(z)\, R(z)\, dz = 0 \qquad 3.40$$

Substituting Eqns. (3.35), (3.36) and (3.37) into Eqns. (3.38), (3.39) and (3.40);

$$\int_\Omega \left(w_x \frac{\dot{u}}{\Delta t} + w_x \left(u_0 \frac{\partial u}{\partial x} + v_0 \frac{\partial u}{\partial y} + w_0 \frac{\partial u}{\partial z} \right) - \frac{\mu}{\rho} w_x \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial w_x}{\partial x} p - w_x f_x \right) dx = 0 \qquad 3.41$$

$$\int_\Omega \left(w_y \frac{\dot{v}}{\Delta t} + w_y \left(u_0 \frac{\partial v}{\partial x} + v_0 \frac{\partial v}{\partial y} + w_0 \frac{\partial v}{\partial z} \right) - \frac{\mu}{\rho} w_y \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial w_y}{\partial y} p - w_y f_y \right) dy = 0 \qquad 3.42$$

$$\int_\Omega \left(w_z \frac{\dot{w}}{\Delta t} + w_z \left(u_0 \frac{\partial w}{\partial x} + v_0 \frac{\partial w}{\partial y} + w_0 \frac{\partial w}{\partial z} \right) - \frac{\mu}{\rho} w_z \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial w_z}{\partial z} p - w_z f_z \right) dz = 0 \qquad 3.43$$

The equation above is known as the residual statement of the differential equation where w_x, w_y, w_z are user selected weight functions.

When a C^0 continuous solution is utilized in the weighted residual statement, there will be a problem in the diffusion term since its second order derivatives cannot be computed properly. Therefore the differentiation requirements of unknown in the weighted residual statement should be lowered which can be achieved by applying integration by parts to the diffusion term of the equation.

$$\int_\Omega \frac{\mu}{\rho} w_x \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) dx = -\left(\frac{\mu}{\rho} \int_\Omega \frac{\partial w_x}{\partial x} \frac{\partial u}{\partial x} + \frac{\partial w_x}{\partial y} \frac{\partial u}{\partial y} + \frac{\partial w_x}{\partial z} \frac{\partial u}{\partial z} dx \right) + \int_\Gamma \frac{\mu}{\rho} w_x \left(n_x \frac{\partial u}{\partial x} + n_y \frac{\partial u}{\partial y} + n_z \frac{\partial u}{\partial z} \right) d\Gamma \qquad 3.44$$

The last term of equation above is called boundary integral which is a by-product of integration by parts. Boundary integral is evaluated at the boundaries (Γ) of the problem domain (Ω), where n_x, n_y, n_z are the x, y, z component of the unit outward normal of the boundary respectively. Boundary integrals that are the by-products of integration by parts include the following traction terms;

$$t_x = \frac{\mu}{\rho}\left(n_x \frac{\partial u}{\partial x} + n_y \frac{\partial u}{\partial y} + n_z \frac{\partial u}{\partial z}\right) \tag{3.45}$$

$$t_y = \frac{\mu}{\rho}\left(n_x \frac{\partial v}{\partial x} + n_y \frac{\partial v}{\partial y} + n_z \frac{\partial v}{\partial z}\right) \tag{3.46}$$

$$t_z = \frac{\mu}{\rho}\left(n_x \frac{\partial w}{\partial x} + n_y \frac{\partial w}{\partial y} + n_z \frac{\partial w}{\partial z}\right) \tag{3.47}$$

Applying integration by parts to the diffusion term of all the equations (Eqns. (3.41), (3.42) and (3.43));

$$\begin{gathered}\int_\Omega \left(w_x \frac{\dot{u}}{\Delta t} + w_x \left(u_0 \frac{\partial u}{\partial x} + v_0 \frac{\partial u}{\partial y} + w_0 \frac{\partial u}{\partial z} \right) + \frac{\mu}{\rho}\left(\frac{\partial w_x}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial w_x}{\partial y}\frac{\partial u}{\partial y} + \frac{\partial w_x}{\partial z}\frac{\partial u}{\partial z} \right) \right) dx \\ = \int_\Omega \left(-\frac{1}{\rho}\frac{\partial w_x}{\partial x} p \right) dx + \int_\Omega (w_x f_x) dx + \int_\Gamma w_x t_x d\Gamma \end{gathered} \tag{3.48}$$

$$\begin{gathered}\int_\Omega \left(w_y \frac{\dot{v}}{\Delta t} + w_y \left(u_0 \frac{\partial v}{\partial x} + v_0 \frac{\partial v}{\partial y} + w_0 \frac{\partial v}{\partial z} \right) + \frac{\mu}{\rho}\left(\frac{\partial w_y}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial w_y}{\partial y}\frac{\partial v}{\partial y} + \frac{\partial w_y}{\partial z}\frac{\partial v}{\partial z} \right) \right) dy \\ = \int_\Omega \left(-\frac{1}{\rho}\frac{\partial w_y}{\partial y} p \right) dy + \int_\Omega (w_y f_y) dy + \int_\Gamma w_y t_y d\Gamma \end{gathered} \tag{3.49}$$

$$\begin{gathered}\int_\Omega \left(w_z \frac{\dot{w}}{\Delta t} + w_z \left(u_0 \frac{\partial w}{\partial x} + v_0 \frac{\partial w}{\partial y} + w_0 \frac{\partial w}{\partial z} \right) + \frac{\mu}{\rho}\left(\frac{\partial w_z}{\partial x}\frac{\partial w}{\partial x} + \frac{\partial w_z}{\partial y}\frac{\partial w}{\partial y} + \frac{\partial w_z}{\partial z}\frac{\partial w}{\partial z} \right) \right) dz \\ = \int_\Omega \left(-\frac{1}{\rho}\frac{\partial w_z}{\partial z} p \right) dz + \int_\Omega (w_z f_z) dz + \int_\Gamma w_z t_z d\Gamma \end{gathered} \tag{3.50}$$

The above equation is called the weak form of the problem since it has lower differentiability requirements compared to the original weighted residual statements. By transferring a given DE into the weak form one can use C^0 continuous solution and also natural boundary conditions (NBC) will automatically be included into the formulation. This is a unique property of FEM.

C^0 continuous approximate solutions for velocity and pressure unknowns take the following form;

$$u^h(x,y,z) = \sum_{j=1}^{NENv} u_j\, S_j(x,y,z) \tag{3.51}$$

$$v^h(x,y,z) = \sum_{j=1}^{NENv} v_j\, S_j(x,y,z) \tag{3.52}$$

$$w^h(x,y,z) = \sum_{j=1}^{NENv} w_j\, S_j(x,y,z) \tag{3.53}$$

$$p^h(x,y) = \sum_{j=1}^{NENp} p_j\, \hat{S}_j(x,y) \tag{3.54}$$

where NENv and NENp are the number of velocity and pressure nodes over an element. Because NENv and NENp are different for the used elements, different shape fuctions are used for velocity and pressure components and they are denoted as S and $\hat{S}$.

Substituting u^h, v^h, w^h, p^h into the elemental weak forms (Eqns. (3.48), (3.49) and (3.50));

$$\begin{aligned}
&\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(w_x\frac{\dot{u}_j\, S_j}{\Delta t}\right)d\Omega\right)+\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(w_x\left(u_0u_i\frac{\partial S_j}{\partial x}+v_0u_i\frac{\partial S_j}{\partial y}+w_0u_i\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)\\
&\quad+\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(\frac{\mu}{\rho}\left(\frac{\partial w_x}{\partial x}u_i\frac{\partial S_j}{dx}+\frac{\partial w_x}{\partial y}u_i\frac{\partial S_j}{dy}+\frac{\partial w_x}{\partial z}u_i\frac{\partial S_j}{dz}\right)\right)d\Omega\right)\\
&\quad=\sum_{j=1}^{NENp}\left(\int_{\Omega^e}\left(-\frac{1}{\rho}\frac{\partial w_x}{\partial x}\,\hat{S}_j\right)d\Omega\right)+\int_{\Omega}(w_xf_x)d\Omega+\int_{\Gamma^e}w_xt_xd\Gamma
\end{aligned} \tag{3.55}$$

$$\begin{aligned}
&\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(w_y\frac{\dot{v}_j\, S_j}{\Delta t}\right)d\Omega\right)+\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(w_y\left(u_0v_i\frac{dS_j}{dx}+v_0v_i\frac{dS_j}{dy}+w_0v_i\frac{dS_j}{dz}\right)\right)d\Omega\right)\\
&\quad+\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(\frac{\mu}{\rho}\left(\frac{\partial w_y}{\partial x}v_i\frac{\partial S_j}{dx}+\frac{\partial w_y}{\partial y}v_i\frac{\partial S_j}{dy}+\frac{\partial w_y}{\partial z}v_i\frac{\partial S_j}{dz}\right)\right)d\Omega\right)\\
&\quad=\sum_{j=1}^{NENp}\left(\int_{\Omega^e}\left(-\frac{1}{\rho}\frac{\partial w_y}{\partial y}\,\hat{S}_j\right)d\Omega\right)+\int_{\Omega}\left(w_yf_y\right)d\Omega+\int_{\Gamma^e}w_yt_yd\Gamma
\end{aligned} \tag{3.56}$$

$$\begin{aligned}
&\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(w_z\frac{\dot{w}_j\, S_j}{\Delta t}\right)d\Omega\right)+\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(w_z\left(u_0w_i\frac{\partial S_j}{dx}+v_0w_i\frac{\partial S_j}{dy}+w_0w_i\frac{\partial S_j}{dz}\right)\right)d\Omega\right)\\
&\quad+\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(\frac{\mu}{\rho}\left(\frac{\partial w_z}{\partial x}w_i\frac{\partial S_j}{dx}+\frac{\partial w_z}{\partial y}w_i\frac{\partial S_j}{dy}+\frac{\partial w_z}{\partial z}w_i\frac{\partial S_j}{dz}\right)\right)d\Omega\right)\\
&\quad=\sum_{j=1}^{NENp}\left(\int_{\Omega^e}\left(-\frac{1}{\rho}\frac{\partial w_z}{\partial z}\,\hat{S}_j\right)d\Omega\right)+\int_{\Omega}(w_zf_z)d\Omega+\int_{\Gamma^e}w_zt_zd\Gamma
\end{aligned} \tag{3.57}$$

Galerkin FEM (GFEM) is the most common variation of FEM, which is also utilized in the present study. The weight functions are selected to be the same as shape functions at GFEM, it can be demonstrated as;

$$w(x) = S_i(x) \tag{3.58}$$

With using GFEM, Eqns. (3.55), (3.56) and (3.57) are transformed into these equations below;

$$\begin{aligned}\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(S_i\frac{S_j}{\Delta t}\right)d\Omega\right)\dot{u}_j + \sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(S_i\left(u_0\frac{\partial S_j}{\partial x}+v_0\frac{\partial S_j}{\partial y}+w_0\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)u_j \\ +\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(\frac{\mu}{\rho}\left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x}+\frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y}+\frac{\partial S_i}{\partial z}\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)u_j \\ =\sum_{j=1}^{NENp}\left(\int_{\Omega^e}\left(-\frac{1}{\rho}\hat{S}_j\frac{\partial S_i}{\partial x}\right)d\Omega\right)p_j+\int_{\Omega}(S_i f_x)d\Omega+\int_{\Gamma^e}S_i t_x d\Gamma\end{aligned} \tag{3.59}$$

$$\begin{aligned}\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(S_i\frac{S_j}{\Delta t}\right)d\Omega\right)\dot{v}_j + \sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(S_i\left(u_0\frac{\partial S_j}{\partial x}+v_0\frac{\partial S_j}{\partial y}+w_0\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)v_j \\ +\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(\frac{\mu}{\rho}\left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x}+\frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y}+\frac{\partial S_i}{\partial z}\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)v_j \\ =\sum_{j=1}^{NENp}\left(\int_{\Omega^e}\left(-\frac{1}{\rho}\hat{S}_j\frac{\partial S_i}{\partial y}\right)d\Omega\right)p_j+\int_{\Omega}(S_i f_y)d\Omega+\int_{\Gamma^e}S_i t_y d\Gamma\end{aligned} \tag{3.60}$$

$$\begin{aligned}\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(S_i\frac{S_j}{\Delta t}\right)d\Omega\right)\dot{w}_j + \sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(S_i\left(u_0\frac{\partial S_j}{\partial x}+v_0\frac{\partial S_j}{\partial y}+w_0\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)w_j \\ +\sum_{j=1}^{NENv}\left(\int_{\Omega^e}\left(\frac{\mu}{\rho}\left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x}+\frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y}+\frac{\partial S_i}{\partial z}\frac{\partial S_j}{\partial z}\right)\right)d\Omega\right)w_j \\ =\sum_{j=1}^{NENp}\left(\int_{\Omega^e}\left(-\frac{1}{\rho}\hat{S}_j\frac{\partial S_i}{\partial z}\right)d\Omega\right)p_j+\int_{\Omega}(S_i f_z)d\Omega+\int_{\Gamma^e}S_i t_z d\Gamma\end{aligned} \tag{3.61}$$

It can be seen that other than pressure gradient and body force terms all terms are same for x, y, z momentum equations. The compact matrix notation may help simplifying the Eqns. (3.59), (3.60) and (3.61);

$$\begin{aligned}[M]^e\frac{\{U^{n+\frac{1}{2}}\}^e-\{U^n\}^e}{\Delta t}+[A(U^n)]^e\{U^{n+\frac{1}{2}}\}^e+[K]^e\{U^{n+\frac{1}{2}}\}^e \\ =-[G]_x^e[P^n]^e+\{F^n\}_x^e\end{aligned} \tag{3.62}$$

$$[M]^e \frac{\{U^{n+\frac{1}{2}}\}^e - \{U^n\}^e}{\Delta t} + [A(U^n)]^e\{U^{n+\frac{1}{2}}\}^e + [K]^e\{U^{n+\frac{1}{2}}\}^e = -[G]_y^e[P^n]^e + \{F^n\}_y^e \qquad 3.63$$

$$[M]^e \frac{\{U^{n+\frac{1}{2}}\}^e - \{U^n\}^e}{\Delta t} + [A(U^n)]^e\{U^{n+\frac{1}{2}}\}^e + [K]^e\{U^{n+\frac{1}{2}}\}^e = -[G]_z^e[P^n]^e + \{F^n\}_z^e \qquad 3.64$$

where;

$$[M]^e = \sum_{j=1}^{NENv} \left(\int_{\Omega^e} \left(S_i \frac{S_j}{\Delta t} \right) d\Omega \right) \qquad 3.65$$

$$[A(U^n)]^e = \sum_{j=1}^{NENv} \left(\int_{\Omega^e} \left(S_i \left(u_0 \frac{\partial S_j}{\partial x} + v_0 \frac{\partial S_j}{\partial y} + w_0 \frac{\partial S_j}{\partial z} \right) \right) d\Omega \right) \qquad 3.66$$

$$[K]^e = \sum_{j=1}^{NENv} \left(\int_{\Omega^e} \left(\frac{\mu}{\rho} \left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x} + \frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y} + \frac{\partial S_i}{\partial z}\frac{\partial S_j}{\partial z} \right) \right) d\Omega \right) \qquad 3.67$$

$$[G]_x^e = \sum_{j=1}^{NENp} \left(\int_{\Omega^e} \left(-\frac{1}{\rho} \hat{S}_j \frac{\partial S_i}{\partial x} \right) d\Omega \right) \qquad 3.68$$

$$[G]_y^e = \sum_{j=1}^{NENp} \left(\int_{\Omega^e} \left(-\frac{1}{\rho} \hat{S}_j \frac{\partial S_i}{\partial y} \right) d\Omega \right) \qquad 3.69$$

$$[G]_z^e = \sum_{j=1}^{NENp} \left(\int_{\Omega^e} \left(-\frac{1}{\rho} \hat{S}_j \frac{\partial S_i}{\partial z} \right) d\Omega \right) \qquad 3.70$$

$$\{F^n\}_{x,y,z}^e = \int_{\Omega} (S_i f_{x,y,z}) d\Omega + \int_{\Gamma^e} S_i t_{x,y,z} d\Gamma \qquad 3.71$$

The global versions of these stiffness matrices are nothing but assembled version of their elemental counterparts. Size of the global stiffness matrices, $[M]$, $[A(U)]$ and $[K]$ are NNxNN; $[G]$ are NNxNNp; $\{F\}$ are NNx1, where NN is number of nodes (velocity nodes), NNp is number of pressure nodes.

Galerkin FEM discretization of Eqn. (3.4) and Eqn. (3.5) are very similar to do Eqn. (3.3).

3.5 Solution of the Energy Conservation Equation

Galerkin FEM discretization of the continuity and the momentum equations are given in the previous section. For non-isothermal problems that include heat transfer effects also the energy equation needs to be solved. In this study only steady state non-isothermal problems are solved and the dependency of fluid properties with temperature is not considered. For such cases, the energy conservation equation becomes decoupled from the continuity and momentum conservation equations and it can be solved by itself after obtaining the velocity field. The equation that needs to be solved to get the temperature field in an incompressible flow is given as

$$\rho c_p\left(\vec{V}\cdot\nabla\right)T = k\nabla^2 T + \Phi \qquad 3.72$$

where ρ, c_p and k are the constant viscosity, specific heat and conductivity of the fluid, respectively. Velocity vector $\vec{V}$ of Eqn. (3.72) is considered to be known. Φ is the viscous dissipation term, which is negligibly small in many practical applications and it is also neglected in the current study for simplicity.

The developed solver supports conjugate heat transfer problems where heat transfer takes place over both fluid and solid regions. For such cases, each element is labeled to be either a fluid element or a solid element and for the ones corresponding to the solid regions velocity vector is set to zero. All fluid and solid elements are then assembled to form a single linear algebraic system, the solution of which gives the temperature field of the whole problem domain.

Nodal temperature unknowns are stored at the same points as the velocity unknowns. Galerkin FEM discretization of the energy conservation equation results in the following elemental system

$$[K^e\ ]\{T^e\} = \{B^e\} \qquad 3.73$$

where the elemental coefficient matrix is given by

$$\begin{aligned} K_{ij}^e = \int_{\Omega^e} \Bigg[\rho c_p S_i \left(u\frac{\partial S_j}{\partial x} + v\frac{\partial S_j}{\partial y} + w\frac{\partial S_j}{\partial z}\right) \\ + k\left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x} + \frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y} + \frac{\partial S_i}{\partial z}\frac{\partial S_j}{\partial z}\right)\Bigg] d\Omega \end{aligned} \qquad 3.74$$

and the elemental boundary integral vector is

$$B_i^e = \int_{\Gamma^e} S_i \left(\frac{\partial T}{\partial x} n_x + \frac{\partial T}{\partial y} n_y + \frac{\partial T}{\partial z} n_z \right) d\Gamma \qquad 3.75$$

For a steady state problem solution of the energy equation takes a negligibly small time compared to the solution of the continuity and the momentum equations. Therefore, no effort is spent to parallelize it.

CHAPTER 4

VERIFICATION OF THE DEVELOPED SOLVER

The developed solver is tested for accuracy using 5 problems that either have known analytical solutions or previously studied numerically or experimentally. The first benchmark problem is selected to be the flow inside a lid driven cavity. The problem is studied at three different Reynolds numbers and the results are compared with available numerical results in the literature. The second test problem is the flow inside a square duct with a 90^o bend. This one is more challenging than the first one in terms of the three dimensionality of the created flow structures. The results are again compared with an existing numerical study. As the third problem hydrodynamically and thermally developing flow inside a square duct is solved and analytically known fully developed Nusselt number is used to check the correctness of the solution. Fourth problem considered is again for a duct flow, but this time the cross section dimensions are in the order of microns. The results are compared with the available experimental values. The final test is selected to be a conjugate heat transfer problem, for which the problem geometry is a pipe.

4.1 Lid Driven Cavity Flow

This is one the most frequently used benchmark problems to verify newly developed incompressible flow solvers. It is appealing due to its simple geometry that is suitable for a structured mesh and all Dirichlet boundary conditions. Problem domain is a $1 \times 1 \times 1$ sized cube as seen in Fig. 4.1. Top face ($z = 1$) of the cube, which is known as its lid, is pulled in the x direction with a speed of 1.0, while other faces are kept stationary. No slip boundary condition is applied on all 6 faces. Due to the lack of inflow/outflow boundaries, the uniqueness of the pressure field is controlled by specifying 0 pressure, at the center of the bottom wall (point A of Fig. 4.1). Density of the fluid is taken to be 1.0 and the computations are performed for Reynolds numbers of 100, 400 and 1000. The desired Re values are obtained by using dynamic viscosity values of 0.01, 0.0025 and 0.001.

The results are compared with Yang et al.'s [87] numerical study, which used an implicit weighted ENO scheme on a non-uniform grid of $33 \times 33 \times 33$ nodes. A similar non-uniform, structured mesh with 27000 $(30 \times 30 \times 30)$ elements is used in this study. The elements used are the high order ones (see Fig. 4.2), which results in $31^3 = 29{,}791$ pressure nodes and $61^2 = 226{,}981$ velocity nodes. Total solved number of unknowns is calculated as $710{,}734$. Fig. 4.2 shows the distribution of the velocity nodes on the faces of the cube.

For all three Reynolds numbers, time step is used as 10^{-3}. Presented results show the steady state solution, which is determined by the continuous checking of the variations of the unknowns at selected monitoring points. Fig. 4.3 shows the velocity component in the x direction along line AB of Fig. 4.1 for $Re = 100$. Similarly it shows the velocity component in the z direction along line CD of Fig. 4.1. Fig. 4.4 and Fig. 4.5 are similar plots for $Re = 400$ and $Re = 1000$, respectively. As seen in these figures, current results show good agreement with those of the reference study.

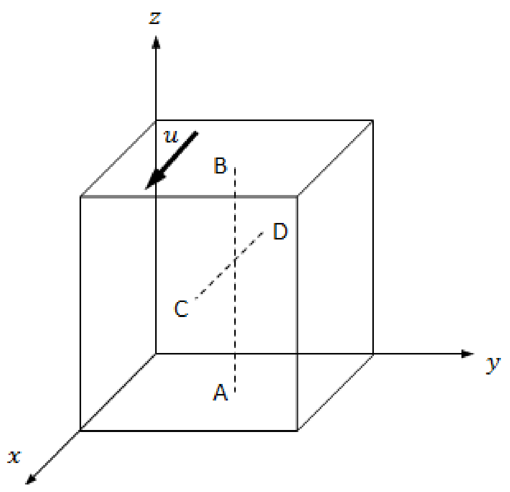


Figure 4.1 Lid Driven Cavity Flow.

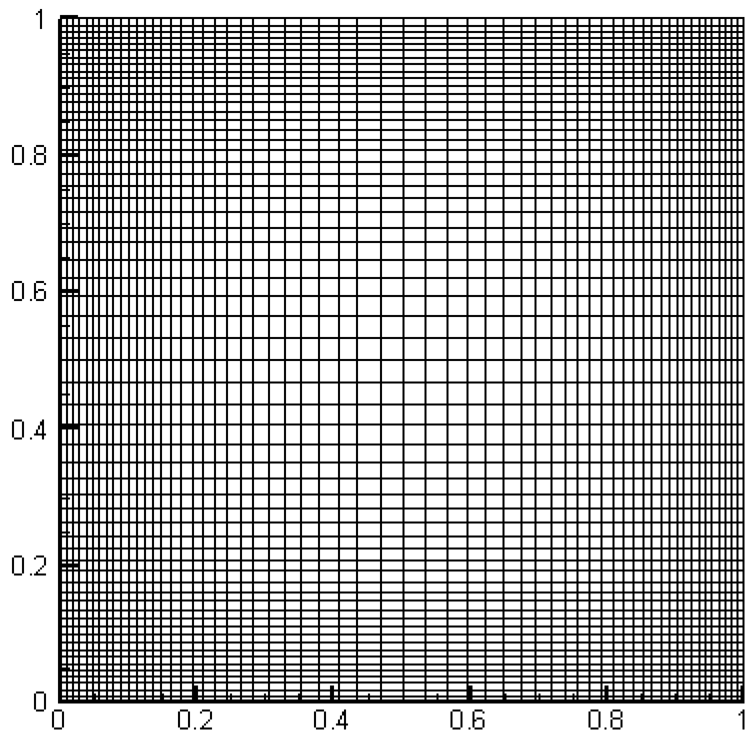
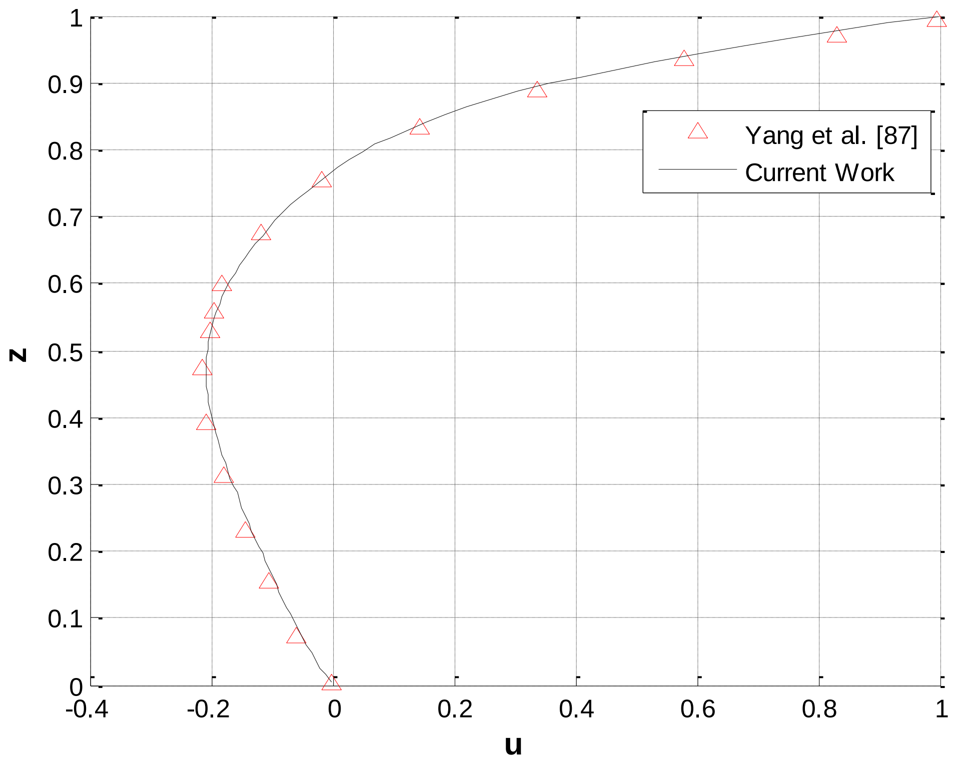


Figure 4.2 Non-Uniform Distribution of 61×61 Velocity Nodes on the Faces of the Lid Driven Cavity.

4.2 Bending Square Duct Flow

The second benchmark test is the flow inside a square duct with a 90° bend. Results obtained are compared with those presented by Yang et al. [87], which is the study already used as the reference for the first test case. Problem geometry can be seen in Fig. 4.6. The duct has two straight sections of length 5 units each, with a square cross-sectional area of 1×1 units. These straight sections have a 90° bend between them, with an inner radius (r_i) of 1.8 and outer radius (r_o) of 2.8. Fully developed inflow velocity profile prescribed at the inlet boundary is given by Eqn. (4.1), which corresponds to a maximum centerline velocity of 2.25 and average velocity of 1.0. Pressure at the midpoint of the exit plane is set to zero. Density of the fluid is taken as 1.0 and by using a dynamic viscosity of 0.0012658, the required Reynolds number of 790 is obtained.

$$u_{inlet} = 36(y - y^2)(z - z^2) \qquad 4.1$$



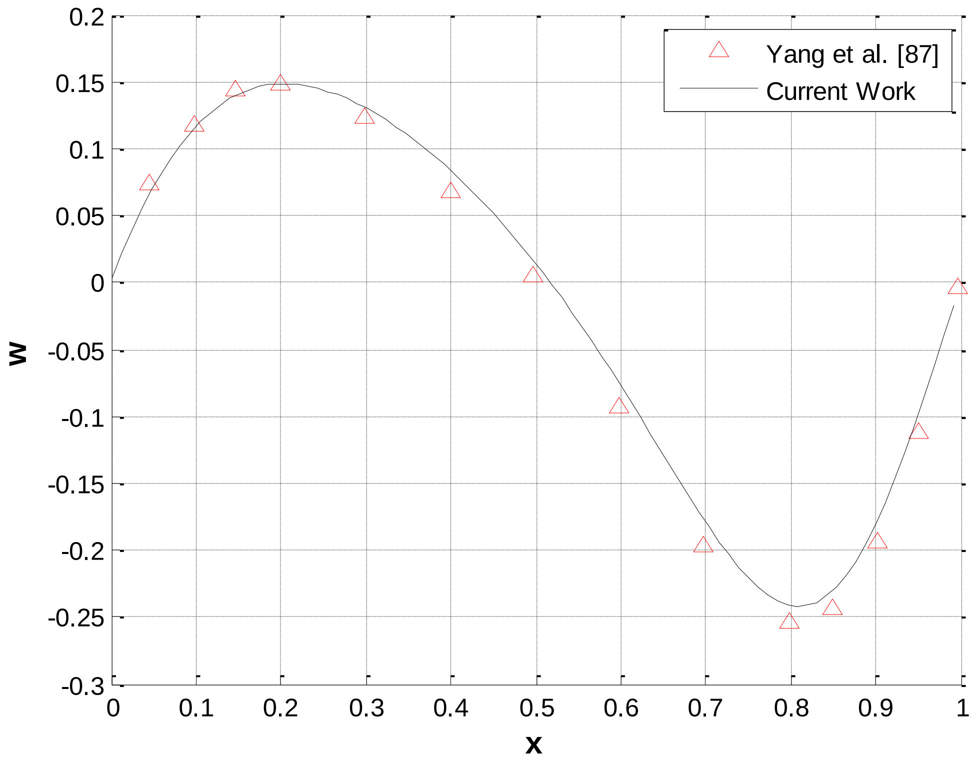
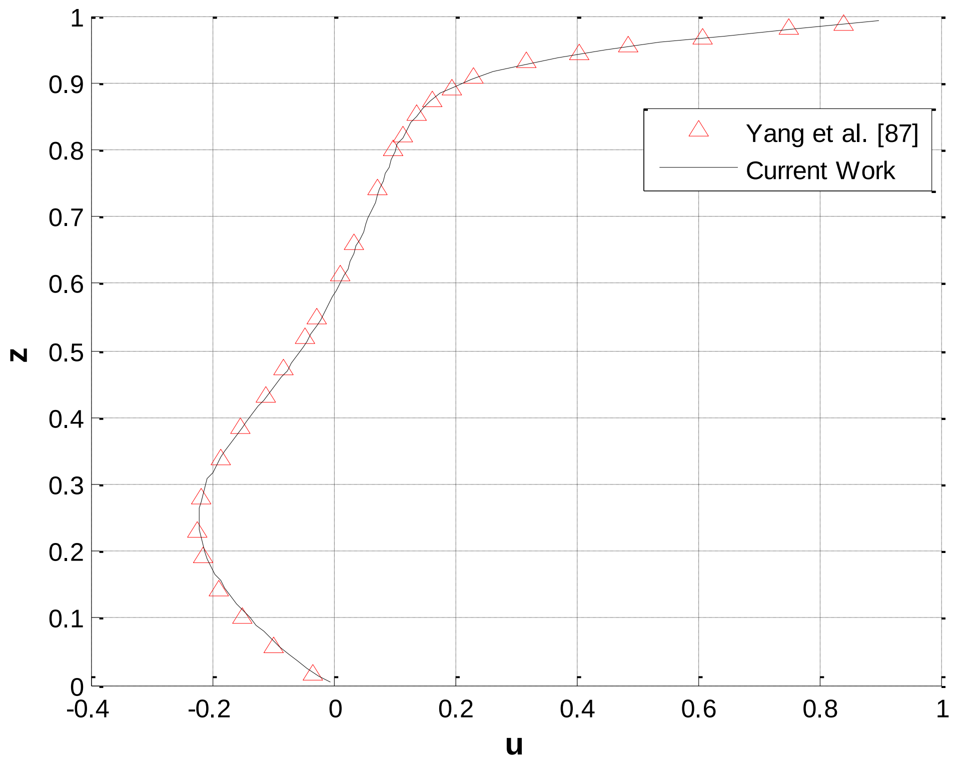


Figure 4.3 Velocity Components in the x and z Directions Along Lines AB and CD of the Lid Driven Cavity. $Re = 100$.



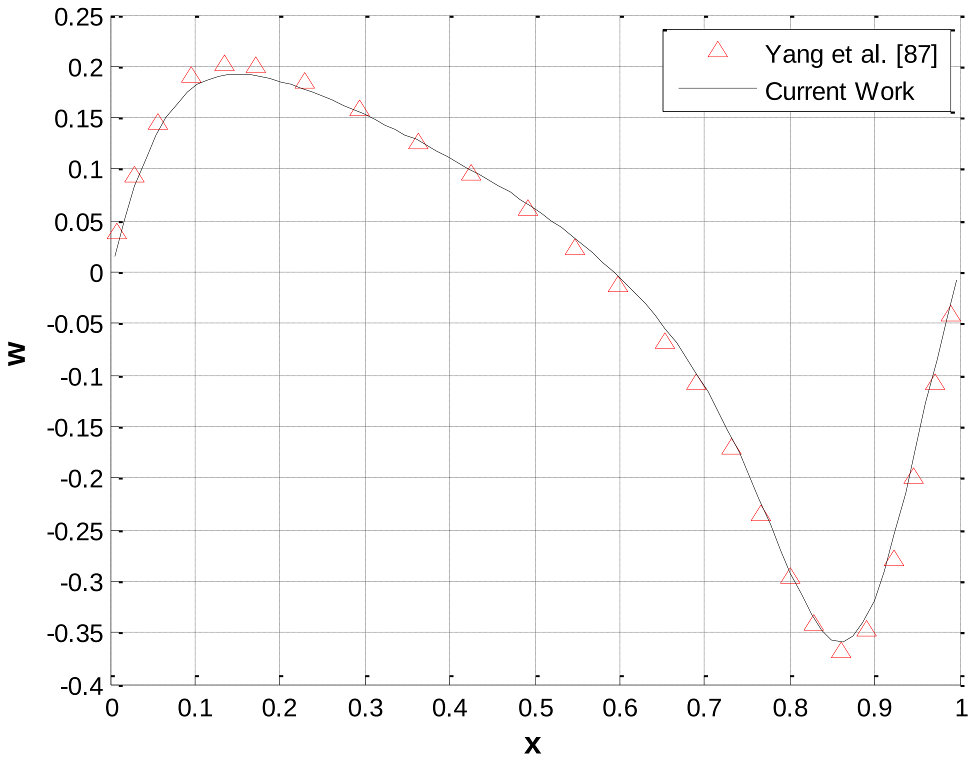
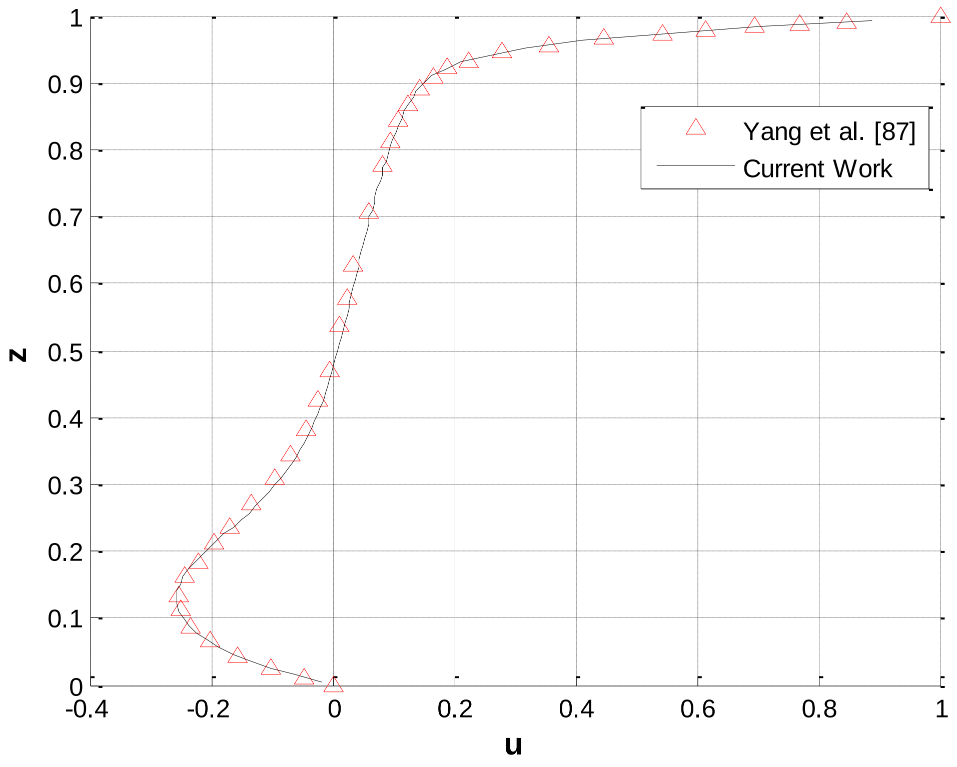


Figure 4.4 Velocity Components in the x and z Directions Along Lines AB and CD of the Lid Driven Cavity. $Re = 400$.



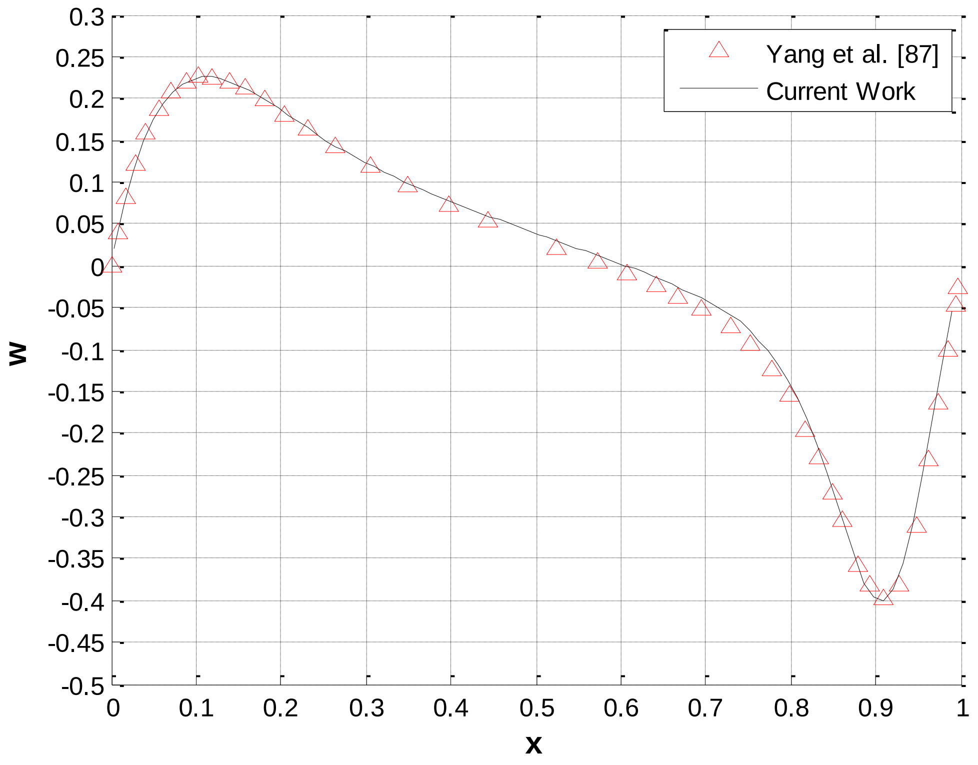


Figure 4.5 Velocity Components in the x and z Directions Along Lines AB and CD of the Lid Driven Cavity. $Re = 1000$.

The results of the bending square duct problem are obtained with a non-uniform, structured mesh of 53,361 ($49 \times 33 \times 33$) elements, which is very similar to the one used by Yang et al. [87]. As seen in Fig. 4.7 elements are finer near the duct walls. Also the elements are concentrated near the bending part. Element type 1 with 8 pressure nodes and 27 velocity nodes are used, resulting in a total of 57,800 pressure and 444,441 velocity nodes. Total number of unknowns for this problem is 1,391,123.

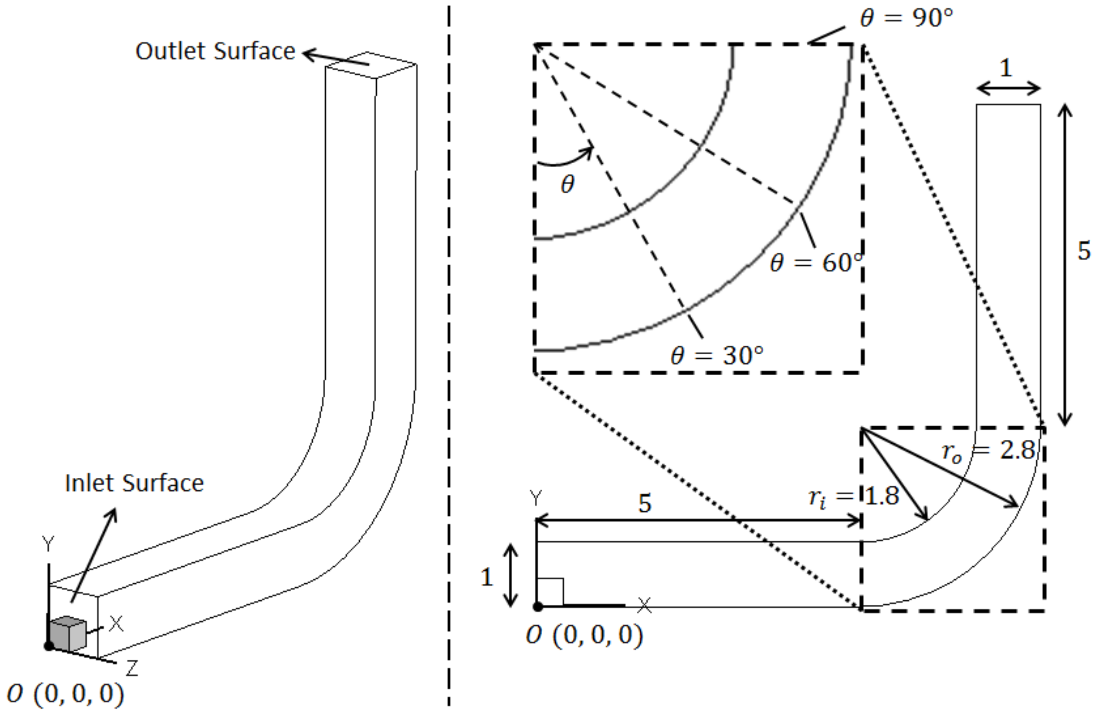
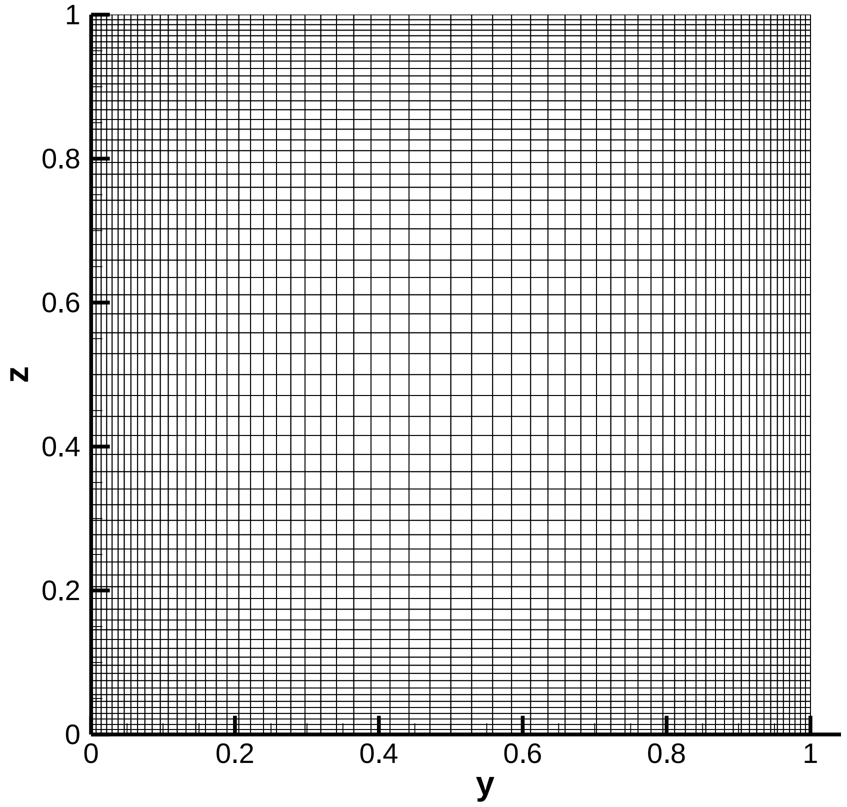


Figure 4.6 The Geometry of the 90° Bending Square Duct.

The stream-wise velocity profiles along three different lines at $z = 0.5$ plane of the bending part of the duct are shown in Fig 4.8. Location of $\theta = 30°, 60°$ and $90°$ planes that are referred in Fig. 4.8 can be seen in Fig. 4.6. Results obtained are in general agreement with the reference study, with small deviations. Contours of the velocity component in the y direction at four different planes after the bending part are shown in Fig. 4.9. The secondary flow, which is known to develop in bending ducts can clearly be seen.



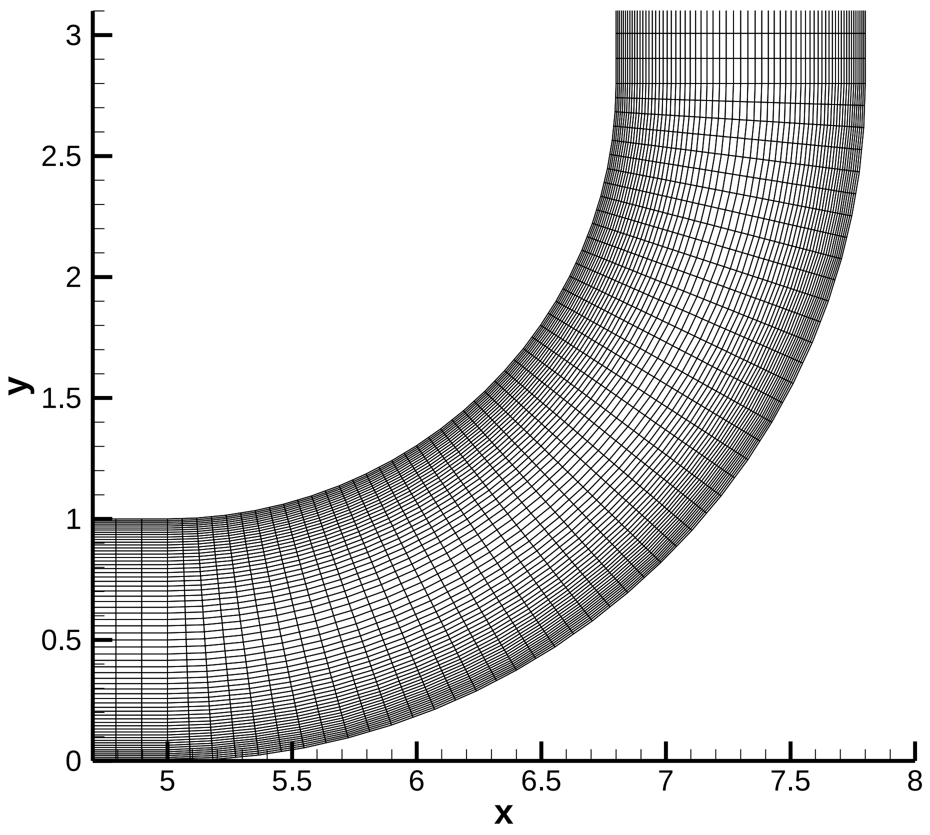
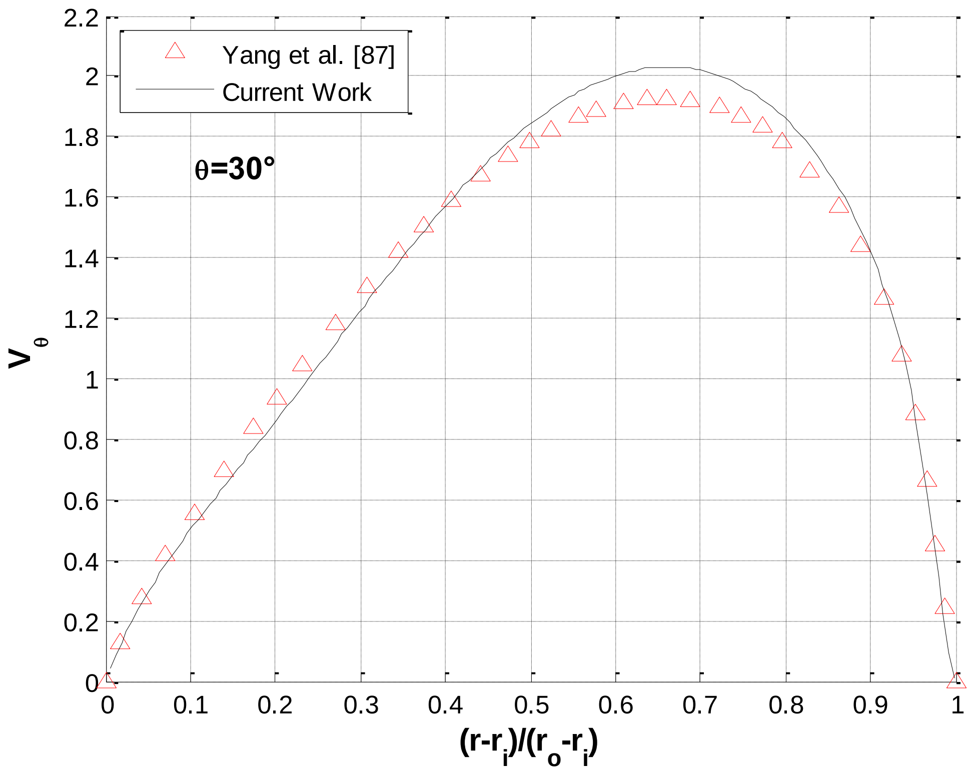


Figure 4.7 Non-Uniform Distribution of 67 × 67 Velocity Nodes on the Inlet Plane and Bend Part of the Bending Square Duct.



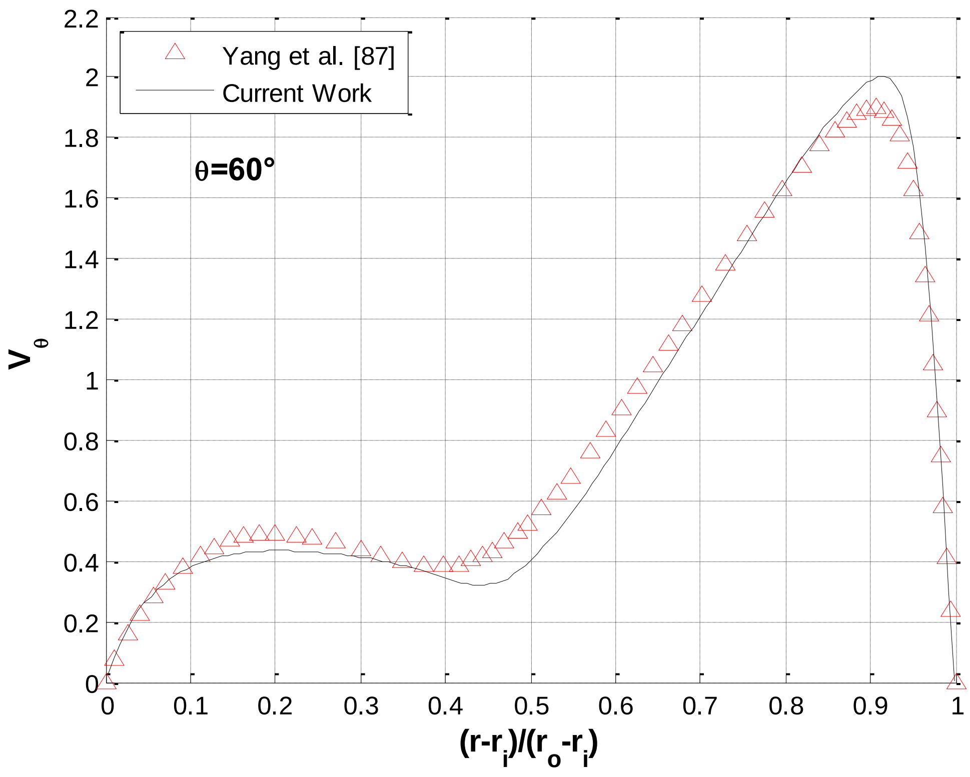


Figure 4.8 Stream-wise Velocity Profiles (V_θ) at the Intersection of $z = 0.5$ Plane and $\theta = 30°, 60°$ and 90° Planes of the Bending Duct.

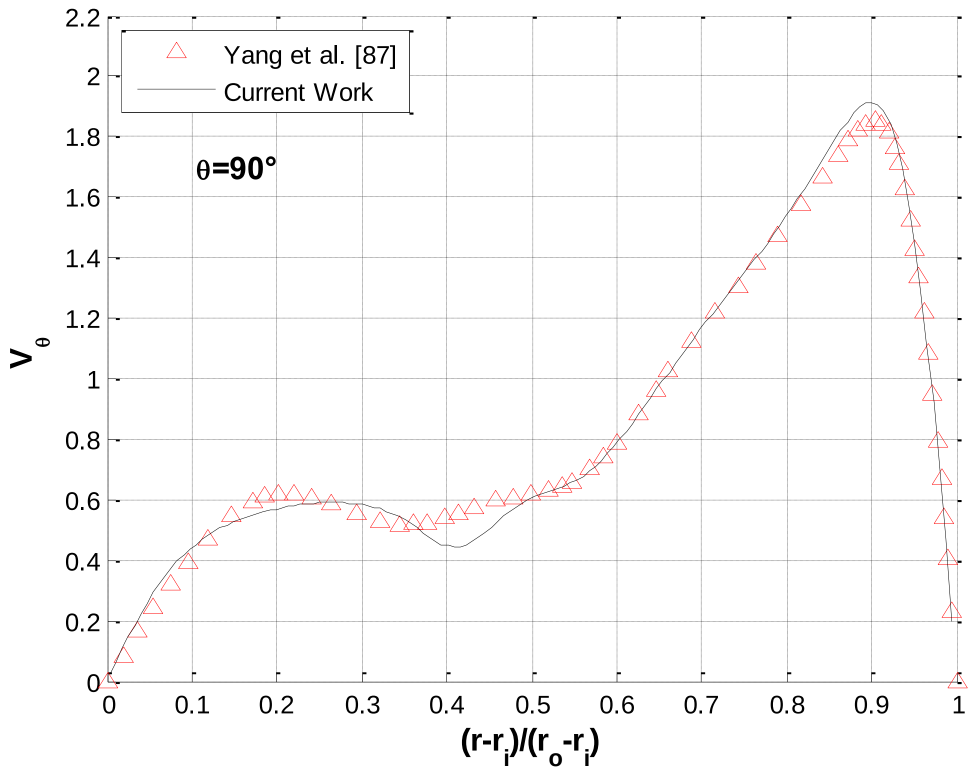


Figure 4.8 (continued) Stream-wise Velocity Profiles (V_θ) at the Intersection of $z = 0.5$ Plane and $\theta = 30°, 60°$ and $90°$ Planes of the Bending Duct.

4.3 Hydrodynamically and Thermally Developing Flow in a Square Duct

Problem domain, shown in Fig. 4.10, is a duct of 1×1 square cross-section, with a length of 10 units. Uniform velocity of magnitude 1.0 is specified at the inlet. Pressure is set to zero at the mid-point of the exit plane. Temperature of the incoming fluid is taken as 1 and wall temperatures are fixed at zero. To study the case of $Re = 100$ and $Pr = 1$, dynamic viscosity, specific heat and thermal conductivity of the fluid are taken as 1 and its density is selected to be 0.01.

To check the solution's correctness, analytically known Nusselt number (Nu) value of 2.98 given for the thermally fully developed region of the duct is used [88]. Nusselt number is defined as

$$Nu = \frac{hD_h}{k} \tag{4.2}$$

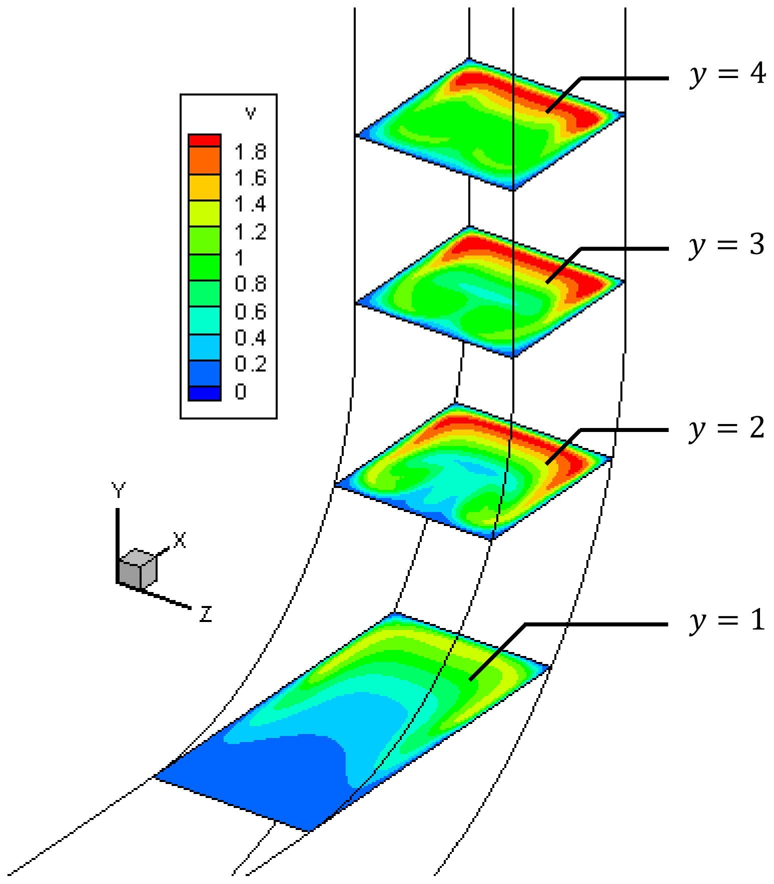


Figure 4.9 Contours of Velocity in the y Direction at Four Different Planes.

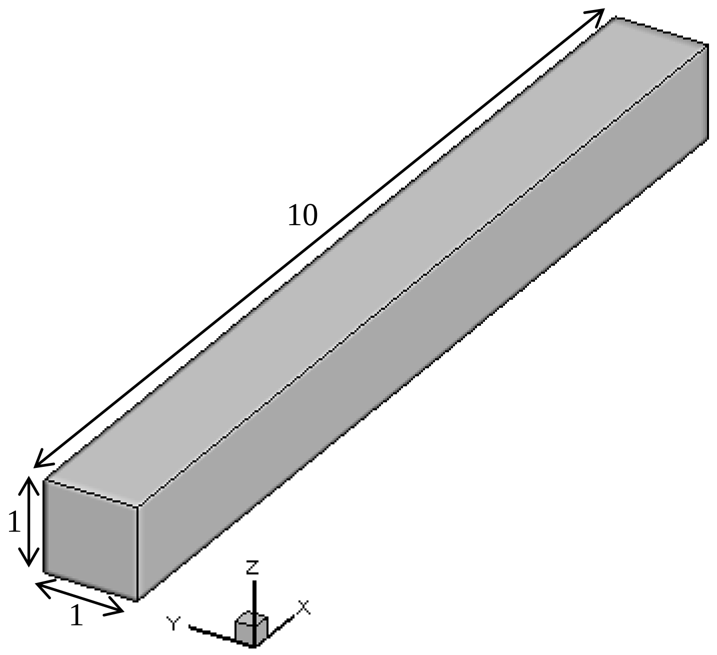


Figure 4.10 Geometry of the Square Duct.

where h is the convective heat transfer coefficient and D_h is the hydraulic diameter of the duct, which is 1 for the problem being solved. h is needed in order to calculate Nu. Average heat transfer coefficient ($\bar{h}$) between two cross sections (x_1, x_2) can be calculated using [89]

$$\frac{T_s - T_m(x_2)}{T_s - T_m(x_1)} = \exp\left(-\frac{P(x_2 - x_1)}{\dot{m}c_p}\bar{h}\right) \qquad 4.3$$

where $\dot{m}$ is the mass flow rate, P is the perimeter of the duct, T_s is the constant wall temperature and T_m is the bulk fluid temperature defined as

$$T_m = \frac{1}{Q}\int_{\mathrm{A}} u\, T\, d\mathrm{A} \qquad 4.4$$

where u is velocity component in the stream-wise direction and Q is volumetric flow rate.

Because the analytically calculated Nu values are valid for the thermally fully developed region, x_1 and x_2 of Eqn. (4.3) have to be selected in this region. The following thermal entrance length (L_T) estimation can be used for this purpose;

$$L_T = 0.05(Re)(D_h)(Pr) \qquad 4.5$$

which gives the value of 5 for the problem being studied. Accordingly x_1 and x_2 values are selected as 7 and 9, respectively. After obtaining the velocity and temperature fields using the developed solver, $T_m(x_1)$ and $T_m(x_2)$ are calculated using Eqn. (4.4) and these values are used in Eqn. (4.3) to find $\bar{h}$. Finally, Nusselt number of the thermally fully developed region is obtained using Eqn. (4.2). These are repeated for 4 different meshes and the Nu values obtained are shown in Table 4.1.

As seen from Table 4.1 calculated Nu values approach to the analytical value of 2.98 [88] as the mesh gets finer. For the $25 \times 25 \times 100$ grid, exact value is obtained. As an additional result temperature contours at the mid cross-section of the duct is given in Fig. 4.11.

Table 4.1 Nu Values for the Thermally Fully Developed Region of the Square Duct Obtained with 4 Different Grids.

Grid	Nu
$10 \times 10 \times 40$	2.997
$15 \times 15 \times 60$	2.988
$20 \times 20 \times 80$	2.987
$25 \times 25 \times 100$	2.980

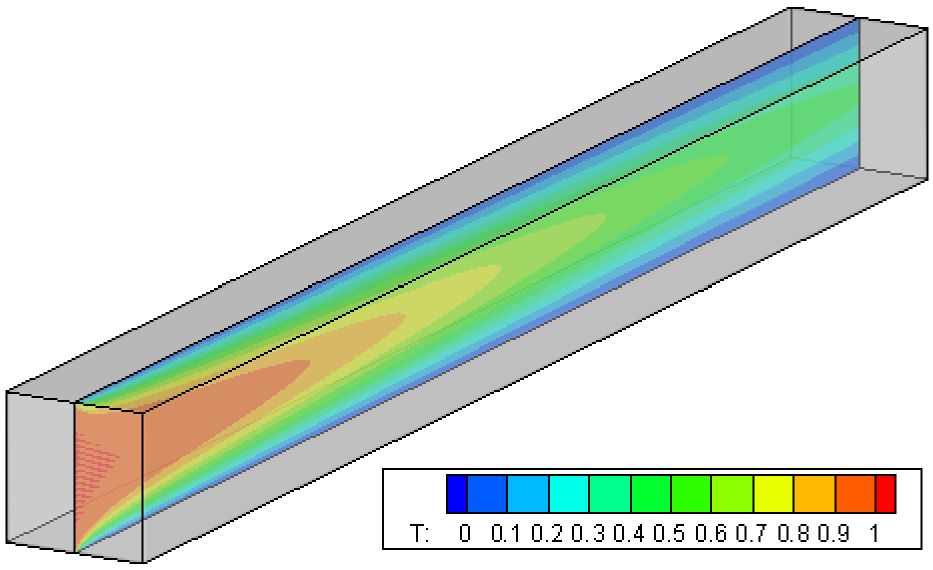


Figure 4.11 Temperature Contours at the Mid Cross-Section of the Square Duct.

4.4 Heat Transfer in a Micro Channel

This problem is based on the work of Lee and Garimella, which includes both numerical and experimental results [90]. Problem geometry that can be seen in Fig. 4.12 is a rectangular channel with dimensions of 25.4 mm $\times$ 194 μm $\times$ 884 μm. Similar to the reference numerical study, fully developed velocity profile is provided at the inlet section, geometric details of which are given in Fig. 4.12. Eqn. (4.7), suggested by Natarajan and Lakshmanan [91], is used to calculate the inlet velocity profile.

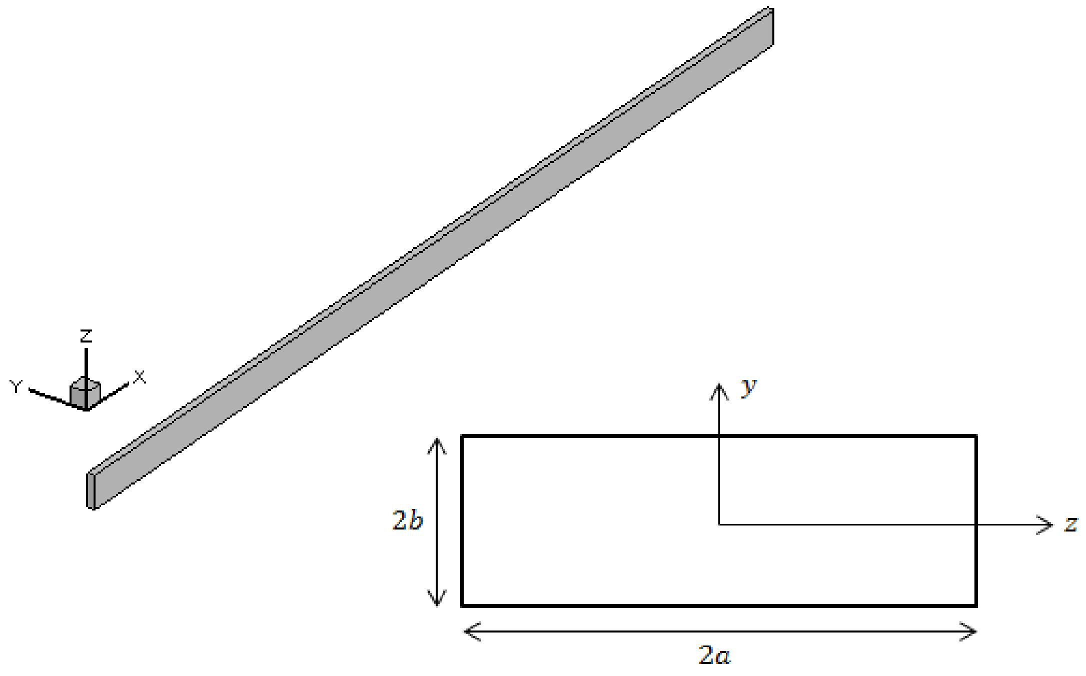


Figure 4.12 Geometry of the Micro Channel and Details of its Inlet Plane.

$$u = u_{max}\left[1-\left(\frac{y}{b}\right)^n\right]\left[1-\left(\frac{z}{a}\right)^n\right]$$

$$u_{max} = u_{avg}\left(\frac{m+1}{m}\right)\left(\frac{n+1}{n}\right) \qquad 4.7$$

$$\alpha = \frac{b}{a}, m = 1.7 + 0.5\alpha^{-1.4}, n = \begin{cases} 2 & \text{for } \alpha \le 1/3 \\ 2 + 0.3\left(\alpha - \frac{1}{3}\right) & \text{for } \alpha \ge 1/3 \end{cases}$$

Simulations are done for three different Reynolds numbers, based on the hydraulic diameter of the duct, which is 318 μm. Used Re values, as well as the corresponding average and maximum inlet velocities are given in Table 4.2. The working fluid is water and its properties at 300 $°K$ are used as $\rho = 997\ kg/m^3$, $\mu = 10^{-3}\ Pa\ s$, $c_p = 4181\ J/(kgK)$ and $k = 0.61\ W/(mK)$. Temperature at the inlet is taken to be constant as 295 $°K$ and wall temperatures are assigned as 350 $°K$.

Table 4.2 Three Different Reynolds Numbers and the Corresponding Average and Maximum Velocities at the Inlet of the Micro Channel.

Re	u_{avg} [m/s]	u_{max} [m/s]
500	1.577	2.748
750	2.365	4.123
1000	3.154	6.184

For the numerical simulations of the reference study, commercial CFD software Fluent was used and only one quarter of the whole problem domain is solved. Unfortunately this simplification was not possible for the developed solver due to its lack of support for the symmetry boundary condition. For the whole domain a mesh with $50 \times 16 \times 40 = 32{,}000$ elements is used. Similar to the previous problems the mesh is structured and non-uniform, getting finer close to the walls. Using the first element type total number of unknowns turned out to be $952{,}107$.

Average Nusselt numbers of the duct are calculated as explained in the previous section. Results obtained for three different Reynolds numbers are illustrated in Fig. 4.13 and compared with the experimental and numerical results of Lee and Garimella [90]. As seen in the figure the current results are close to the numerical results of the reference, but considerable deviation is seen between the numerical and experimental results due to the simplifications on the boundaries.

For the Re values investigated, temperature contours at $y = 0$ and $z = 0$ planes are shown in Fig. 4.14. As seen, temperature gradients near the walls are getting sharper as Re increases, making it harder to resolve these regions accurately.

4.5 Conjugate Heat Transfer in a Tube

As the last test problem, conjugate heat transfer on a thick walled tube with constant outer wall temperature is considered. Numerical solution of Zhang et al. [92] is used for comparison. Inner radius of the tube is 1.0 and its wall thickness is 0.84. For $Re = 50$, for which the solution is obtained, hydrodynamically fully developed flow occurs at $0.05(Re)(D) = 5$. Tube length is selected as 25, which is 5 times the

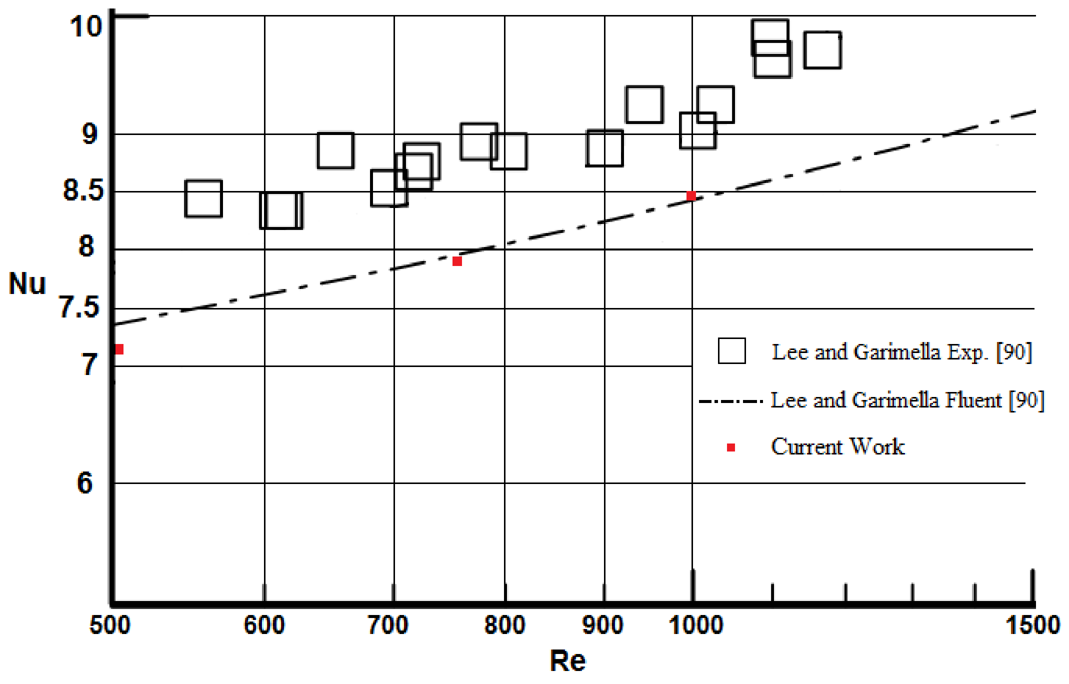


Figure 4.13 Nusselt Numbers Obtained for the Micro Channel Problem at Three Different Re Values.

entrance length. Unit velocity is defined at the inlet of the tube and zero pressure is defined at the center of the outlet plane. In order to simulate $Re = 50$ case, density and dynamic viscosity are taken as 1 and 0.04, respectively. Inlet temperature is taken as 0 and wall temperatures are fixed at 1. For the solid parts temperature gradient in the axial direction is taken as zero at the inlet and outlet planes. Simulations areperformed for $Pr = 1$, which is obtained by using $c_p = 1$ and $k_{fluid} = 0.04$.

Solutions are obtained for two different conductivity values for the tube material, $k_{solid} = 0.04$ and $k_{solid} = 1$. These values correspond to $k_{sf} = k_{solid}/k_{fluid}$ ratio of 1 and 25, respectively.

Unfortunately, the details of the mesh used in the reference study are not shared. Details of the mesh used in this study are shown in Fig. 4.15. It contains 21,000 fluid elements and 160,00 solid elements. Using the first element type, 173,821 pressure and 521,463 velocity unknowns are solved for the flow analysis part. Energy equation had 303,101 temperature unknowns.

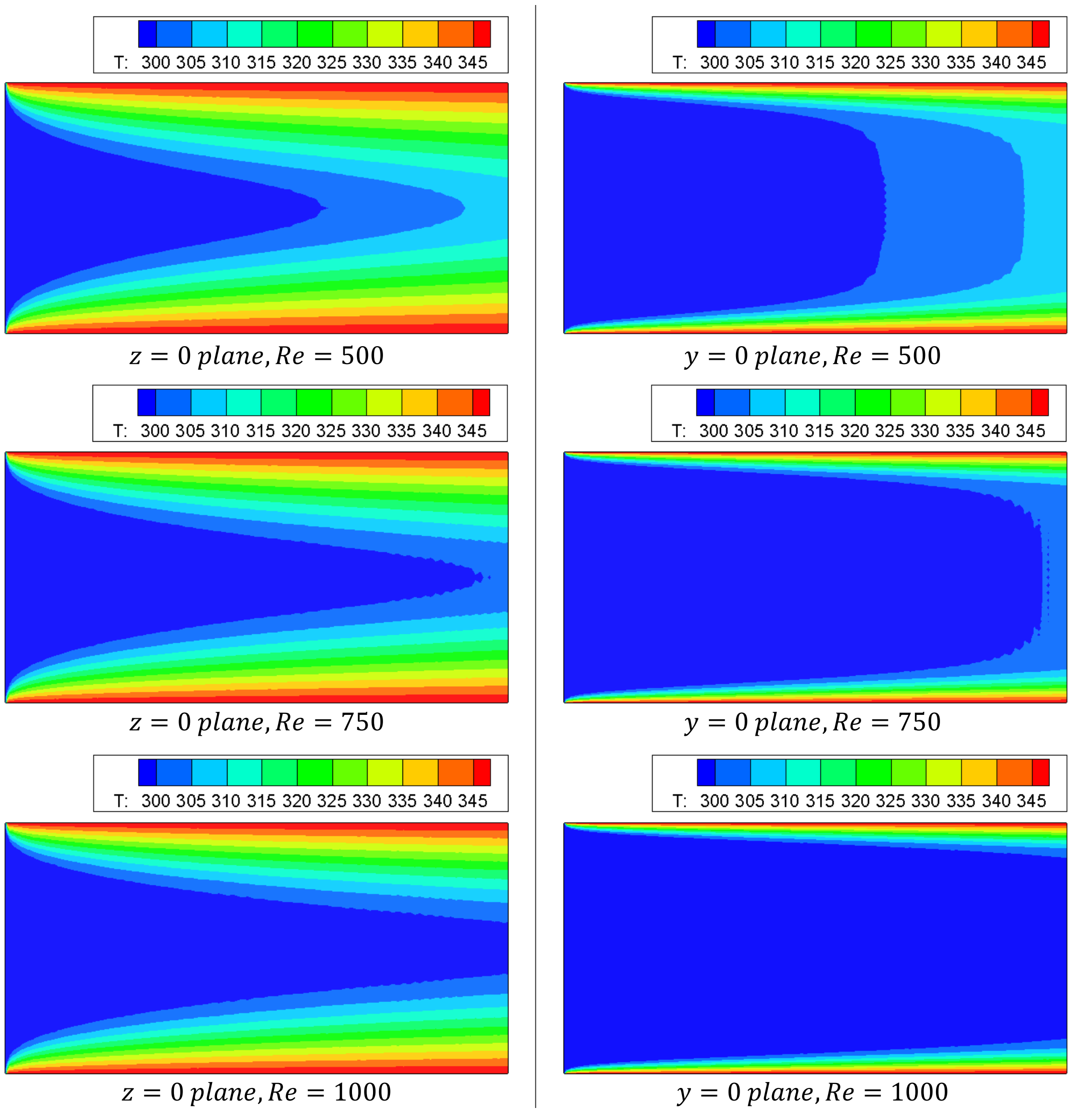


Figure 4.14 Temperature ($°K$) Contours at $y = 0$ and $z = 0$ Planes for Three Different Re Values of the Micro Channel Problem (Axes Are Out of Scale).

In Fig. 4.16, temperature profiles on the inner wall (T_{wi}) along the tube is presented for $k_{sf} = 1$ and $k_{sf} = 25$. For $k_{sf} = 25$, high thermal conductivity of the solid causes very rapid increase of T_{wi}, as expected. Both results show good agreement with the reference results, which are obtained with the finite volume method where discretization of the convective term is done with the SGSD scheme and the SIMPLEC algorithm is utilized for velocity-pressure coupling. In Fig. 4.17 some oscillations are seen at the solid-fluid interface for $k_{sf} = 25$ case. The reason behind this is thought to be the large difference between the solid and fluid conductivity values, which may require a very carefully crafted mesh close to the interface for accurate capturing of the heat flow. Fig. 4.17 shows the axisymmetric temperature contours. The oscillations for $k_{sf} = 25$ can also be seen in this figure.

Finally, variation of bulk fluid temperature (T_{fb}) along the tube is shown in Fig. 4.18. For $k_{sf} = 25$, T_{fb} increases rapidly along the tube and almost reaches to the specified outer wall temperature at the exit. On the other hand, variation of T_{fb} is more linear for $k_{sf} = 1$. Results are in a good agreement with the reference results. Oscillations seen in other figures are not present in this figure because T_{fb} is an integral parameter with inherent smoothing in it.

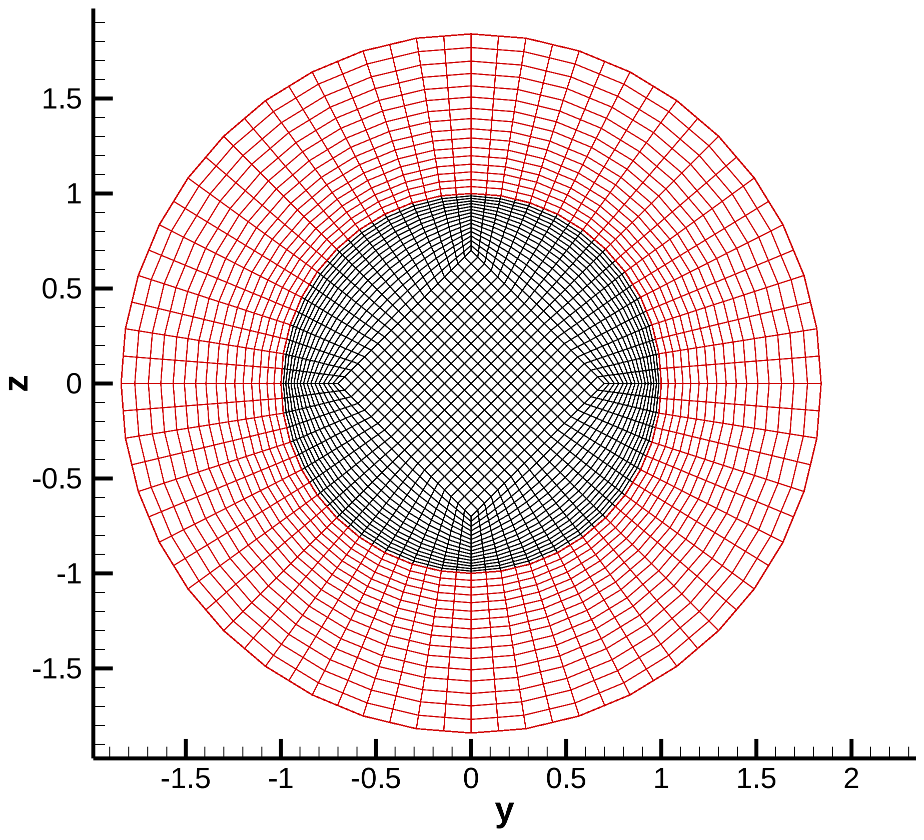
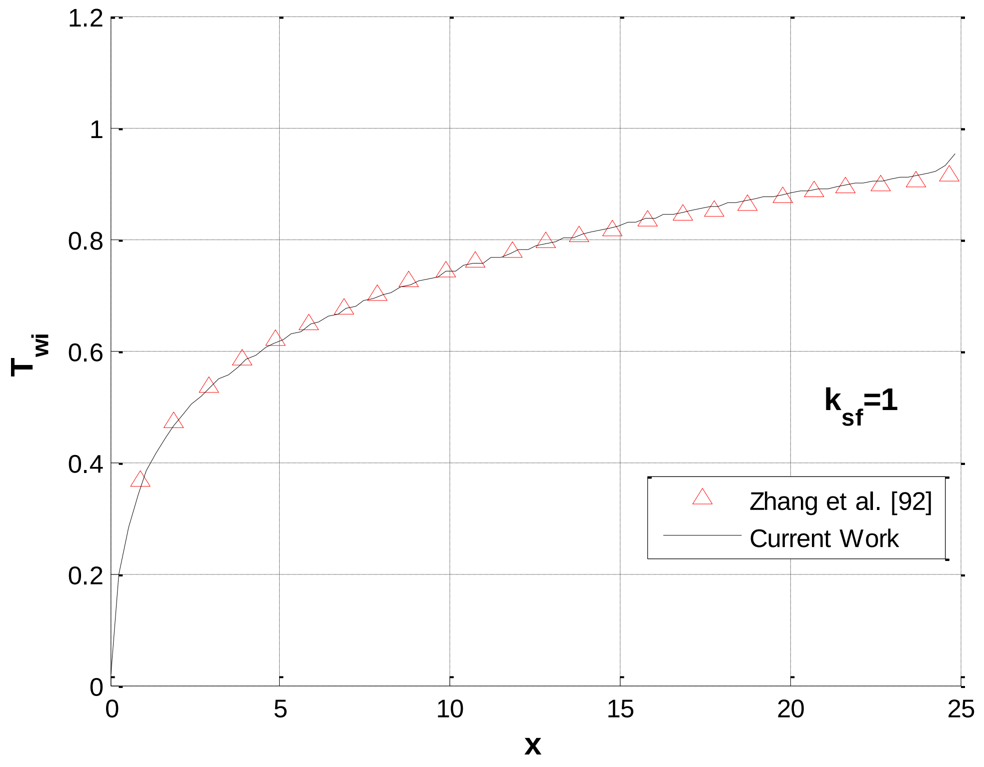


Figure 4.15 A View of the Mesh Used for the Tube Flow Problem. Red Color Shows the Solid Tube Wall.



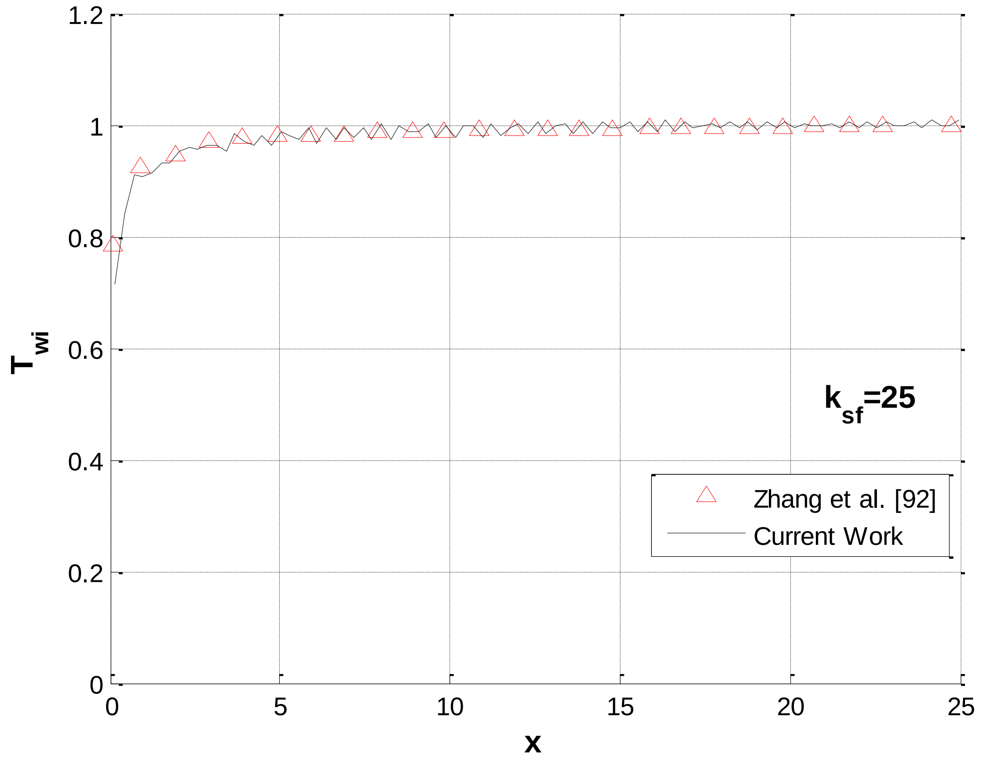


Figure 4.16 Temperature Profiles on the Inner Wall along the Tube.

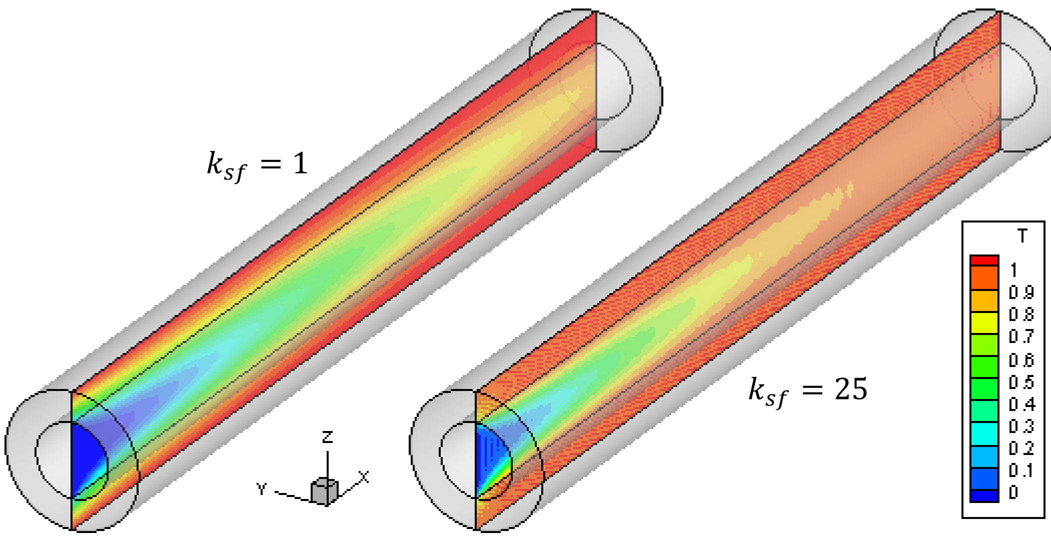


Figure 4.17 Axisymmetric Temperature Contours of the Tube Flow Problem.

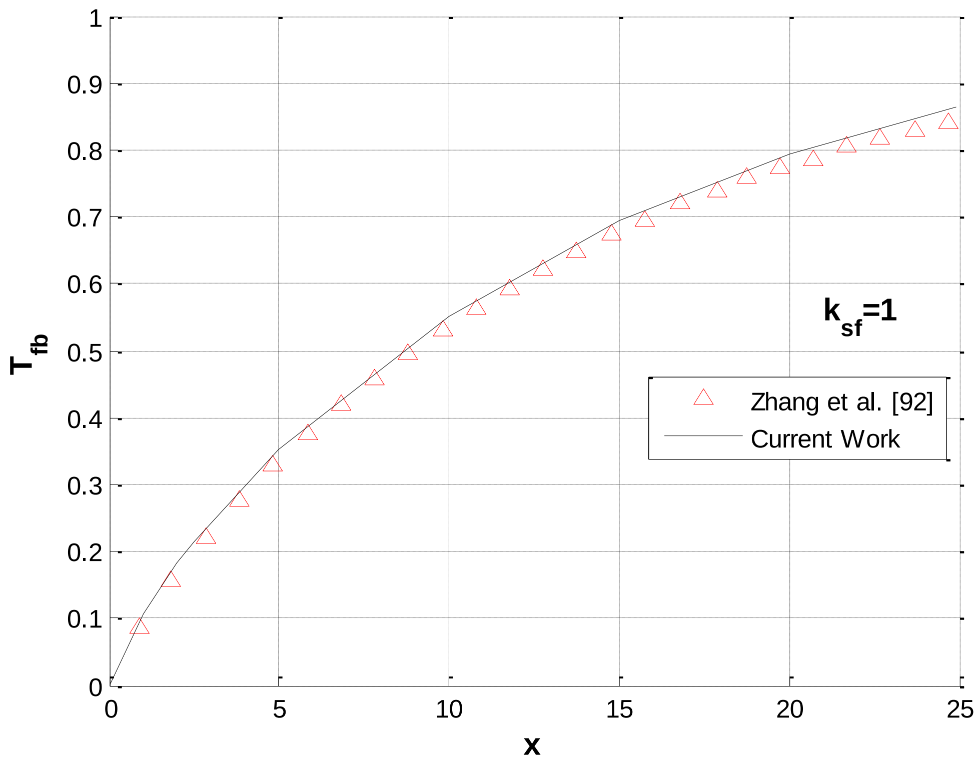


Figure 4.18 Change of the Bulk Fluid Temperature along the Tube.

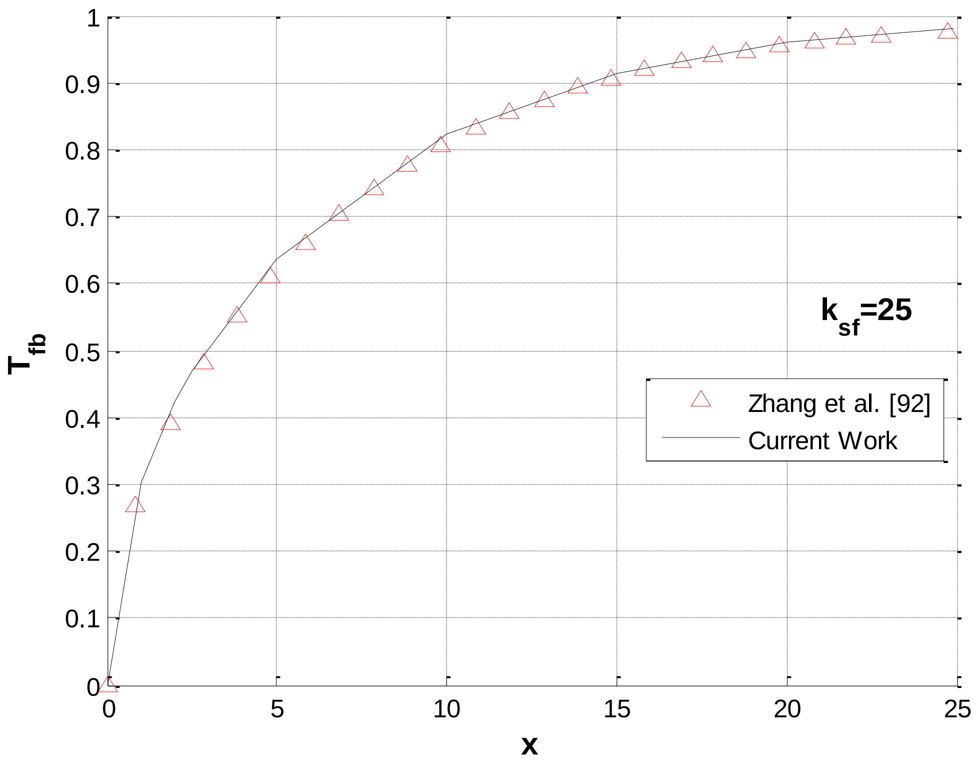


Figure 4.18 (continued) Change of the Bulk Fluid Temperature along the Tube.

CHAPTER 5

PARALLEL PERFORMANCE ANALYSIS OF THE DEVELOPED SOLVER

For parallel performance analysis of the developed solver, lid driven cavity problem that was used before as a benchmark problem is used. Unless otherwise mentioned, results presented in these sections are obtained using the explicit version of the solver. Three different meshes with details given in Table 5.1 are created to represent a small, medium and large size problem. Elements used are of type 1, with 27 velocity nodes and 8 pressure nodes. These three meshes will help to identify how the time consumed by different parts of the flow solver scales with the problem size. Considering the NN values given in the last column of the table, the Mesh II is 3.1 times larger than Mesh I, whereas Mesh III is 3.0 times larger than Mesh II. The closeness of these ratios is important in analyzing the importance of problem size on run time performance. Similar ratios can be calculated by using the values of the other columns of Table 5.1, too.

Table 5.1 Details of the Meshes Used for Parallel Performance Analysis.

Mesh No.	Number of Elements (NE)	Number of Pressure Nodes (NNp)	Number of Velocity Nodes (NNv)	Total Number of Unknowns (NN)
I	$30 \times 30 \times 30 = 27{,}000$	$31 \times 31 \times 31 = 29{,}791$	$61 \times 61 \times 61 = 226{,}981$	$NNp + 3NNv = 710{,}734$
II	$44 \times 44 \times 44 = 85{,}184$	$45 \times 45 \times 45 = 91{,}125$	$89 \times 89 \times 89 = 704{,}969$	$NNp + 3NNv = 2{,}206{,}032$
III	$64 \times 64 \times 64 = 262{,}144$	$65 \times 65 \times 65 = 274{,}625$	$129 \times 129 \times 129 = 2{,}146{,}689$	$NNp + 3NNv = 6{,}714{,}692$

5.1 Determining the Most Time Consuming Parts of the Solver

Before making any parallelization or optimization on the code, time consumed by different parts of it are examined. This is done with a code that works serially on a single core of the CPU, which is Intel Xeon E5-2670. At this point it is good to remember the following major tasks performed by the code, which were previously mentioned in Chapter 3.

Task 0: Major calculations that are done only once outside the time loop. Includes the calculation of the global matrices $[M], [M_d], [M_d]^{-1}, [K], [G], [G]^T$ and calculating $[Z] = [G]^T [M_d]^{-1}[G]$.

Task 1a: Calculating the resulting vector from $[A(U^n)]\{U_i^{n+\frac{1}{2}}\}$ multiplication at the element level, which is used to calculate $\{R_1\}$ of Eqn. (3.18).

Task 1b: Calculating $\{R_1\}$ of Eqn. (3.18) and solving this equation to get $\{U_{i+1}^{n+\frac{1}{2}}\}$.

Task 2: Calculating $\{R_2\}$ of Eqn. (3.19) and solving this equation to get $\{P_{i+1}^{n+1}\}$.

Task 3: Calculating $\{R_3\}$ of Eqn. (3.20) and solving this equation to get $\{U_{i+1}^{n+\frac{1}{2}}\}$.

Task 1 contains two different types of operations namely MIMD type elemental level matrix vector multiplications and SIMD type sparse matrix vector operations. Due to this difference Task 1 is divided into two as Tasks 1a and 1b. Maximum number of iterations per time step is selected as 5. Eqn. (3.16) with a tolerance value of 10^{-3} is used to check the convergence of the iterations. At the beginning of a typical run, usually 3-4 iterations are seen to be enough for convergence and the number drops to 1 as the solution converges to steady state.

Although both Cholesky decomposition and Jacobi Preconditioned Conjugate Gradient (PCG) are used for Task 2, only the results obtained with PCG will be presented for brevity. Compared to PCG, Cholesky decomposition has the major drawback of high memory usage, which becomes especially critical when GPUs are used. On the other hand, being an iterative method, PCG presented a minor issue in performance comparisons of different implementations on the CPU (in MKL) and the GPU (in CUSP) due to their use of different converge criteria.

Table 5.2 Time Spent for Different Tasks During 1 Iteration of a Time Step. Obtained Using One Core of the CPU. Values Are Based on a Single Iteration of One Time Step.

Mesh No.	Task 1a		Task 1b		Task 2		Task 3	
	Time [s]	Ratio to Mesh I	Time [s]	Ratio to Mesh I	Time [s]	Ratio to Mesh I	Time [s]	Ratio to Mesh I
I	0.452	–	0.067	–	0.355	–	0.017	–
II	1.427	3.2	0.209	3.1	1.681	4.7	0.056	3.3
III	4.382	9.7	0.654	9.8	7.294	20.5	0.173	10.2

Time spent for different tasks in one iteration of a single time step are presented in Table 5.2. Presented times are averaged values obtained during the initial few time steps of a solution. The values are wall clock times that are calculated by high accuracy timers inside the code. Task 0 is not shown in this table because it is only executed once per solution so it takes a negligible amount of time compared to other tasks. Determining time consumption of the PCG solver used in Task 2 needs special care because of the iterative nature of the process and the dependency of the time spent by it on the selected tolerance criteria. PCG convergence tolerances are selected based on the accuracy and efficiency observations made during the verification runs of Chapter 4.

Results given in Table 5.2 can be summarized as follows;

- For the case of a single iteration per time step, the most time consuming task is 1a for small meshes. As the mesh gets finer, Task 2 becomes more dominant. It should be noted that one iteration per time step is the best case scenario for a simulation.
- Task 3 is the least time consuming part among others. Time spent for Task 1b is also small, but not negligible compared to Task 1a and Task 2.

- As the mesh is refined, time spent for Tasks 1a, 1b and 3 increase almost linearly with the total number of unknowns (see NN of Table 5.1). For example, time spent for Task 1a for Mesh II and Mesh III are 3.2 and 9.7 times those spent for Mesh I, whereas the number of total unknowns for Mesh II and Mesh III are 3.1 and 9.45 times that of Mesh I, respectively. But the time spent for Task 2 increases at a higher rate as the mesh is refined, which means that Task 2 becomes more and more critical as the problem size increases.
- To sum up, Tasks 1a and 2 stand out when run time is considered. Task 1b should also be watched carefully.

5.2 Multi Thread Usage for the CPU

The workstation used in this study has two Intel Xeon E5-2670 processors, each having 8 cores (16 threads). For the CPU version of the code, this multi core environment is taken advantage of by the use of Intel's MKL library and OpenMP, which have multi-threading support. MKL is used for all matrix-vector operations and the PCG solver. Task 1a cannot be performed in parallel with an already available library. A special OpenMP code is written for it from scratch. The speed-up values obtained for different tasks on different meshes by using multiple threads are shown in Table 5.3.

When Table 5.3 is examined, it is seen that using 8 threads gives the best results for almost all tasks, even though the computer used has 32 threads on 2 CPUs. For Tasks 1b and 3, even the difference between using 4 and 8 threads is very small. Although the saturation at the performance after 8 threads seems rather unexpected, similar results were also reported in a study by Venetis et al. [93]. In their FEM based study, a similar Intel Xeon E5-2658 processor was used.

According to Table 5.3, Task 1a seems to benefit the most from multi-threading and Task 3 seems to benefits the least.

Table 5.3 Achieved Speed-ups by Multi-Threading with Respect to 1 Thread Usage on the CPU. Values are Based on a Single Iteration of One Time Step.

	Task 1a				Task 1b				Task 2				Task 3			
# of Threads	2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
Mesh I	1.9	3.4	6.3	3.5	1.3	1.8	2.4	2.3	1.6	2.5	3.4	2.7	1.3	1.9	2.1	1.9
Mesh II	1.8	3.5	6.3	3.5	1.2	2.2	2.3	2.0	1.4	2.3	2.6	2.3	1.2	2.0	2.0	1.8
Mesh III	1.9	3.5	6.2	3.5	1.5	2.2	2.2	2.2	1.7	2.3	2.5	2.3	1.4	2.0	2.0	2.0

5.3 Performance Comparisons between CPU and GPU

All four main time consuming parts of the code that are executed repetitively inside the time loop are migrated to the GPU. Parallel coding on the GPU is mostly done using freely available libraries CUBLAS, CUSPARSE and CUSP, and new parallel GPU kernels are coded when needed, such as the parallelization of Task 1a. Graphics card used is a single NVIDIA Tesla C2075. After the observation made in the previous section, GPU performance is compared with the performance of the CPU with 1 and 8 threads. In this section, in addition to three lid-driven cavity meshes, performance results for a bending square duct (shown as BSD in Table 5.4 and Table 5.5) mesh is also examined. This mesh has 53,361 elements and 1,391,123 unknowns, which locates it between Mesh I and Mesh II in terms of number of unknowns.

Two different algorithms are tried for Task 1a as explained in Section 3.3. Results obtained by the one that calculates $[A]\{U\}$ multiplication at the element level without creating the global $[A]$ are presented because it is more memory efficient and faster than the one that first assembles the global $[A]$ and then multiplies it with $\{U\}$. It is important to note that the latter algorithm calculates elemental $[A]$ matrices and assembles them into the global $[A]$ only once per time step, but due to GPU memory limitations the former one calculates the elemental $[A]$ matrices at each iteration inside a time step. The former one is still faster because it bypasses the assembly of $[A]$ and iteration number per time step decreases to 1 as the solution approaches to steady state. Moreover at CPU performance analyses, it is seen that making the multiplication elemental level is also beneficial for CPU hence at both CPU and GPU this approach is chosen. For an unsteady problem, this selection needs reconsideration.

Speed-ups obtained by the GPU with respect to 1 and 8 thread usage on the CPU are shown in Table 5.4. Values larger than 1 indicate faster operation on the GPU compared to the CPU. When the performances of GPU and single thread CPU are compared, a maximum of 16.5 times speed-up is seen, which is obtained for Task 1a on the finest mesh. Although in the literature GPU-CPU performance comparisons are usually done for a single thread CPU usage, this is not a fair comparison. When 8 thread CPU usage is considered, speeds-up values decrease and the maximum speed-up is now 4.0, which is seen for Task 3 on Mesh I.

For the finest mesh, GPU vs. 8-thread CPU speed-ups for Tasks 1a, 1b, 2 and 3 are 2.35, 2.79, 1.69 and 3.78, respectively. As seen in the previous sections, Task 2 becomes the most critical part of the code as the mesh gets finer and from that perspective 1.69 times speed up on the finest mesh suggests that there may be room for improvement. PCG solver of the CUSP library is used for Task 2 on the GPU. As an alternative PCG can be coded by using the CUSPARSE library or another third party library can be tried. Another important point here is the possible differences in the residual calculations and the convergence checks of MKL's and CUSP's PCG implementations. In that case providing the same tolerance value for both might not result in a fair comparison. This issue needs further control and clarification. The good news is though, that there is an overall trend of increase in speed-up values as the problem size increases.

Time consumption and speed-up values of all tasks together are shown in Table 5.5. It is seen that GPU outperforms 8 threads of CPU for all three lid-driven cavity meshes and bending square duct mesh. Moreover performance of GPU increases

Table 5.4 Achieved Speed-ups by GPU with Respect to 1 and 8 Thread Usage on the CPU. Values Are Based on a Single Iteration of One Time Step.

	Task 1a		Task 1b		Task 2		Task 3	
# of Threads	1 thread	8 threads	1 thread	8 threads	1 thread	8 threads	1 thread	8 threads
Mesh I*	14.58	2.32	6.09	2.55	2.80	0.82	8.50	4.00
Mesh II*	14.56	2.33	5.97	2.63	4.02	1.53	7.00	3.50
Mesh III*	14.61	2.35	6.06	2.79	4.22	1.69	7.52	3.78
BSD**	16.46	2.32	8.19	2.52	4.42	1.11	11.00	4.00

*Lid-driven cavity meshes, see Table 5.1.
**Bending square duct mesh.

with increasing number of unknowns. For the GPU versus 8 threads of CPU case 1.24, 1.76 and 1.86 times speed-ups are obtained for Mesh I, Mesh II and Mesh III of the lid-driven cavity problem, respectively. When the performances of GPU and single thread and 8 threads of CPU are compared, a maximum of 6.39 (BSD mesh) and 1.86 (Lid-driven cavity, Mesh III) times speed-ups are seen respectively.

An important point to keep in mind is the price and power consumption of GPU and CPU while comparing their performances. The current retail prices of Intel Xeon E5-2670 CPU and NVIDIA Tesla C2075 GPU are around 1750 $ and 1900 $, respectively. So there is no significant difference between prices. On the other hand, thermal design powers of the used CPU and the GPU are 115 W [4] and 225 W [5], respectively. GPU needs two times more power to run compared to the CPU, which is a considerable difference. The power comparison becomes more dramatic considering that the CPU can be used alone without the GPU, but to operate a GPU you also need a running CPU.

Table 5.5 Speed-Up Values for the Total Time Spent for all Tasks. (Considering Single Iteration of One Time Step)

Mesh No.	CPU (1 thread) [s]	CPU (8 threads) [s]	GPU [s]	Speed-up GPU vs CPU (1 thread)	Speed-up GPU vs CPU (8 threads)
I*	0.891	0.212	0.171	5.21	1.24
II*	3.373	0.986	0.559	6.03	1.76
III*	12.503	4.012	2.162	5.79	1.86
BSD**	2.525	0.548	0.395	6.39	1.39

*Lid-driven cavity meshes, see Table 5.1.
**Bending square duct mesh.

5.4 Effect of Using Single Precision on Performance and Accuracy

For all of the runs that were presented in the previous sections floating point numbers are stored in double precision (DP). Considering that GPUs are originally designed for single precision (SP) arithmetic and their DP support is getting attention only recently, it is logical to compare the performances of CPUs and GPUs for the use of SP. Again the lid driven cavity problem is used for this purpose. The problem is solved for $Re = 100$ with the three meshes defined in Table 5.1. Steady state results obtained with SP are almost identical to the ones obtained previously with DP. It is possible to conclude that the use of SP has no undesired effect on the accuracy.

Speed-up values obtained by using SP instead of DP are given in Table 5.6. Both the CPU and GPU versions of the code benefited from SP usage. For the Tasks 1b, 2 and 3, GPU and CPU speed-ups are nearly same but for Task 1a there is a huge gap in speed-up values between GPU and CPU in favor of the GPU. GPU-CPU speed-up values for the total time spent for Tasks 1b, 2 and 3 when SP is used is given in Table 5.7. When compared with Table 5.5, it is seen that using SP increases the GPU versus CPU speed-up in favor of GPU. Now for the finest mesh GPU outperforms 8 threads of CPU by 2.08 times.

Table 5.6 Speed-up Values when Single Precision is Used Compared to Double Precision. (Considering Single Iteration of One Time Step)

	Task 1a		Task 1b		Task 2		Task 3	
	CPU 8 threads	GPU	CPU 8 threads	8 threads	CPU 8 threads	GPU	CPU 8 threads	GPU
Mesh I	1.11	1.94	1.56	1.57	1.89	1.76	1.60	2.00
Mesh II	1.09	2.04	1.48	1.67	1.96	1.86	1.56	2.00
Mesh III	1.11	2.04	1.53	1.66	2.23	2.10	1.55	1.77

Table 5.7 Speed-Up Values for the Total Time Spent for all Tasks When Single Precision is Used. (Considering Single Iteration of One Time Step)

Mesh No.	CPU (8 threads) [s]	GPU [s]	Speed-up GPU vs CPU (8 threads)
I	0.143	0.096	1.49
II	0.614	0.298	2.06
III	2.185	1.048	2.08

5.5 Performance Comparisons Between Explicit and Implicit Fractional Step Formulations

All the results presented in the previous section of this chapter are obtained using the explicit formulation version of the developed solver. But due to its severe time step restrictions, also an implicit version is developed. It is first tested for accuracy and stability when large time steps are used. It was possible to solve the lid driven cavity problem accurately with Mesh I of Table 5.1 with time steps in the order of unity.

With the explicit formulation this problem required the use of $\Delta t = 0.001$ s. To compare the run time performances of the explicit and implicit formulations for steady state problems the following steady state convergence check is used;

$$min(|\{U^{n+1}\} - \{U^{n}\}|) \leq \epsilon_{ss} \quad 5.1$$

Because of the different natures of the methods, particularly about the time step constraints, convergence to the steady state solution is considered and different tolerance values are used for them. Analyses are performed using Mesh I and Mesh II of Table 5.1 and results of these analyses are presented in Table 5.8. Although the implicit solver executes a single time step much slower than the explicit one, due to its ability to use larger time steps it can finish the overall solution faster. The main reason behind the difference in the time spent for a single time step is the time spent for the solution of non-symmetric linear system arises from Eqn. (3.27). The implicit formulation also requires more memory than the explicit one because it needs to keep the left hand side matrix ($[(1/\Delta t)[M] + [A(U^{n})] + [K]]$) of Eqn. (3.23) in memory. Due to this, the lid driven cavity problem cannot be solved on the GPU with Mesh III, which was possible for the explicit formulation. It can be concluded that the implicit formulation is better for the solution of steady state problems if they can be fit into the memory of the available hardware. Explicit formulation can be preferred for transient solutions, where time step size is constrained by accuracy rather than stability.

Table 5.8 Performance Comparisons of Implicit and Explicit Formulations on the GPU

Mesh No.	Method	Δt [s]	# of Iterations	Time for 1 Time Step [s]	Total Time [s]
Mesh I	Implicit	0.2	155	2.16	335
	Explicit	0.001	4,409	0.16	727
Mesh II	Implicit	0.02	433	4.51	1,954
	Explicit	0.0002	27,110	0.38	10,404

5.6 Memory Usage of the Fractional Step Solver

In this part, memory usage of developed flow solver is examined. Table 5.9 represents the main parameters that determine the memory requirement. In this table NE is the number of elements, NNp and NNv are the number of pressure and velocity nodes inside each element respectively, NN is total number of unknowns, and $Mnnz$, $Gnnz$ and $Znnz$ are number of non-zeros in $[M], [G]$ and $[Z]$ matrices, respectively. Number of non-zeros in $[A]$ and $[K]$ matrices are same as the number of non-zeros in $[M]$. For a three-dimensional problem $[M], [A]$ and $[K]$ matrices are formed by three identical sub-matrices and keeping only one of them in the memory is enough. Considering this, actual $Mnnz$ value is one third of the value given in Table 5.9.

Among many sparse storage schemes, compressed row storage (CSR) [94] is used for the storage of the global matrices both on the CPU and the GPU. In addition to its efficient memory handling, it is supported by Intel MKL, CUSP, CUSPARSE,

Table 5.9 Parameters That Determine the Memory Requirements of the Developed Solver

	Mesh I	Mesh II	Mesh III
NE	27,000	85,184	262,144
NNp	29,791	91,125	274,625
NNv	226,981	704,969	2,146,689
NN	710,734	2,206,032	6,714,692
Mnnz	41,992,563	131,960,931	405,017,091
Gnnz	10,328,853	32,381,583	99,228,483
Znnz	1,668,870	5,297,292	16,368,192

CUBLAS and Csparse libraries. CSR scheme consists of 3 arrays as shown below for a sample 4x4 matrix

$$\mathrm{A} = \begin{bmatrix} 2 & 5 & 0 & 0 \\ 3 & 1 & 2 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 5 & 3 \end{bmatrix}$$

$$val = [2\ 5\ 3\ 1\ 2\ 4\ 5\ 3]$$

$$col = [0\ 1\ 0\ 1\ 2\ 2\ 2\ 3]$$

$$rowStarts = [0\ 2\ 5\ 6\ 8]$$

val array stores the non-zero values of the matrix in a row-by-row order. col array stores the column indices of these non-zero values. $rowStarts$ array is the list of non-zero value indices where each row of $[A]$ starts. With the CSR scheme, memory requirement for storing a $(N \times N)$ matrix is $(NN + 1 + NNZ)$ integers and NNZ floating points, where NNZ is the number of non-zeros. Because the col and $rowStarts$ arrays are same for $[M], [A]$ and $[K]$ matrices, only one set of col and $rowStarts$ arrays are kept in memory for them.

There are many other large vectors that must be kept in memory in order to solve the systems in Eqns. (3.17-3.22). However, compared to global matrices memory requirements of these vectors are very small. Moreover, there are vectors that keep elemental to global node mapping information, which is used during the assembly process. These mapping vectors only contain integers but they grow quickly with the problem size so they also need attention. Lastly the vector that stores derivatives of the shape functions with respect to x, y and z at all Gauss Quadrature points (integrals are computed using 8 point Gauss Quadrature) of all elements are stored in memory. These derivatives could be calculated whenever they are needed, but because $[A]$ matrix is calculated every time step, calculating these derivatives again and again could be an inefficient process. Another idea is to take advantage of geometrically similar (both in size and orientation) elements and storing shape function derivatives only once for them, which is not utilized in this study. After all these details memory requirement of the largest mesh, Mesh III in Table 5.9, is almost 6 GB.

An important point about Table 5.9 is that it presents the values for hexahedral elements with second order velocity and first order pressure interpolation. This element, which can be seen in Figure 3.4, has 27 velocity nodes and 8 pressure nodes. Table 5.9 needs to be modified when a different element type is used, such as the one that uses linear interpolation for both pressure and velocity. Another important parameter that affects memory usage is the precision of floating-point numbers. The 6 GB value given in the previous paragraph is based on double precision usage. If instead single precision is used, memory requirement drops by almost a factor of two.

CHAPTER 6

SUMMARY AND CONCLUSION

Computational power is one of the most important limiting factors in the development and use of CFD. For many years, scientists parallelized their CFD codes on computers with shared and distributed memory architectures using tools such as OpenMP, MPI and PVM. For many decades the only computing power had been the standard Central Processing Units (CPUs). Over the last two decades the major development in the CPU technology was about their inherent parallelization features. First, single core CPUs gained hyper-threading support, followed by multi-core CPUs. Nowadays standard PCs come with 4, 6 or 8 core CPUs. In the last decade another concept known as many-core computing is introduced to the high performance computing community. GPUs become a major driving force behind this new wave due to their hardware supporting 100s of cores that can work with 1000s of parallel running threads. With the release of CUDA Toolkit by NVIDIA on 2007, general purpose computing on GPUs became a very appealing parallelization alternative for the scientific codes including CFD solvers. Today Top500 supercomputer list [95] includes many computers with GPU support. GPU computing community is very active with specialized conferences, free and commercial third party linear algebra libraries, increasing number of commercial codes with GPU support, etc.

Researchers previously demonstrated that compared to CPUs, GPUs can provide tens of times of speed-ups. Of course these claims are very dependent on the algorithm being parallelized. For the case of CFD, it is possible to see publications that report up to 100 times speeds-ups with the utilization of GPUs. These very high performance gains are usually limited to methods that have a very high parallelization potential such as Smoothed Particle Hydrodynamics (SPH), Lattice

Boltzmann Method (LBM) or Discontinuous Galerkin (DG). Also in the literature it is possible to find many unfair GPU-CPU comparisons and one needs to be very cautious with very high speed-ups [21]. Other than SPH, LBM or DG based works most CFD codes that are ported to GPUs are compressible solvers. There are very limited number of incompressible flow studies on the GPU and similarly there are very limited number of FEM based solvers ported to the GPU. This forms the motivation behind the current work, in which a finite element based incompressible flow solver is developed to work parallel on the GPU.

Three dimensional, unsteady, laminar flows with possible heat transfer affects are solved using two different fractional step formulations based on the classical Galerkin finite element formulation. Two different versions of the solvers are written, one working on the CPU and the other working on the GPU. Different from many of the studies available in the literature GPU version is not written as a modification of the CPU version, but it is written from scratch. First the CPU version is used for verification purposes using 5 benchmark problems, including one microchannel duct flow and a tube flow with conjugate heat transfer. After verifying the accuracy of the code, speed tests are performed using the lid driven cavity benchmark problem with three different grids. Coarse grid had about 700,000 total pressure and velocity unknowns. The number was about 2.2 million for the medium grid and about 6.7 million for the fine grid.

First the most time consuming parts of the code are detected and the scaling of the amount of time spent on these parts to the total number of unknowns is studied. Next, multi core performance of the CPU version of the code is tested. Parallelization on the CPU is mainly achieved by the use of Intel’s MKL library and sometimes using OpenMP pragma’s. Speed-ups obtained for the major time consuming tasks that are calculated repetitively inside the time loop are studied separately. Overall it was seen that even Intel’s own MKL library on an Intel CPU could not make use of the full potential of the available cores and for most of the tasks 8-thread usage resulted in the lowest run times, although the machine used can utilize up to 32 threads.

Parallelization on the GPU is mainly done by the use of CUBLAS and CUSPARSE libraries that come with the CUDA Toolkit and the CUSP library, which is a freely available third party library developed by NVIDIA employees. For certain non-standard tasks new GPU kernels are written from scratch. For all three grids mentioned above, it was possible to obtain a speed-up by the use of GPU compared to the use of 8-cores on the CPU. For the largest grid that had more than 6.7 million unknowns, GPU usage resulted in 5.79 and 1.86 times speed-ups compared to single-thread and 8-thread CPU solutions. These values are similar with results of Göddeke et al.'s study [59], which is one of the a few studies that uses GPUs for finite element based incompressible flow solutions.

Considering the fact that especially early generation GPUs have very poor double precision support, the developed solver is transferred to single precision accuracy both on the CPU and the GPU. First the effect of this conversion on the accuracy of the solution is tested with a benchmark solution and no difference is seen in the results obtained by single and double precision. Then the run times of single and double precision codes are compared on the CPU and the GPU. It is seen that switching from double to single precision resulted in performance increases on both platforms. When compared to double precision, it is seen that using single precision increases the GPU versus CPU speed-up in favor of GPU. For the finest mesh GPU outperforms 8 threads of CPU by 2.08 times, which was 1.86 times for double precision. With further accuracy tests for single precision using different problems, double precision usage may completely be dropped. Other than 2 times speed-up memory usage also drops significantly when single precision accuracy is used.

The fractional step method used throughout the study was based on an explicit time integration scheme, which puts severe limitations on the allowable time step. This resulted in quite long total run times to complete a solution. To improve this, an implicit fractional step formulation, which has no time step restriction due to stability concerns, is tried towards the end of the study. The implicit scheme can make use of much larger time steps compared to the explicit version but it requires the solution of a non-symmetric linear algebraic system. However solving a non-symmetric system is costly, using much larger time steps leads to less iteration numbers hence implicit solver converges faster than explicit one. On the other hand the implicit solver suffered from higher memory requirement, which especially became an issue considering the limited global memory of the GPUs.

During the study an appreciable amount of time is spent for learning the basics of GPU architecture and CUDA programming. The learning curve sometimes turned out to be quite steep. A major disadvantage was the very rapid development of GPU hardware and parallel to that very frequent releases of the CUDA Toolkit. In the three year time span of this study NVIDIA made several major architectural changes to their GPUs. Also they released 3 major Toolkit versions, together with several minor ones. At times it was very difficult to keep track of the updates and make advantage of the new tools. Many times, the reference books and other online resources used for learning GPU hardware internals and CUDA programming lack the most recent information.

Another important observation is about the lack of supporting linear algebra libraries on the GPU side. BLAS and LAPACK libraries are essential tools for CFD codes. It is not logical to self-code the vector and matrix operations provided by these libraries. They have highly optimized sequential and parallel versions. Of equal importance are the linear algebra libraries that work with sparse matrices. Intel's MKL library used in this study for computations on the CPU is a popular implementation of these libraries. It has been developed for many years and comes with an excellent documentation. On the GPU side CUBLAS replaces BLAS and CUSPARSE provides some essential sparse matrix support. But they are not as developed as their CPU counterparts. For example they have no multi-GPU support and today CUDA programming on multiple GPUs is still a challenge. Other than CUBLAS and CUSAPRSE it is not easy to find freely available linear algebra libraries for GPUs. CUSP library was very critical for this study, but it is a work of a few researches and has no documentation. Support is available only through a discussion list.

This work will be concluded with a list of possible future work ideas to improve the developed solver.

- Although the available workstation had two GPUs only one of them was used. The code will benefit a lot by the use of multiple GPUs, especially if the doubling of the global GPU memory is considered. With more memory many alternative solution ideas will be implementable, such as the use of direct Cholesky factorization instead of the iterative PCG technique for the solution of the pressure equation.

- Although two different fractional step techniques are tested, the search for a more efficient one should continue. This will be critical especially for problems that require long time integration.
- More efficient preconditioners such as algebraic multi-grid or incomplete Cholesky should be used with the Conjugate Gradient solver.
- Making the solver completely matrix-free without any assembly of the global matrices can be very effective when the limited GPU memories are considered.
- The developed solver can make use of only hexahedral elements. Support for tetrahedral and other types of elements will be useful for the solution of problems on complex geometries. Each different type of element comes with its own parallel performance details due to the different number of unknowns on them and the sparsity pattern of the resulting global systems.
- The main limitation behind the solution of more realistic real-life problems is turbulence modeling. The code will benefit a lot from the implementation of a possibly RANS type, turbulence model.

During this study the developed solver is kept at the following code repository and future updates can be followed there.

https://code.google.com/p/cfd-with-cuda

REFERENCES

[1] Milne-Thomson, L.M., 1973, "*Theoretical Aerodynamics*", Dover Publications.

[2] Reddy, J.N., 1993, "*An Introduction to the Finite Element Method*", McGraw-Hill, New York.

[3] Tatourian, A., 2013, "NVIDIA GPU Architecture & CUDA Programming Environment", http://tatourian.com/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/, last visited on July 2014.

[4] 2014, "Intel® Xeon® Processor E5-2670 (20M Cache, 2.60 GHz, 8.00 GT/s Intel® QPI)", http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI, last visited on August 2014

[5] 2014, "NVIDIA® TESLA™ C2075 COMPANION PROCESSOR", http://www.nvidia.com.tr/docs/IO/43395/NV-DS-Tesla-C2075.pdf, last visited on August 2014.

[6] 2014, "Intel® Xeon Phi™ Product Family, Peak Theoretical Performance", http://www.intel.my/content/www/my/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html, last visited on November 2014.

[7] 2014, "CUDA C Programming Guide", http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz38TBxSZKv, last visited on July 2014.

[8] Ramirez, S., 2010, "Understanding Latency versus Throughput", http://community.cadence.com/cadence_blogs_8/b/sd/archive/2010/09/13/understanding-latency-vs-throughput, last visited on November 2014.

[9] 2014, "The Open Standard for Parallel Programming of Heterogeneous Systems", https://www.khronos.org/opencl/, last visited on November 2014.

[10] 2014, "APP SDK – A Complete Development Platform", http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/, last visited on November 2014.

[11] 2014, "cuBLAS", http://docs.nvidia.com/cuda/cublas/, last visited on November 2014.

[12] 2014, "cuSPARSE", http://docs.nvidia.com/cuda/cusparse/, last visited on November 2014.

[13] 2014, “Matrix Algebra on GPU and Multicore Architectures”, http://icl.cs.utk.edu/magma/, last visited on November 2014.

[14] Bell, N., Garland, M., 2014, “Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations”, http://cusplibrary.github.io/, last visited on November 2014.

[15] 2014, “CULA | Tools”, http://www.culatools.com/, last visited on November 2014.

[16] Flynn, M., 1972, “Some Computer Organizations and Their Effectiveness.” Computers, IEEE Transactions on, 100 (9), 948-960.

[17] 2014, “Flynn's Taxonomy”, http://en.wikipedia.org/wiki/Flynn's_taxonomy, last visited on July 2014.

[18] Govindaraju, N. K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J., “High Performance Discrete Fourier Transforms on Graphics Processors”, Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 1–12.

[19] Bennemann, C., Beinker, M., Egloff, D., Gauckler, M., 2008, “Teraflops for Games and Derivative Pricing”, Wilmott Magazine, 36, 50-54.

[20] Tolke, J., Krafczyk, M., 2008, “TeraFLOP Computing on a Desktop PC with GPUs for 3D CFD”, In International Journal of Computational Fluid Dynamics, 22, 443–456.

[21] Lee, Victor W., et al., 2010, "Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU." ACM SIGARCH Computer Architecture News, 38-3.

[22] Sanders, J., Kandrot, E., 2010, “*CUDA by Example An Introduction to General-Purpose GPU Programming* ”, Addison-Wesley, Upper Saddle River, NJ.

[23] 2009, “CUDA – Tutorial 3 – Thread Communication”, http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/, last visited on November 2014.

[24] Zienkiewicz, O.C., Cheung, Y.K., 1965, “Finite Elements in the Solution of Field Problems”, The Engineer, 220, 507-510.

[25] Hood, P., 1970, “A Finite Element Solution of the Navier-Stokes Equations for Incompressible Contained Flow”, M.Sc. Thesis, Wales University, Swansea, United Kingdom.

[26] Huyakorn, P.S., Taylor, C., Lee, R.L., Gresho, P.M., 1978, “A Comparison of Various Mixed-Interpolation Finite Elements in the Velocity-Pressure Formulation of the Navier-Stokes Equations”, Comput. & Fluids, 6, 25-35.

[27] Zahedi, S., Kronbichler, M., Kreiss, G., 2012, "Spurious Currents in Finite Element Based Level Set Methods for Two-Phase Flow.", Int. J. for Numer. Methods in Fluids, 69, 1433-1456.

[28] Taylor, C., Hood, P., 1973, "A Numerical Solution of the Navier-Stokes Equations Using the Finite Element Technique", Comput. & Fluids, 1, 73-100.

[29] Barragy, E., Carey, G.F., 1997, "Stream Function-Vorticity Driven Cavity Solution Using *p* Finite Elements.", Comput. & Fluids, 26, 453-468.

[30] Hughes, T.J.R., Liu, W.K., Brooks, A., 1979, "Finite Element Analysis of Incompressible Viscous Flow by the Penalty Function Formulation", J. Comput. Phys., 30, 1-60.

[31] Reddy, J.N., 1982, "On Penalty Function Methods in the Finite-Element Analysis of Flow Problems.", Int. J. for Numer. Methods in Fluids, 2, 151-171.

[32] Brooks, A.N., Hughes, T.J.R., 1982, "Streamline Upwind/Petrov-Galerkin Formulations for Convection Dominated Flows with Particular Emphasis on the Incompressible Navier-Stokes Equations", Comp. Meth. Applied Mech. Engrg, 32, 199-259.

[33] Hannai, S.K., Stanislas, M., Dupont, P., 1995, "Incompressible Navier-Stokes Computations Using SUPG and GLS Formulations – A Comparison Study", Comput. Methods Appl. Mech. Engrg. 124, 153-170.

[34] Donea, J., 1984, "A Taylor-Galerkin Method for Convection Transport Problem", Int. J. Numer. Methods in Fluids, 4, 1043-1063.

[35] Rice, J.G., Schipke, R.J., 1986, "An Equal Order Velocity-Pressure Formulation That Does Not Exhibit Spurious Pressure Modes", Comput. Methods Appl. Mech. Engrg., 58, 135-149.

[36] Prakash, C., Patankar, S.V., 1985, "A Control Volume-Based Finite-Element Method for Solving the Navier-Stokes Equations Using Equal-Order Velocity-Pressure Interpolation", Numer. Heat Transfer, 8 (3), 259-280.

[37] Hughes, T.J., Franca, L.P., Balestra, M., 1986, "A New Finite Element Formulation for Computational Fluid Dynamics: V. Circumventing the Babuška-Brezzi Condition: A Stable Petrov-Galerkin Formulation of the Stokes Problem Accommodating Equal-Order Interpolations.", Comput. Methods Appl. Mech. Engrg., 59, 85-99.

[38] Haroutunian, V., Engelman, M.S., Hasbani, I., 1993, "Segregated Finite Element Algorithms for the Numerical Solution of Large-Scale Flow Problems", Int. J. Numer. Methods Fluids, 17, 323-348.

[39] Shaw, C.T., “Using a Segregated Finite Element Scheme to Solve the Incompressible Navier-Stokes Equations.”, Int. J. Numer. Methods Fluids, 12, 81-92.

[40] Du Toit, C.G., “Finite Element Solution of the Navier-Stokes Equations for Incompressible Flow Using a Segregated Algorithm.”, Comput. Methods Appl. Mech. Engrg, 30, 53-73.

[41] Donea, J., Giuliani, S., Laval, H., Quartepelle, L., 1982, “Finite Element Solution of Unsteady Navier-Stokes Equations by a Fractional Step Method”, Comput. Methods Appl. Mech. Engrg, 30, 53-73.

[42] Blasco, J., Codina, R., Huerta, A., 1998, “A Fractional-Step Method for the Incompressible Navier-Stokes Equations Related to a Predictor-Multi Corrector Algorithm”, Int. J. Numer. Methods in Fluids, 28, 1391-1419.

[43] Guermond, J.L., Minev, P., Shen, J., 2006, “An Overview of Projection Methods for Incompressible Flows”, Comput. Methods Appl. Mech. Engrg., 195, 6011-6045.

[44] Volker, J., 2002, “Higher Order Finite Element Methods and Multigrid Solvers in a Benchmark Problem for the 3D Navier–Stokes Equations”, Int. J. for Numer. Methods in Fluids, 40, 775-798.

[45] Whitling, C.H., Jansen, K.E., 2001, “A Stabilized Finite Element Method for the Incompressible Navier-Stokes Equations Using a Hierarchical Basis”, Int. J. Numer. Methods in Fluids, 35, 93-116.

[46] Nithiarasu, P., Mathur, J.S., Weatherill, N.P., Morgan K., 2004. “Three Dimensional Incompressible Flow Calculations Using the Characteristic Based Split (CBS) Scheme”, Int. J. for Numer. Methods Fluids, (44), 1207-1229.

[47] Cockburn, B., 2003, “Discontinuous Galerkin Methods”, ZAMM - Journal of Applied Mathematics and Mechanics, 83 (11), 731-754.

[48] Löhner, R.K., Morgan, J., Peraire, J., Zienkiewicz, O.C., 1985, “Finite Element Methods for High Speed Flows”, AIAA-85-1531-CP.

[49] Hauke, G., Hughes, T. J. R., “A Unified Approach to Compressible and Incompressible Flows.”, Comput. Methods Appl. Mech. Engrg., 113, 389-395.

[50] Liu, Y., Liu, X., Wu, E., 2004, “Real-time 3D Fluid Simulation on GPU with Complex Obstacles”, Proc. Pacific Graphics, 247–256.

[51] Herault, A., Bilotta, G., Dalrymple, R.A., 2010, “SPH on GPU with CUDA”, J. Hydraulic Research, (48) Extra Issue, pp. 74–79.

[52] Hagen, T.R., Lie, K.A., Natvig, J.R., 2006, “Solving the Euler Equations on Graphics Processing Units”, Proc. 6th Int. Conf. Comput. Sci., Vol. 3994 of Lecture Notes in Computer Science, 220–227, Springer.

[53] Brandvik, T., Pullan, G., 2007, “Acceleration of a Two-Dimensional Euler Flow Solver Using Commodity Graphics Hardware”, Proc. Inst. Mech. Engineers Part C – J. Mech. Eng. Sci., 221 (12), 1745-1748.

[54] Elsen, E., LeGresley, P., Darve, E., 2008, “Large Calculation of the flow Over a Hypersonic Vehicle Using a GPU”, J. Comp. Phys., 227, 10148-10161.

[55] Tölke, J., Krafczyk, M., 2008, “TeraFLOP Computing on a Desktop PC with GPUs for 3D CFD”, Int. J. Comput. Fluid Dynamics, 22 (7), 443-456.

[56] Riegel, E., Indiger, T., Adams, N.A., 2009, “Numerical Simulation of Fluid Flow on Complex Geometries using Lattice-Boltzmann Method and CUDA-Enabled GPUs”, SIGGRAPH, New Orleans, Lousiana.

[57] Bolz, J., Farmer, I., Grinspun, E., Schroder, P., 2003, “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”, ACM Trans. Graph. (Proceedings of ACM SIGGRAPH) 22 (3), 917-924.

[58] Bell, N., Garland, M., 2009, “Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors.”, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 18.

[59] Göddeke, D., Buijssen, S.H.M., Wobker, H., Turek, S., 2009, “GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver”, Proceedings of the 2009 Int. Conf. on High Performance Computing and Simulation, HPCS 2009 - 5191718, 12-21.

[60] Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D., 2009, “Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units”, Proc. 47th AIAA Aerospace Sciences Meeting, AIAA 2009-565.

[61] Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.S., 2009, “Nodal Discontinuous Galerkin Methods on Graphics Processors”, J. Comp. Phys., 228 (21), 7863-7882.

[62] Cecka, C., Lew, A.J., Darve, E., 2011, “Assembly of Finite Element Methods on Graphics processors”, Int. J. Numer. Meth. Engrg., 85 (5), 640-669.

[63] Refsnæs, R.H., 2010, “Matrix-Free Conjugate Gradient Methods for Finite Element Simulationson GPUs”, M.Sc. Thesis, Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway.

[64] Jespersen, D.C., 2009, “Acceleration of a CFD Code with a GPU”, NAS Technical Report NAS-09-003.

[65] Corrigan, A., Camelli, F., Löhner, R., Mut, F., 2010, “Porting of an Edge-Based CFD Solver to GPUs”, 48th AIAA Aerospace Sciences Meeting, Orlando FL.

[66] Malecha, Z., Miroslaw, L., Tomczak, T., Koza, Z., Matyka, M., Tarnawski, W., Szczerba, D., 2011, “GPU-based Simulation of 3D Blood Flow in Abdominal Aorta Using OpenFOAM”, Archives of Mechanics, 62, 137-161.

[67] Asouti, V.G., Trompoukis, X.S., Kampolis, I.C., Giannakoglou, K.C., 2011, “Unsteady CFD Computations Using Vertex-Centered Finite Volumes for Unstructured Grids on Graphics Processing Units”, Int. J. Numer. Meth. Engrg., 67, 232-246.

[68] Lefebvre, M., Guillen, P., Le Gouez, J.M., Basdevant, C., 2012, “Optimizing 2D and 3D Structured Euler CFD Solvers on Graphical Processing Units”, Computers & Fluids, 70, 136-147.

[69] Niemeyer, K.E., Sung, C.J., 2014, “Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics”, The Journal of Supercomputing, 67, 528-584.

[70] Habich, J., Zeiser, T., Hager, G., Wellein, G., 2011, “Performance Analysis and Optimization Strategies for a D3Q19 Lattice Boltzmann Kernel on nVIDIA GPUs Using CUDA”, Advances in Engineering Software, 42, 266-272.

[71] Štumbauer, V., Petera, K., Štys, D., 2013, “The Lattice Boltzmann Method in Bioreactor Design and Simulation”, Mathematical and Computer Modelling, 57, 1913-1918.

[72] Domínguez, J.M., Crespo, A.J.C., Gómez-Gesteira, M., 2013, “Optimization Strategies for CPU and GPU Implementations of a Smoothed Particle Hydrodynamics Method”, Computer Physics Communications, 184, 617-627.

[73] 2014, “ArrayFire”, http://arrayfire.com/, last visited on December 2014.

[74] 2014, “PARALUTION”, http://www.paralution.com/, last visited on December 2014.

[75] Reinder, J., Jeffers, J., 2013, “*Intel Xeon Phi Coprocessor High Performance Programming*”, Morgan Kaufmann.

[76] Nithiarasu, P., 2003, "An Efficient Artificial Compressibility (AC) Scheme Based on the Characteristic Based Split (CBS) Method for Incompressible Flows." International Journal for Numerical Methods in Engineering, 1815-1845.

[77] Van Kan, J. J. I. M., 1986, "A Second-Order Accurate Pressure-Correction Scheme for Viscous Incompressible Flow", SIAM Journal on Scientific and Statistical Computing 7.3, 870-891.

[78] Guermond, J. L., Minev, P., Shen, J., 2006, "An Overview of Projection Methods for Incompressible Flows", Computer Methods in Applied Mechanics and Engineering, 6011-6045.

[79] Donea, J., Huerta, A., 2003, "*Finite Element Methods for Flow Problems*", John Wiley & Sons.

[80] Guermond, J-L., Quartapelle, L., 1997, "Calculation of Incompressible Viscous Flows by an Unconditionally Stable Projection FEM", Journal of Computational Physics, 12-33.

[81] Cecka, C., Adrian, J. L., Darve, E., 2011, "Assembly of Finite Element Methods on Graphics Processors", International Journal for Numerical Methods in Engineering, 640-669.

[82] Markall, G. R., et al., 2011, "Finite Element Assembly Strategies on Multi-and Many-Core Architectures", International Journal for Numerical Methods in Fluids, 1-8.

[83] Komatitsch, D., Michéa, D., Erlebacher, G., 2009, "Porting a High-Order Finite-Element Earthquake Modeling Application to NVIDIA Graphics Cards Using CUDA", Journal of Parallel and Distributed Computing, 451-460.

[84] 2014, "The OpenMP® API Specification for Parallel Programming", http://openmp.org/, last visited on December 2014.

[85] 2014, "Intel® Math Kernel Library", https://software.intel.com/en-us/intel-mkl, last visited on November 2014.

[86] Davis, T. A., 2006, "Direct Methods for Sparse Linear Systems", SIAM, Part of the SIAM Book Series on the *Fundamentals of Algorithms*, Philadelphia.

[87] Yang, J.-Y., Yang, S.-C., Chen, Y.-N., Hsu, C.-A, 1998, "Implicit Weighted ENO Schemes for the Three-Dimensional Incompressible Navier–Stokes Equations", J. Comput. Physics, 146, 464-487.

[88] Kays, W.M., Crawford, M.E., 1980, "*Convective Heat and Mass Transfer*", McGraw-Hill Book Company, New York.

[89] Incropera, F.P., DeWitt, D.P., 1996, "*Fundamentals of Heat and Mass Transfer*", John Wiley & Sons, New York.

[90] Lee, P.-S., Garimella, S. V., Liu, D., 2005, "Investigation of Heat Transfer in Rectangular Microchannels", Int. J. Heat Mass Transfer, 43, 1688-1704.

[91] Natarajan, N. M., Lakshmanan, S. M., 1972, “Laminar Flow in Rectangular Ducts: Predictions of Velocity Profiles and Friction Factor”, Indian J. Technol., 10, 435-438.

[92] Zhang, S.X., He, Y.L., Lauriat, G., Tao, W.Q., 2010, “Numerical Studies of Simultaneously Developing Laminar Flow and Heat Transfer in Microtubes with Thick Wall and Constant Outside Wall Temperature”, Int. J. Heat and Mass Transfer, 53, 3977-3989.

[93] Venetis, I. E., Goumas, G., Geveler, M., Ribbrock, D., 2014, “Porting FEASTFLOW to the Intel Xeon Phi: Lessons Learned”, Partnership for Advanced Computing in Europe (PRACE), 139.

[94] Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H., 2000, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia.

[95] 2014, “Top 500 The List”, http://www.top500.org/lists/2014/11/, last visited on November 2014.

[96] Farber, R., 2011, “*CUDA Application Design and Development*”, Elsevier.

APPENDIX

GLOSSARY OF TERMS IN PROGRAMMING

Term	**Explanation [3, 7, 22, 96]**
CPU	A computer unit that the logical, arithmetical, input/output and control operations of the system takes place according to instructions of a computer program.
GPU	A computer unit that specialized in rapidly manipulating graphics intended for output to a display. Also their highly parallel structure makes them effective for algorithms where processing of large chunks of data is performed in parallel.
Processor Core/Core	Units that read and execute program instructions.
Host	The CPU
Device	The GPU
Kernel	A subroutine executed on the CUDA device.
Thread	Smallest sequence of programmed instructions.
Thread Block/Block	A set of threads which have a common shared memory, and thread synchronization primitives.
Register	Fastest memory on the GPU. Exclusive to a single thread.
Shared Memory	Slower than registers, faster than global memory. Can be shared among threads of the same block.
Global Memory	Slowest but largest memory on GPU. Accessible to all threads.
Race Condition	Arises when more than one thread attempt to access the same memory location at the same time and at least one access is a write. Causes uncertainty about the final condition/value on the memory.
Throughput	The amount of output (data) that can be produced in a given period of time.
Latency	The time, the device (CPU or GPU) waits to obtain the data from memory.
Cache	A small on-die storage that stores data from earlier requests so that future requests for that data can be served faster

	(Because getting data from external memory is way slower.)
Bandwidth	The amount of data that can be transmitted in a fixed amount of time.
Speed-up	A metric for relative performance improvement when executing a task. (In this thesis study speed-up values are given according to execution time)