# POLITECNICO DI TORINO

Corso di Laurea Magistrale

in Ingegneria Aerospaziale

Tesi di Laurea Magistrale

---

# Numerical simulations of compressible turbulent flows using modern GPU architecture

---


**Relatori** :

Francesco Larocca

Andrea Ferrero

**Candidato:**

Alfredo Bianchi

Anno Accademico 2019/2020

# Abstract

The purpose of this Master Thesis is the development of numerical software to solve compressible turbulent flows using recent GPU architecture. Performance of parallel computing has been evaluated and the architecture of numerical codes has been studied in order to identify the key features for a proper use of these computational systems.

# Contents

# List of Figures

# List of Tables

# Introduction

In modern days, Computational Fluid dynamics ( **CFD** ) is widely used even in the preliminary phase of an aerospace project , as computing performance of modern machines has rapidly increased in the last decades and numerical investigation of fluid motion became a reliable tool to replace experimental collection of data in various situations . Many CFD algorithms have been developed to solve both the compressible and incompressible formulation of the flow field and with the advent of open-source software it is possible to investigate complex physic interactions, such as turbulent compressible wall-bounded flows at high Mach number , using solvers released under the GNU General Public License .

However , given the dimensions of the physical domain, we know that the numerical solution computed by these solvers is deeply influenced by the grid size . To achieve an accurate representation of the flow field we need to rise the number of grid points to reduce the truncation error, which is due to the difference between the set of Partial Differential Equations ( **PDE** ) that describes the physics of the problem and the finite numerical schemes implemented to solve them.

If the numerical schemes involved are convergent , we can ideally tend to the real solution of the problem by simply generating a really fine mesh of the fluid domain . On the other hand , it is impossible to carry out simulations on a mesh with tens of millions of grid points by writing serial source code running on ordinary Central Processing Units (**CPU**s) without facing issues related to time of the integration and memory usage .

To solve the problem with present technologies , we can parallelize the source code using Graphic Processing Units (**GPU**s) instead of CPUs . Frequently , in fluid dynamics we perform independent arithmetic operations in each node of the grid . Thus , we can think about using GPUs which are made of thousands of processing units (cores) designed to maximize throughput and not to minimize latency like CPUs.

Moreover , to achieve higher performance , we can divide the domain into several subdomains and assign each of them to a different physical device that can communicate boundary values and synchronize with the others .

This is why we need High Performance Computing centers ( **HPC** ) , especially in the academic community, to shed some light on compressible flows at high speed . HPC systems are servers made of multiple computing nodes ( CPUs and GPUs ) connected through a network and running on Linux or other Unix-like operating systems .

In the first part of this Master Thesis we present STREAmS , an open-source solver for "*large-scale, massively parallel direct numerical simulations (DNS) of compressible turbulent flows on graphical processing units (GPUs) . STREAmS is written in the Fortran 90 language and it is tailored to carry out DNS of canonical compressible wall-bounded flows* " [1]. In this work we analyze the interaction between an oblique shock wave and a turbulent compressible boundary layer ( **SBLI** ) at supersonic Mach number, focusing on evaluating strong and weak scalability of the solver. We executed the code on HPC@POLITO Legion cluster , where the single node has four NVIDIA Tesla V100 GPUs.

In the second part of the thesis we parallelized a simple 2D CFD solver of Euler equations by implementing the CUDA Fortran directives in the source code to use a single GPU of the cluster.

A comparison of physical time needed to conclude the integration of equations is reported for both the CPU and GPU versions of the code , in order to show the benefits of these technologies in future implementations of more sophisticated CFD numerical solvers.

# Chapter 1

# CUDA Fortran

In the past , GPUs were generally made for graphic applications and one of the most common personal use of them deals with the entertainment industry. In the early ‘90s these devices had specific architecture for specific fixed functions and operations on 2D or 3D data structures, like shading triangles and pixels for graphic rendering of images . Later on people realized that we can use the same hardware design and software applications in different fields that involve the processing of huge multi-dimensional data .

The newer generation of GPUs have evolved into computational units with highly parallel programming capabilities, that are easier to access through the use of libraries released both by the same GPU hardware developers , such as NVIDIA Corporation with their CUDA library , and the open source community. In scientific computing , GPUs are deeply impacting performances in applications that require the processing of massive arrays and matrices , especially when we use multiple devices connected together in HPC clusters .

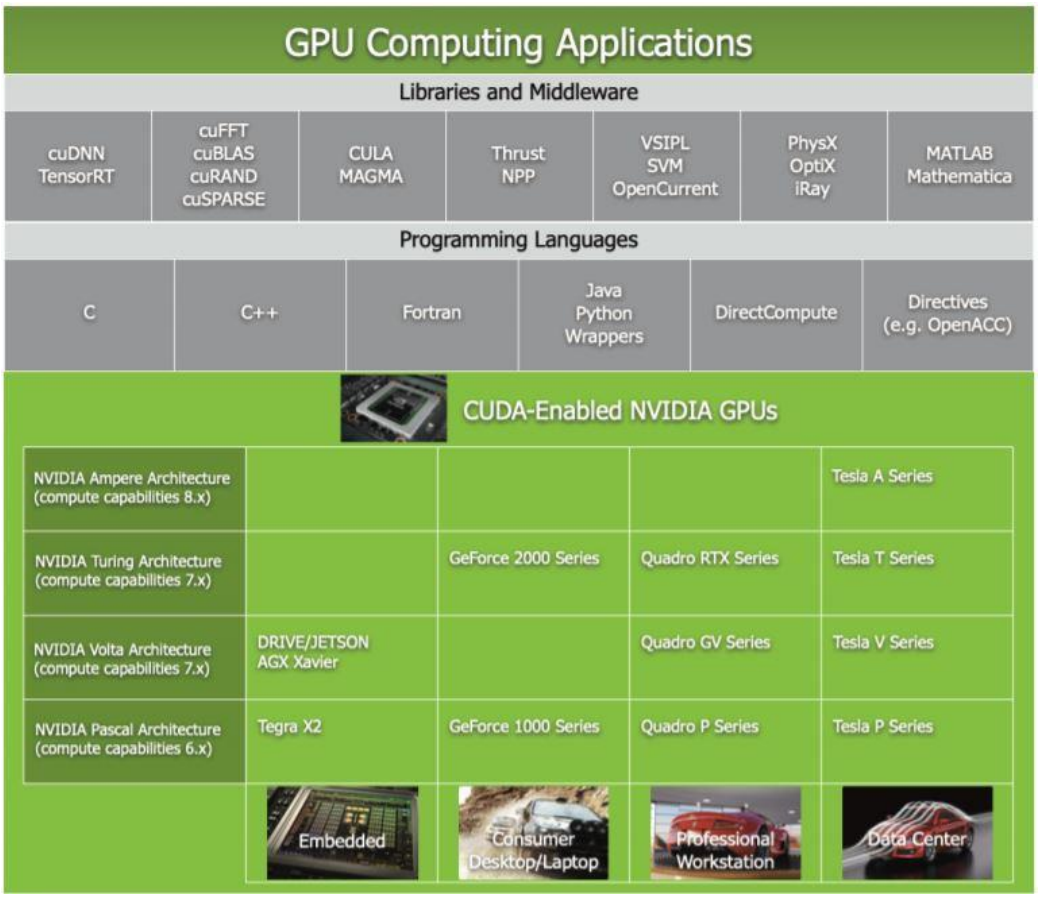



*Figure 1-1 Overview of NVIDIA GPU applications and libraries. Credits from [2]*

## 1.1 CPU and GPU architectures

A typical parallel computing system requires both a CPU and a GPU . The first one is optimized to minimize **latency ,** to be able to switch between different operations in a very short time , while the latter is optimized to maximize **throughput,** to push as many independent operations as possible through the device to process a huge amount of data .

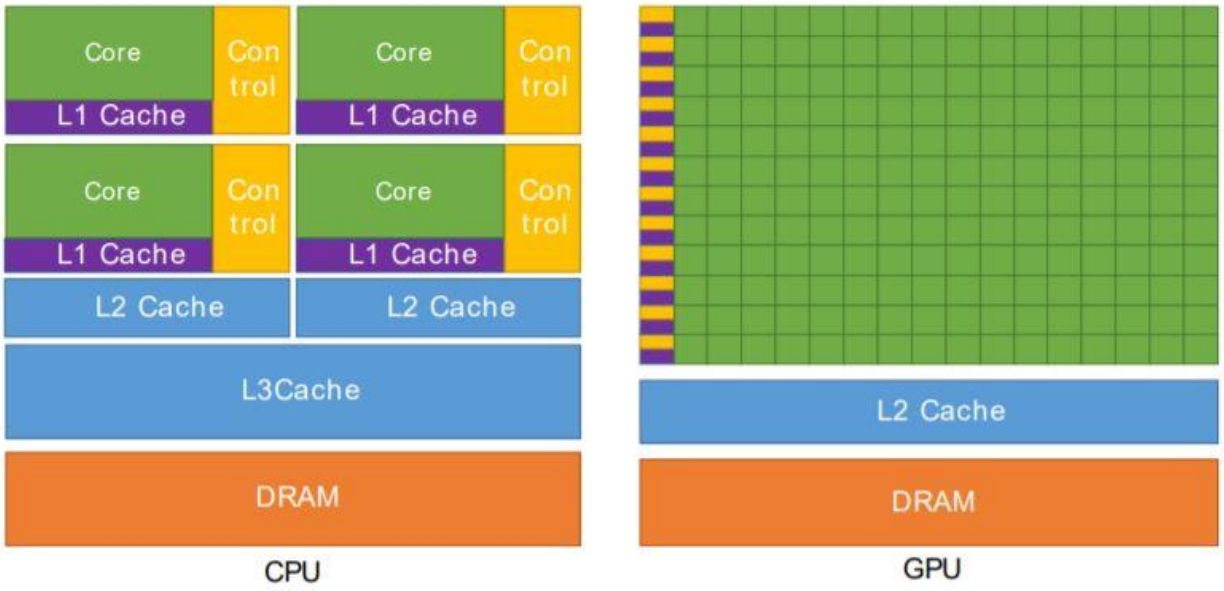



*Figure 1-2 Difference in architecture schemes between CPUs and GPUs. Credits from [2]*

The main differences in architectures are the followings :

- CPU is designed to excel at executing a sequence of operations , called a **thread** , as fast as possible , but it can execute in parallel only a few tens of these threads because of the small amount of processing units that are embedded on the chip . Generally , the infrastructure of the chip is made of :
    1. large **Cache memory** , to have readily available data that are copied from the DRAM;
    2. control flow transistors to order the execution of operations;
    3. A few Arithmetic Logic Units (**ALU** ) or **cores** , which read , process and write data back to the L Cache memory space .
- GPU is designed to excel at executing thousands of threads in parallel , amortizing the slower single-thread performance to achieve greater throughput . In GPUs the balance of CPU hardware elements is shifted :
    1. there are tons of ALUs to have sufficient work on the device , to hide the bigger latency which comes with this architecture ;

2 The L Cache is shrinked , so we have more data to be copied back and forth the DRAM;

3 Cores are grouped into arrays of Streaming Multiprocessors ( SM ) to distribute the workload efficiently across the multiple processing units .

Basically , in recent CPUs we can have about twelve or sixteen cores and we want to switch among different tasks as fast as we can by making memory accesses as short as possible to cache the data and continuing processing .

On the other hand , in GPUs it can take a longer time for processors to receive data from the memory space , but as we are running thousands of them in parallel to hide this latency and produce a massive data output , it seems like the single GPU thread has better performance than the CPU one . However, all this is possible only when we have independent operations, like graphic renderings or loops over points of a mesh grid as in CFD applications .

## 1.2 CUDA Fortran program structure

In this section we introduce and discuss the programming model for Computing Unified Device Architecture (**CUDA**) using the high-level Fortran language . We point out the fact that CUDA is a free downloadable library developed by NVIDIA Corporation, which is currently supported on Linux Operating Systems and offers routines for most common programming languages such as C , C++ or Fortran F77/F90.

**Fortran** is a key programming language for high performance computing developers , especially in the field of CFD simulations of complex systems such as high-speed turbulent compressible flows or weather and ocean modeling , rather than applications regarding finite-element analysis , molecular dynamics and quantum chemistry .

The NVIDIA CUDA Fortran Compiler provides Fortran language support for NVIDIA's CUDA-enabled GPUs . To build a Fortran source code with GPU capability we used the Portland Group Inc. ( PGI ) **PGF90** compiler in the HPC@POLITO Legion cluster with the **-Mcuda** option activated to recognize and implement the CUDA Fortran directives .

In CUDA programs we can identify the CPU as the **host** , which executes the source code and manages allocation of variables and parallel subroutine invocation on the **device** , that is the GPU.

To give a brief overview, CUDA programs must perform the following steps [3]:

a) Select the GPU to run on ;

b) Allocate memory for data on the GPU using the *attributes(device)* syntax ;

c) Move data from the host to the device via the Peripheral Component Interconnect ( **PCI** ) bus;

d) Launch kernels ( subroutines ) from the host to run on the multiple cores of the GPU ;

e) Gather results back from the GPU for further processing in the host program , again via the PCI bus ;

f) Deallocate the data on the GPU to free the device memory space.

Through the *use cudafor* syntax in the host code , we can access a Fortran module that makes possible kernel launches on the GPU and where all special CUDA memory attributes are defined .

## 1.3 Memory Hierarchy

In CUDA Fortran programs , we can allocate variables into different memory spaces . The CPU can access data into the host main memory and transfer copies of the variables to and from the device global memory [3].

The CUDA threads are organized into blocks and can cooperate using shared memory, a low-latency, high bandwidth cache memory . We can organize threads in a block using one-, two-, or three-dimensional indices accessed through the built-in *threadidx* variable.

Blocks of threads are organized in a one-, two-, or three-dimensional grid , so each thread has a thread index within the block and a block index within the grid [3] .

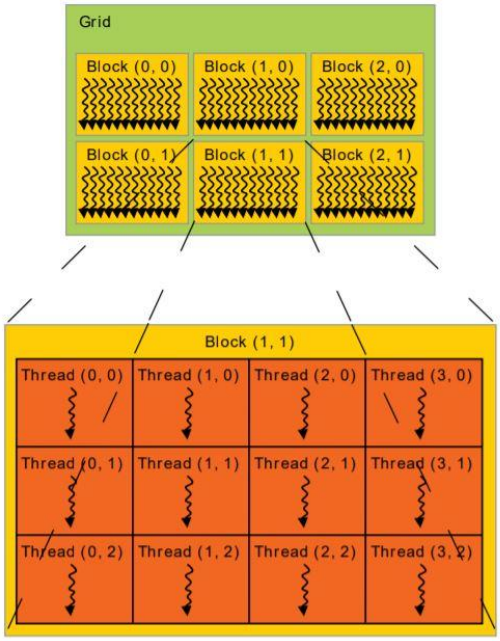



*Figure 1-3 Organization of CUDA threads on the device hardware. Credits from [2]*

For the one-dimensional case , we may know the global index of a thread by using the built-in functions :

$$i = blockidx\%x * blockdim\%x + threadidx\%x$$

Through the *dim3* derived type , defined in the *cudafor* module , we can specify variables in the host code to manage the group configuration of threads and blocks within the grid , in order to control the mapping of subroutines into CUDA cores .

On the GPU side , each thread has its private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Furthermore, all threads have access to the same global memory as shown in the following scheme [2]:

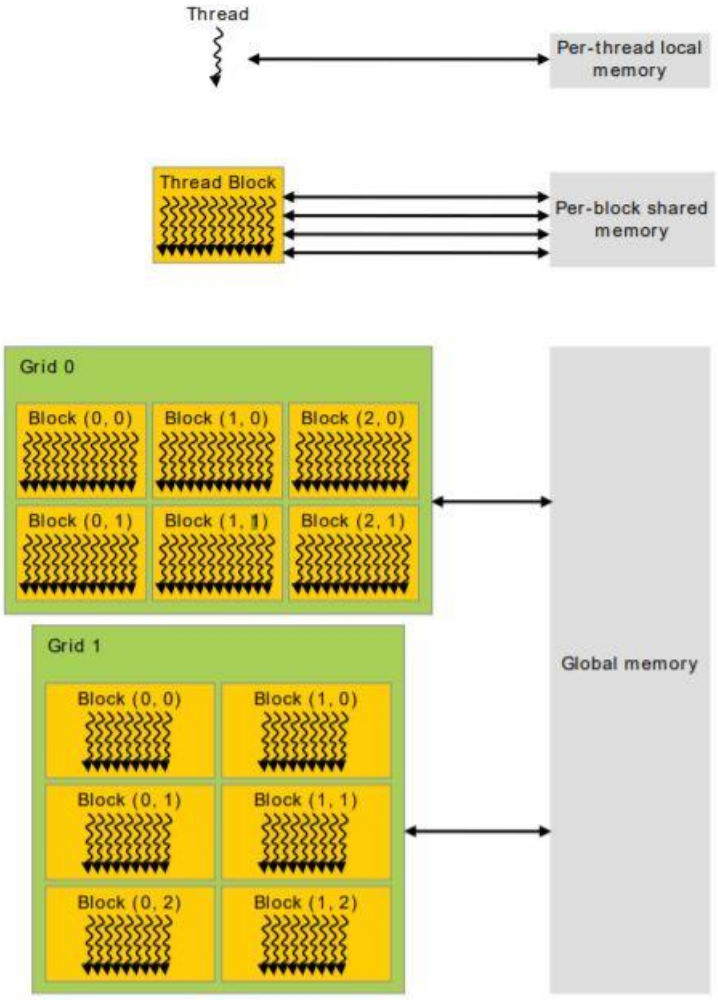



*Figure 1-4 Memory space organization on the device. Credits from [2]*

We can carry out dynamic allocation of multi-dimensional arrays through the attribute *allocatable* in the variable declaration , to avoid segmentation fault errors with high numbers of array elements.

Variables resident in the GPU memory have the *device* attribute , in Fortran we can simply copy variables from the CPU to the GPU and back with explicit assignment syntax , for instance:

$$A\_gpu = A$$

To optimize the code , we can use the *pinned* attribute for variables which are frequently transferred between host and device , to put them directly in special page-locked host physical memory, when it is available . In this way Direct Memory Access of host memory is faster, as variables are copied from virtual to physical memory only when page-locked memory is not available [3].

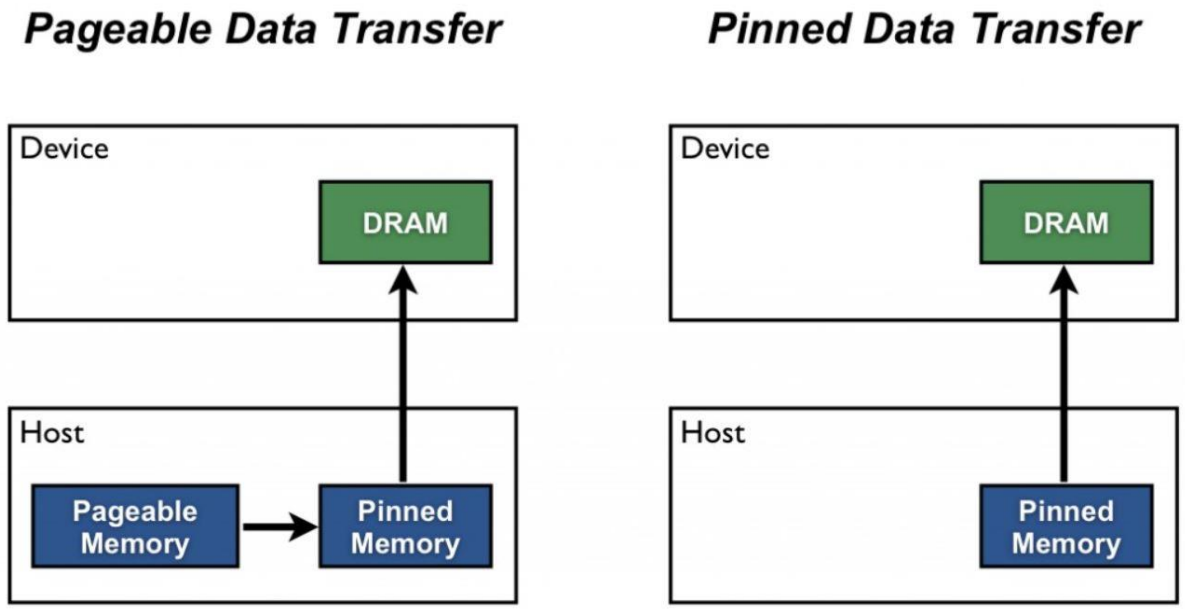



*Figure 1-5 Transfer of variables from host to device over the PCI bus , credits from [4]*

GPU read-only variables are allocated in the device constant memory space , which allows really fast accesses . These data may not be assigned or modified in any device subprogram, but can be modified in host parts of the source code and have lifetime of the entire application.

Some scalar variables may be allocated in local thread private memory , which can be implemented as processor registers or may be allocated in the global device memory , the slowest GPU memory space [3].

## 1.4 Kernel Subroutines

CUDA Fortran allows the definition of Fortran subroutines which can be executed in parallel by different CUDA threads . A thread is a computational unit on the GPU which reads data from the memory space , executes an arithmetic operation and writes data back to the memory . As GPUs are made to maximize throughput , there are thousands of cores which can run subroutines in parallel.

To define a kernel we can use the *attributes(global)* specifier on the subroutine statement , in the host code we call the subroutine to run in parallel on the GPU using special chevron syntax to specify the number of thread blocks in the grid and the threads within each block , in parenthesis we specify the subroutine parameters , for instance :

$$type(dim3) :: blocks, threads$$
$$blocks = dim3(n/256, n/16, 1)$$
$$threads = dim3(16,16,1)$$
$$call\ devkernel <<< blocks, threads >>> (\dots)$$

However , CUDA Fortran allows automatic Kernel generation and invocation from a region of host code containing nested loops [3], the kernel directive is:

$$!\$cuf\ kernel\ do(n)\ <<< *, \quad *, \quad [stream] >>>$$

After a host program launches a kernel subroutine on a queue for execution by the device , it can continue executing the following lines of the source code . By calling the routine :

$$iercuda = cudaDeviceSynchronize()$$

the host program waits until all previously Kernel executions are completed and stores integer error values in *iercuda* .

The CUDA Fortran application can manage more levels of concurrency by using multiple CUDA streams . Basically we can place operations on different queues that execute concurrently with each other on the GPU. For instance in STREAmS the evaluation of convective fluxes for internal points can be performed before receiving the boundary values from other MPI processes . Thus, the exchange of ghost node variables can be overlapped with the flux evaluation by placing it on a different CUDA stream.

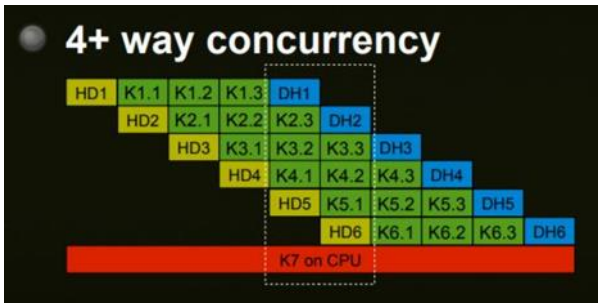



*Figure 1-6 Multiple concurrency scheme running on GPU and CPU using CUDA Streams , credits from [5]*

It is possible to create multiple streams through the routine :

$$cudaStreamCreate(stream\_number)$$

## 1.5 1D Vectorial Addition

To give an example of a typical CUDA Fortran program structure , a simple source code that executes in parallel the addition of two one-dimensional arrays A and B has been written and compiled to make an executable file which can run on a single GPU of the Legion cluster .

Firstly , in the variabili.F90 module , we define CPU and GPU versions of the arrays using special attributes introduced in the previous sections :

```
module variabili
      use cudafor
      implicit none

      integer :: N, iercuda !number of array elements
      real(4),dimension(:),allocatable,pinned :: A , B , C
      real(4),dimension(:),allocatable,device :: A_gpu , B_gpu , C_gpu

endmodule variabili
```

This module can be used in any host subprogram, but we need to remember that some variables are allocated just on the device and other ones in the host memory space , therefore we need to make sure that kernels launched on the GPU may only have access to the device version of variables otherwise a segmentation fault may occur .

In the main.F90 program we use the module and call a series of subroutines to perform the addition . We need to read with the CPU the two vectors from an input.dat file , where in the first lines we specify the number of array elements to allocate them both in the host and the device , and store the values in the CPU memory space :

```
program main

      use variabili

      call readinp

      call solver

      call writeout

endprogram main
```

In particular the solver.F90 is written as followed :

```
subroutine solver
      use variabili
```

```
      implicit none

      integer :: i

      call copy_cpu_to_gpu()

      !$cuf kernel do(1) <<<*,*>>>
      do i =1,N
            C_gpu(i) = A_gpu(i) + B_gpu(i)
      enddo
      !@cuf iercuda=cudaDeviceSynchronize()

      call copy_gpu_to_cpu()

endsubroutine solver
```

It is interesting to note that CUDA Fortran directives are preceded by a comment sign , which in Fortran language is the exclamation mark . In this way, GPU kernels are generated by the pgf90 compiler only when the -Mcuda option is activated in the makefile running on Linux . If we carefully set the preprocessor directives we may think about writing a CUDA program which can run only on the CPU when CUDA is not available , without releasing multiple versions of the source code. For the moment this is not the case as in GPU variables the special device attribute has to be defined.

We can easily identify the copying procedure from CPU to GPU in Fortran using a simple explicit assignment:

```
subroutine copy_cpu_to_gpu()


      use variabili
      implicit none

      A_gpu      = A
      B_gpu      = B

endsubroutine copy_cpu_to_gpu
```

However , we should limit these transfers as much as we can to avoid a bottleneck in computing performance , in fact the PCI bus which links the CPU and the GPU has limited bandwidth and data passage over it can slow down tremendously the Fortran application .

To make a comparison between the serial CPU version of the addition using one-dimensional loops and the GPU version exploiting multi-thread kernels , we evaluated the job time of the application through a call to the *cpu_time* routine before and after the solver subroutine in the main program .

In this way we try to compare time performance of completely serial loops with CUDA Fortran kernels , cutting off the time needed to read and write files with the CPU . We need to keep in mind

that the more workload we leave on the device without copying data back to the host to perform serial processing , the more benefits we gain regarding time to complete the application .

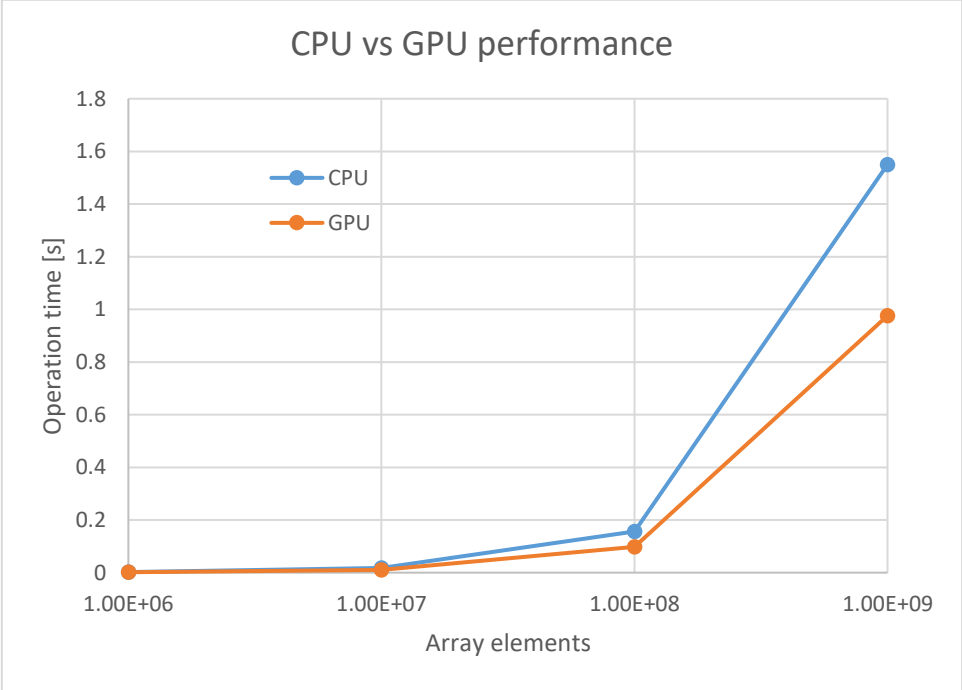



*Figure 1-7 CPU and GPU time performance without serial writing of output data.*

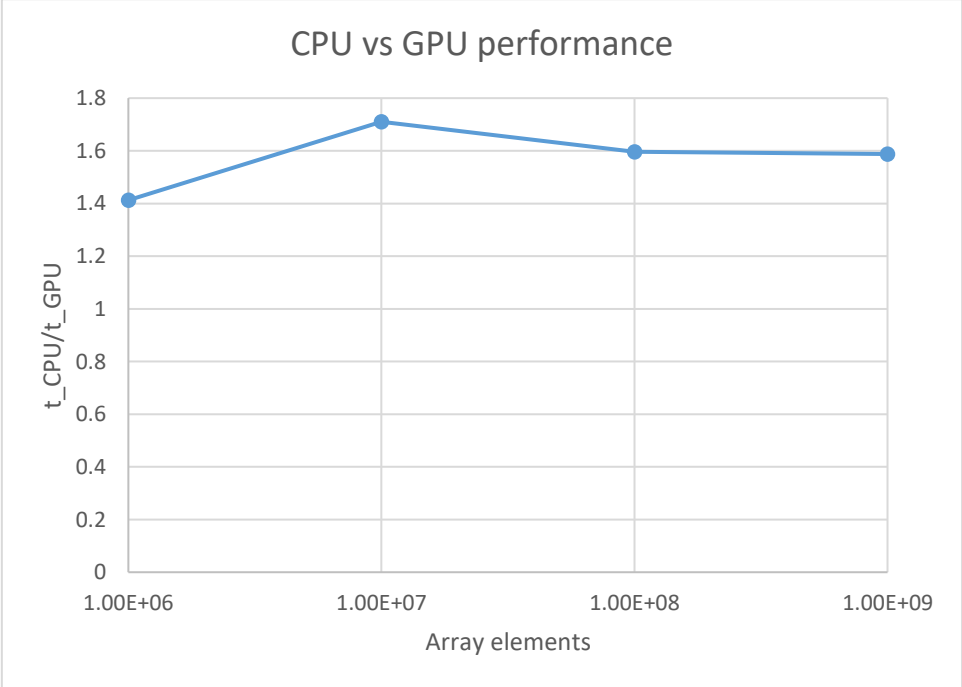



*Figure 1-8 Ratio between the CPU and the GPU runtime of the application.*

# Chapter 2

# Legion and MPI

## 2.1 Cluster technical specifications

All simulations in this master thesis have been carried out in HPC@POLITO Academic Computing Center , which provides computational resources for research activities and didactical purposes . In particular the cluster used for parallel computing was Legion , that has the following technical specifications [6] :

*Table 1 Legion - technical specification*

| | |
|---|---|
| Architecture | Cluster Linux Infiniband-EDR MIMD Distributed Shared-Memory |
| Node Interconnect | Infiniband EDR 100 Gb/s |
| Service Network Gigabit | Ethernet 1 Gb/s |
| CPU Model | 2x Intel Xeon Scalable Processors Gold 6130 2.10 GHz 16 cores |
| GPU Node | 8x nVidia Tesla V100 SXM2 - 32 GB - 5120 cuda cores |
| Performance | 21.1 TFLOPS (July 2019) |
| Computing Cores | 448 |
| Number of Nodes | 14 |
| Total RAM Memory | 5.4 TB DDR4 REGISTERED ECC |
| OS | CentOS 7.6 - OpenHPC 1.3.8.1 |
| Scheduler | SLURM 18.08 |

*Table 2 Storages technical specification*

| | |
|---|---|
| Home Storage | 140 TB on RAID 6, throughput near 200 MB/s |
| Lustre Storage | 87 TB. throughput greater than 2.2 GB/s |
| Storage Interconnect | Ethernet 10 Gb/s |

## 2.2 How to launch a Job

Each account logs into a frontend node of the cluster using a simple **ssh** client from any Linux , Unix, OSX or Windows terminal by simply typing the following line:

*$ ssh username@legionlogin.polito.it*

and inserting then the account password .

After accessing the cluster , the first directory shown is the user's */home* directory located in the Home Storage , this is one of the two directories where data have to be put in to start a job and where results will be written at the end of the task. There is one TB available for each user .

All users can access a high performance storage, implemented with the Lustre filesystem, that is really good to improve tasks making consistent I/O operations on files. This secondary directory , accessible with the command : *cd /work/username* , was set as the starting directory of each job launched on the cluster , due to the massive amount of output data computed by the CFD solvers in this thesis. Also this storage space has one TB per user by default.

Normally , jobs are launched on the cluster as *batch processes* , that means they are non interactive and their execution is managed by the SLURM scheduler , which distributes processes among cluster nodes and allocates memory resources . After submission of a job using the command :

*sbatch file_batch_legion*

it is queued on a priority line waiting for resources that my be drained by other jobs running on the same cluster partition . The special partition for GPU tasks in Legion is **cuda** , where there are 4 GPU slots per node and jobs can have a maximum duration of 5 days . This is why it is highly recommendable to use CFD numerical codes which have restarting options to continue a really long time integration .

As an example , a typical batch file with some scheduler directives (starting with **#SBATCH** syntax) for the execution of the job is shown below. It can be expected that #SBATCH commands will not be run by the Linux shell but only interpreted by the SLURM resource manager :

```
#!/bin/bash
#SBATCH --job-name=SBLI
#SBATCH --mail-type=ALL
#SBATCH --mail-user=mail@studenti.polito.it
#SBATCH --partition=cuda
#SBATCH -o output.out
#SBATCH -e output.err
#SBATCH --time=10:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
#SBATCH --gres=gpu:2
#SBATCH --mem=2GB
mpirun -n 2 ./streams.exe
```

The most important directives that have been used for simulations in this thesis are the following:

*Table 3 #SBATCH directives*

| | |
|---|---|
| --partition | Indicates the partition where the job has to be scheduled (default=global). |
| --output | the standard output is redirected to the file name specified, by default both standard output and standard error are directed to the same file. |
| -error | instruct Slurm to connect the batch script's standard error directly to the file name specified. |
| --time | Indicates the hard run time limit, which is about the time that processes needs to reach the end. This value must be less than 5 days for tasks on cuda partition. |
| --nodes | Each server is referred as a node and is associated with the –nodes directive |
| --ntasks-per-node | Give the possibility to control the number of task on one single node |
| --gres=gpu:N of gpu | Generic resource scheduling, used for specify the required number of GPU(s) per node . |
| --mem | Specify the real memory required per node, default units are megabytes. |

We need to be careful in specifying memory for the job , which is treated as a consumable resource. If requests of memory are not congruous with the job real usage , the overall throughput of the system is penalized along with our position on the priority line computed by the algorithm , therefore we need to be enough precise in demanding more memory than the default value for the job , which is one GB per core . It is advisable to estimate memory usage analyzing precedent completed jobs .

## 2.3 Message Passing Interface

MPI stands for Message Passing Interface . Generally , it is software that runs on a server and simplifies the network activity among nodes of the cluster . In this way , the application source code can run parallelly in different nodes and the exchange of information is made easier by this software library , which is available for most common programming languages such as C , C++ , Fortran 77/90.

There are different implementations of MPI , some of them are Open Source such as **OpenMPI**.

The main purpose of parallel computing is sharing the job across multiple nodes in a server . For instance, in fluid dynamics we can divide the computational domain into different subdomains and assign each of them to a different CPU/GPU of the cluster . The evaluation of numerical fluxes for internal points of the subdomains can run independently , but at a certain point we need to exchange boundary values of the conservative variables and this is done through MPI communication across different processes .

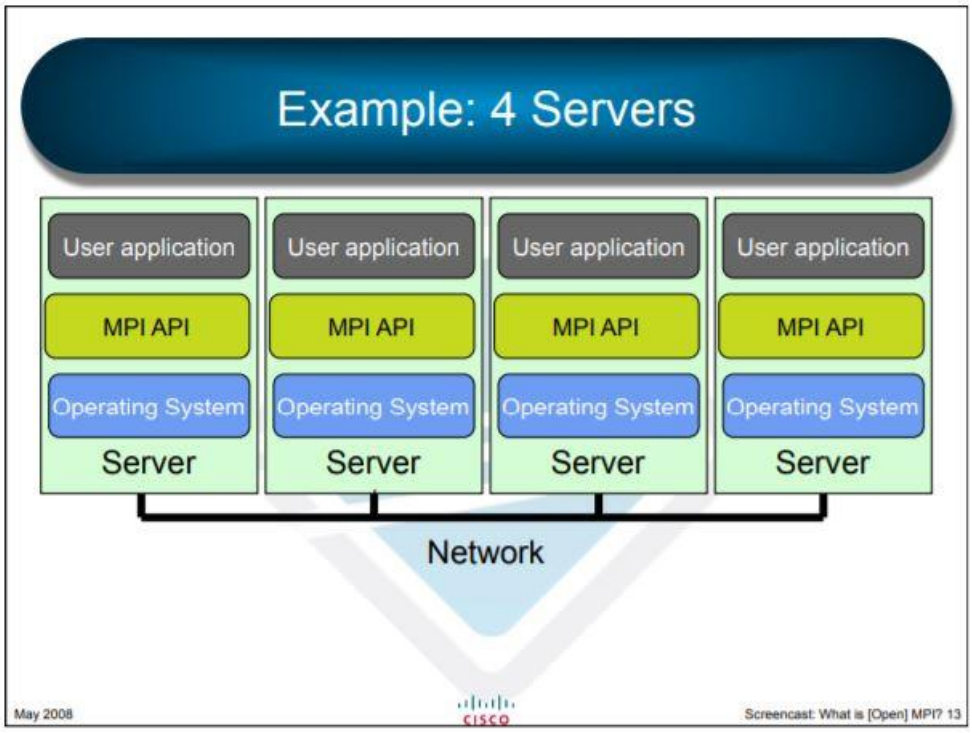



*Figure 2-1 Distribution of the job among 4 nodes of the cluster . MPI software is midware the user application and the operating system. From [7]*

However, to deliver high performance computing we need a network with low latency and high bandwidth , otherwise the communication process may overcome the benefits of sharing the job. In the past, in the first tens of seconds after the job launch there was a massive exchange of information ( Megabytes of data ) among processes running on different cluster nodes needed just to set the MPI environment .

Basically , through the MPI programming module we run in parallel multiple versions of the source code across different CPUs of the cluster , in this way in a CFD application each process can work on a single portion of the domain and communicate with other processes through the network.

For instance , in the numerical solver STREAmS used in this thesis we can set the MPI decomposition of the fluid domain along streamwise (x) and spanwise (z) direction in the input.dat file , but not in the wall-normal direction y . We need to specify the number of processes with the **-n** option when executing the **mpirun** command in the Linux shell . Then , in the program main.F90 a call to the startmpi.F90 subroutine initialize the MPI environment . Each process is assigned a **rank** number (integer), which is set through a call to the mpi_comm_rank subroutine , to identify it in the MPI communicator . Using the rank we can write if statements to make a **root process** . In this way , parts of the code can run differently among the cluster nodes when the logical variable masterproc is true, for instance:

$if(rank == 0)\ then$

$masterproc\ = .true.$

$endif$

$if\ (masterproc)\ then$

$\ldots\ operations\ to\ perform\ with\ the\ root\ process$

$else$

$operations\ to\ perform\ with\ all\ other\ processes$

$endif$

A typical format to initialize MPI processes in Fortran F90 is the following , where the input parameters are all integers :

```
call mpi_init(iermpi)
call mpi_comm_rank(mpi_comm_world,nrank,iermpi)
call mpi_comm_size(mpi_comm_world,nproc,iermpi)
```

To send and receive a message (for example boundary values ) between different processes we need to know the source and destination rank in the MPI communicator world.

Through the OpenMPI syntax we can give different structures to the default communicator . In STREAmS with startmpi.F90 subgroups of MPI processes with Cartesian topology are created using mpi_cart_create and mpi_cart_sub routines , furthermore it is possible to identify them based on their coordinates in the cartesian grid , which are set through a call to mpi_cart_coords subroutine (ncoords is a three-dimensional array of integers ) :

```
call mpi_cart_create(mpi_comm_world,ndims,nblocks,pbc,reord,mp_cart,iermpi)
call mpi_cart_coords(mp_cart,nrank,ndims,ncoords,iermpi)

call mpi_cart_sub(mp_cart,remain_dims,mp_cartx,iermpi)
call mpi_comm_rank(mp_cartx,nrank_x,iermpi)
```

To easily identify the source and destination process rank of a buffer array within the cartesian topology we use the mpi_cart_shift subroutine and store the values in ileft and iright variables to find neighbors of the considered process . For example , in the x direction .

```
call mpi_cart_shift(mp_cartx,0,1,ileftx,irightx,iermpi)
```

A single process can broadcast data to other processes in the communicator by using the *mpi_bcast* subroutine , this is the case for the target Reynolds stress matrix needed for cross-correlation of turbulence in digital filtering , computed only by the root process.

Generally, only the root process has to write standard error and standard output which are redirected to the .out and .err files. In addition to that , the master process has to allocate variables , to read the input.dat file and to generate the mesh grid.

Furthermore , we can choose only a certain subgroup of processes in the cartesian topology by selecting them based on their $ncoord$ value. This is the case in the writing of 2D statistics in STREAmS where only ( *ncoords(3) == 0 )* processes access and write to the stat output files.

## 2.4 Boundary values exchange across multiple MPI processes

As an example we analyze the bcswap.F90 subroutine in STREAmS intended to exchange boundary values across different MPI processes . Firstly , we copy ghost cell variables of each subdomain into buffer arrays through the use of **cuf** kernels to run in parallel the reading/writing procedure on the GPU.

```
!$cuf kernel do(3) <<<*,*>>>
 do k=1,nz
  do j=1,ny
   do i=1,ng
    do m=1,nv
     wbuf1s_gpu(i,j,k,m) = w_gpu(i,j,k,m)
     wbuf2s_gpu(i,j,k,m) = w_gpu(nx-ng+i,j,k,m)
    enddo
   enddo
  enddo
 enddo
 !@cuf iercuda=cudaDeviceSynchronize()
```

Secondly, we move the send buffer from the GPU to the CPU by using the cudaMemcpyAsync routine , specifying the CUDA stream2 to run the operation concurrently with Eulerian flux evaluation for internal points. The cudaStreamSynchronize call stops the execution of the host code until the copying procedure is completed.

```
cudaMemcpyAsync(wbuf1s, wbuf1s_gpu, indx, cudaMemcpyDeviceToHost, stream2)

cudaStreamSynchronize(stream2)
```

Thirdly , we can send/receive buffer arrays resident on the CPU to/from neighbor MPI processes using the *mpi_sendrecv* routine . To find the source and destination process rank within the cartesian topology we used the *mpi_cart_shift* routine in the startmpi.F90 file.

```
call
mpi_sendrecv(wbuf1s,indx,mpi_prec,ileftx,1,wbuf2r,indx,mpi_prec,irightx,1,
mp_cartx,istatus,iermpi)
```

Finally , we copy the received buffer back to the GPU using again the cudaMemcpyAsync routine . Again another call to cudaStreamSynchronize is necessary before the execution of the following lines. To conclude , the received buffer arrays are copied into ghost nodes using again the cuf kernels to run the reading/writing procedure in parallel on the GPUs.

However , in the most recent CUDA implementations we can pass device-resident variables across different MPI processes if CUDA-Aware MPI is available . The structure of the code is similar but we need to specify device-resident buffer arrays in the mpi_sendrecv subroutine.

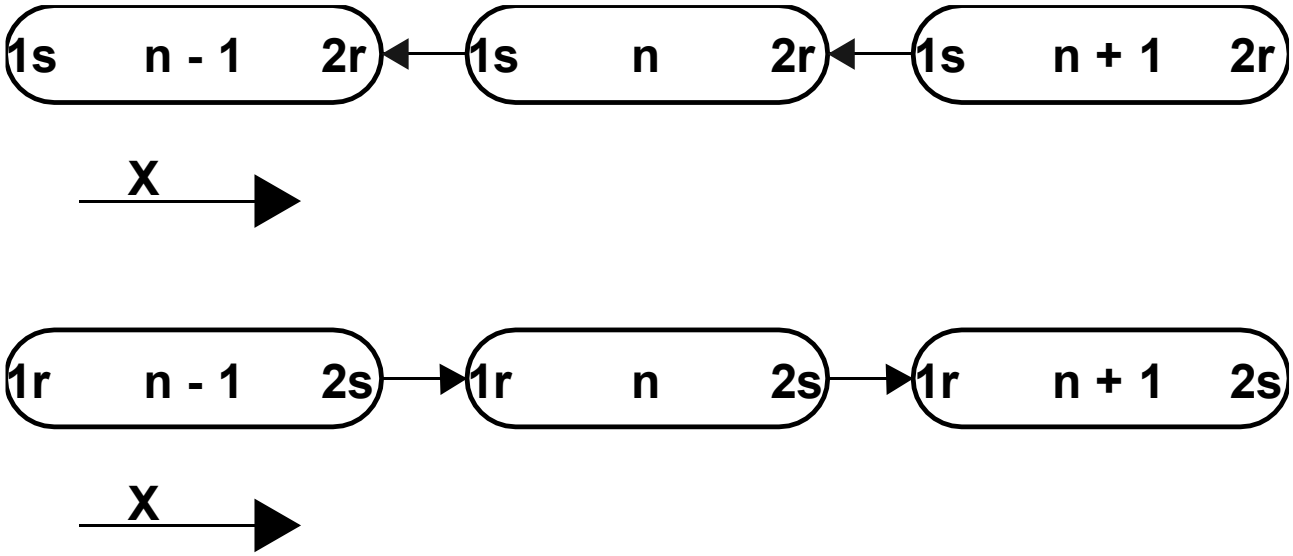



*Figure 2-2 Scheme for MPI flow of buffer arrays in X direction*

# Chapter 3

# STREAmS

STREAmS is an open-source CFD solver developed by D. Modesti , F. Salvadore, M. Bernardini and S. Pirozzoli from the "*Dipartimento di Ingegneria Meccanica e Aerospaziale, Sapienza Università di Roma" ; "Aerodynamics Group , Faculty of Aerospace Engineering , Delft"* and *"HPC Department, Cineca"* . It can be downloaded from [https://github.com/matteobernardini/STREAmS](https://github.com/matteobernardini/STREAmS) and it is used to investigate three complex cases of canonical turbulent compressible wall-bounded flows:

1. compressible turbulent channel flow
2. compressible zero-pressure-gradient turbulent boundary layer
3. supersonic oblique shock wave/turbulent boundary-layer interaction ( **SBLI** )

Fully compressible Navier-Stokes equations are solved with a finite difference approach on a Cartesian mesh , for a perfect heat-conducting gas . In this section we briefly describe the methodology followed to solve the set of equations , in particular we present a digital-filter technique that has been implemented by the authors of the program to generate artificial turbulence for the case under investigation , that is the SBLI at a supersonic Mach number . After that , some parametric studies have been carried out in Legion cluster and results have been collected and reported in this thesis , focusing on the visualization of the flow field through the open-source program **Visit** and reporting some graphs of the flow statistics using **MATLAB** .

## 3.1 Navier-Stokes equations

In fluid mechanics we can write the governing equations of viscous compressible flows in local conservative form . Given an elementary volume fixed in cartesian space, we can explicit the time derivative of conservative variables by considering the fluxes of mass , momentum and energy, for a perfect heat-conducting gas, across the multiple walls of the infinitesimal volume .

Therefore , the continuity equation in divergence form is :

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \bar{q}) = 0$$

where ρ is density and q is the velocity vector field , while ∇ is the divergence operator :

$$\bar{q} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

$$\nabla \cdot (\rho \bar{q}) = \frac{\partial (\rho u)}{\partial x} + \frac{\partial (\rho v)}{\partial y} + \frac{\partial (\rho w)}{\partial z}$$

Similarly , we can write the vectorial equation of momentum for a viscous compressible flow:

$$\frac{\partial(\rho\bar{q})}{\partial t} + \nabla \cdot (\rho\bar{q} \otimes \bar{q}) = \nabla \cdot \bar{\bar{\sigma}} + \rho\bar{f}$$

where $\bar{\bar{\sigma}}$ is the stress tensor , which has an hydrostatic and a deviatoric component :

$$\bar{\bar{\sigma}} = -p\bar{\bar{I}} + \bar{\bar{\tau}}$$

The viscous stress tensor in the relation above can be written as a function of velocity gradients as:

$$\tau_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{2}{3}\mu\,\nabla \cdot \bar{q}\,\delta_{ij}$$

$\bar{q}\bar{q}$ is a tensor of velocity components :

$$\bar{q}\otimes\bar{q} = \begin{bmatrix} uu & uv & uw \\ vu & vv & vw \\ wu & wv & ww \end{bmatrix}$$

$\mu$ is the molecular viscosity, which is a function of temperature T that can be accounted for through Sutherland's law , while $\delta_{ij}$ is the Kronecker delta . $\bar{f}$ is the vector of body mass forces such as gravity or centrifugal forces . p is pressure and $\bar{\bar{I}}$ is the identity tensor .

Finally we can write the energy scalar equation as:

$$\frac{\partial(\rho E)}{\partial t} + \nabla \cdot (\rho E\bar{q}) = \nabla \cdot (\bar{\bar{\sigma}} \cdot \bar{q}) + \rho\bar{f} \cdot \bar{q} - \nabla \cdot \bar{q}_T$$

$$\bar{q}_T = -k\nabla T = -k\frac{\partial T}{\partial x_i}\ is\ the\ heat - flux\ vector$$

$$E = c_v T + \frac{q^2}{2}$$

Where E is the total energy per unit mass , $e = c_v T$ is the fluid internal energy per unit mass and therefore the total enthalpy H can be written as a function of total energy E , pressure p and density ρ:

$$H = E + \frac{p}{\rho}$$

k is the thermal molecular conductivity which in STREAmS is deeply linked to $\mu$ with the Prandtl number Pr = 0.72 through the expression :

$$k = \frac{c_p\mu}{Pr}$$

It is impossible to find analytical procedures to solve this system of PDE equations , apart for really simple canonical cases . Hence , discrete numerical schemes are needed to approximate the real solution using algorithms on computing machines.

## 3.2 Spatial discretization

The convective terms in the Navier-Stokes equations are discretized using a hybrid energy-conservative, shock-capturing scheme in locally conservative form [1] .

Given the transported quantity φ it is possible to define the convective flux in one space direction:

$$f_x = \rho u \varphi$$
$$\varphi = 1 \; for \; continuity \; equation$$
$$\varphi = u \; for \; momentum \; equation$$
$$\varphi = H \; for \; energy \; equation$$

To discretize the convective derivatives in a mesh with equally spaced grid points we can use a finite difference approach based on Tylor expansion formula for first order derivatives:

$$\left.\frac{\partial f_x}{\partial x}\right|_i = \frac{1}{\Delta x}\left(\hat{f}_{x,i+1/2} - \hat{f}_{x,i-1/2}\right)$$

The numerical fluxes at intermediate nodes i + ½ of the mesh are obtained by defining the three-point averaging operator [1] :

$$\left(\widetilde{F,G,H}\right)_{i,l} = \frac{1}{8}(F_i + F_{i+l})(G_i + G_{i+l})(H_i + H_{i+l})$$

Therefore , it is possible to recast in conservative form the split formulation of the Eulerian fluxes [1]:

$$\hat{f}_{x,i+1/2} = 2\sum_{l=1}^{L} a_l \sum_{m=0}^{l-1} (\widetilde{\rho,u,\varphi})_{i-m,l}$$

where $a_l$ are standard coefficients for central finite-difference approximations of the first derivative , yielding order of accuracy 2 L . In smooth (shock-free) regions of the flow we use a fourth-order energy-consistent flux, which guarantees that the total kinetic energy is discretely conserved in the limit case of inviscid incompressible flow [1] .

Viscous terms are expanded to Laplacian form and also approximated with fourth-order formulas to avoid odd-even decoupling phenomena [1] :

$$\frac{\partial}{\partial x}\left(\mu\frac{\partial u}{\partial x}\Big|_i\right)\Big|_i = \frac{\partial \mu}{\partial x}\Big|_i \frac{\partial u}{\partial x}\Big|_i + \mu\frac{\partial^2 u}{\partial x^2}\Big|_i =$$

$$\frac{1}{\Delta x^2}\sum_{l=-L}^{L} a_l^2 \mu_{i+l} u_{i+l} + \mu_i \frac{1}{\Delta x^2}\sum_{l=-L}^{L} b_l u_{i+l}$$

where $b_l$ are the finite difference coefficients for the second derivative of order 2 L .

Equations are solved in a physical rectangular box of dimensions $rlx \times rly \times rlz$ where the three cartesian axes of the reference system points to streamwise x , wall-normal y and spanwise z directions. The mesh spacing is constant in wall-parallel directions x and z , while to cluster grid points towards the wall-resolved region , where velocity gradients are stronger and viscous effect are dominant for the non-slip condition , an hyperbolic sine mapping is chosen from 0 to the distance *rlywr* , which is computed after choosing the number of grid points in the wall-region in the input.dat file . Then a geometric progression is applied from y=rlywr up to y = rly to have constant spacing in the upper wall-normal direction.

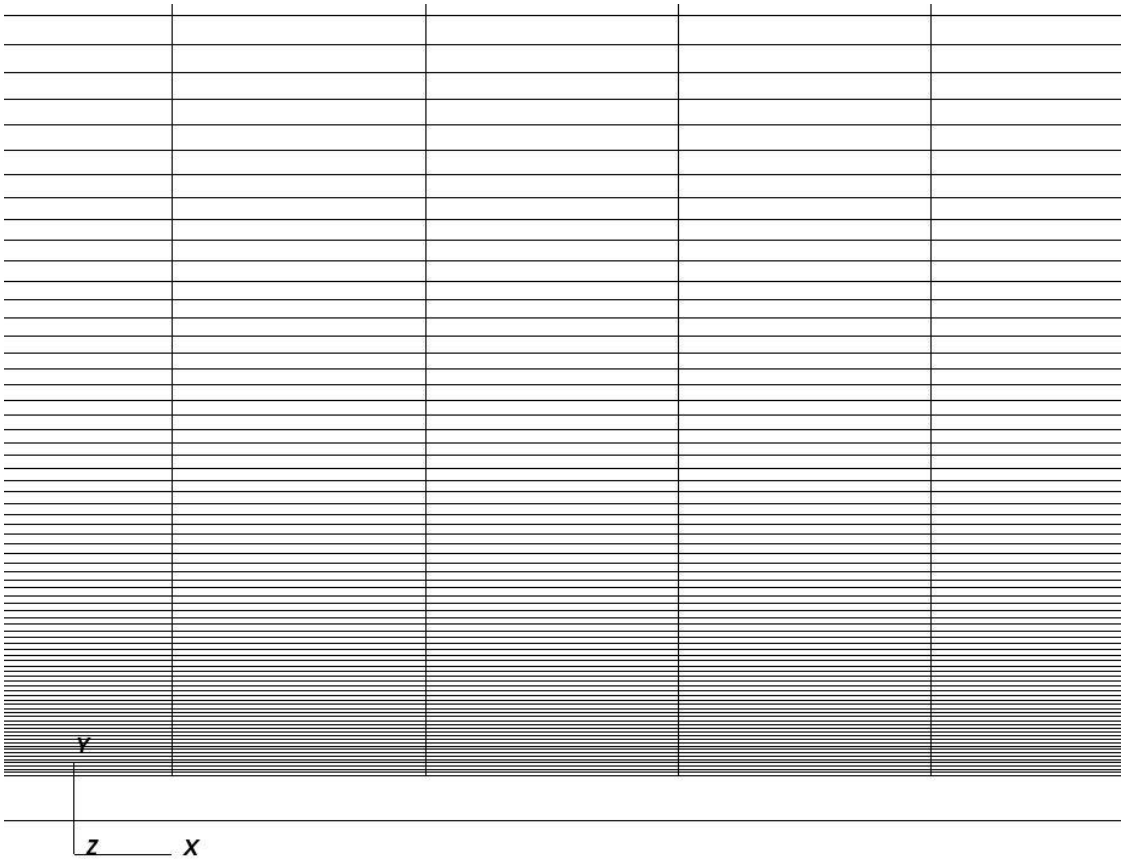



*Figure 3-1 Mesh detail of the wall region in the x-y plane*

## 3.3 Time integration

The vector of conservative variables is defined as :

$$w = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}$$

after discretizing the convective Eulerian and viscous fluxes in all space directions it is possible to perform a discrete time integration . In STREAmS the system of equations is advanced in time using Wray's three-stage third-order scheme [1] ,

$$w^{(l+1)} = w^{(l)} + \alpha_l \Delta t R^{(l-1)} + \beta_l \Delta t R^{(l)}, l = 0,1,2$$
$$w^{(0)} = w^n$$
$$w^{n+1} = w^{(3)}$$

where **R** is the vector of the residuals.

The integration coefficient are:

$$\alpha_l = \begin{bmatrix} 0 \\ 17/60 \\ -5/12 \end{bmatrix}$$
$$\beta_l = \begin{bmatrix} 8/15 \\ 5/12 \\ 3/4 \end{bmatrix}$$

It is possible to calculate primitive variables such as velocity components **(u,v,w)** , velocity module **q**, temperature **T**, pressure **p**, speed of sound ***a*** and entropy **s** using the **w** components , given the gas properties , such as $\gamma = \frac{c_p}{c_v}$ , and closing the system with the state equation for a perfect heat-conducting gas.

In STREAmS equations are in non-dimensional form . In order to avoid the specification of ambient pressure and temperature at the inlet plane their static values are set to unity , therefore the wall temperature in adiabatic conditions is equal to the recovery value :

$$\frac{T_r}{T_\infty} = 1 + \frac{(\gamma - 1)}{2} r M_\infty^2$$
$$r = Pr^{1/3} \; is \; the \; recovery \; factor$$

The inlet speed of sound and streamwise velocity are simply :

$$a_\infty = \sqrt{\gamma}$$
$$u_\infty = \sqrt{\gamma} \cdot M_\infty$$

## 3.4 Boundary conditions

It is important to point out the fact that the discrete solution is deeply affected by the boundary conditions imposed by the physics of the problem . Hence , we need to be careful in selecting those conditions for the case under investigation , otherwise the convective numerical fluxes may be completely wrong and we can never converge to the real solution of the problem .

In our case , an oblique shock wave is generated artificially from the upper wall of the rectangular domain by imposing the Rankine-Hugoniot relations , valid for inviscid compressible flows for a given flow deflection angle θ and an inlet Mach number *rm* specified in the input.dat file . In this way the discontinuity in the flow field is treated explicitly , as shock relations stem from conservation of mass , momentum and energy applied to an elementary volume fixed in space . For the case :

$$[\rho u]_1 = [\rho u]_2$$
$$[\rho u^2 + p]_1 = [\rho u^2 + p]_2$$
$$[(\rho E + p)u]_1 = [(\rho E + p)u]_2$$

By rearranging the above equations it is possible to calculate cinematic and thermodynamic properties of the flow after the shock as a function of quantities in normal direction to the shock :

$$M_{1n} = M_1 sin(\sigma)$$
$$\frac{p_2}{p_1} = 1 + \frac{2\gamma}{\gamma + 1}\left({M_{1n}}^2 - 1\right)$$
$$\frac{\rho_2}{\rho_1} = \frac{u_{1n}}{u_{2n}} = \frac{(\gamma + 1)M_{1n}^2}{(\gamma - 1)M_{1n}^2 + 2}$$

$$v_{1t} = v_{2t}$$
$$u_2 = q_2 cos(\theta)$$
$$v_2 = -q_2 sin(\theta)$$

Where $M_{1n}$ is the component of the Mach number normal to the oblique shock . The shock angle σ can be evaluated using classic $\theta - \sigma - M$ relations :

$$tan(\theta) = \frac{2}{tan(\sigma)} \cdot \left[\frac{M_1^2 (sin\sigma)^2 - 1}{M_1^2(\gamma + cos2\sigma) + 2}\right]$$

The interaction region with the turbulent boundary layer corresponds to the lower wall , where a recirculation bubble forms due to the adverse pressure gradient and a reflected shock generates from a nominal abscissa known as the impinging point .

In the rest of the domain , the solver applies shock-capturing schemes based on Lax-Friedrichs flux vector splitting , using a weighted essentially non-oscillatory ( WENO ) reconstruction of positive and negative characteristic fluxes at the interfaces . WENO is active if the shock sensor value exceeds $0 < tresduc < 1$ , that can be set in the input.dat file along with the shock deflection angle and the nominal impinging abscissa of the reflected shock . To judge the local smoothness of the numerical solution , the classic shock sensor expression is [1] :

$$\theta = max\left(\frac{-\nabla \cdot u}{\sqrt{\nabla \cdot u^2 + \nabla \times \mathrm{u}^2 + \mathrm{u}_0^2/\mathrm{L}_0}}, 0\right) \in [0,1]$$

where $u_0, L_0$ are suitable velocity and length scales , $\theta \approx 0$ in smooth regions and $\theta \approx 1$ in presence of a shock [1].

In the spanwise direction z the flow is assumed to be statistically homogeneous , therefore periodic boundary conditions are applied . At the lower wall of the rectangular domain the non-slip condition is applied to take into account viscous effects on the flow field .

At the outflow plane non-reflecting boundary conditions are imposed by performing characteristic decomposition in the direction normal to the boundary [1].

To gain a more in-depth knowledge about turbulence , artificial generation of velocity fluctuations through digital filter techniques has been studied to further investigate the boundary condition at the inlet plane of the domain .

## 3.5 Digital filter design techniques for turbulence generation

The most simple inlet condition for SBLI would only satisfy the mean turbulent velocity profile, discarding fluctuations. It may be possible to pass from a laminar to a turbulent state by inserting a region before the shock interaction where transition occurs . Unfortunately, this would lead to unnatural flow behavior with a much delayed transition to realistic turbulence [8] . From experiments we know the intensity of perturbations are of the order of 1-3 % of the averaged streamwise and wall-normal velocity profiles.

Turbulent eddies have different sizes with different energies and wave numbers associated , a typical scheme of how turbulent kinetic energy is transferred from larger size eddies to smaller isotropic ones and then dissipated into heat is shown in Figure 3-2 :

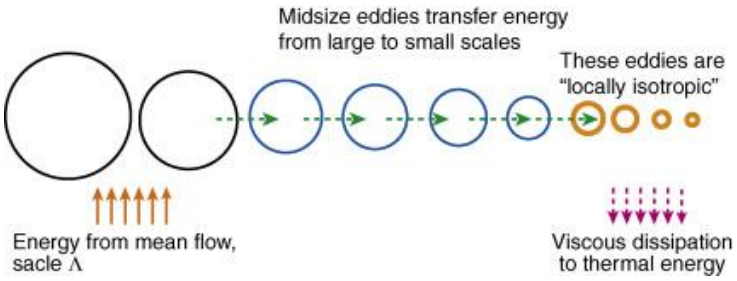



*Figure 3-2 Energy Cascade scheme for turbulent eddies , Credits from [9]*

Random white noise may be super-imposed to an average turbulent profile at the inlet to introduce perturbations to the flow and represent velocity fluctuations; however, the noise spectral density is a flat profile , because the signals are equally distributed within a fixed bandwidth , and contain far too much energy at the high (under-resolved) wave-numbers associated to smaller size eddies, with little or no energy at the well-resolved low wave-numbers associated to larger ones , which are dependent from the geometry and mean flow settings of the problem [8]. Due to this lack of energy in the low wave number range, viscous dissipation would cause a rapid damping of the perturbations, resulting in a laminar inflow even before the interaction with the oblique shock wave.

To solve the issue, Klein et al. [10] have suggested low-pass filtering and re-scaling of the white noise to obtain a more reasonable spectrum for the inlet velocity fluctuations. To give an impression of what spatial filtering of random data is , in Figure 3-3 is shown the change in greyscale picture of random white noise applying filtering firstly in the j-direction and subsequently in the i-direction .

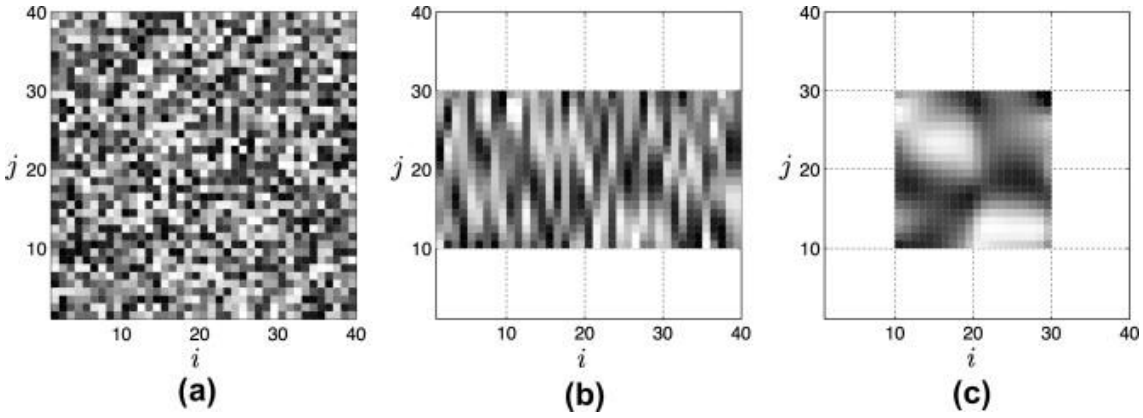



*Figure 3-3 Visualization of two-dimensional filtering of random data using greyscale picture, Credits from [8]*

Klein et al. [10] Proposed a method for generating inflow data for DNS based upon digital filters, correlation functions and length scales. The one-dimensional theory for this procedure is presented.

Given a set of random data (white noise) at each grid point *m* with mean values:

$$\overline{r_m} = 0$$

$$\overline{r_m r_m} = 1$$

we can define velocity fluctuations by creating a two-point correlation using digital filter coefficients:

$$u_m = \sum_{n=-N}^{N} b_n r_{m+n}$$

Because $\overline{r_m r_n} = 0$ for $m \neq n$ we can derive a relation between the two-point autocorrelation function and the filter coefficients:

$$\frac{\overline{u_m u_{m+k}}}{\overline{u_m u_m}} = \sum_{j=-N+k}^{N} b_j b_{j-k} \Big/ \sum_{j=-N}^{N} b_j^2$$

Based on the assumption that the two-point autocorrelation function for homogeneous turbulence in a late stage takes a Gaussian form :

$$R_{uu}(r, 0,0) = \exp\left(-\frac{\pi r^2}{4L^2}\right)$$

Where the length scale L can be prescribed and r is the distance between two grid points , in this one-dimensional case in the x direction , if $\Delta x$ is the grid spacing and $L = n\Delta x$ is the desired length scale, we can rewrite :

$$\frac{\overline{u_m u_{m+k}}}{\overline{u_m u_m}} = \exp\left(-\frac{\pi k^2}{4n^2}\right)$$

Thus , the filter coefficients can be approximated by the following analytical expression:

$$b_k \approx \widetilde{b_k} \Big/ \left(\sum_{j=-N}^{N} b_j^2\right) \; and \; \widetilde{b_k} = \exp\left(-\frac{\pi k^2}{4n^2}\right)$$

In the multi-dimensional case , we can choose for each coordinate direction corresponding to the inflow plane the length scales : $L_y = n_y \Delta y$ ; $L_z = n_z \Delta z$ and the time scale Lx . Moreover , as filter coefficients are limited in number, it is necessary to select also a filter width N according to the constraint $N_\alpha \geq 2n_\alpha$ , $\alpha = x, y, z$

Klein's algorithm first creates and stores three independent fields of random numbers $r_\alpha$ with dimensions $[-N_x : N_x, -N_y + 1 : M_y + N_y, -N_z + 1 : M_z + N_z]$ where $M_y \times M_z$ is the size of the computational grid at the inflow plane . Each field will be used for fluctuations in a separate component of the flow velocity ($\alpha$ = [$u$, $v$, $w$]).

After that it is possible to calculate filter coefficients $B_{i,j,k}$ with the prescribed integral length scales.

Considering a single slice of data at streamwise station x, for each point $j = 1, \dots, M_y$ , $k = 1, \dots, M_z$ the random fields $r_\alpha$ are then convoluted (smoothed) with the discrete low-pass filters $B_{i,j,k}$ to obtain the spatially correlated data $R_\alpha$ .

$$R_\alpha(j,k) = \sum_{i'=-N_x}^{N_x} \sum_{j'=-N_y}^{N_y} \sum_{k'=-N_z}^{N_z} b(i',j',k')\, r_\alpha(i', j+j', k+k')$$

After having defined the three-dimensional signal of correlated random data $\Psi_\alpha(x,y,z)$, $\alpha = 1,2,3$ with prescribed two point statistics within each point of the rectangular domain , we can perform the following transformation to cross-correlate the velocity fluctuations , as suggested by Lund et al. [11]:

$$u_i = \bar{u}_i + a_{ij}\Psi_j(x,y,z)$$

Where $u_i$ is the final needed velocity signal and with :

$$(a_{ij}) = \begin{pmatrix} (R_{11}^{\frac{1}{2}}) & 0 & 0 \\ R_{21}/a_{11} & (R_{22} - a_{21}^2)^{\frac{1}{2}} & 0 \\ R_{31}/a_{11} & (R_{32} - a_{21}a_{31})/a_{22} & (R_{33} - a_{31}^2 - a_{32}^2)^{\frac{1}{2}} \end{pmatrix}$$

$R_{ij}$ is the one point correlation tensor , which may be known from interpolation of experimental target Reynolds stress tensors, defined as a function of root mean squared velocity fluctuations :

$$\tau'_{ij} = \rho\overline{u'_i u'_j} = \rho\sqrt{\frac{1}{T}\int_0^T u'_i u'_j \, dt}$$

In STREAmS this step is accomplished by the subroutine *target_reystress.F90* that provides the $a_{ij}$ coefficients as a function of the wall-normal distance and the friction Reynolds number $R_\tau$ .

Now it is possible to copy velocity components $u_\alpha$ to the inflow plane and start solving the equations.

For the next iteration we need to discard the first y,z-plane of $r_\alpha$ and shift the whole data $r_\alpha(i,j,k) = r_\alpha(i+1,j,k)$ , then we can fill again the plane $r_\alpha(N_x,j,k)$ with new random numbers and repeat the spatial convolution for each time step.

As an alternative to Klein procedure , which is actually implemented in STREAmS for its lower need of computational resources , Xie e Castro [12] chose to calculate the digital filter coefficients with an exponential form rather than a Gaussian curve like Klein :

$$b_k \approx \widetilde{b_k} \Big/ \left( \sum_{j=-N}^{N} b_j^2 \right) \; and \; \widetilde{b_k} = \exp\left( -\frac{\xi \pi |k|}{n} \right)$$

The filter coefficients are normalized to ensure that $\overline{u_m u_m} = 1$ and the value of $\xi$ is chosen to minimize the following standard deviation:

$$\sigma \left[ \sum_{j=-N+k}^{N} b_j b_{j-k} \Big/ \sum_{j=-N}^{N} b_j^2 - \exp\left( -\frac{\pi |k|}{2n} \right) \right] \; for \; N \geq 2n \; and \; n = 2, \ldots 200$$

For simplicity a value of $\xi = 1$ is chosen but it is not universal. One may seek other values of ξ for other types of flows. The difference in shapes of the two-point autocorrelation function for the two methods is reported , for a fixed value of n = 1 . It can be noticed that turbulent fluctuations are less spatially correlated when a decaying exponential form is chosen :

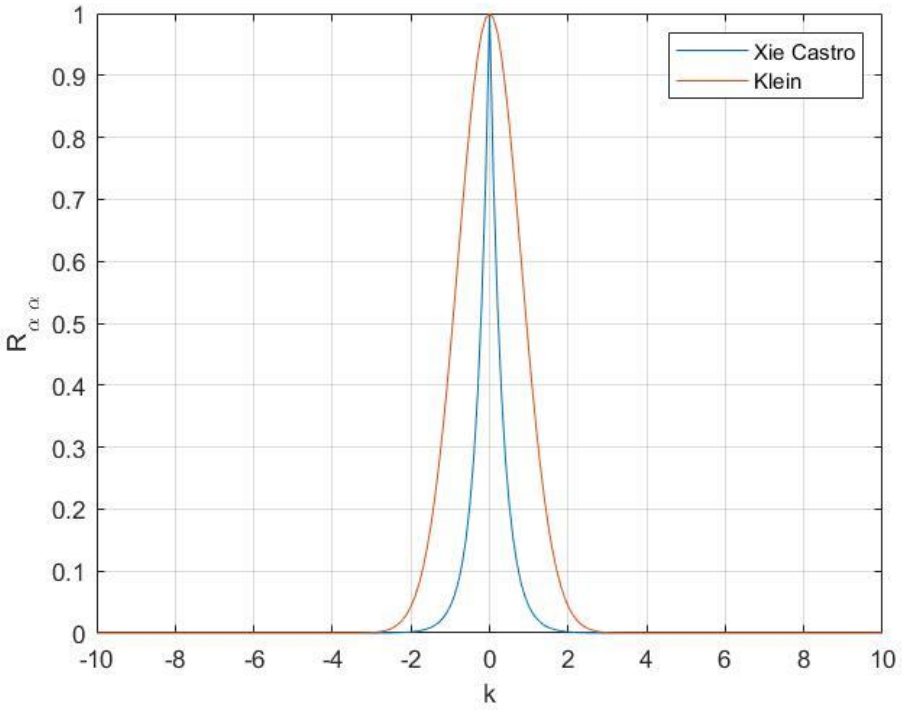



*Figure 3-4 Difference in shape of the autocorrelation functions*

Based on the work of Xie e Castro a two dimensional slice of random data of dimensions [-Ny + 1 : My + Ny , -Nz + 1 : Mz + Nz ] is generated, where My x Mz is the size of the grid, while $N_\alpha \geq 2n_\alpha$ , $\alpha = y, z$ is the filter support , which needs to be at least twice the integral length scale L.

After calculating the digital filter coefficients with prescribed integral length scales in the two directions (y, z) , we can perform a two dimensional convolution to generate spatially correlated random data , similarly to Klein procedure :

$$\Psi_\beta(t,y,z) = \underbrace{\sum_{j=-N}^{N} b_j \underbrace{\sum_{k=-N}^{N} b_k r_\beta(t, J+j, K+k)}_{filter\ in\ k-direction}}_{filter\ in\ j-direction}$$

Now at the first time step we have a slice of correlated random data for each velocity component:

$$\Psi_\beta(t,y,z), \qquad \beta = 1,2,3$$

For the next iteration, we need to generate only one more two-dimensional slice of spatially correlated random data $\psi(t,y,z)$ and we can update the inflow plane by adopting the following relation:

$$\Psi_\beta(t+\Delta t,y,z) = \Psi_\beta(t,y,z)\exp\left(-\frac{\pi\Delta t}{2T}\right) + \psi(t,y,z)\left[1-\exp\left(-\frac{\pi\Delta t}{T}\right)\right]^{0.5}$$

Where T is the Lagrangian time scale, which can be derived heuristically from previous simulations. In this expression we can notice that even the time correlation of data between $t+\Delta t$ and t assumes a decaying exponential form.

The procedure by Xie e Castro is equivalent to Klein e al. original procedure, where a three-dimensional set of random data was needed along with a three-dimensional convolution using digital filter coefficients in the three cartesian dimensions. This method calculates only a two-dimensional slice of data, but we need to know the lateral and vertical length scales as well as the Lagrangian time scale T , which can be held constant or differ depending on the inflow zone. It is generally called a Forward Stepwise Method (**FSM**).

## 3.6 Implementation of digital filters in STREAmS

In the GPU-accelerated version of STREAmS the digital filter coefficients are defined in the subroutine df_par.F90 . Here the z length scale is locally defined using outer and inner constant z length scales and a blending function of the wall normal coordinate y (*ftany*) . For each velocity component m=1,3 and wall normal index j=1,ny :

$$z_{len}(m,j) = z_{lenin}(m) + ftany * \left(z_{lenou}(m) - z_{lenin}(m)\right)$$

$$ftany = \frac{1}{2}\left[1 + tanh\left(\frac{y - 0.2}{0.03}\right)\right]$$

The inner z length scale can be expressed as a function of the inlet friction Reynolds number, which can be set in the input.dat file:

$$z_{lenin} \sim \frac{1}{Re_\tau}$$

$$Re_\tau = \frac{u_\tau \delta}{\upsilon} = \frac{\delta}{\delta_\upsilon}$$

Where δ is the boundary layer thickness at the inflow station , computed considering 99 % of the free-stream velocity , ν is the kinematic viscosity and $u_\tau = \sqrt{\tau_w/\rho_w}$ is the friction velocity, which is a function of the wall shear stress $\tau_w$ and wall density $\rho_w$ . $\delta_\upsilon = \upsilon/u_\tau$ are viscous units .

*Table 4 Inner and outer integral length scales in z direction*

| | |
|---|---|
| $z_{lenout}(1)$ | 0.4 |
| $z_{lenout}(2)$ | 0.3 |
| $z_{lenout}(3)$ | 0.4 |
| $z_{lenin}(1)$ | $min(150/Re_\tau , z_{lenout}(1))$ |
| $z_{lenin}(2)$ | $min(75/Re_\tau , z_{lenout}(2))$ |
| $z_{lenin}(3)$ | $min(150/Re_\tau , z_{lenout}(3))$ |

Following Xie e Castro we can derive the y length scale which is proportional to the z length scale:

$$y_{len}(m,j) = 0.7 * z_{len}(m,j)$$

The x length scale is assumed constant , thus the Langrangian time scale T can be expressed as a function of the inlet free-stream velocity $u_0$ . The values for the three velocity components are:

*Table 5 Streamwise and Lagrangian length scales*

| | |
|---|---|
| $x_{len}(1)$ | 0.8 |
| $x_{len}(2)$ | 0.3 |
| $x_{len}(3)$ | 0.3 |
| $T(1)$ | $x_{len}(1)/u_0$ |
| $T(2)$ | $x_{len}(2)/u_0$ |
| $T(3)$ | $x_{len}(3)/u_0$ |

Following Xie e Castro we can evaluate the filter coefficients $b(i,j,k)$ with an analytical expression.

It is important to note that the size of turbulent eddies is limited by the distance to the wall. As we approach the lower boundary, both the wall-normal and spanwise length scales must go to zero. That is why we need a blending function of the wall-normal coordinate to scale the superimposed length scales.

The mixing length $l_m$ in turbulence is linked to the dissipation rate $\epsilon$ of turbulent kinetic energy k . Hence, reducing the integral length scale could generate turbulent velocity fluctuations with a higher energy decay into heat :

$$k = \frac{1}{2}\left(\overline{(u')^2} + \overline{(v')^2} + \overline{(w')^2}\right)$$

$$\epsilon \propto \frac{k^{3/2}}{l_m}$$

We can estimate first and second order velocity moments and integral length scales from DNS of previous simulated cases available in databases. Given a set of two-point correlated data from the database, we can calculate Integral length scales by the following equation:

$$L_i = \int_0^\infty R(x_i)dx_i\,,\ i = 1,\ 2,\ 3,$$

Where $R_{\alpha\alpha}$ is the autocorrelation function of the α velocity component (typically a Gaussian or decaying exponential curve ). For example , Xie e Castro estimated $L_i$ based on DNS of plane channel flows by Kasagi's group (http://www.thtlab.t.u-tokyo.ac.jp/) .

In STREAmS the computation is initialized by prescribing a mean fully developed turbulent compressible boundary layer , obtained by applying the Van Driest transformation to an incompressible velocity profile of the Musker family ($\bar{u}$). For the effective velocity in the outer part of the boundary layer [13] :

$$u^+_{VD} = \frac{1}{k}log(y^+) + C$$

$$du_{VD} = \left(\frac{\bar{\rho}}{\overline{\rho_w}}\right)^{\frac{1}{2}} d\bar{u}$$

$$y^+ = \frac{y}{\delta_v}$$

In the input.dat file one can set the ratio $T_{rat}$ between wall temperature and adiabatic wall temperature, which is equal to the recovery temperature:

$$\frac{T_r}{T_\infty} = 1 + r\frac{(\gamma - 1)}{2} M_\infty^2$$

$$r = P_r^{\frac{1}{3}}$$

Hence, we can consider $T_{rat}$ as another parameter for turbulence, as the Van Driest mean streamwise velocity changes according to density, which is related to the temperature ratio $T_{rat}$ . The kind of velocity profile we expect before the interaction region is the following ( with a logarithmic scale for the horizontal axis ) , where a linear and a log-law region stems from the results in the inner and outer part of the boundary layer:

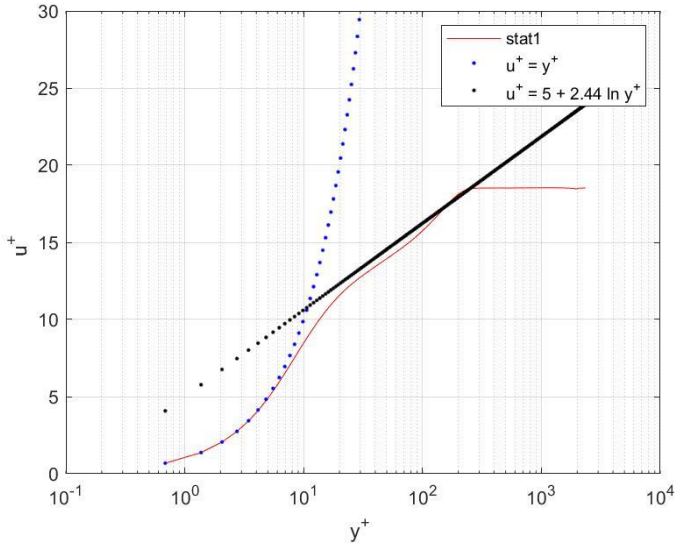



*Figure 3-5 Mean streamwise velocity profile in the inlet region*

In the file initurb.F90 generation of random fields and convolutions in the three cartesian directions are performed to initialize the inflow velocity. Fluctuations are derived following Klein procedure and superimposed to the mean velocity profile . Finally the vector of conservative variables is calculated and the initial condition of the flow field is defined.

In the file bcdf.F90 inflow turbulent boundary conditions are updated in time using Xie e Castro procedure , in this way we have a set of random data spatially and time correlated for each iteration. We can modify the two-point statistics by changing the integral length scales , and the one point cross-correlation function by changing the Reynolds target stress tensor in the database_bl.dat file.

The support of the digital filter nfmax is defined in mod_streams.F90 and it is set to a constant value of 64 .

# Chapter 4

# Strong and weak scalability of STREAmS

## 4.1 Strong scalability

Performance of parallel computing has been evaluated for SBLI numerical simulations incrementing number of MPI processes launched on Legion cluster and therefore scaling up the computational resources ( CPUs and GPUs ) for the job .

The following simulations are carried out on a rectangular domain of lengths :

$$rlx = 70$$
$$rly = 12$$
$$rlz = 3$$

The inflow Mach number is M = 2.28 , the friction Reynolds number is $Re_\tau = 475$ . Wall temperature is set equal to the adiabatic wall temperature and shock wave angle is set equal to 8 deg. The nominal impinging point of the reflected shock is $x_{imp} = 40$.

Time performance is the **Job wall clock-time** , that is the total physical runtime needed to complete the job on the cluster , multiplied by the CPU efficiency . From the results , we can notice that increasing the number of GPUs reduces rapidly the integration time when the job is shared in the same cluster node (where we can use a maximum of 4 GPUs ) , probably because the communication time needed to exchange boundary values across MPI processes is much higher when multiple nodes are involved. Speed-up is time ratio between reference case (1 GPU) and job execution with N GPUs.

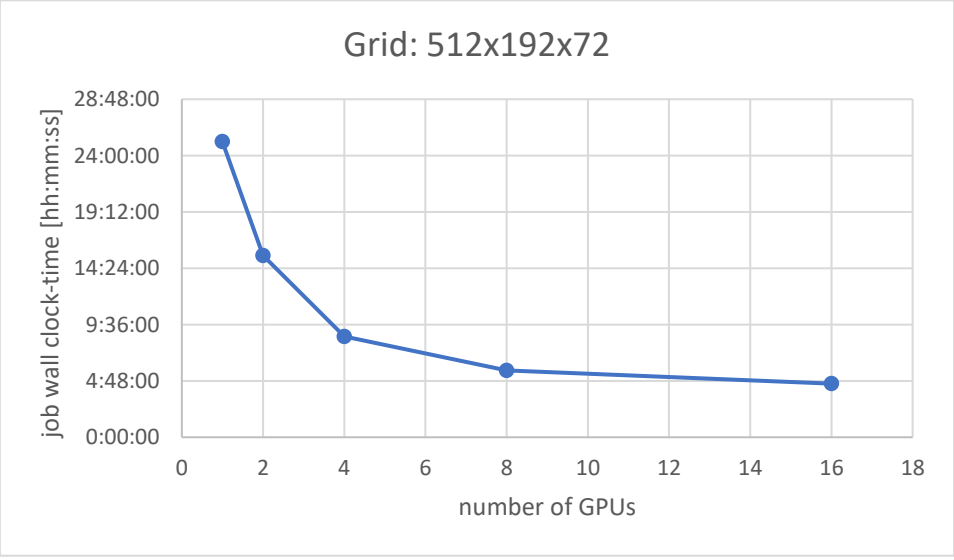



*Figure 4-1 Job time for a grid of 7077888 points incrementing number of GPU and MPI processes , 500000 iterations.*

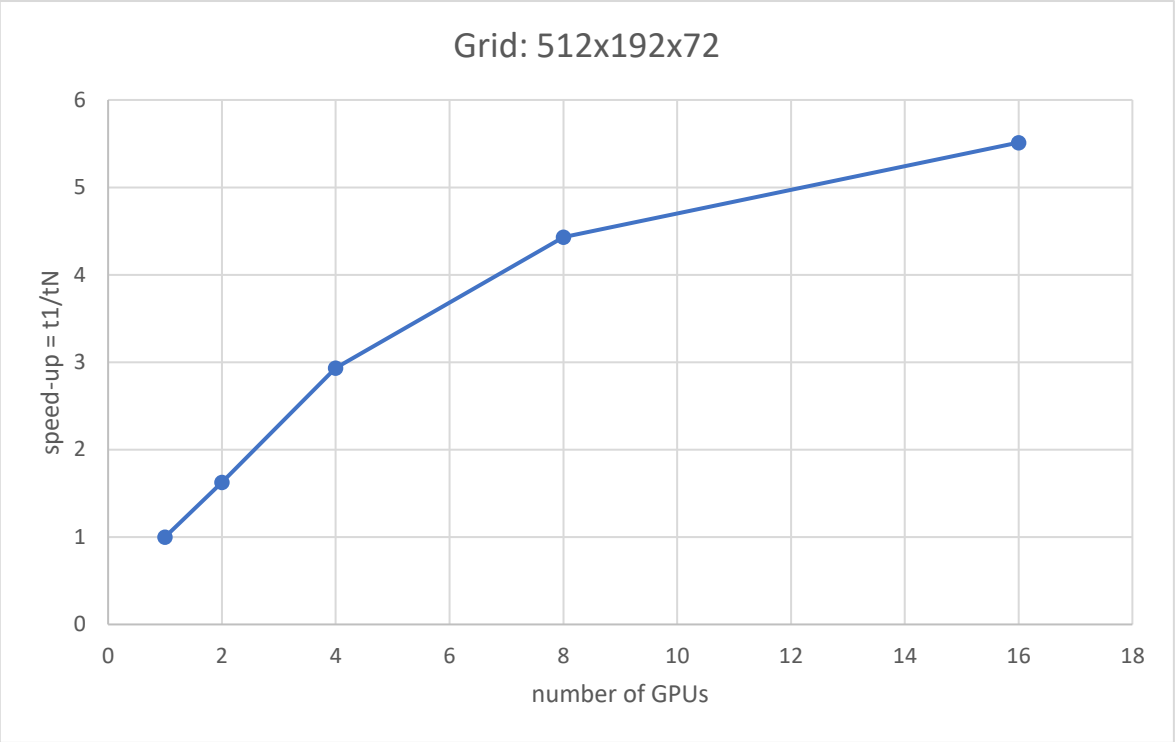



*Figure 4-2 speed-up for a grid of 7077888 points, incrementing number of GPU and MPI processes, 500000 iterations.*

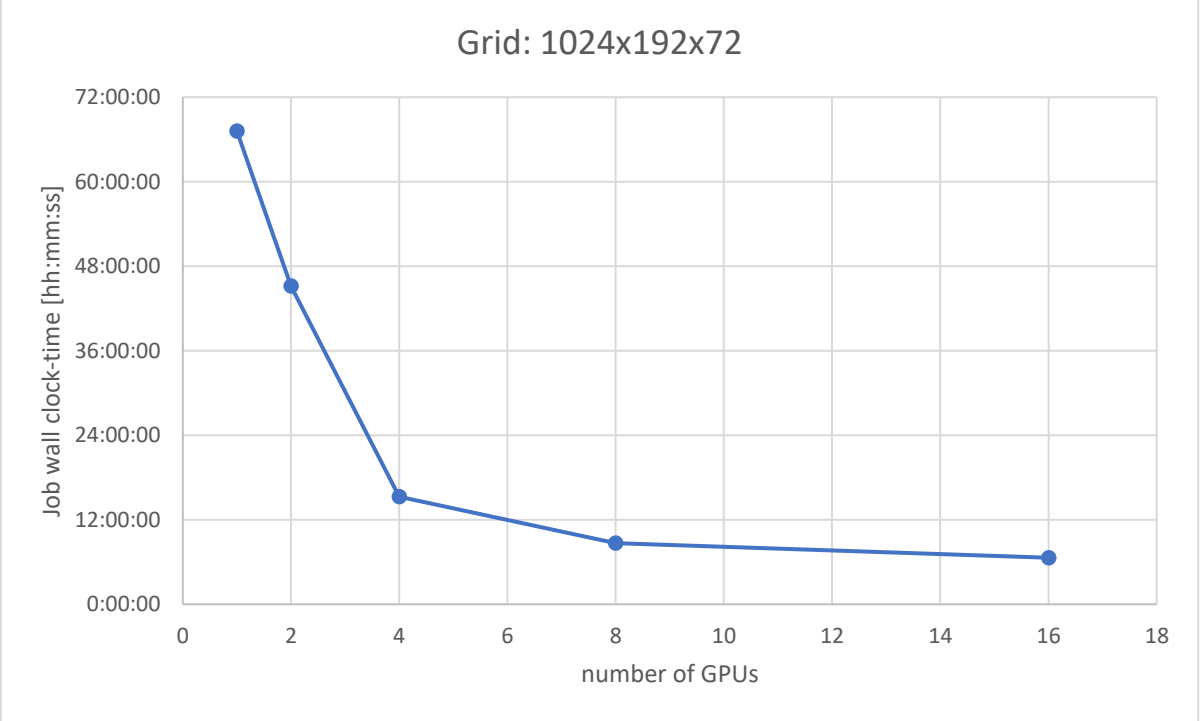



*Figure 4-3 Job time for a grid of 14155776 points incrementing number of GPU and MPI processes , 500000 iterations.*

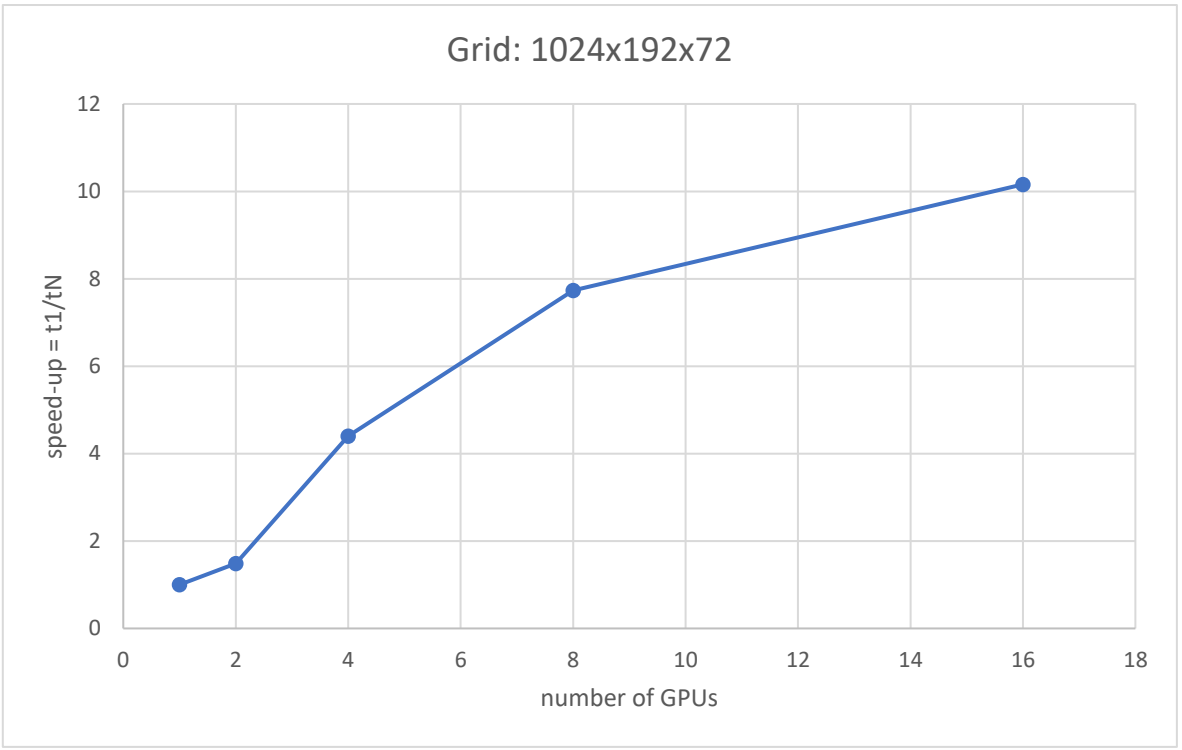



*Figure 4-4 Speed-up for a grid of 14155776 points incrementing number of GPU and MPI processes , 500000 iterations.*

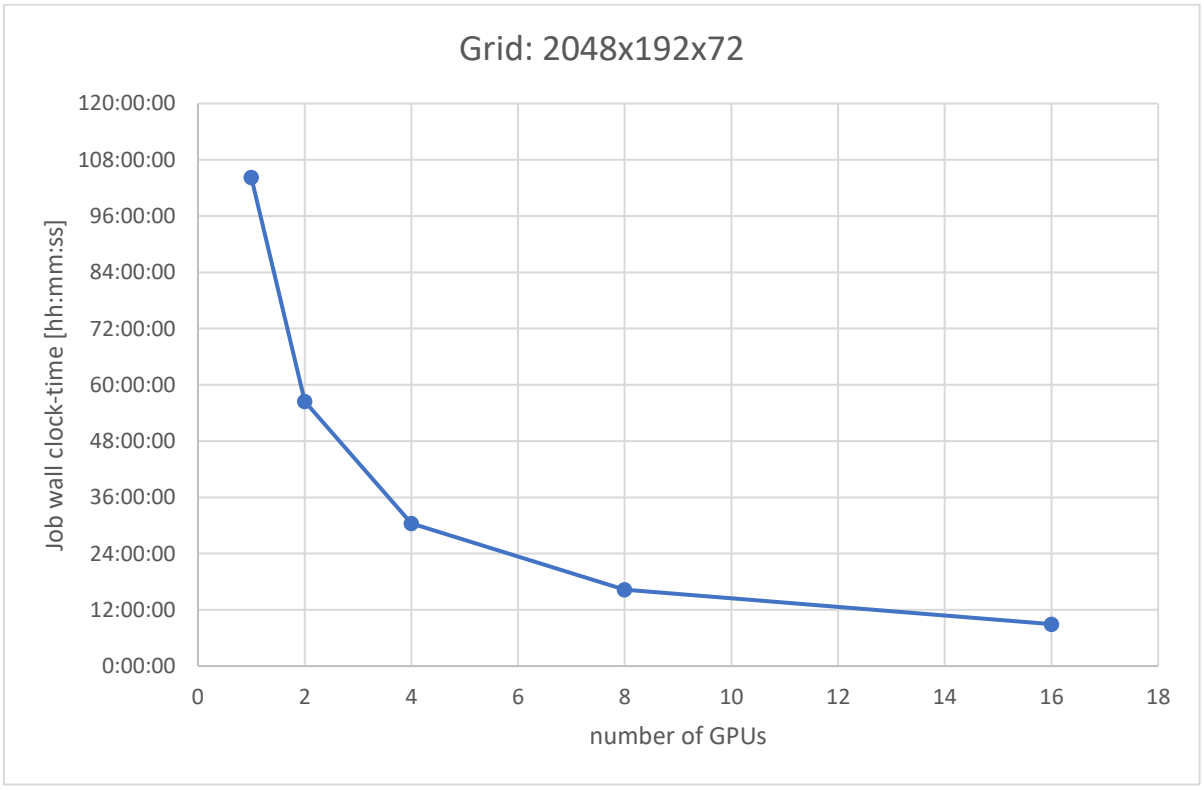



*Figure 4-5 Job time for a grid of 28311552 points incrementing number of GPU and MPI processes , 500000 iterations.*

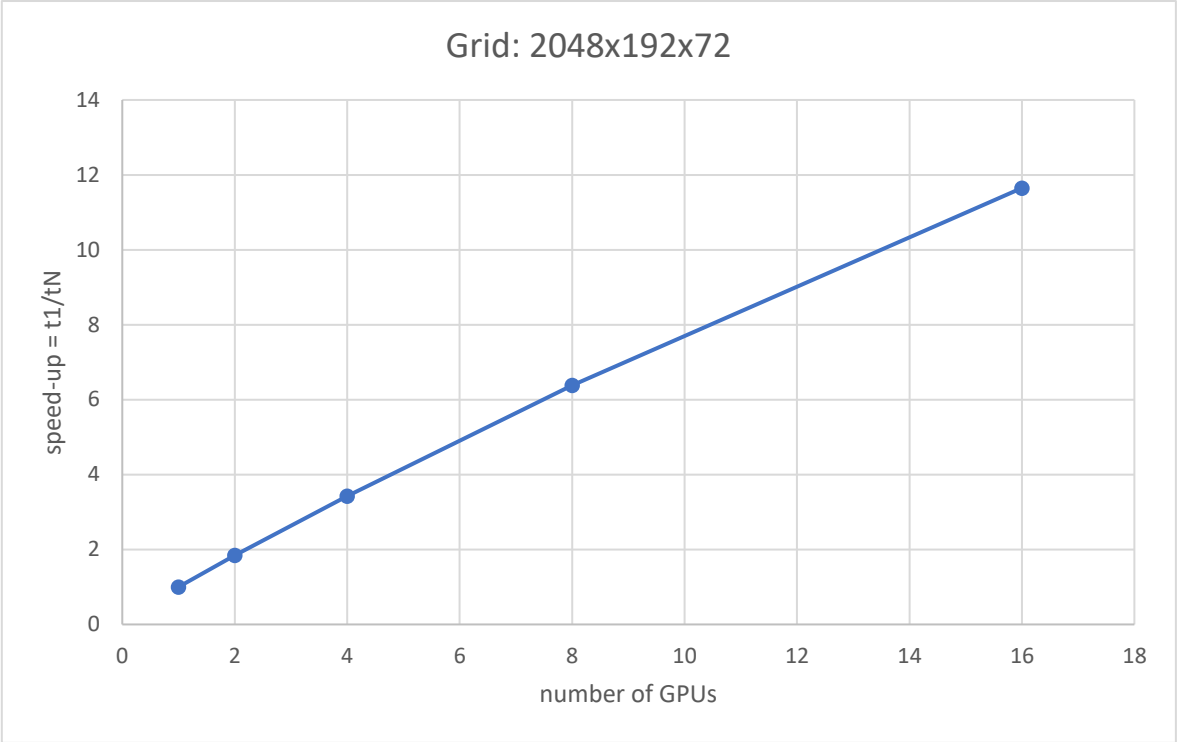



*Figure 4-6 speed-up for a grid of 28311552 points incrementing number of GPU and MPI processes , 500000 iterations.*

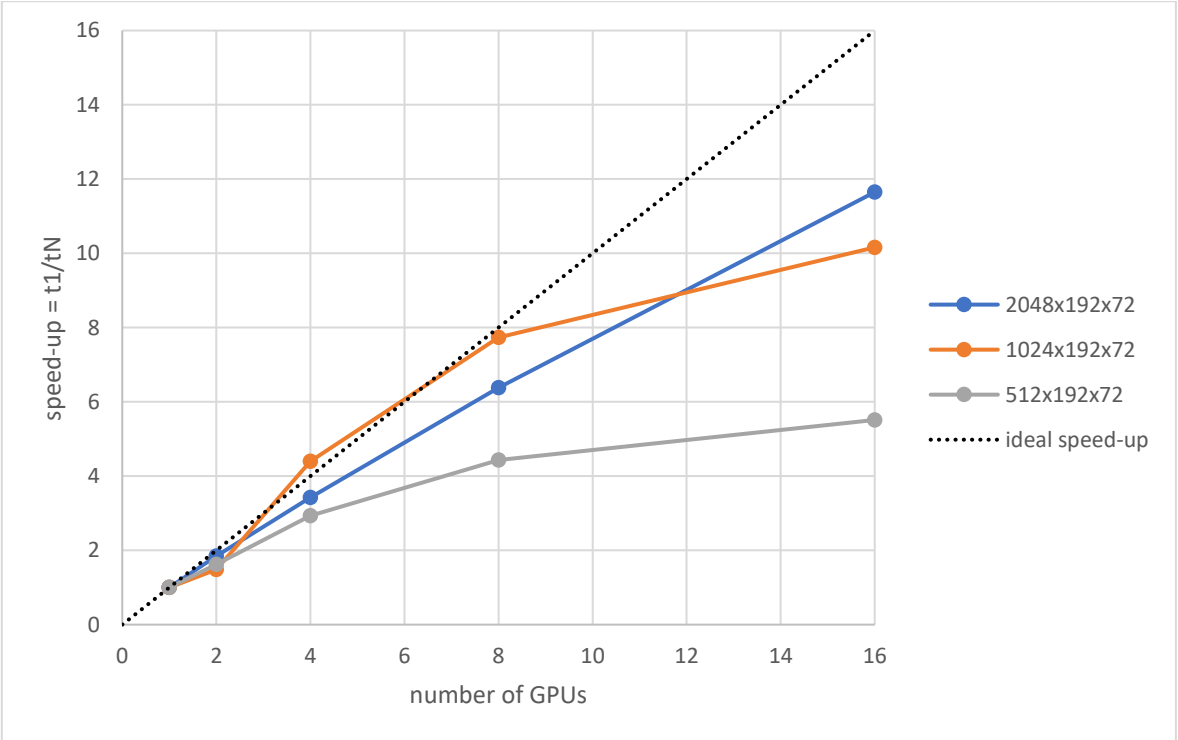



*Figure 4-7 Comparison of speed-up performances for the three grid cases*

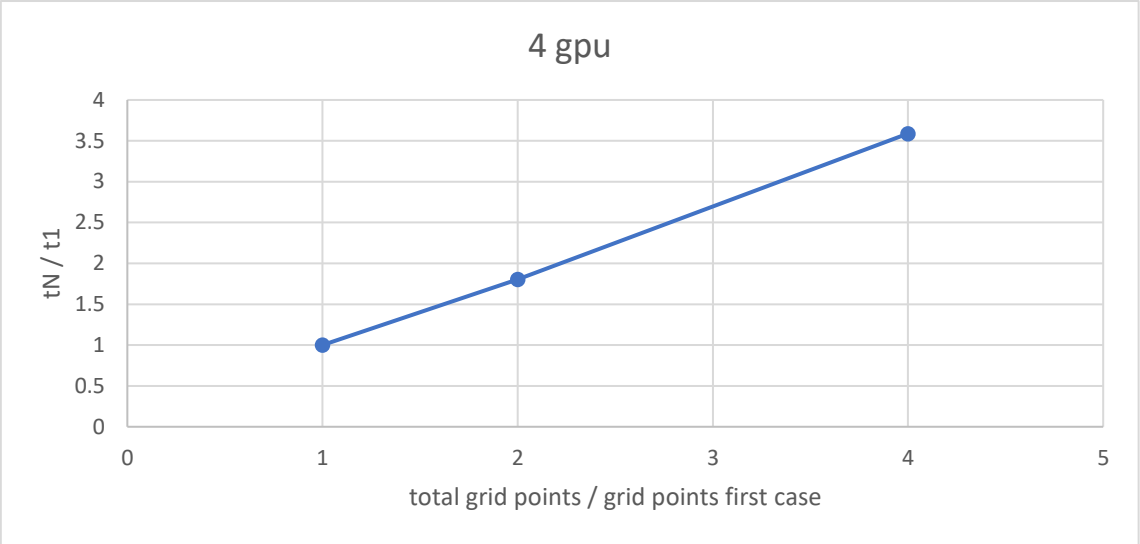


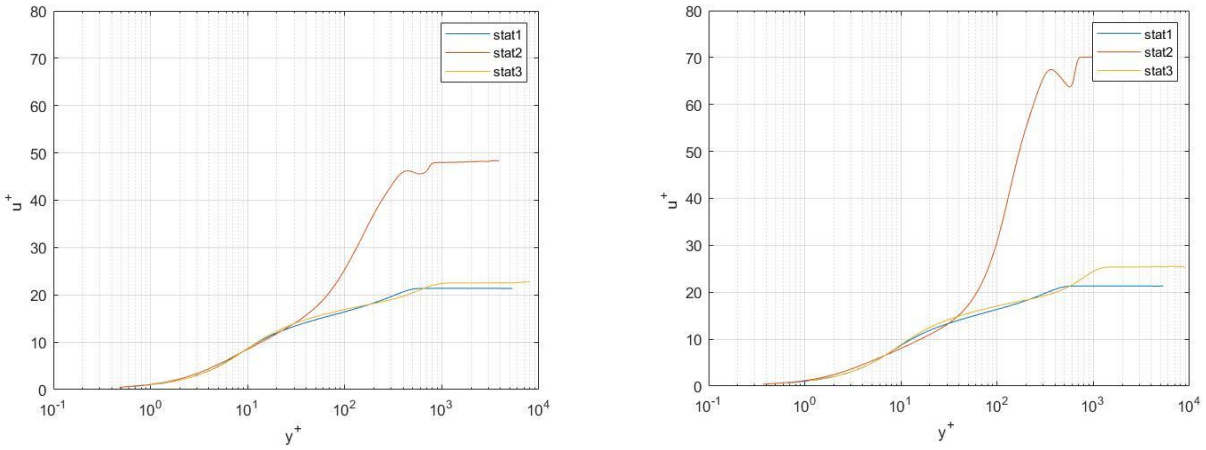

*Figure 4-8 Job time ratio tN/t1 for a fixed number of GPUs ( 1 node , 4 GPUs ) incrementing number of grid point ratio with the first case grid size . 500000 iterations.*

The main statistics computed by the program are the following:

1) Favre averaged streamwise velocity, normalized with the friction velocity. Statistics are collected at the following streamwise coordinates :

$$x = [\,20\,,\ 40,\ 60\,]$$

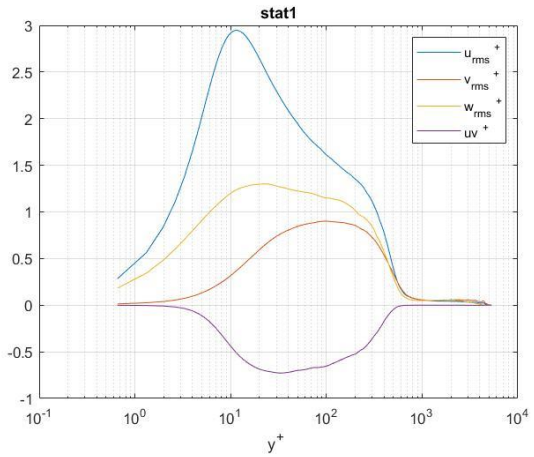
Grid size is : nx = 2048 ; ny = 192 ; nz = 72. Statistics are collected for 100000 and 500000 iterations, corresponding to a physical time of 41 [s] and 205 [s] . $y^+ = \frac{y}{\delta_v}$ is the wall-normal coordinate , normalized with the viscous length scale. It is interesting to note the thickening of the incoming boundary layer caused by the shock in the interaction region , which is close to the x = 40 station , the boundary layer relaxes again to an equilibrium state after the reflected shock .

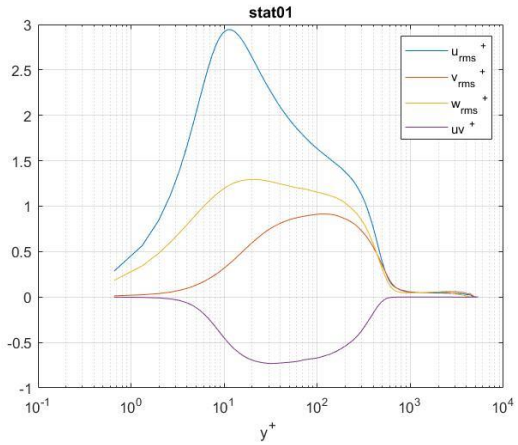



2) Density scaled streamwise , wall-normal and spanwise velocity root mean square , in viscous units . $uv^+$ is Reynolds turbulent shear stress in viscous units. Statistics are collected at

streamwise coordinate $x = 20$ , located in the region before the impinging shock wave, for 100000 and 500000 iterations, corresponding to a phisical time of 41 [s] and 205 [s]. There is almost no change in statistics as the oblique shock wave is located downstream the considered x station. We can notice that the fluctuation level must go to zero in the viscous sublayer nearby the wall , where laminar viscous stresses are dominant , and again at the upper edge of the boundary layer where the flow field is almost irrotational .

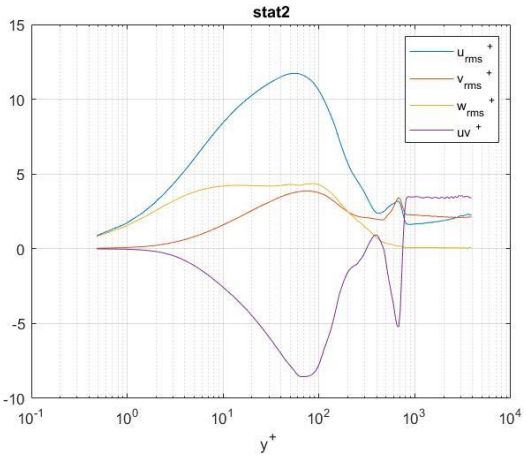
Grid size is : nx = 2048 ; ny = 192 ; nz = 72.

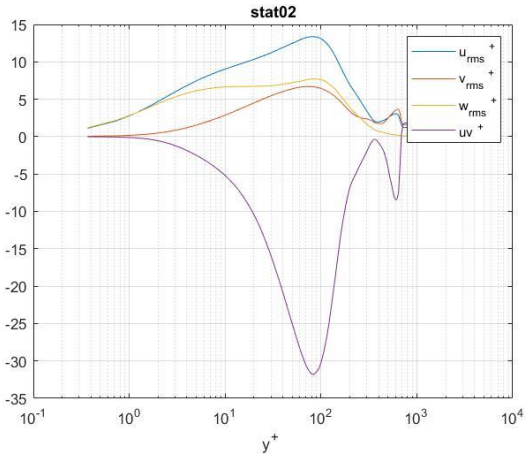


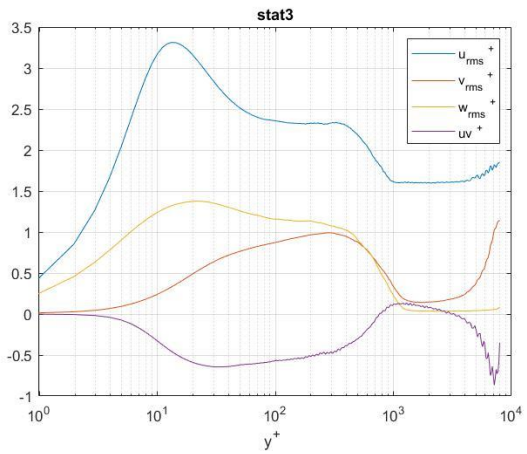

3) Density scaled streamwise , wall-normal and spanwise velocity rms , in viscous units . $uv^+$ is Reynolds turbulent shear stress in viscous units. Statistics are collected at streamwise coordinate $x = 40$ , located in the region between the impinging and reflected shock wave, for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s]. This is the region where we have higher turbulent fluctuations due to the recirculation bubble that forms for the adverse pressure gradient caused by the impinging and reflected shock .

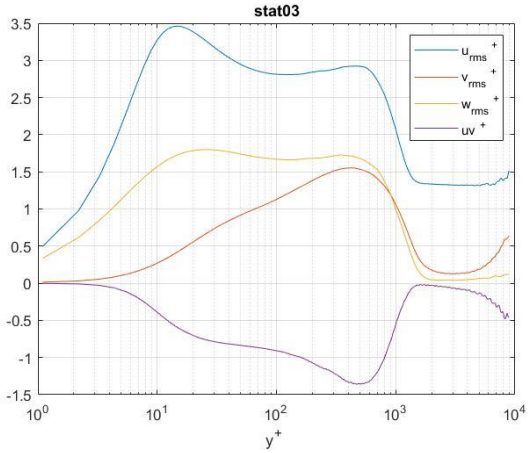
Grid size is : nx = 2048 ; ny = 192; nz = 72.



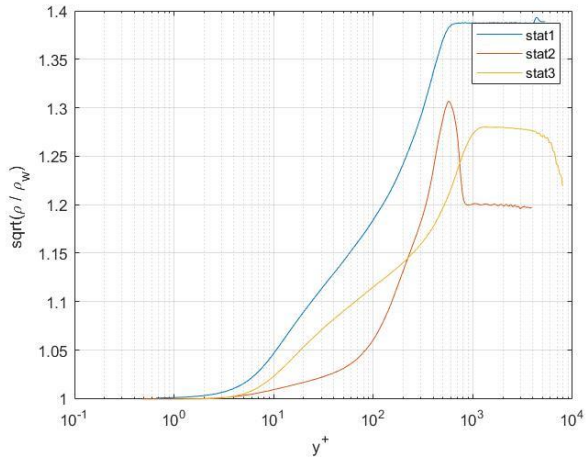



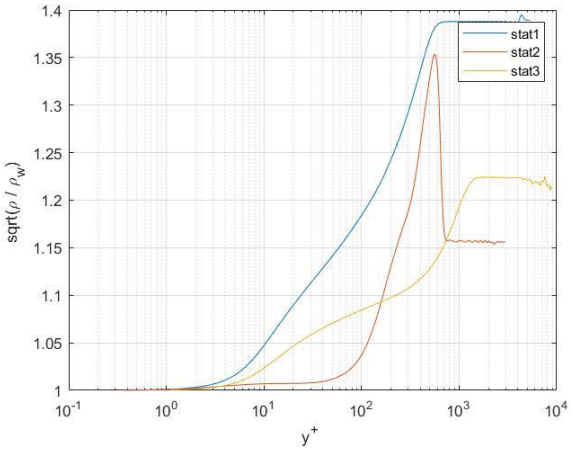
4) Density scaled streamwise , wall-normal and spanwise velocity rms , in viscous units . $uv^+$ is Reynolds turbulent shear stress in viscous units. Statistics are collected at normalized streamwise coordinate $x = 60$ , located in the region after the reflected shock wave. Simulation is carried out for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s]. The fluctuation level decreases as the turbulent boundary layer approaches again an equilibrium state after the interaction region . Grid size is : nx = 2048 ; ny = 192 ; nz = 72



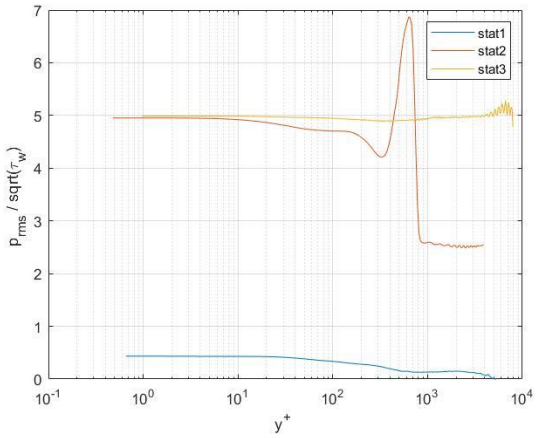



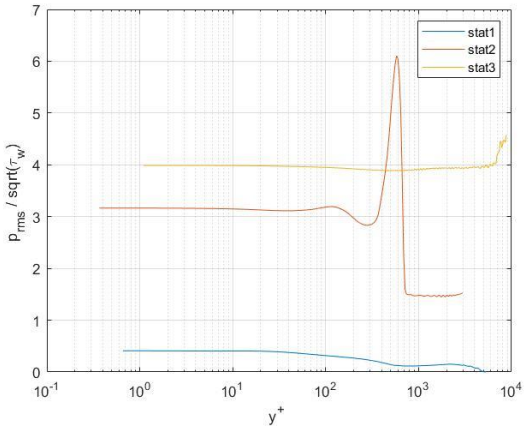

5) Square root of the mean density , normalized with the wall density. Grid size is : nx = 2048 ; ny = 192 ; nz = 72 and simulation is carried out for 100000 and 500000 iterations, corresponding to a physical time of 41 [s] and 205 [s]. Statistics are collected at the same streamwise coordinates of the previous cases.



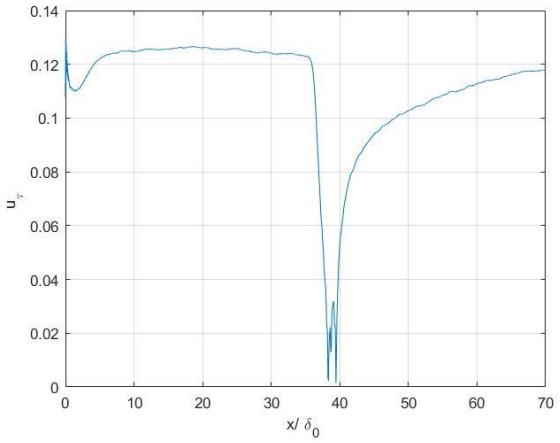



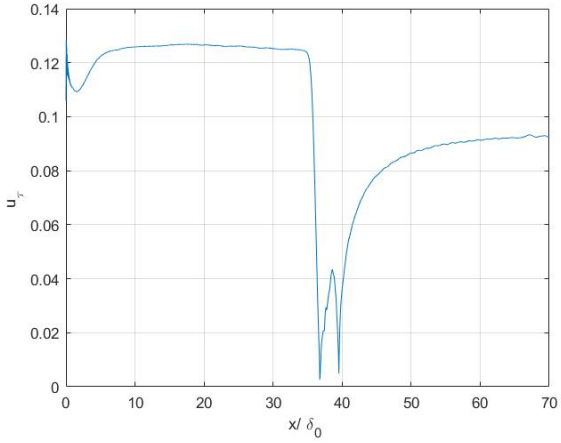

6) Pressure rms , normalized with the square root of the wall shear stress . Grid size is : nx = 2048; ny = 192 ; nz = 72 and simulation is carried out for 100000 and 500000 iterations, corresponding to a physical time of 41 [s] and 205 [s]. Statistics are collected at the same streamwise coordinates of the previous cases.



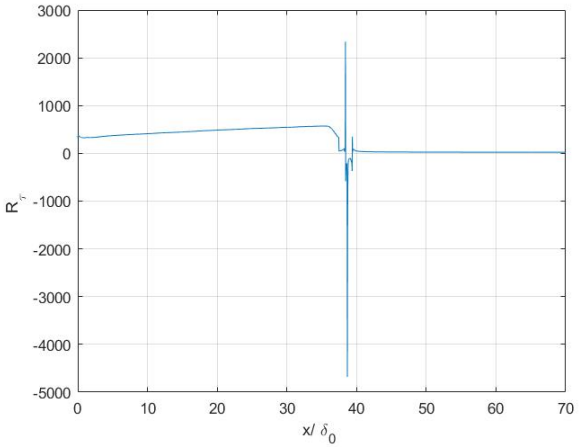



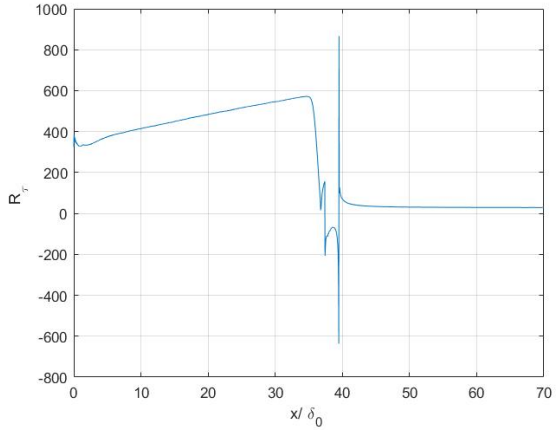

7) Friction velocity as a function of the normalized streamwise coordinate . Simulation is carried out for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s].



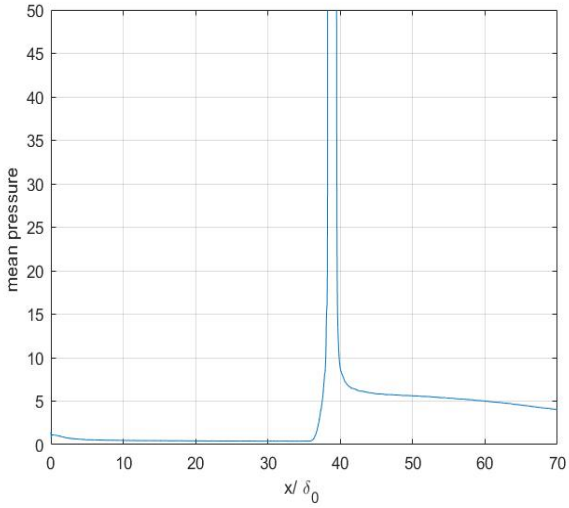



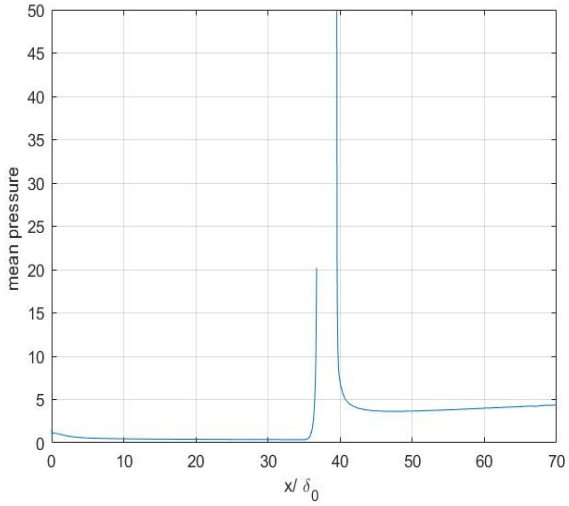

8) Friction Reynolds number as a function of the normalized streamwise coordinate . Simulation is carried out for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s].



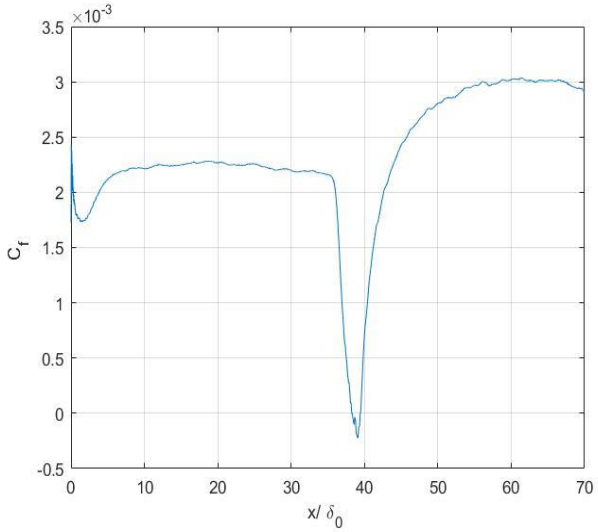



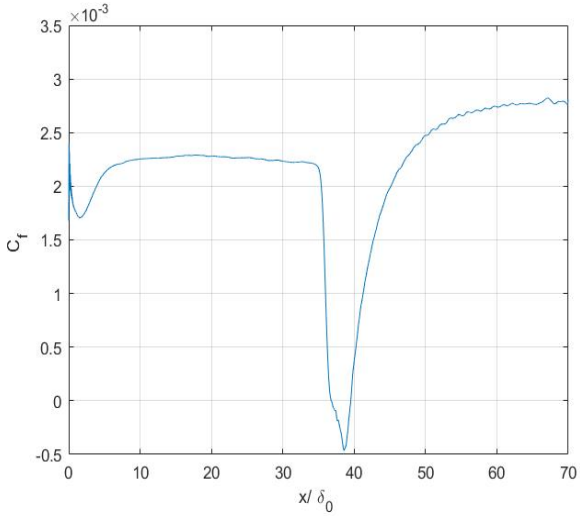
9) Mean pressure at the wall, normalized with the wall-shear stress $\tau_w$, as a function of the normalized streamwise coordinate . In the interaction region between the shock wave and the turbulent boundary layer the statistic goes to infinity and a recirculation bubble originates due to the strong adverse pressure gradient. Grid size is : nx = 2048 ; ny = 192 ; nz = 72. Simulation is carried out for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s].



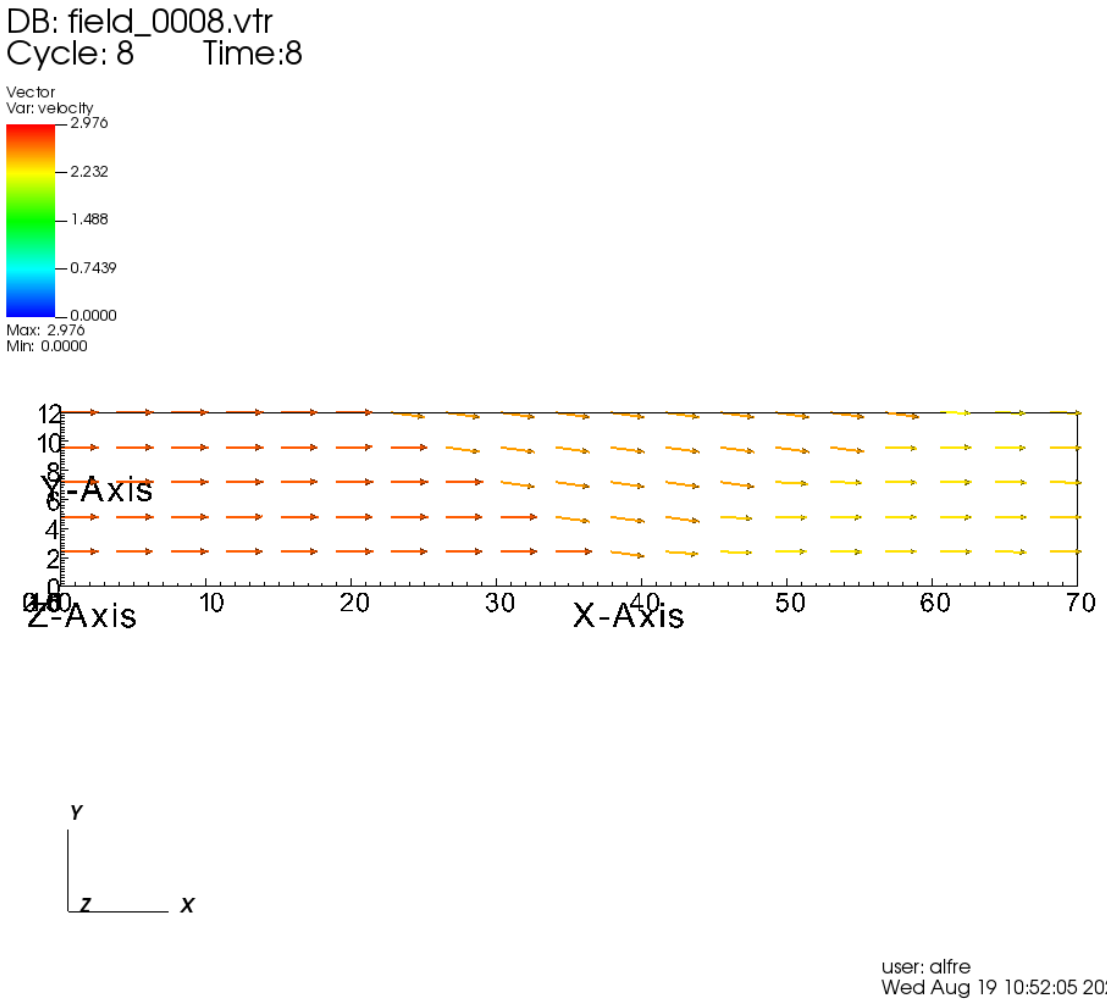




10) Skin-friction coefficient as a function of the normalized streamwise coordinate , for the same grid of the previous cases. Simulation is carried out for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s]. $C_f$ rises after the interaction region between the shock wave and the turbulent boundary layer and stays fairly constant.

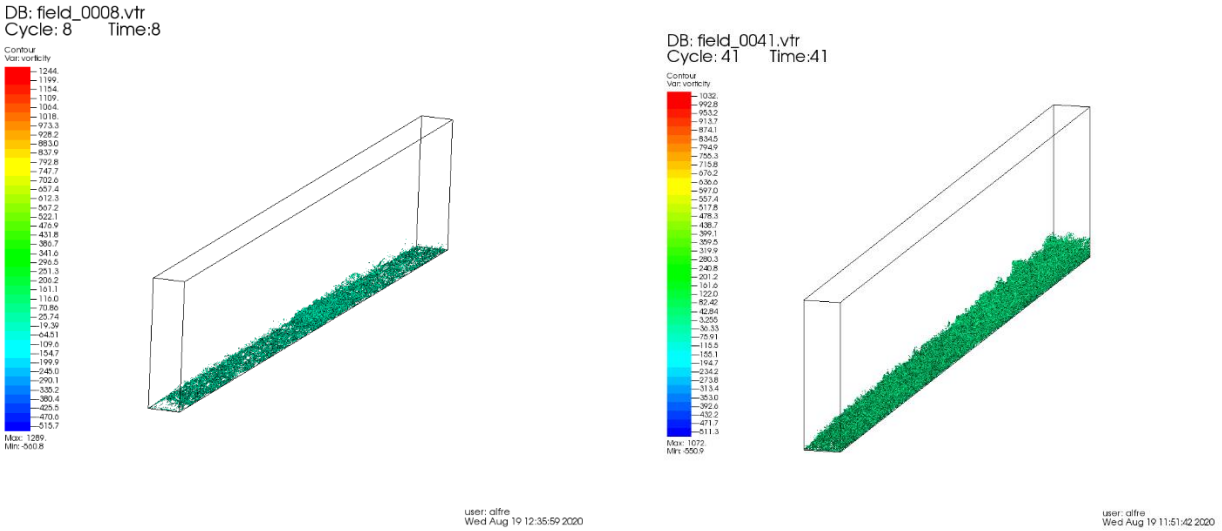


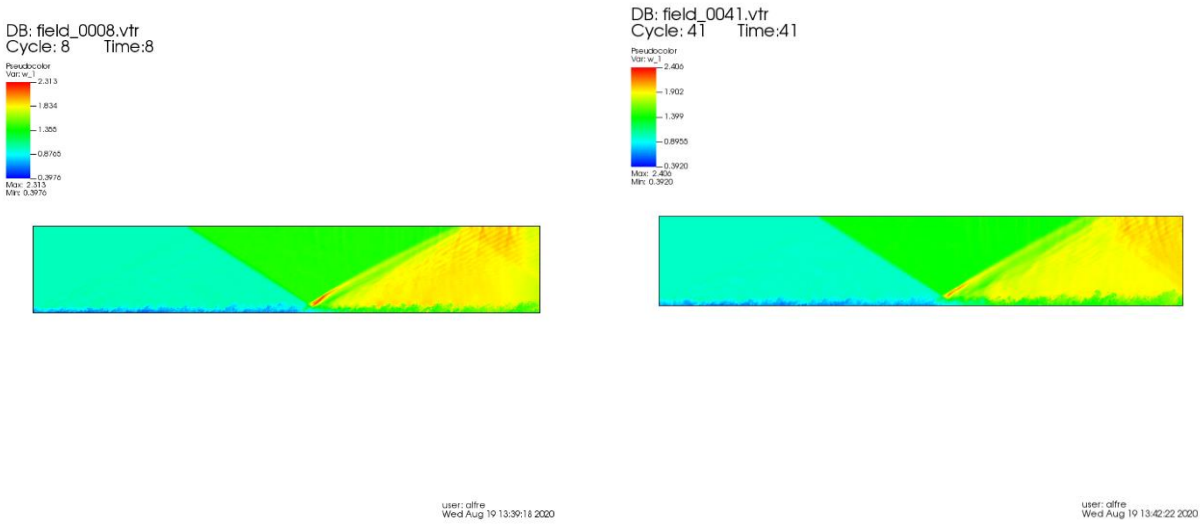
11) Visualisation of the vector velocity field. Grid size : nx = 2048 ; ny = 192 ; nz = 72 . Iterations 100000.




12) Visualization of q-criterion for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s]. Grid size : nx = 2048 ; ny = 192 ; nz = 72 .



13) Visualization of density pseudocolor for 100000 and 500000 iterations , corresponding to a physical time of 41 [s] and 205 [s]. Grid size : nx = 2048 ; ny = 192 ; nz = 72 .




## 4.2 Weak Scalability

To evaluate parallel performances of the numerical code , we should run some simulations with a lower number of iterations ( 1000 ) keeping constant the number of grid points per single GPU .

### 4.2.1 Preprocessing time at constant GPU workload

As we should not take into consideration the time necessary to initialize and preprocess the MPI environment of the computational domain using multiple GPUs , we evaluated it by running just 1 iteration.

Starting from a grid of 1024x96x72 = 7077888 total points , we scaled up the number of points with the number of GPUs . Simulations were carried out for a number of 1,2 and 4 GPUs in the same cluster node. As we note from the results, the time to preprocess data increases with the number of GPUs imposed in the input file, at constant GPU workload.

The number of points was first doubled in y cartesian direction for the second case and then doubled in x cartesian direction for the third case.

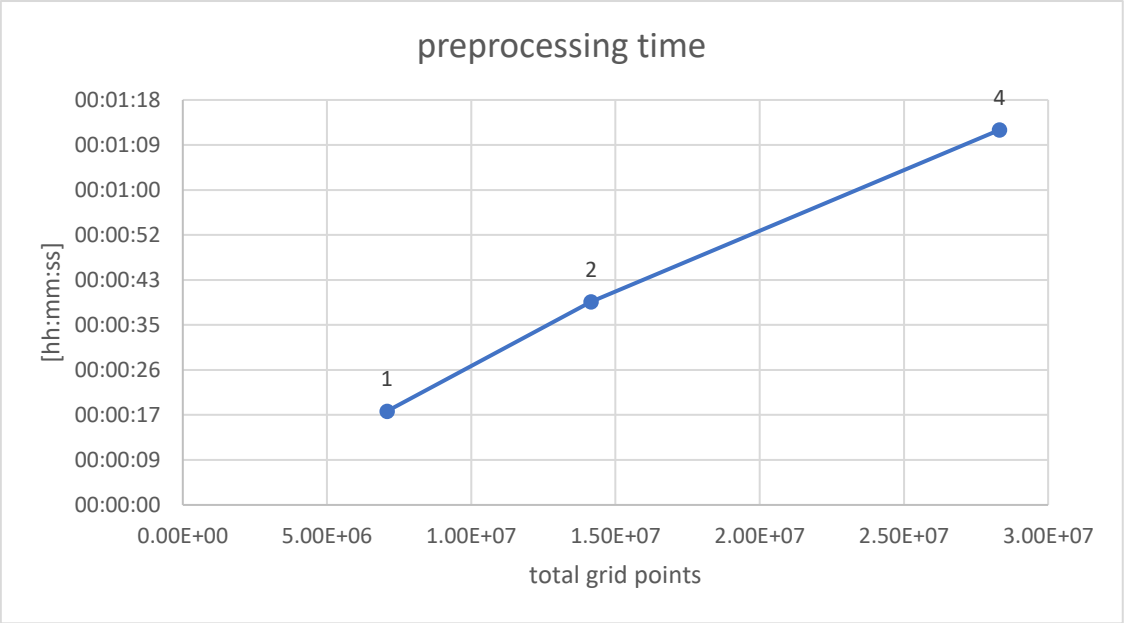



*Figure 4-9 Preprocessing time at constant GPU workload .*

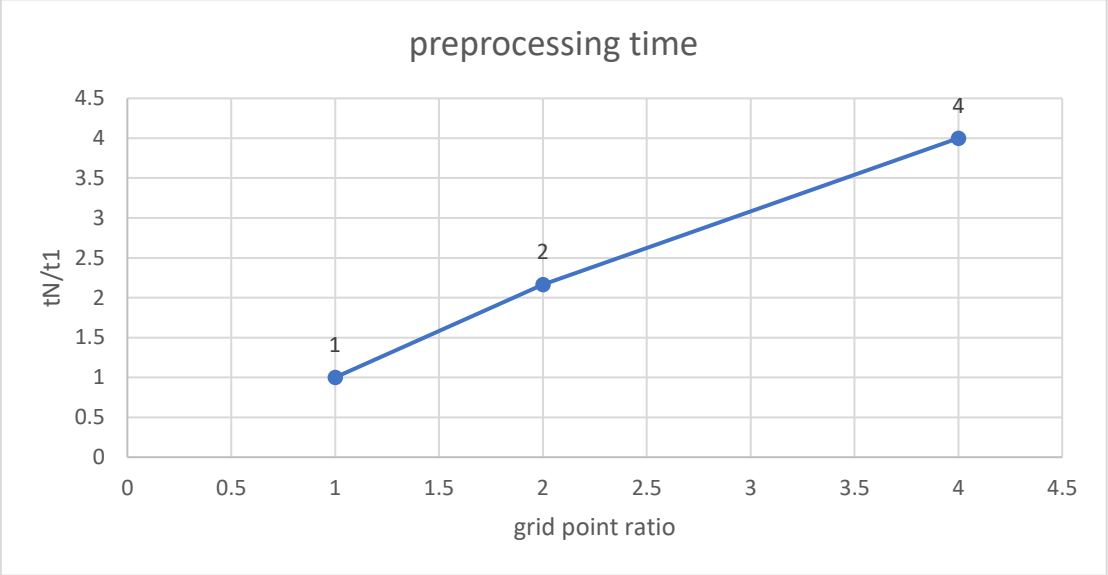



*Figure 4-10 Preprocessing time ratio with reference case at constant GPU workload*

### 4.2.2 Integration time at constant GPU workload

Starting from a grid of 1024x96x72 = 7077888 total points , we scaled up the number of points with the number of GPUs and evaluated the integration time as the Job Wall-clock time minus the preprocessing time. Simulations were carried out for a number of 1,2 and 4 GPUs in the same cluster node. Despite the more communication time needed to exchange boundary values among multiple MPI processes , the integration time stays fairly constant . We carried out simulations for just a few minutes, corresponding to a low number of iterations (1000).

The number of points was first doubled in y cartesian direction for the second case and then doubled in x cartesian direction for the third case.

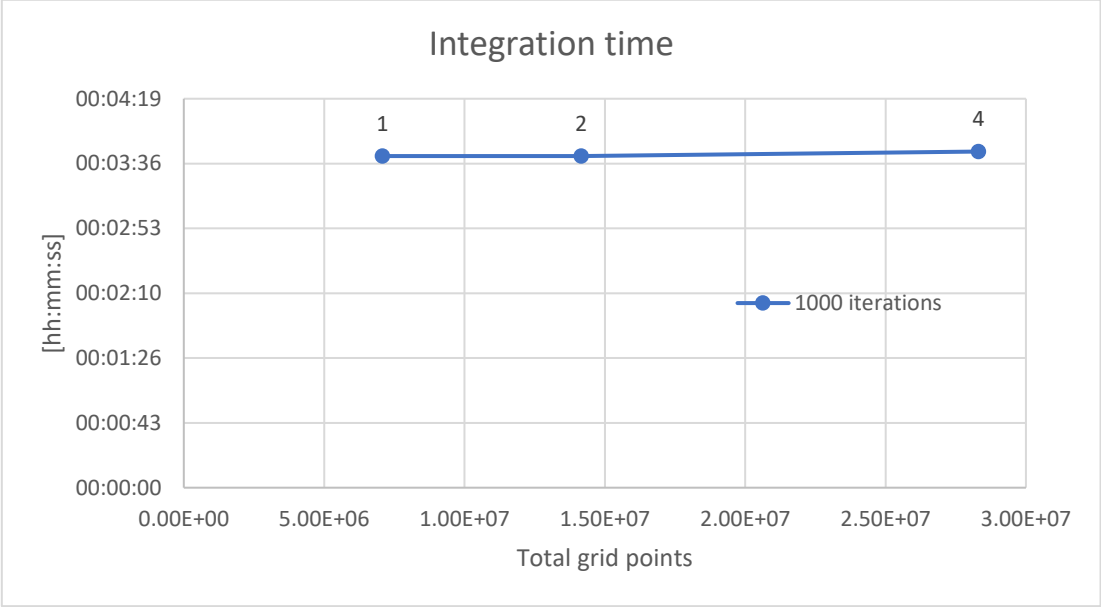



*Figure 4-11 Integration time at constant GPU workload*

### 4.2.3 Job Wall-clock time at constant GPU workload

Starting from a grid of 1024x96x72 = 7077888 total points , we scaled up the number of points with the number of GPUs and evaluated the Job Wall-clock time, which comprises both the preprocessing and integration time. Simulations were carried out for a number of 1,2 and 4 GPUs in the same cluster node. The number of points was first doubled in y cartesian direction for the second case and then doubled in x cartesian direction for the third case.

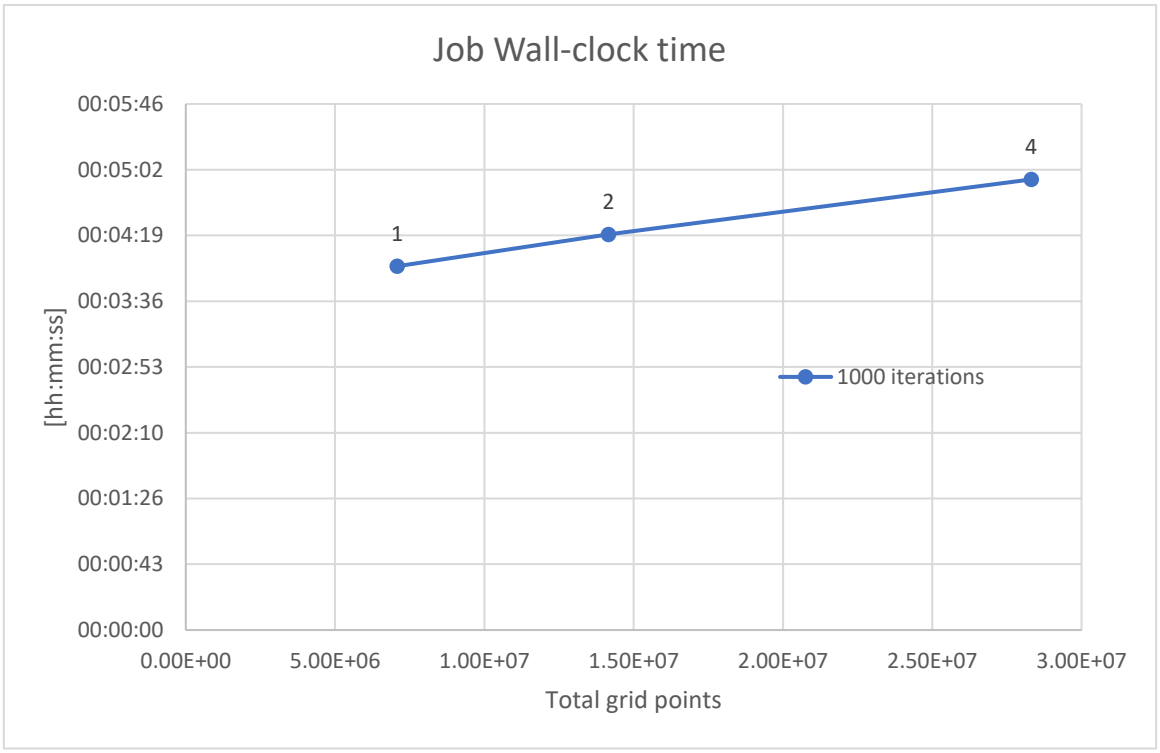



*Figure 4-12 Job Wall-clock time at constant GPU workload*

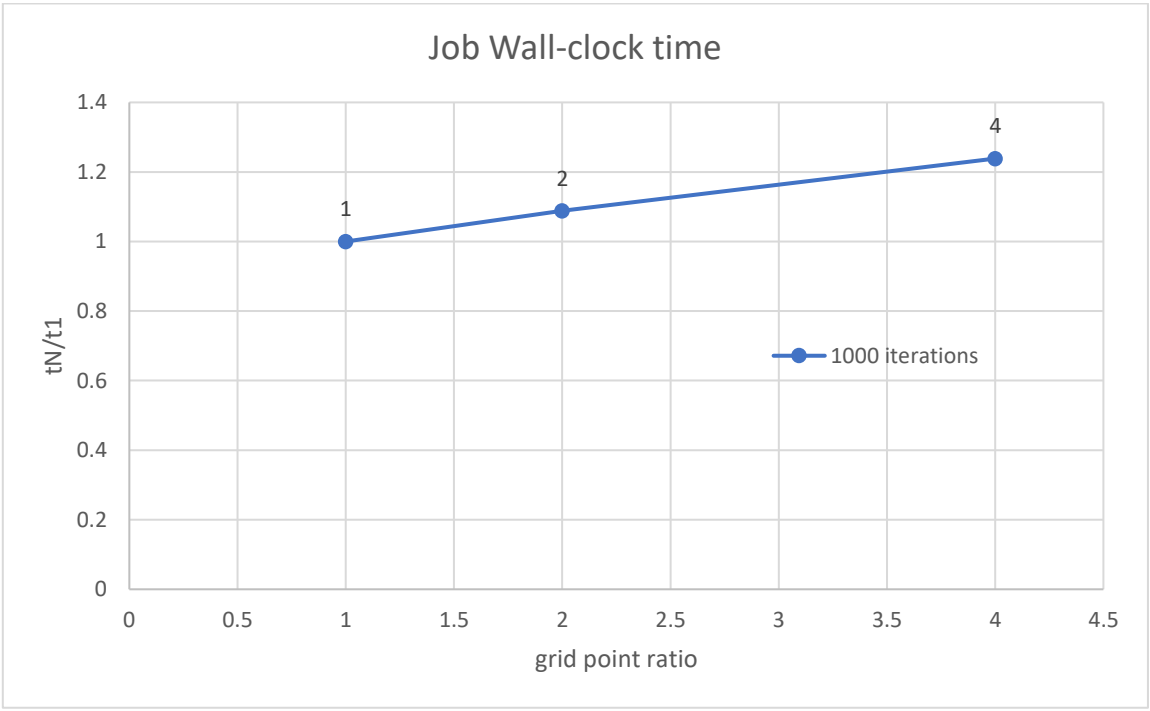



*Figure 4-13 Time ratio with reference case at constant GPU workload*

### 4.2.4 Memory usage per geometrical node at constant GPU workload

Starting from a grid of 1024x96x72 = 7077888 total points , we scaled up the number of points with the number of GPUs and evaluated the mean Bytes per geometrical node as $(Total\ Memory\ Utilized)/(Total\ Number\ of\ Points)$.

Simulations were carried out for a number of 1,2 and 4 GPUs in the same cluster node. Results show a slight increase in the memory usage for the single grid point.

The number of points was first doubled in y cartesian direction for the second case and then doubled in x cartesian direction for the third case.

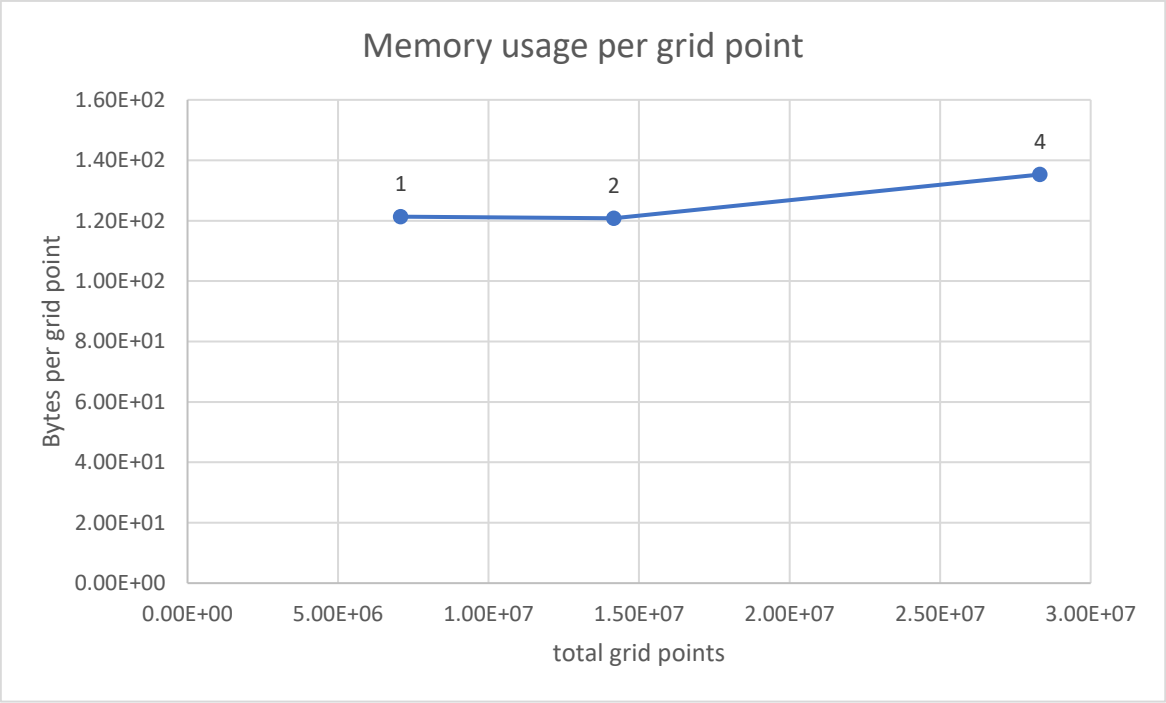



*Figure 4-14 Memory usage per grid point at constant GPU workload*

# Chapter 5

# Parametric studies for SBLI

Three simulations were carried out using four MPI processes and four GPUs in the same cluster node, on a grid of 1024x192x72 points at different friction Reynolds number : $Re_\tau = [200 \quad 500 \quad 800]$ . In this way , the integral length scales of turbulent eddies in z and y direction have been diminished across the three cases , leading to a more isotropic kind of turbulence. The computational domain had lengths rlx = 70 , rly = 12 , rlz = 3.

Statistics of the flow properties should be collected after statistical steadiness is achieved, hence a period $T_0 \approx 100 \frac{\delta_{in}}{u_\infty}$ should be sufficient to wait for the end of the initial transient [13] .

After that, samples of the flow field are periodically stored up to the final time $T_f \approx 220 \frac{\delta_{in}}{u_\infty}$ , where $\delta_{in}$ is the inflow boundary layer thickness estimated from previous simulations (as shown in the following graph , where $\delta_{99}$ has no physical meaning nearby the recirculation bubble zone close to x=40 ) while $u_\infty$ is the streamwise velocity component based on the Mach number imposed in the input file.

In our case $u_\infty = \sqrt{\gamma} * M$ with $M = 2.28$ , as Navier-Stokes equations are in dimensionless form. Shock wave angle is 8 deg and nominal impinging point is x = 40. The CFL number for numerical stability is 0.5 .

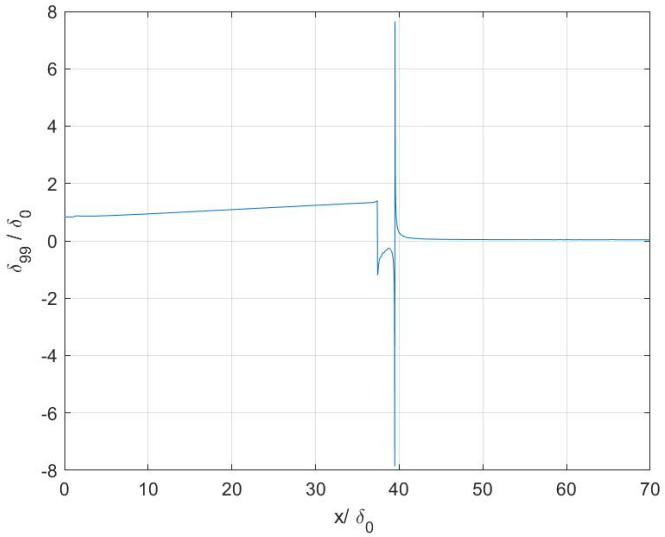



*Figure 5-1 Boundary layer thickness along streamwise coordinate.*

From a simulation on a grid of 2048x192x72 points and 500000 iterations, we estimated $\delta_{in} \approx 0.8$ ($L_y = 12$) , then we calculated the transient period $T_0 \approx 30\,s$ , corresponding

approximately to 80000 iterations . To collect figures during statistically Steady State we restarted the computation by setting the parameter *idisk = 1* in the input file . Then, time integration was carried out until $T_f \approx 65\ s$ , corresponding to at least 100 000 more iterations.

In the following graphs we point out the difference in statistics during the transitory and the statistical convergent state.

For cases with $Re_\tau = [200 \quad 500]$ , the transitory was estimated to last 80000 iterations , while for the case $Re_\tau = 800$ the transitory was longer due to the smaller time step required for the numerical stability, hence 160000 iterations were needed. In this way , all cases had a minimum physical time of the transitory $T_0 > 30\ s$.

For the first case we collected steady-state statistics every 150 iterations for a total of 100000 iterations, for the second case we collected statistics every 250 iterations for a total of 100000 iterations and for the third case we collected statistics every 250 iterations for a total of 200000 iterations.

## 5.1 Difference in statistics during transitory and steady state

The statistics reported are:

1. Mean Streamwise Turbulent Velocity normalized with the friction velocity, in stations $x = [20 \quad 40 \quad 60]$ ;
2. Compressible Friction Coefficient ;
3. Reynolds stress tensor for x = 20;
4. Reynolds stress tensor for x = 40;
5. Reynolds stress tensor for x = 60;
6. Mean pressure normalized with wall shear stress;
7. Pressure root mean square normalized with square root of wall shear stress;
8. Friction Reynolds number;
9. Compressible Reynolds number;
10. Friction velocity.

*Table 6 Integration time table*

| $Re_\tau$ | $T_0$ [s] | $T_f$ [s] | N° Iterations Transitory | N° Iterations statistically Steady State | istat |
|---|---|---|---|---|---|
| 200 | 71 | 158 | 80000 | 180000 | 150 |
| 500 | 32 | 72 | 80000 | 180000 | 250 |
| 800 | 41 | 93 | 160000 | 360000 | 250 |

To post-process data and visualize the flow field we used the software Visit to read the **vtk** files, which are printed out by STREAmS every physical time step $\Delta t_{save}$ specified in the input file.

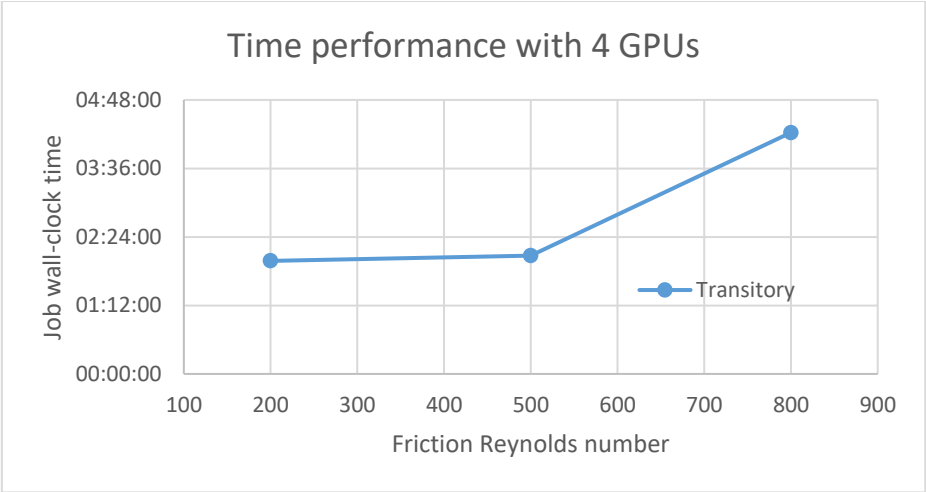
It is necessary to express velocity components (U, V, W) as scalar mesh variables and as a function of conservative quantities of the vector **w .** Then we can define the velocity vector field and the q-criterion using built-in functions in Visit . In particular the syntax :

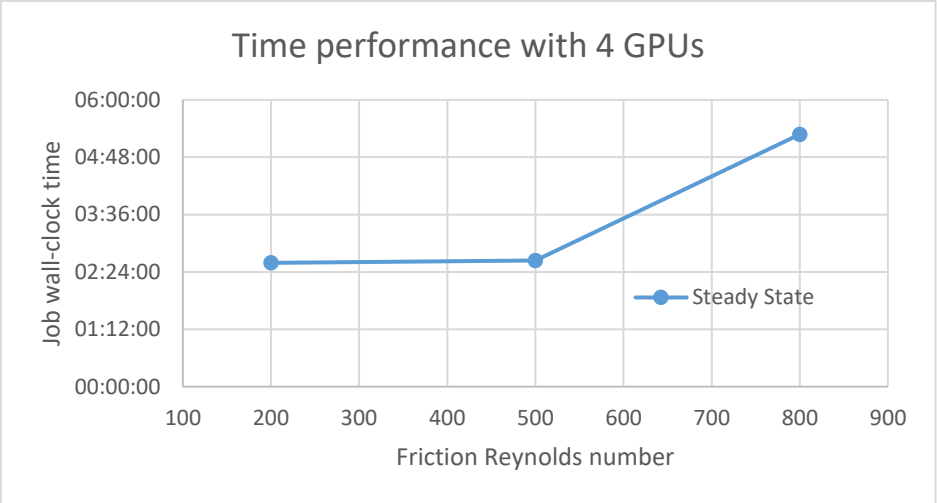
*q_criterion(<gradient(velocity[0])>, <gradient(velocity[1])>, <gradient(velocity[2])>)*

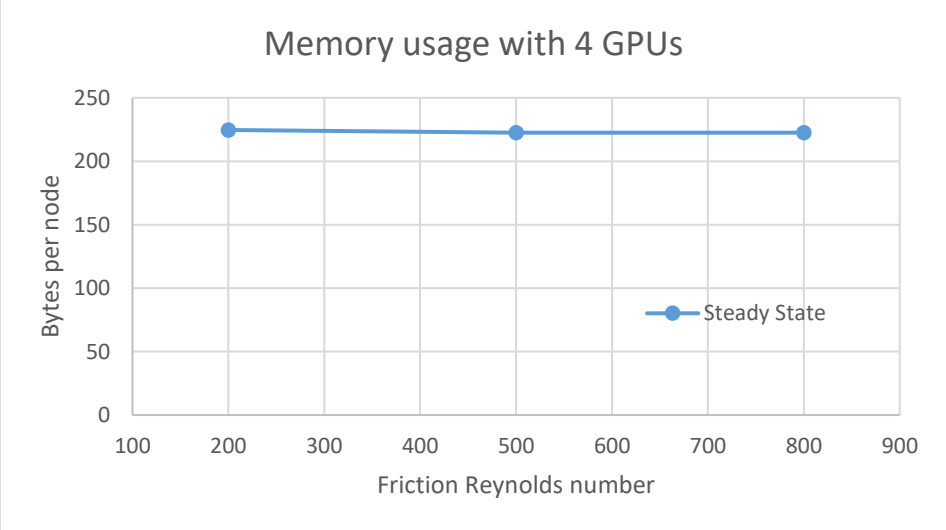
generates the Q-criterion value developed by Hunt et. al.. It is based on the observation that, in regions where the Q-criterion is greater than zero, rotation exceeds strain and, in conjunction with a pressure min, indicates the presence of a vortex. The three arguments to the function are gradient vectors of the x-, y-, and z-velocity. The gradient function can be used to create the gradient vectors. [14]

From the results we notice that the compressible skin friction coefficient diminishes at higher $Re_\tau$, both during transitory and steady state , while the fluctuation level rises especially in the interaction region located close to x = 40 , where the Reynolds shear stress component assumes deeply negative values for $Re_\tau \geq 500$ . As it can be expected , the compressible Reynolds number increases according to $Re_\tau$ , but it can be computed by the solver only until the nominal impinging point of the incoming shock . Even the mean pressure statistic is defined until the recirculation bubble region , afterwards it assumes higher values due to the adverse pressure gradients caused by the impinging and reflected shock waves , which are visible from the Mach contour of the flow field with proper Mach levels set in Visit .

Time performance and memory usage per grid node are reported for the three cases , for the last one we need more iterations to reach the physical final time Tf , as the computed time step for CFL stability is lower :



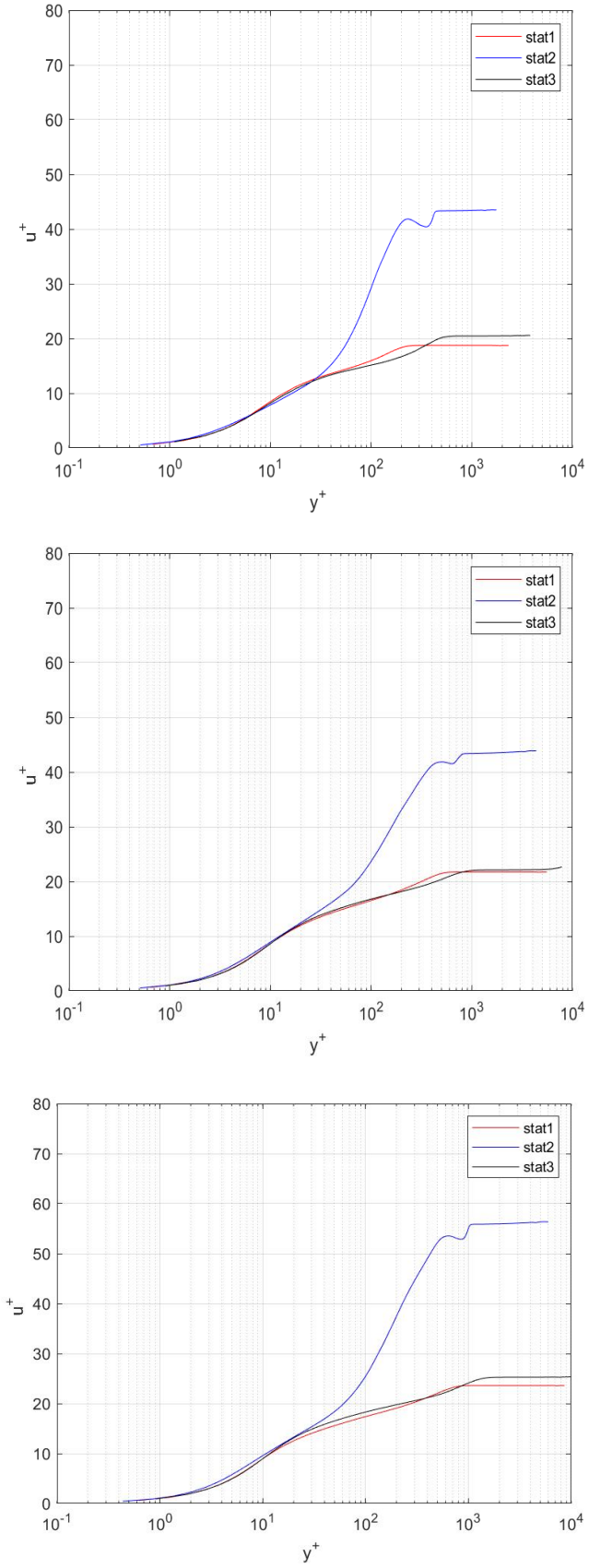



*Figure 5-2 Mean Velocity in streamwise stations* $x = [20 \quad 40 \quad 60]$ *during transitory* $T_0$ , $Re_\tau = [200 \quad 500 \quad 800]$ .

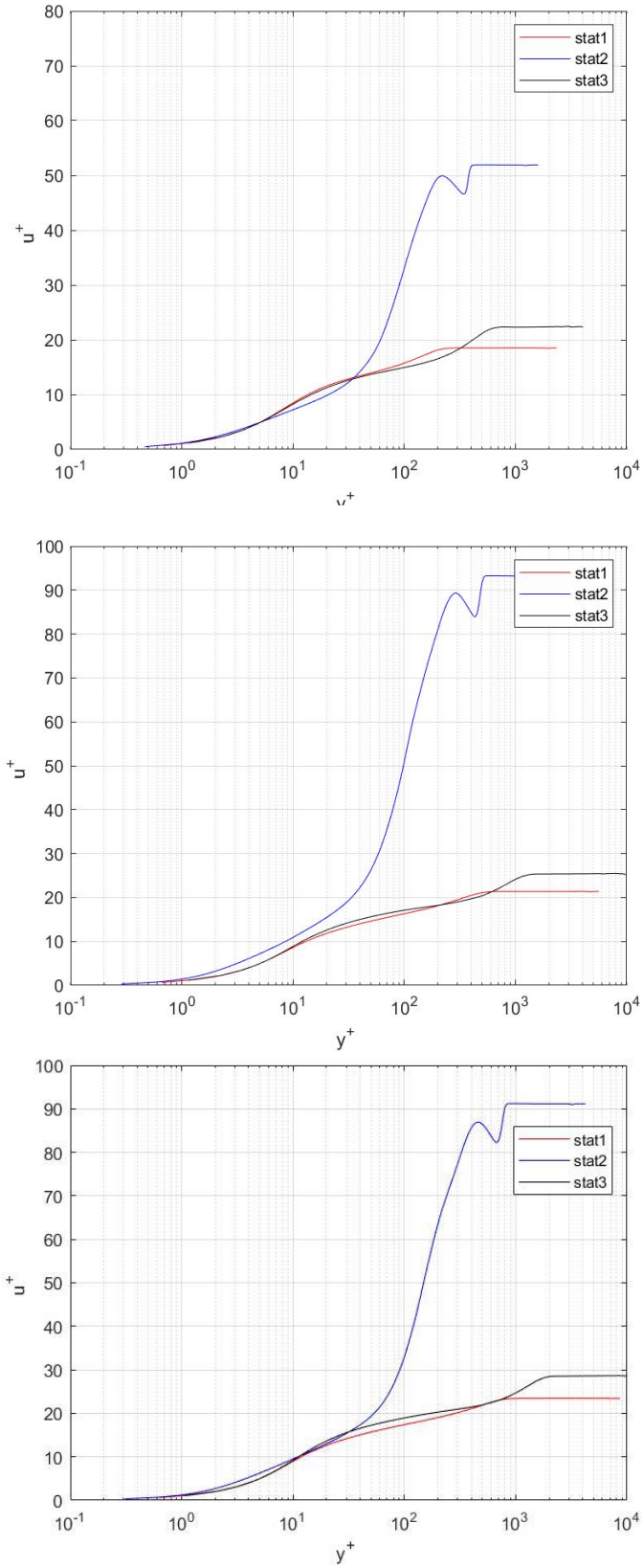
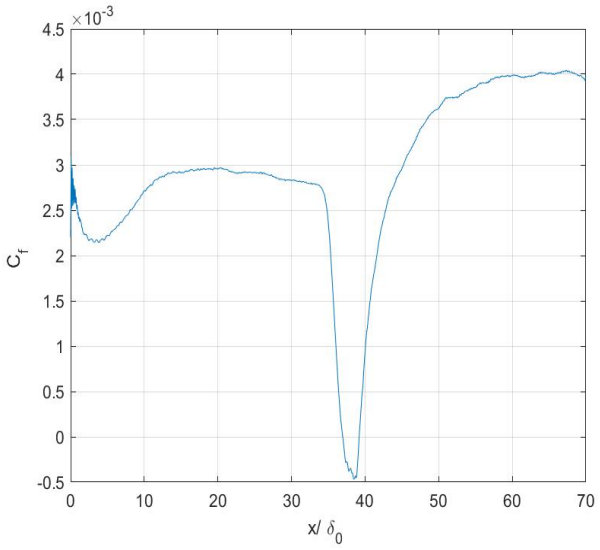
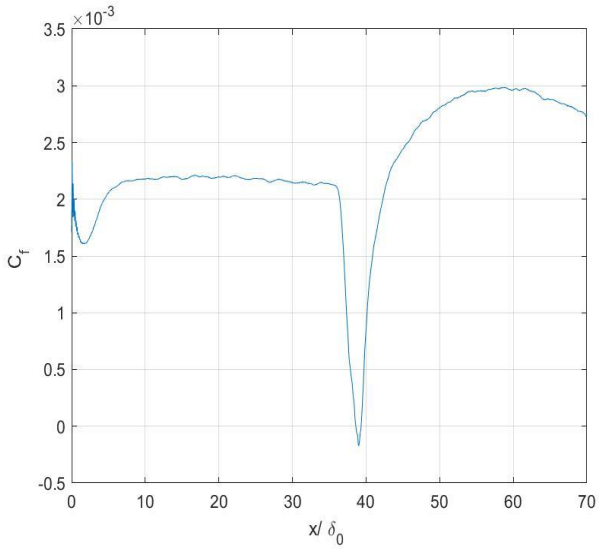



*Figure 5-3 Mean Velocity in streamwise stations* $x = [20 \quad 40 \quad 60]$ *during statistically Steady State* $T_f$. $Re_\tau = [200 \quad 500 \quad 800]$



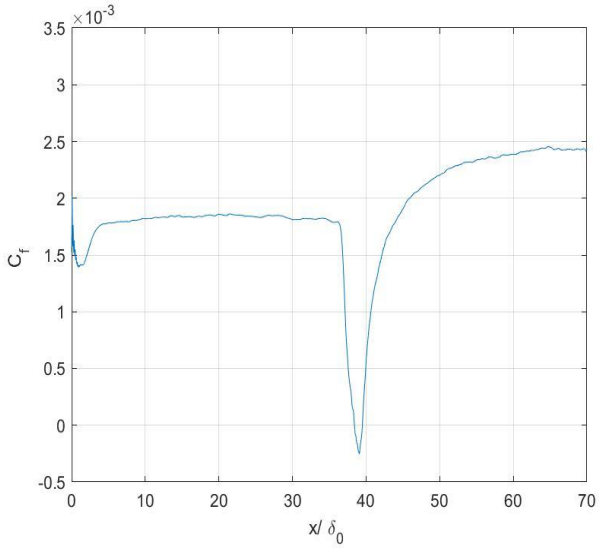




*Figure 5-4 Compressible Friction Coefficient during transitory* $Re_\tau = [200 \quad 500 \quad 800]$

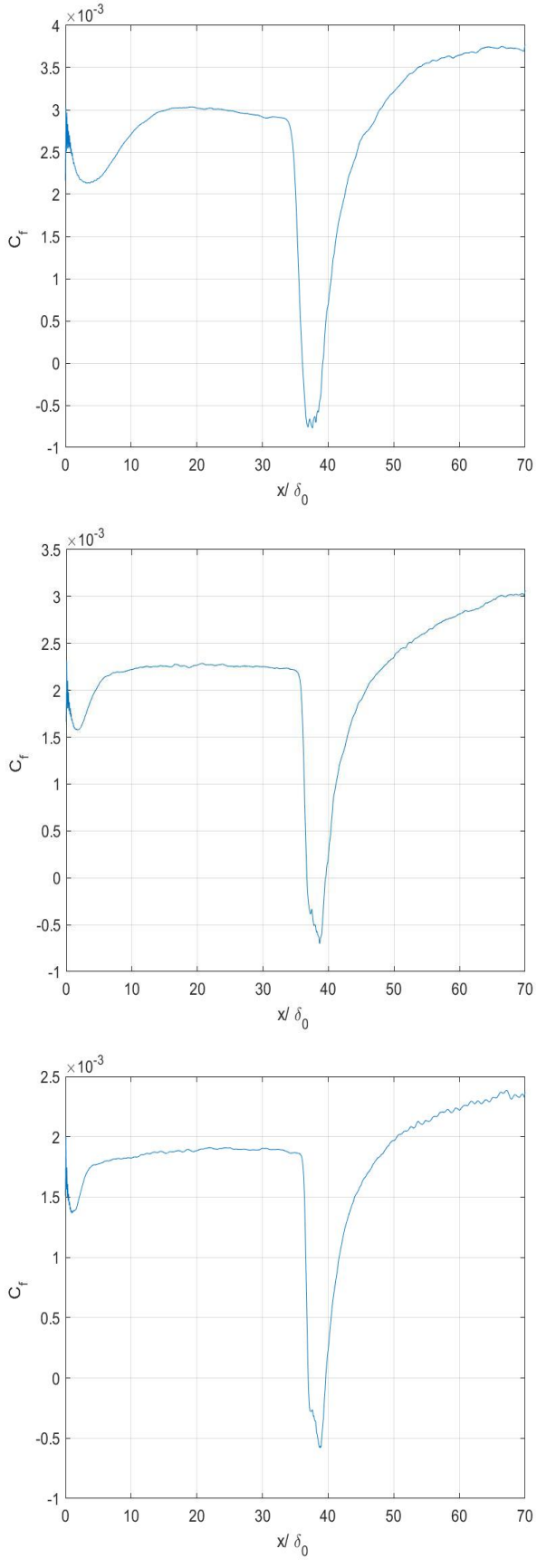



*Figure 5-5 . Compressible Friction Coefficient during statistically Steady State ;* $Re_\tau = [200 \quad 500 \quad 800]$

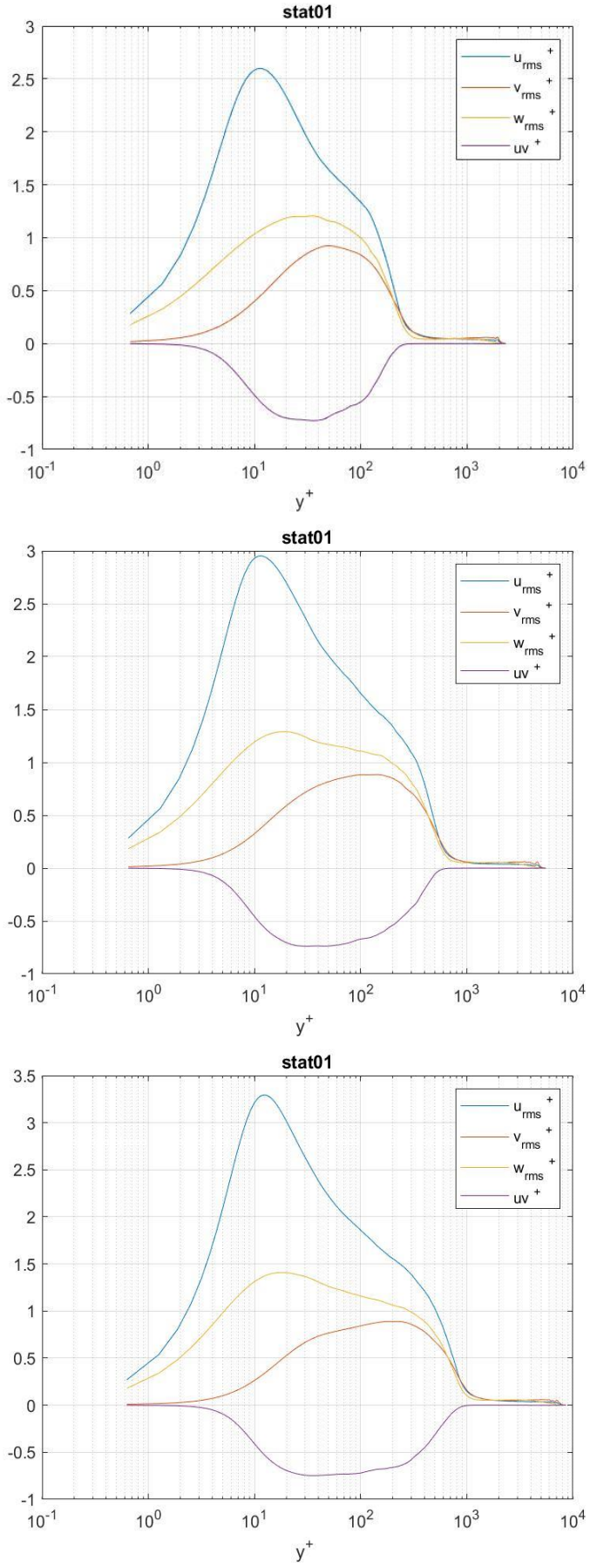



*Figure 5-6 Reynolds stress tensor for x = 20 during transitory* $Re_\tau = [200 \quad 500 \quad 800]$

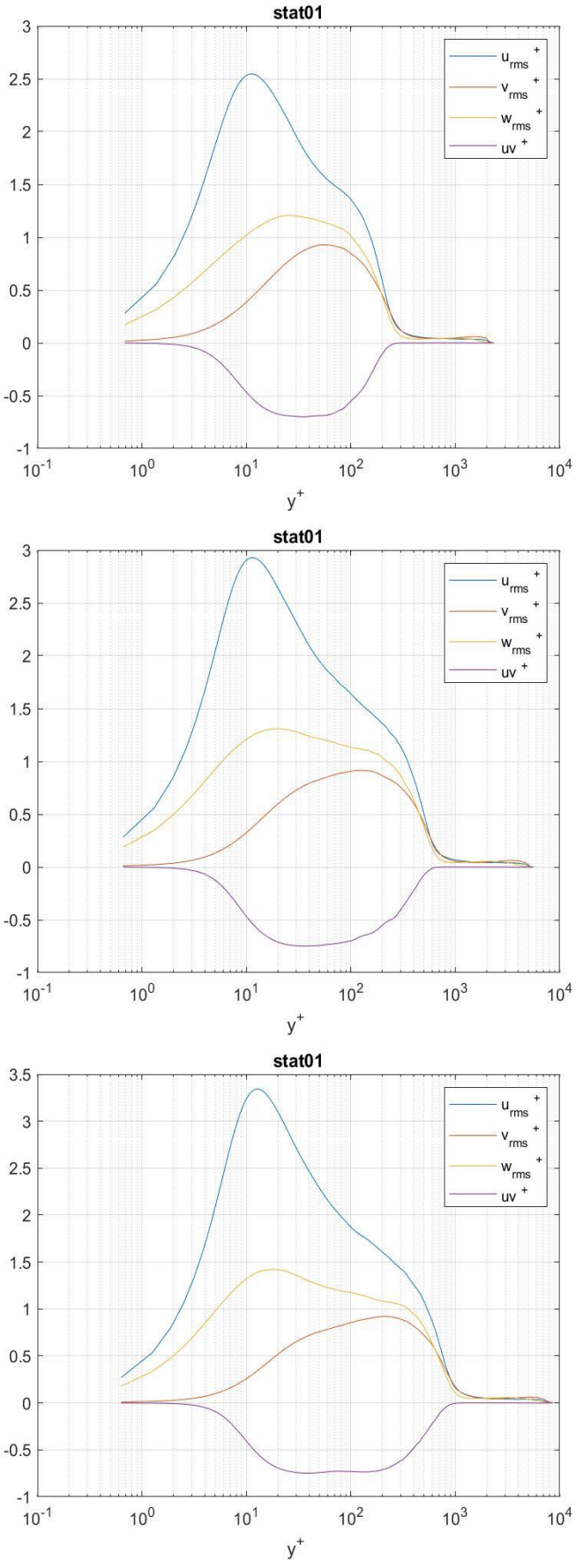



*Figure 5-7 Reynolds stress tensor for x = 20 during statistically Steady State* $Re_\tau = [200 \quad 500 \quad 800]$

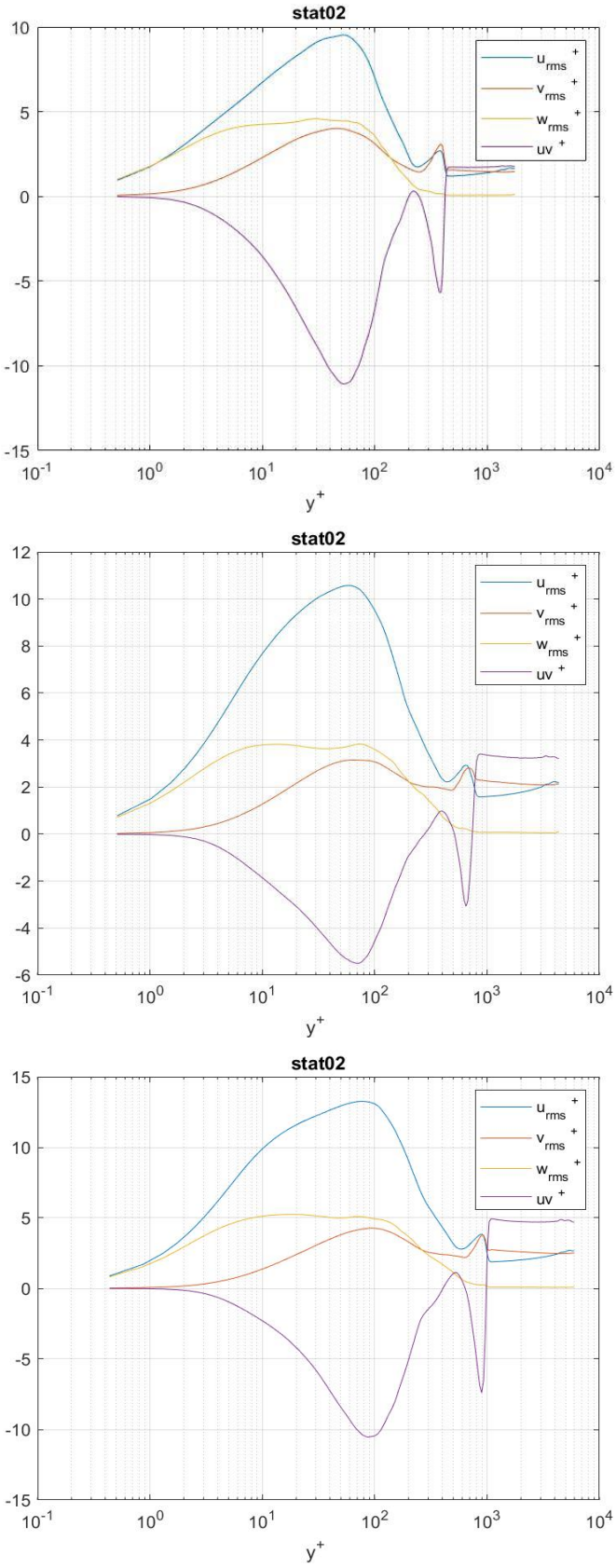



*Figure 5-8 Reynolds stress tensor for x = 40 during transitory;* $Re_\tau = [200 \quad 500 \quad 800]$

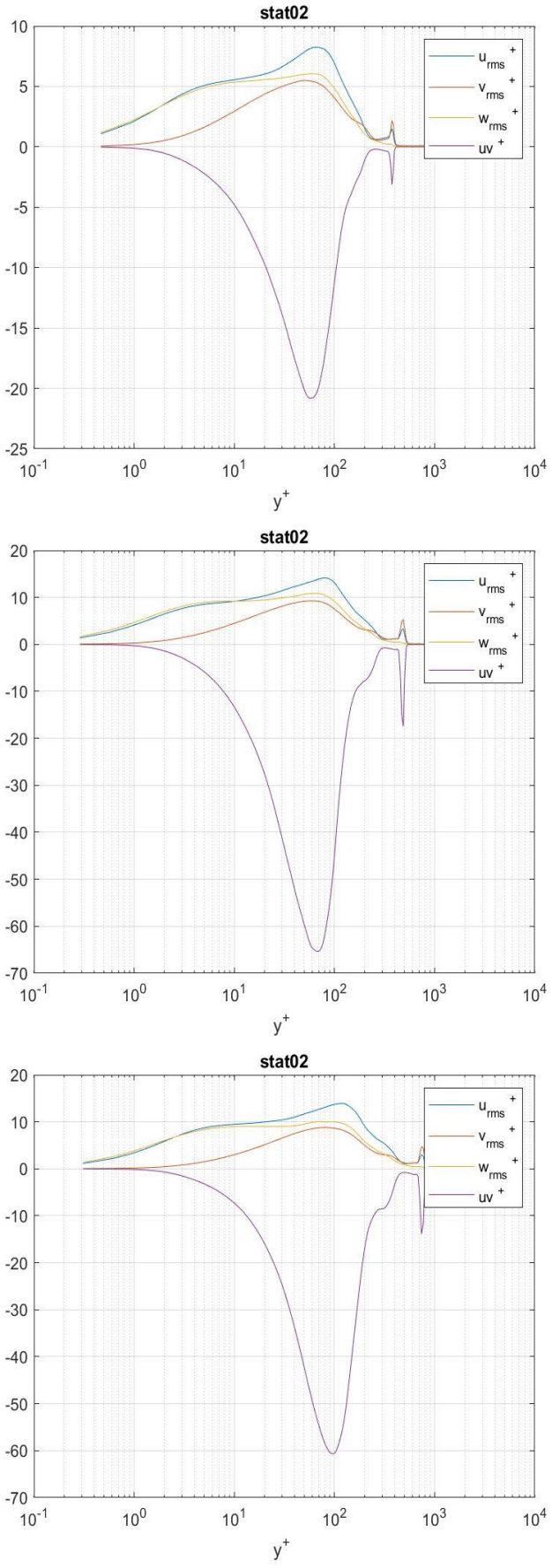



*Figure 5-9 Reynolds stress tensor for x = 40 during statistically Steady State* $Re_\tau = [200 \quad 500 \quad 800]$*;*

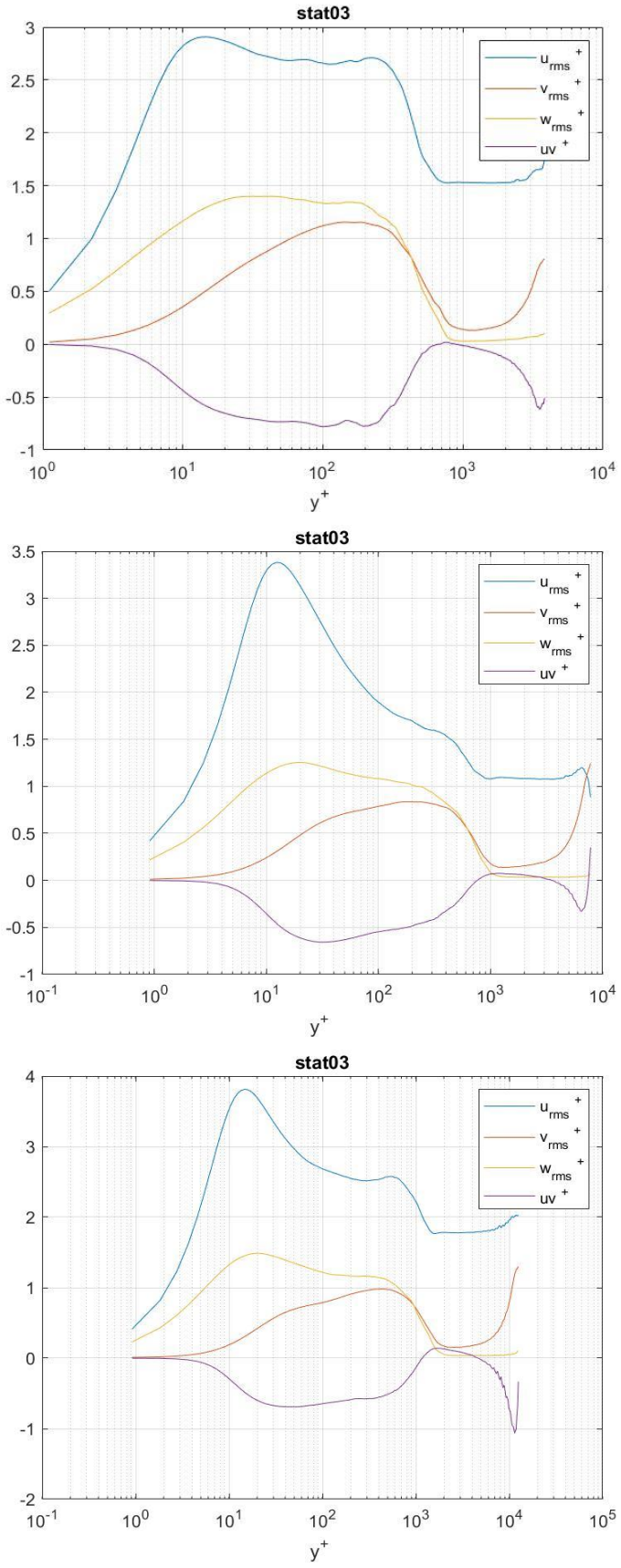



*Figure 5-10 Reynolds stress tensor for x = 60 during transitory* $Re_\tau = [200 \quad 500 \quad 800]$

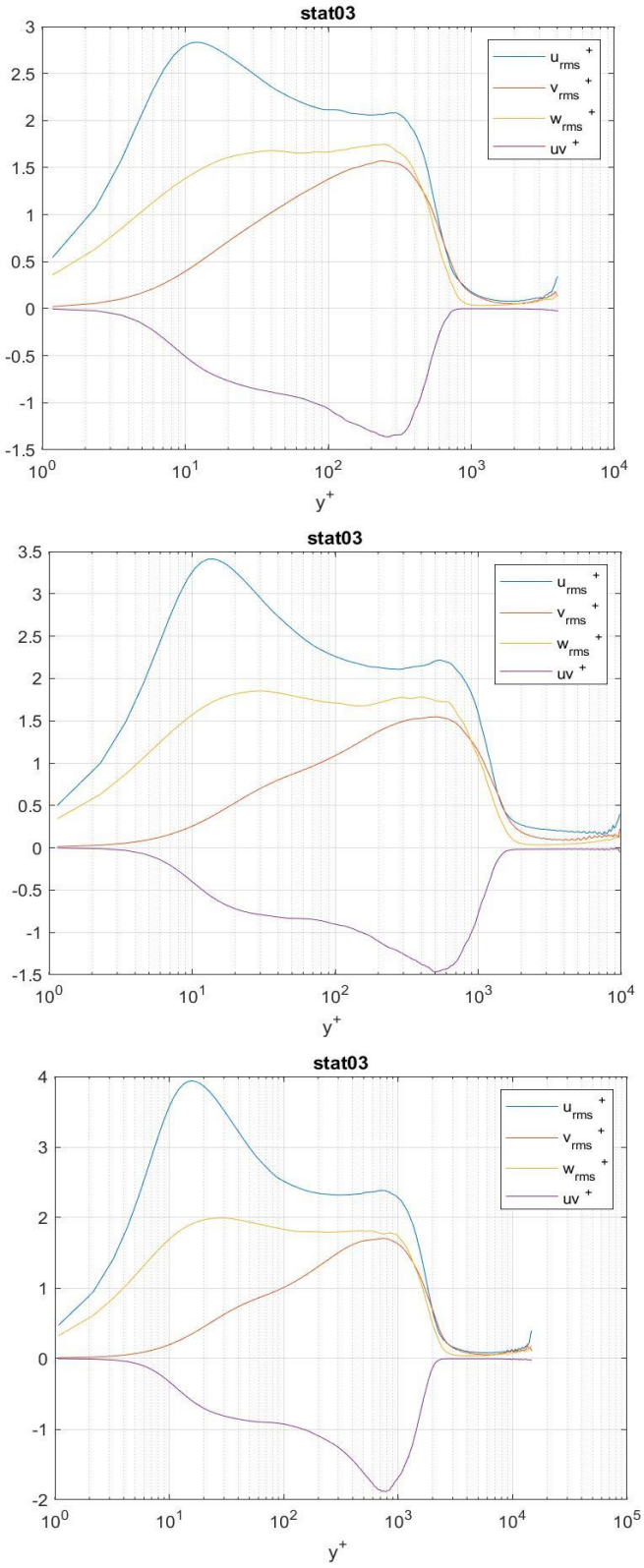



*Figure 5-11 Reynolds stress tensor for x = 60 during statistically Steady State* $Re_\tau = [200 \quad 500 \quad 800]$

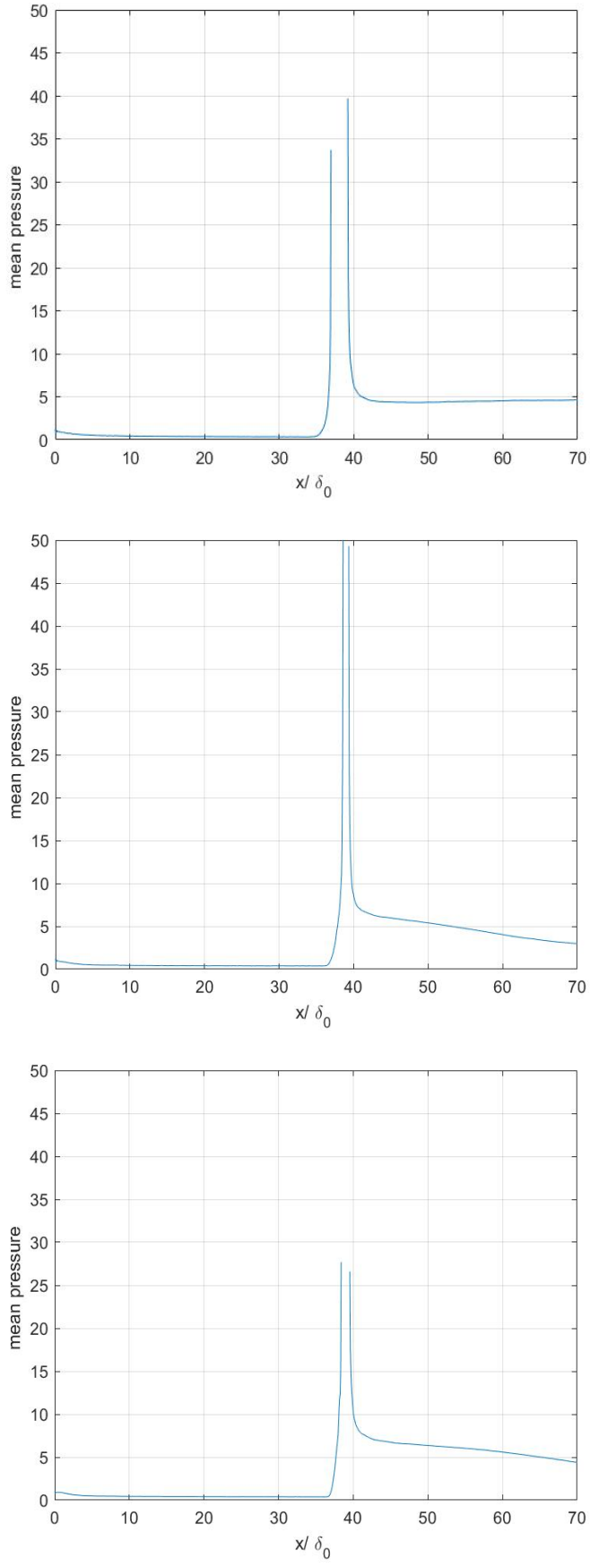



*Figure 5-12 . Mean pressure rms at the wall normalized with the wall-shear stress in streamwise direction during transitory;* $Re_\tau = [200 \quad 500 \quad 800]$

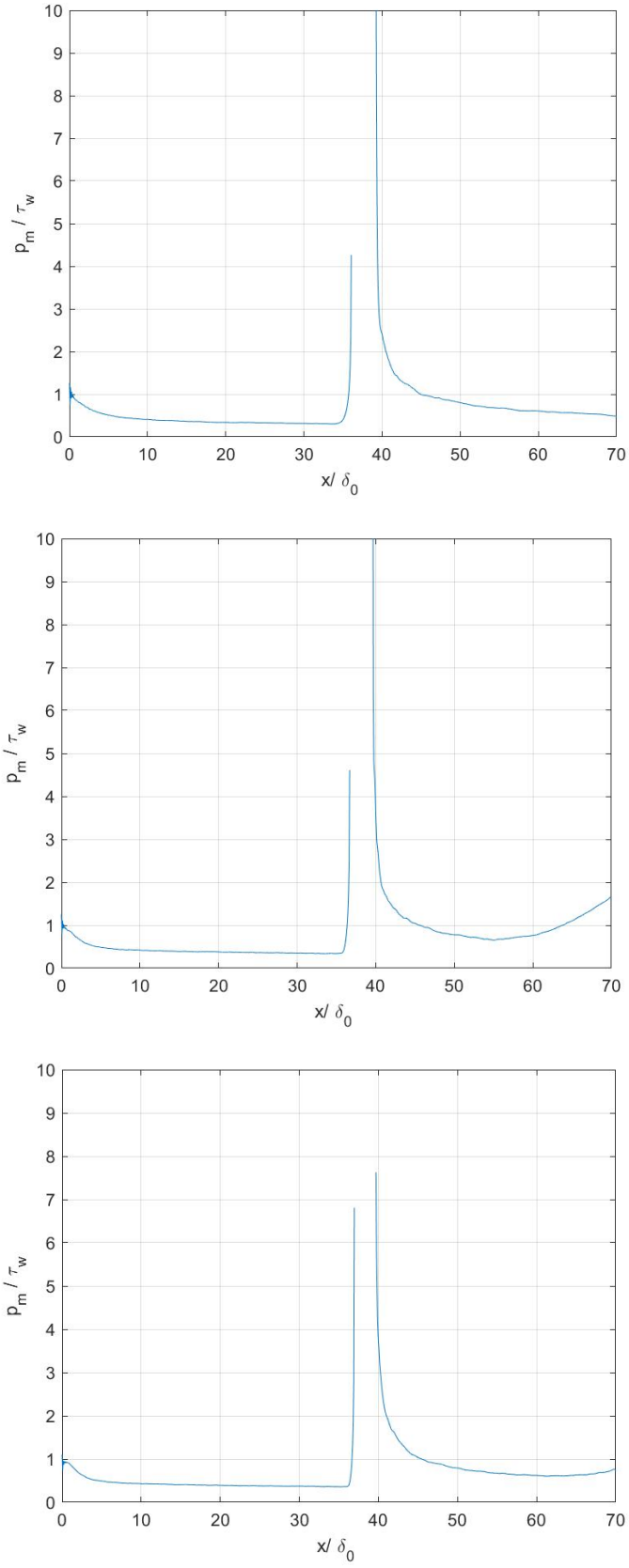



*Figure 5-13 Mean pressure rms at the wall normalized with the wall-shear stress in streamwise direction during statistically Steady State,* $Re_\tau = [200 \quad 500 \quad 800]$

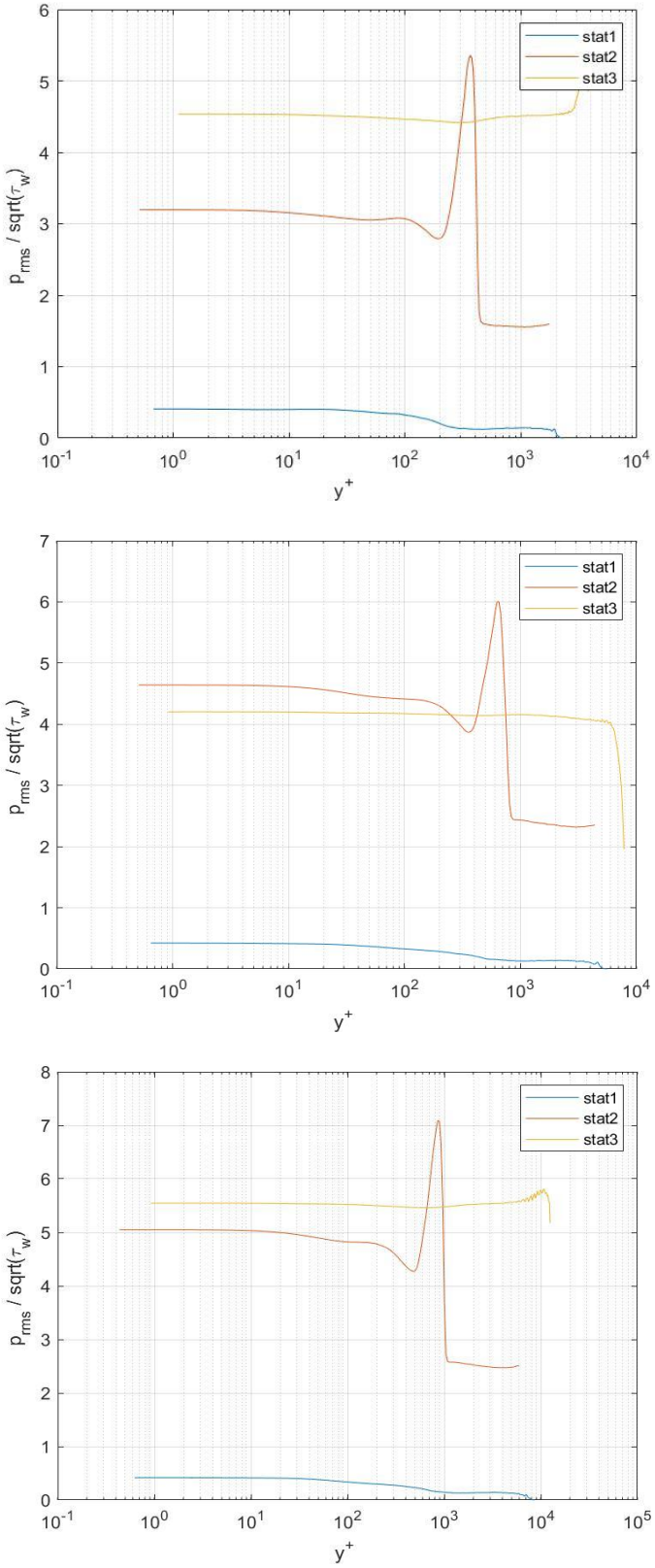



*Figure 5-14 Pressure root mean square normalized with square root of wall shear stress during transitory;* $Re_\tau = [200 \quad 500 \quad 800]$

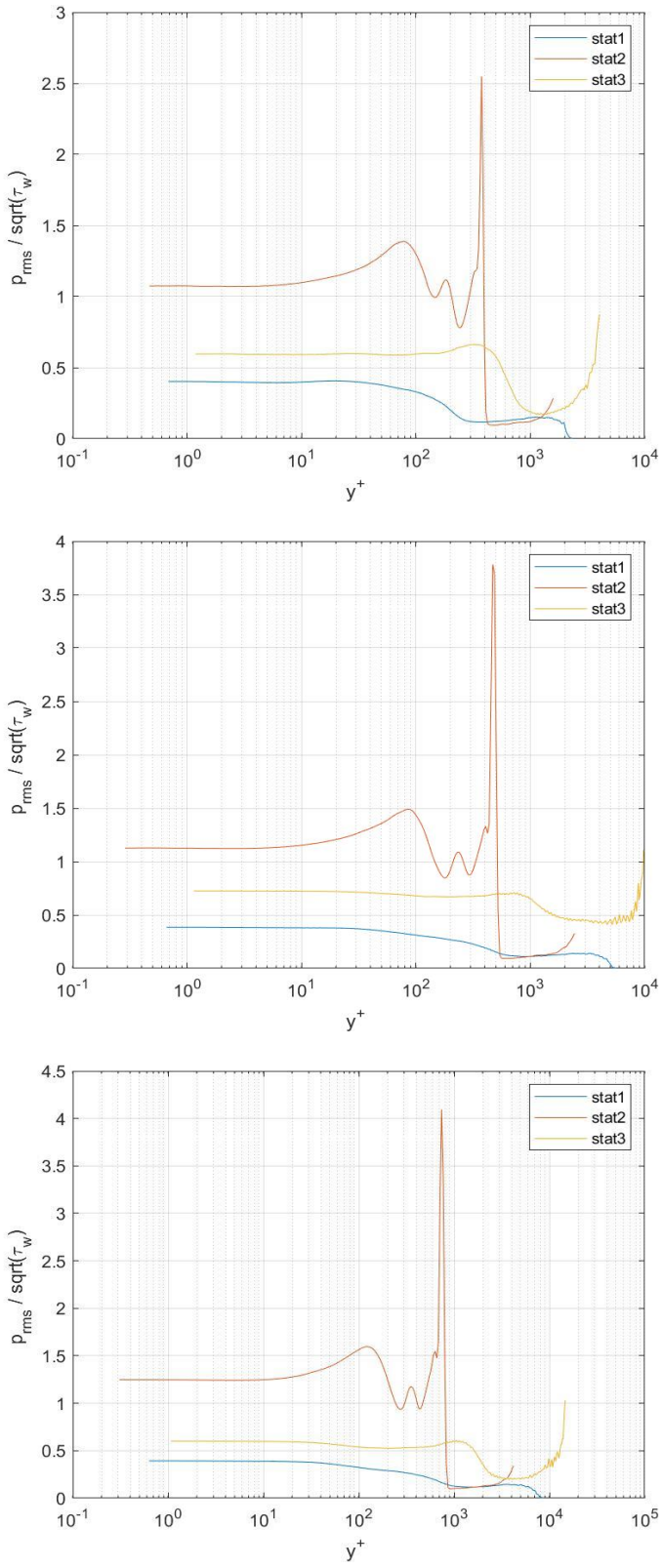



*Figure 5-15 Pressure root mean square normalized with square root of wall shear stress during statistically Steady State;* $Re_\tau = [200 \quad 500 \quad 800]$

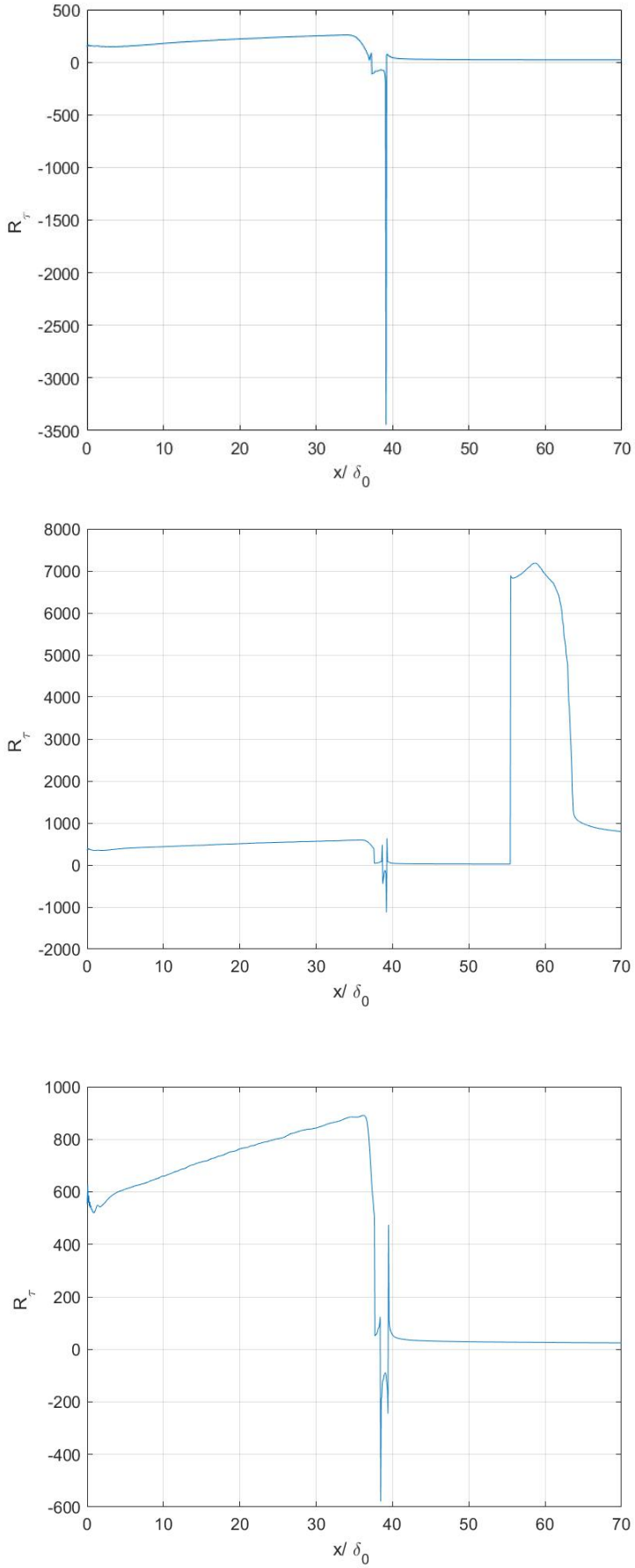



*Figure 5-16 Friction Reynolds number during transitory , $Re_\tau = [200 \quad 500 \quad 800]$*

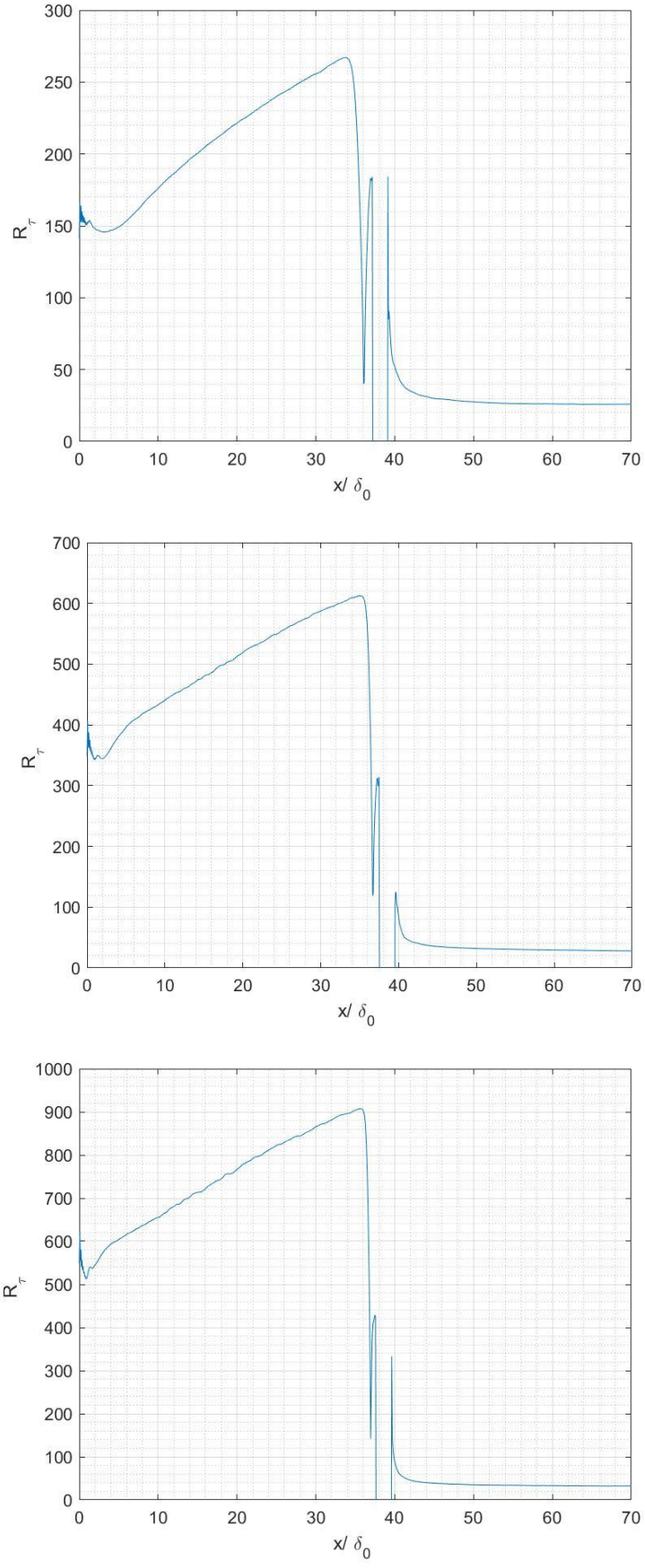



*Figure 5-17 Friction Reynolds number during statistically Steady State;* $Re_\tau = [200 \quad 500 \quad 800]$

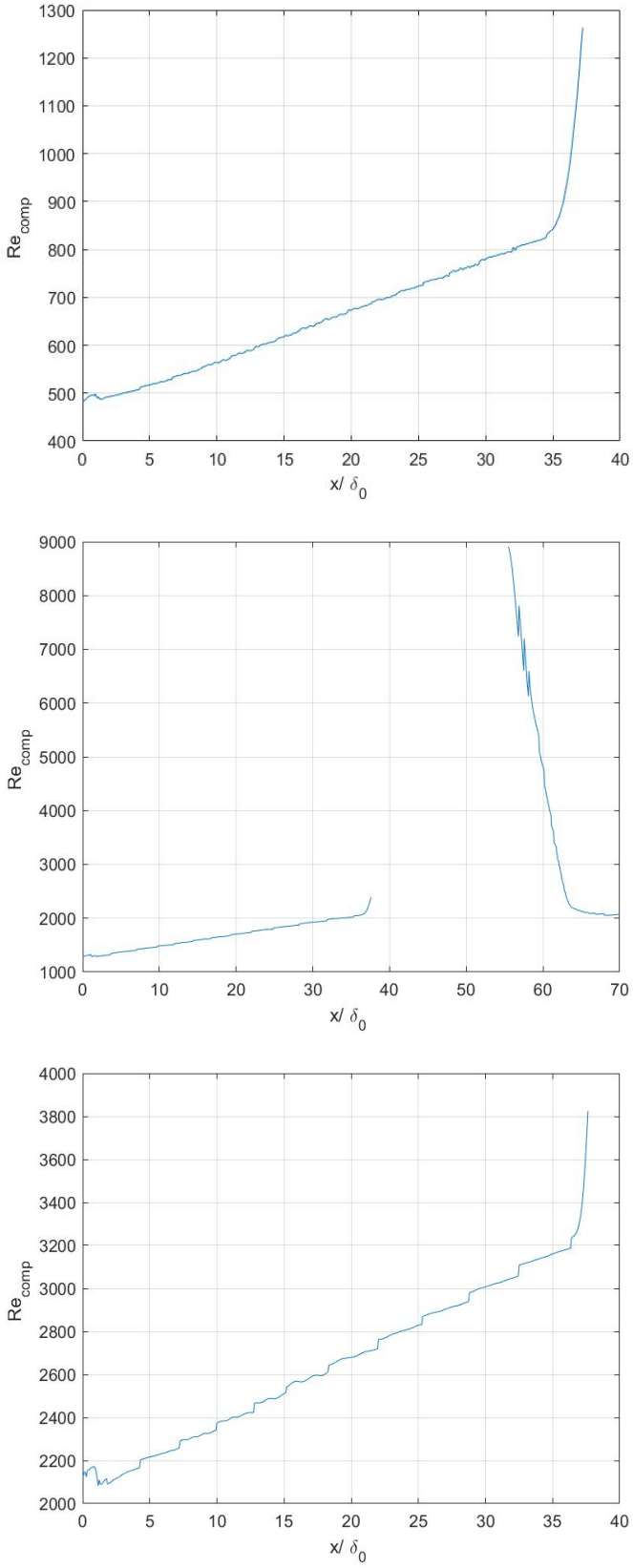



*Figure 5-18 Compressible Reynolds number during transitory , $Re_\tau = [200 \quad 500 \quad 800]$*

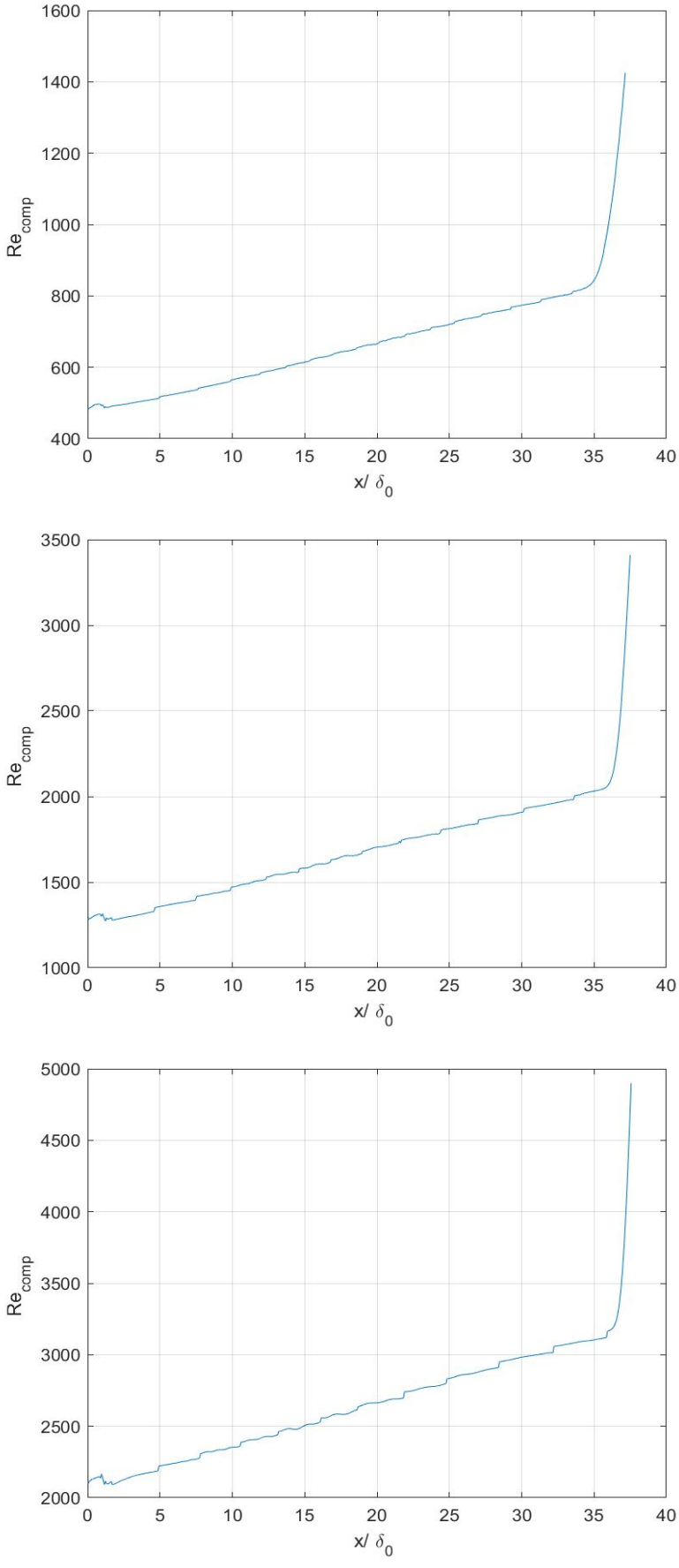



*Figure 5-19 . Compressible Reynolds number during statistically Steady State;* $Re_\tau = [200 \quad 500 \quad 800]$

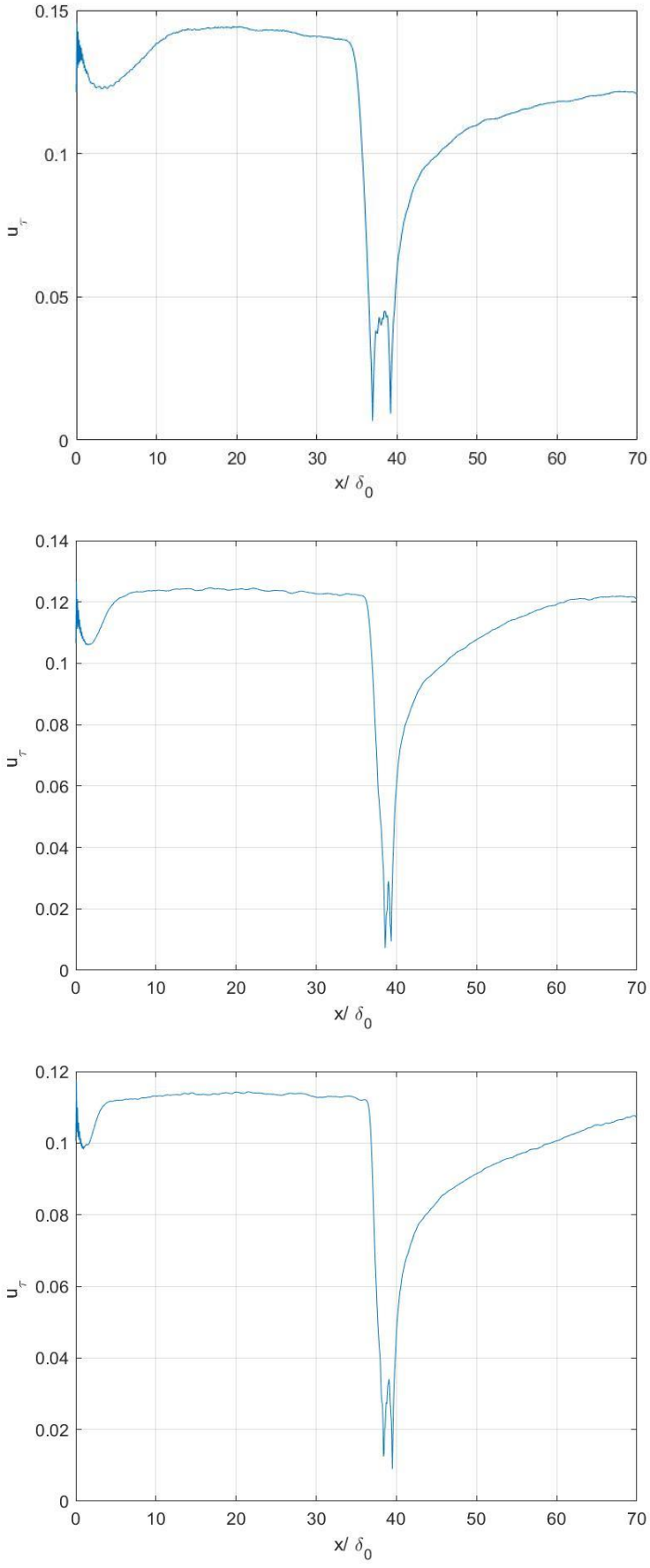



*Figure 5-20 Friction velocity during transitory.* $Re_\tau = [200 \quad 500 \quad 800]$

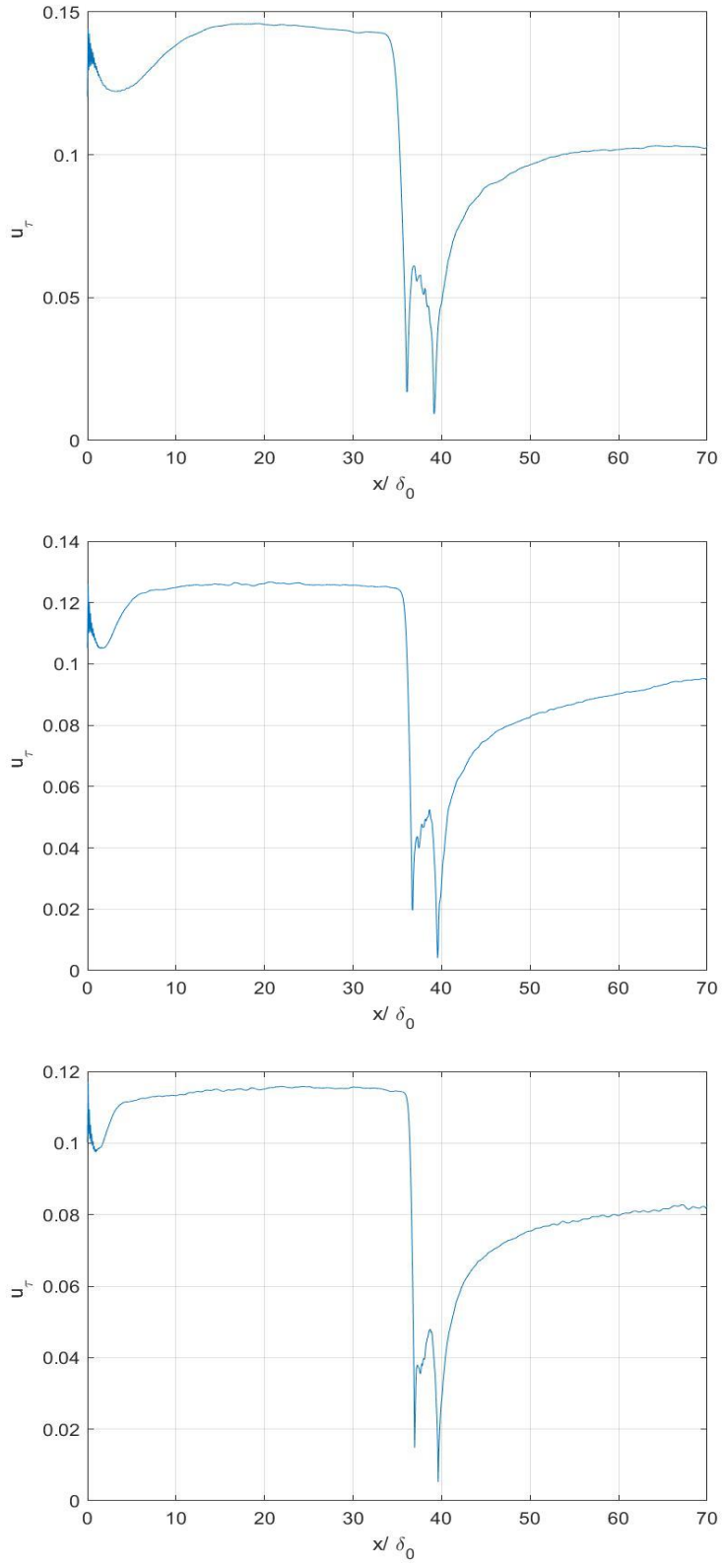



*Figure 5-21 Friction velocity during statistically Steady State* $Re_\tau = [200 \quad 500 \quad 800]$.

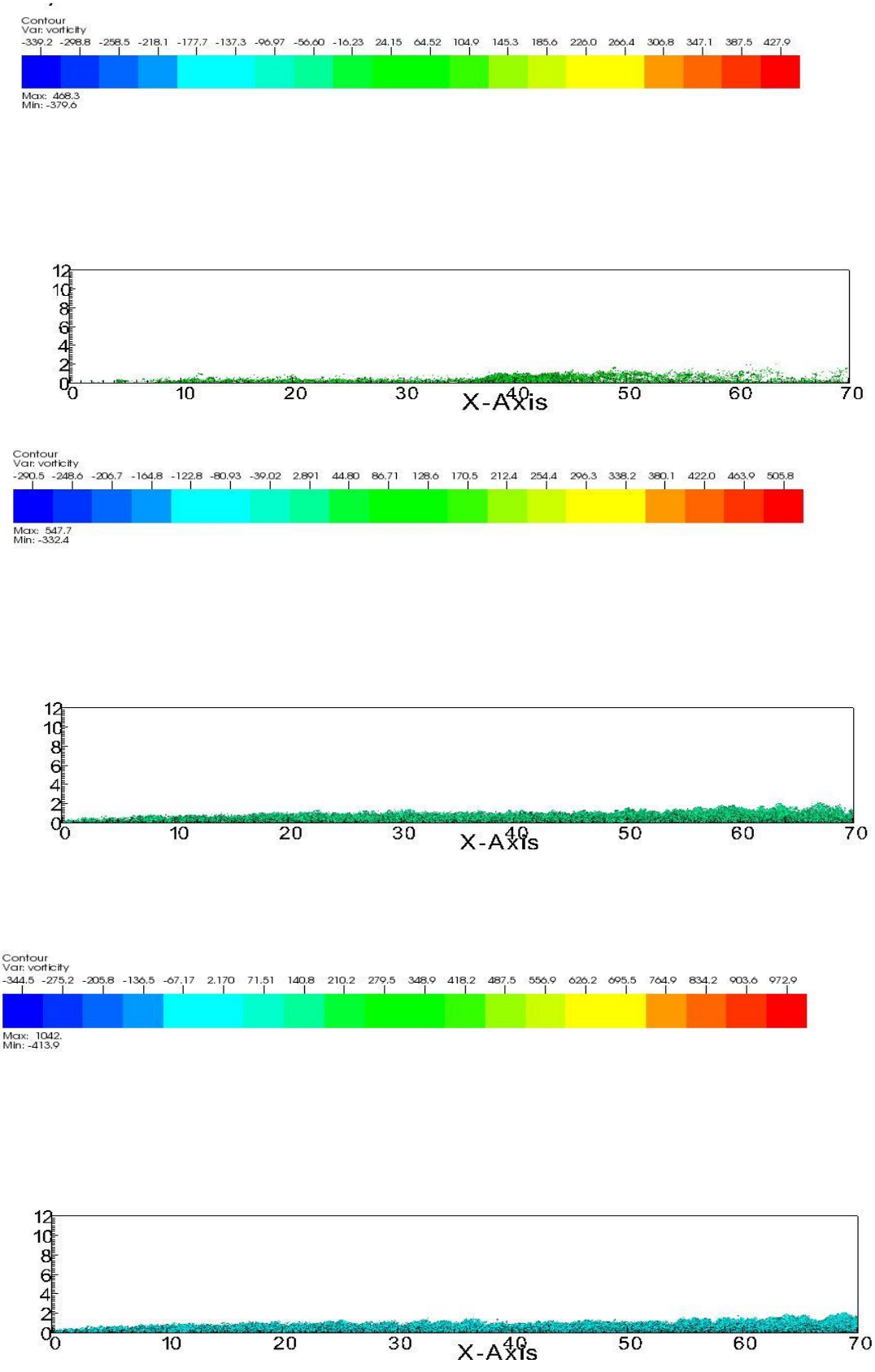



*Figure 5-22 q-criterion applied to the last .vtr file saved by the program.*

$$Re_\tau = [200 \quad 500 \quad 800]$$

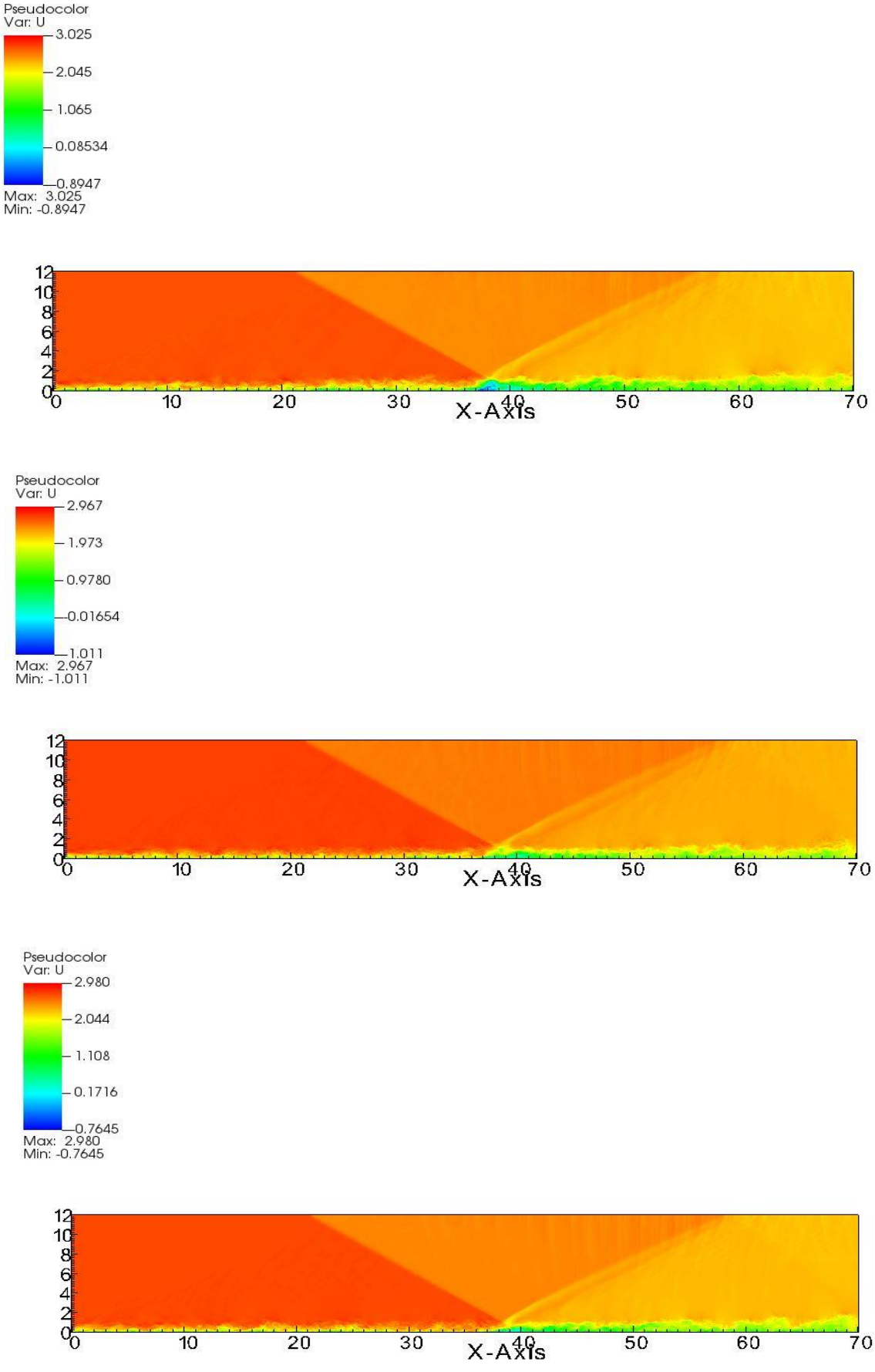



*Figure 5-23 Pseudocolor of the U , $Re_\tau = [200 \quad 500 \quad 800]$*

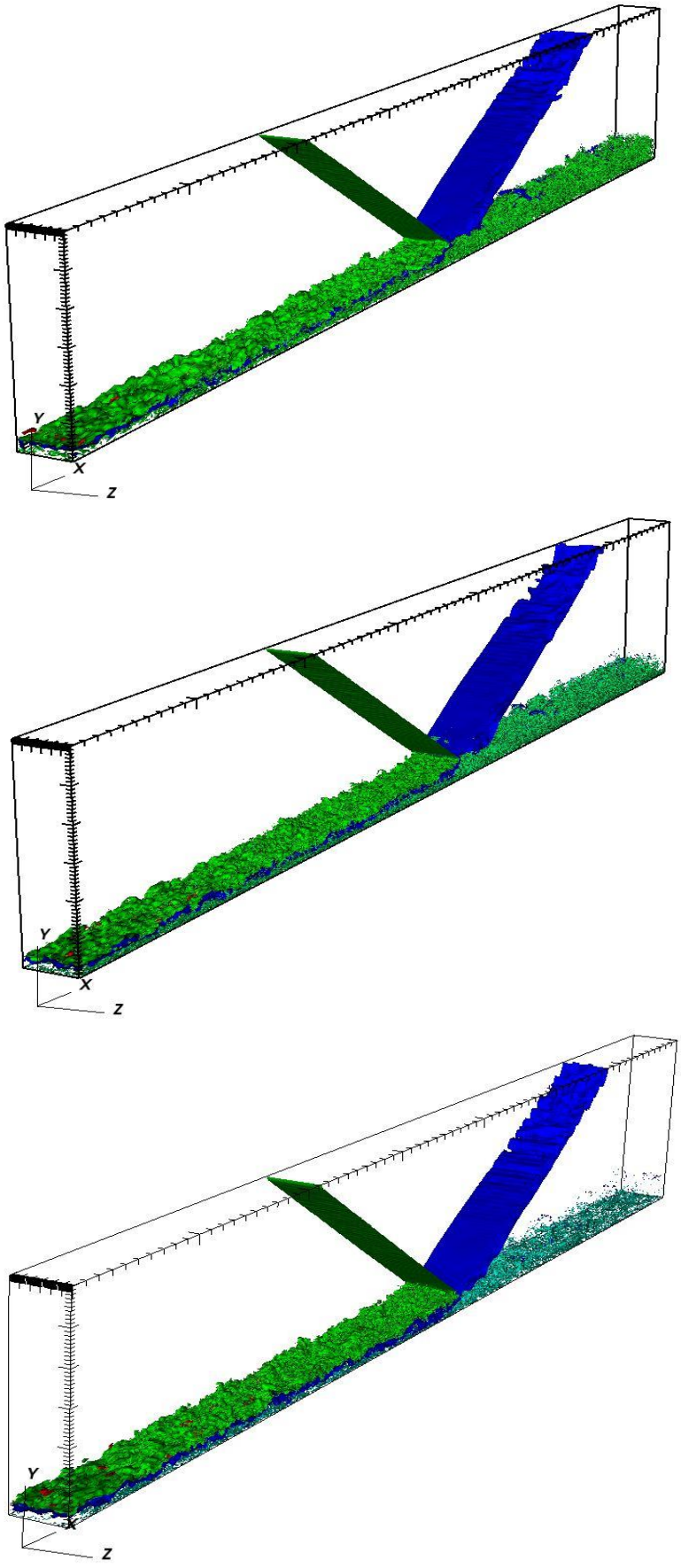



*Figure 5-24 Mach contour and q-criterion contour for* $Re_\tau = [200 \quad 500 \quad 800]$

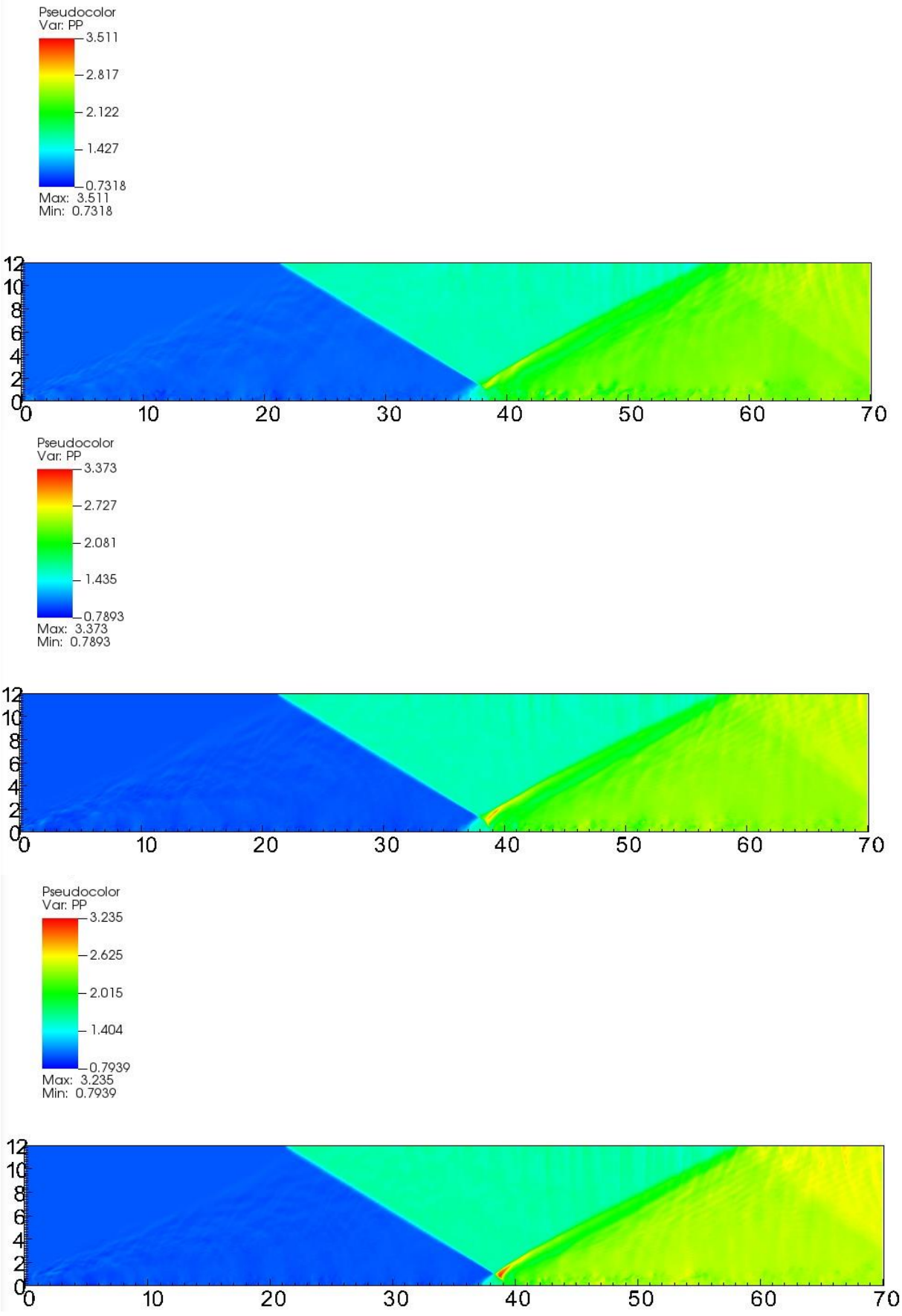



*Figure 5-25 Pressure pseudocolor for* $Re_\tau = [200 \quad 500 \quad 800]$

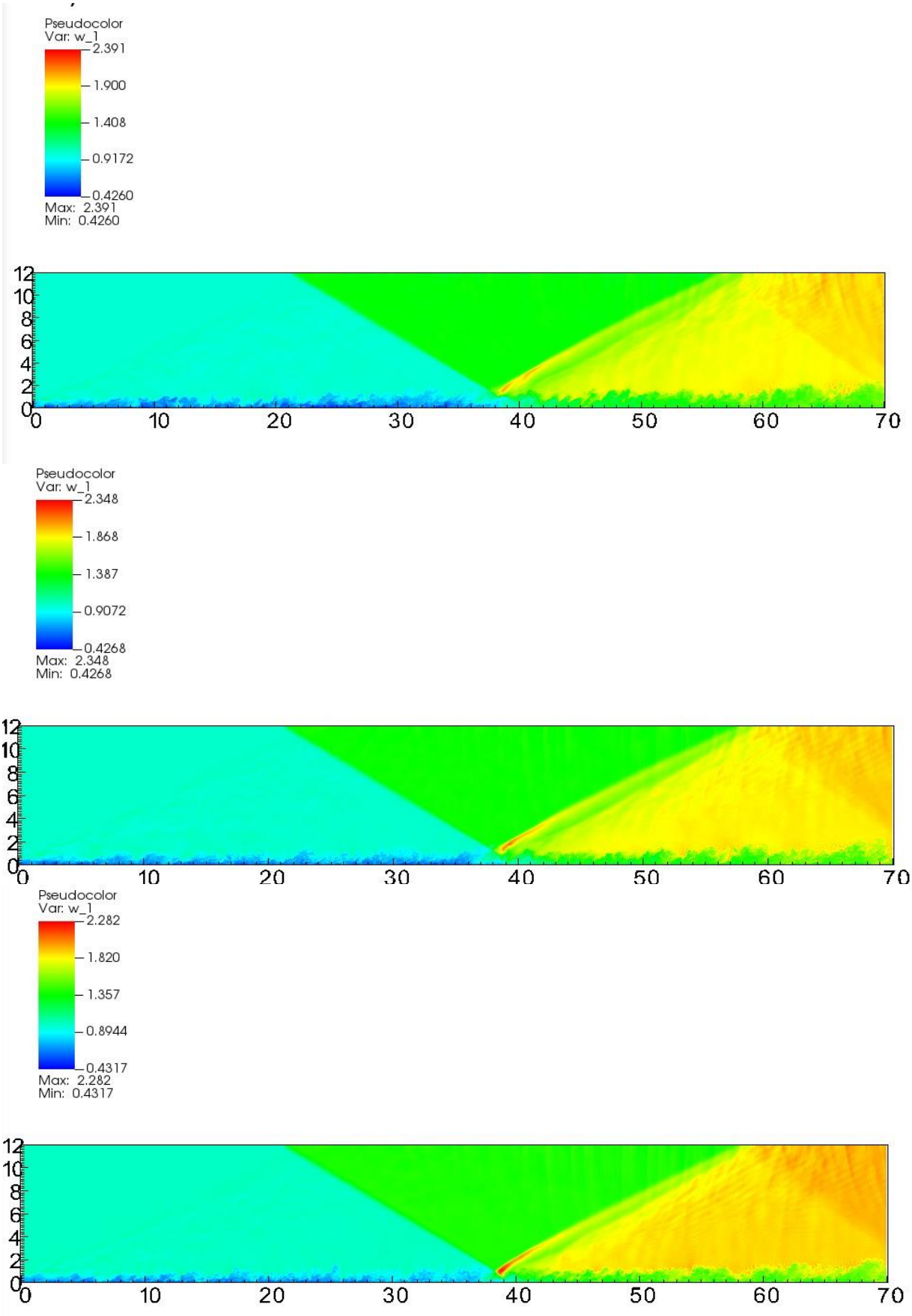



*Figure 5-26 Density pseudocolor for* $Re_\tau = [200 \quad 500 \quad 800]$

## 5.2 Parametric study of SBLI varying temperature ratio $T_{rat}$

We carried out a parametric study by varying the ratio $T_{rat}$ between wall temperature and adiabatic wall temperature. The simulations run over four GPUs in the same cluster node.

The three cases have values of $\frac{T_{wall}}{T_{aw}} = [1 \quad 2 \quad 3]$ and an inflow friction Reynolds number of $Re_\tau = 500$.

The computational domain had lengths rlx = 70, rly = 12, rlz = 3 and grid size was 1024x192x72 points.

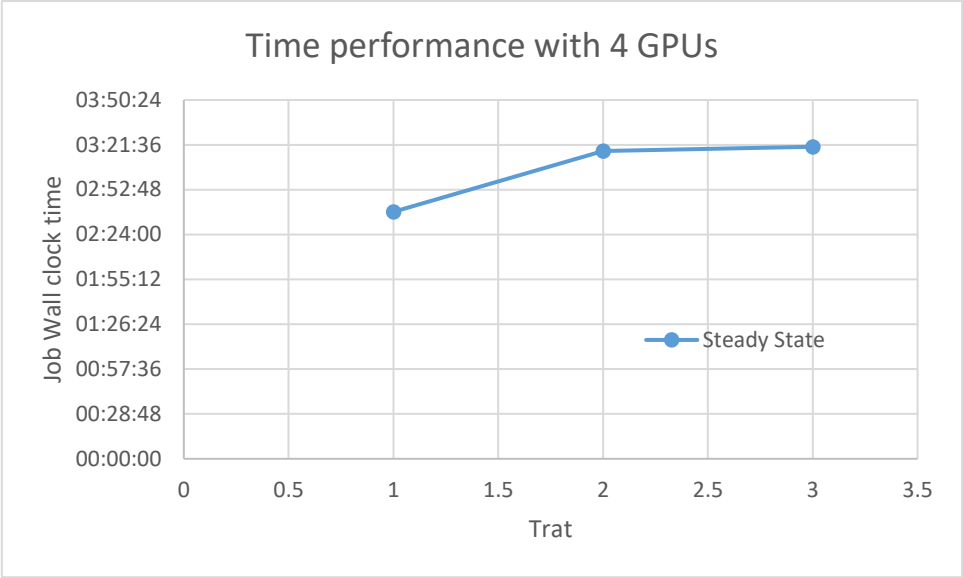
Mach number was set to 2.28. Shock wave angle is 8 deg and nominal impinging point is x = 40. The CFL number for numerical stability is 0.5 .

The statistics shown are:

1. Mean Velocity in streamwise stations $x = [20 \quad 40 \quad 60]$ , normalized with the friction velocity;
2. Compressible Friction Coefficient ;
3. Reynolds stress tensor for x = 20;
4. Reynolds stress tensor for x = 40;
5. Reynolds stress tensor for x = 60;
6. Mean pressure normalized with wall shear stress;
7. Pressure root mean square normalized with square root of wall shear stress;
8. Friction Reynolds number;
9. Compressible Reynolds number;
10. Friction velocity.

Only statistics at convergent steady state have been reported in this thesis , cutting off the transitory with a restart of STREAmS after a prefixed number of iterations equal to 80000. It can be pointed out that the final physical time Tf of the simulation reduces by increasing wall temperature , as the speed of sound rises and therefore the computed time step for CFL stability gets smaller .

Time performance and memory usage per grid point are reported in the following graphs .



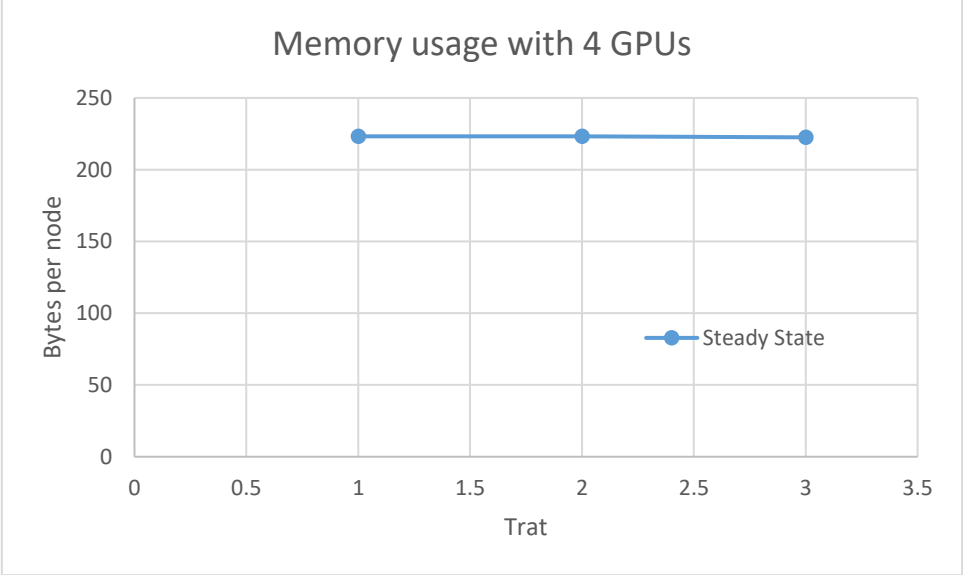




*Table 7 Integration time table for temperature parametric study*

| $\frac{T_{wall}}{T_{aw}}$ | $T_0$ [s] | $T_f$ [s] | N° Iterations Transitory | N° Iterations Steady State | Istat |
|---|---|---|---|---|---|
| 1 | 32 | 72 | 80000 | 180000 | 250 |
| 2 | 24 | 59 | 80000 | 200000 | 250 |
| 3 | 19 | 48 | 80000 | 200000 | 250 |

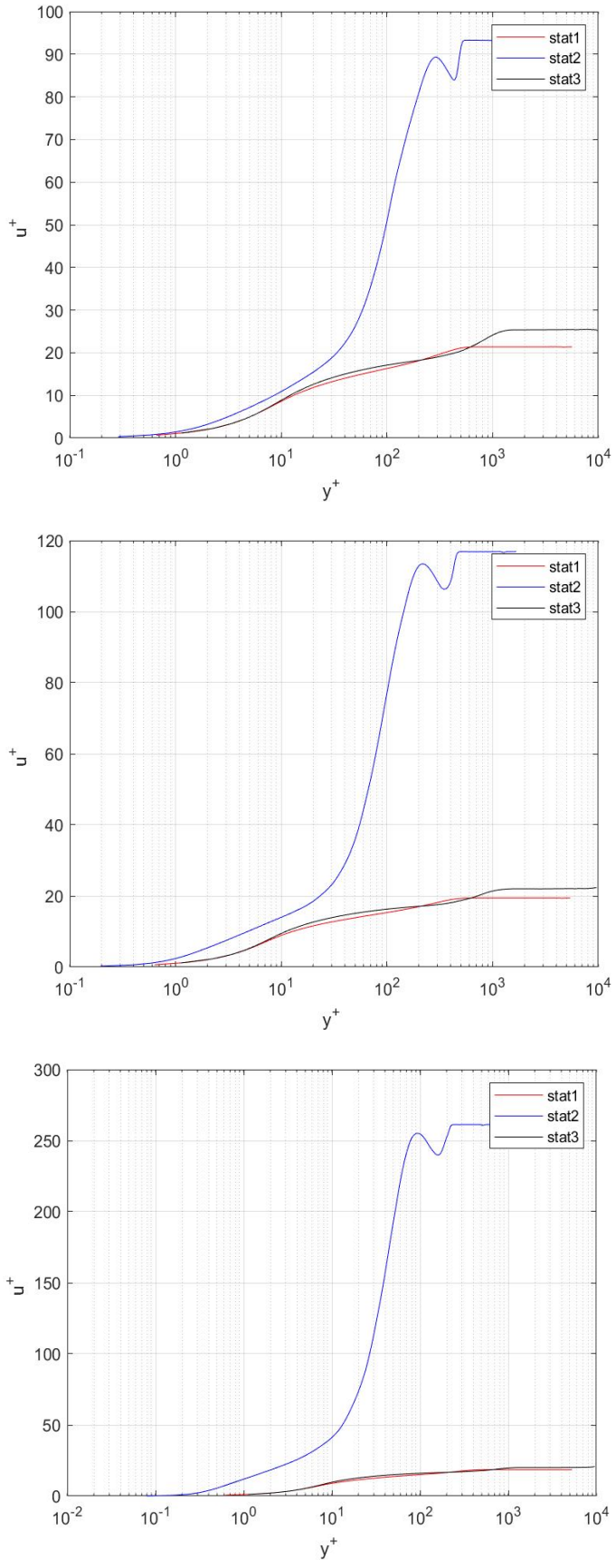



*Figure 5-27 Mean streamwise velocity profile for* $\frac{T_{wall}}{T_{aw}} = [1 \quad 2 \quad 3]$

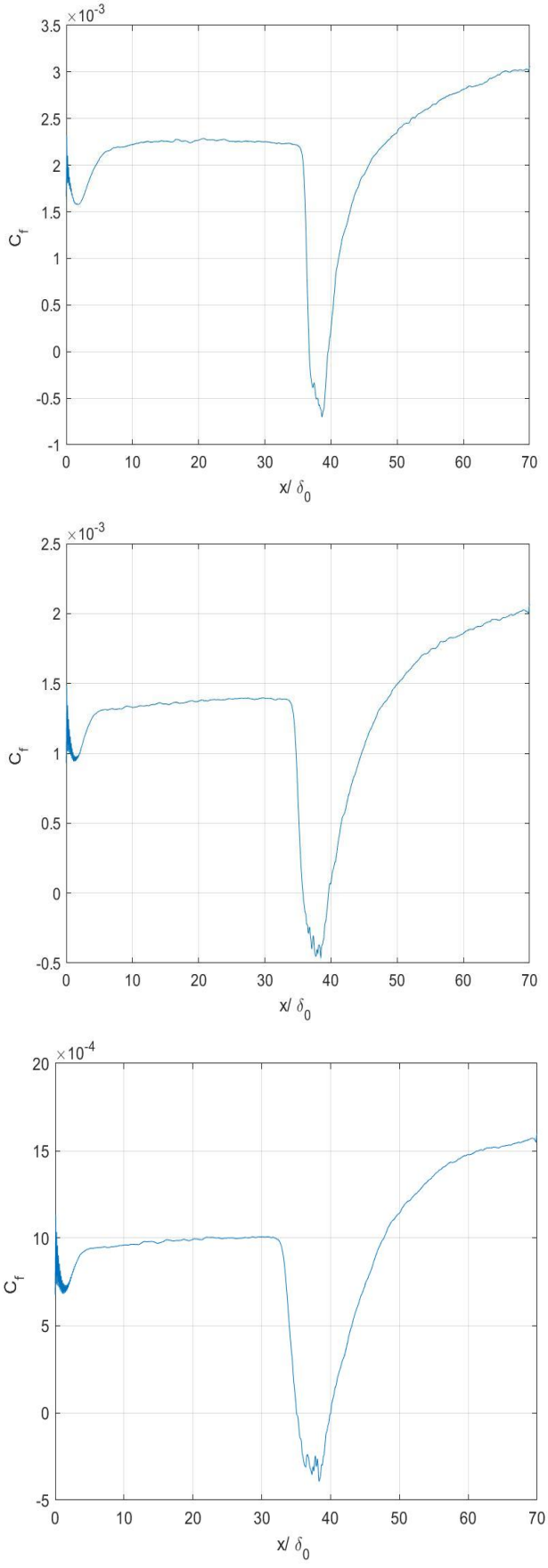



*Figure 5-28 Compressible Friction Coefficient , it gets smaller after the reflected shock by rising the wall temperature.*

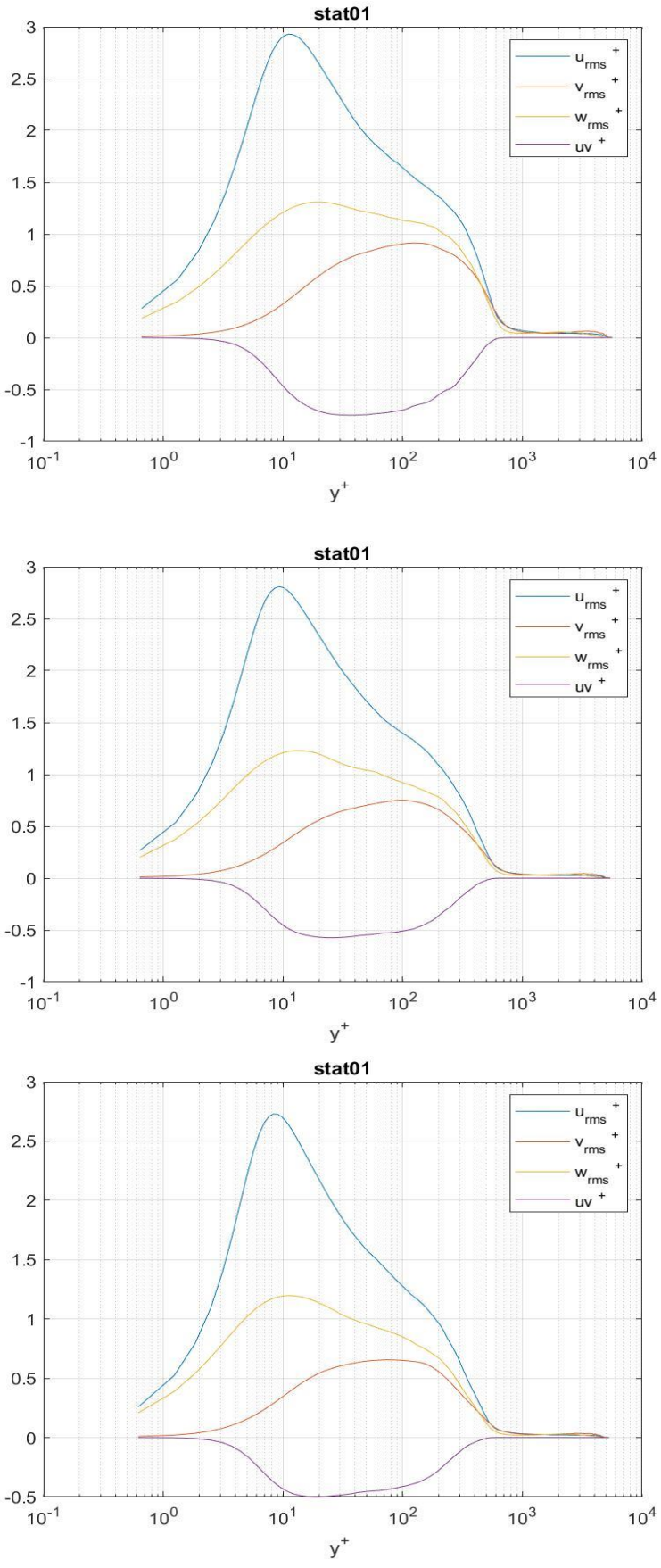



*Figure 5-29 . Reynolds Stress tensor for x = 20*

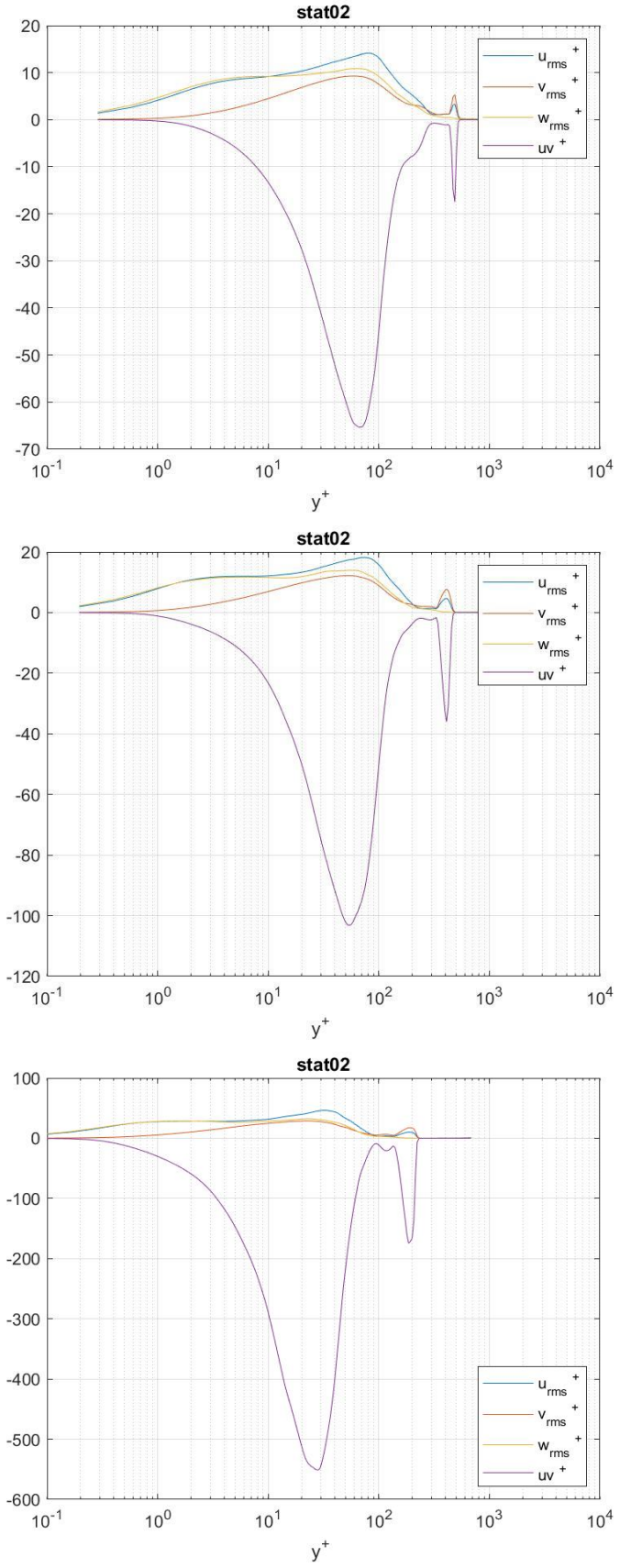



*Figure 5-30 Reynolds Stress Tensor for x = 40 , In the interaction region the Reynolds shear stress gets deeply negative by rising wall temperature.*

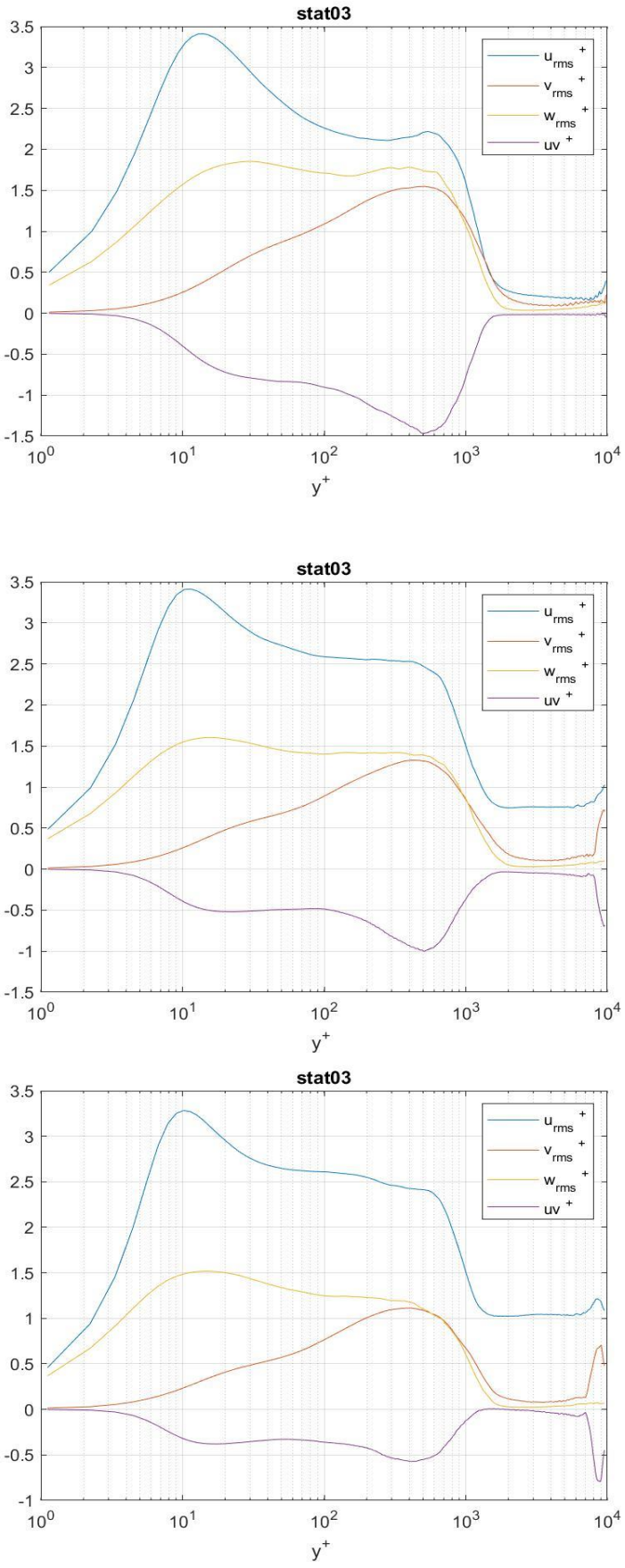



*Figure 5-31 Reynolds Stress Tensor for x = 60*

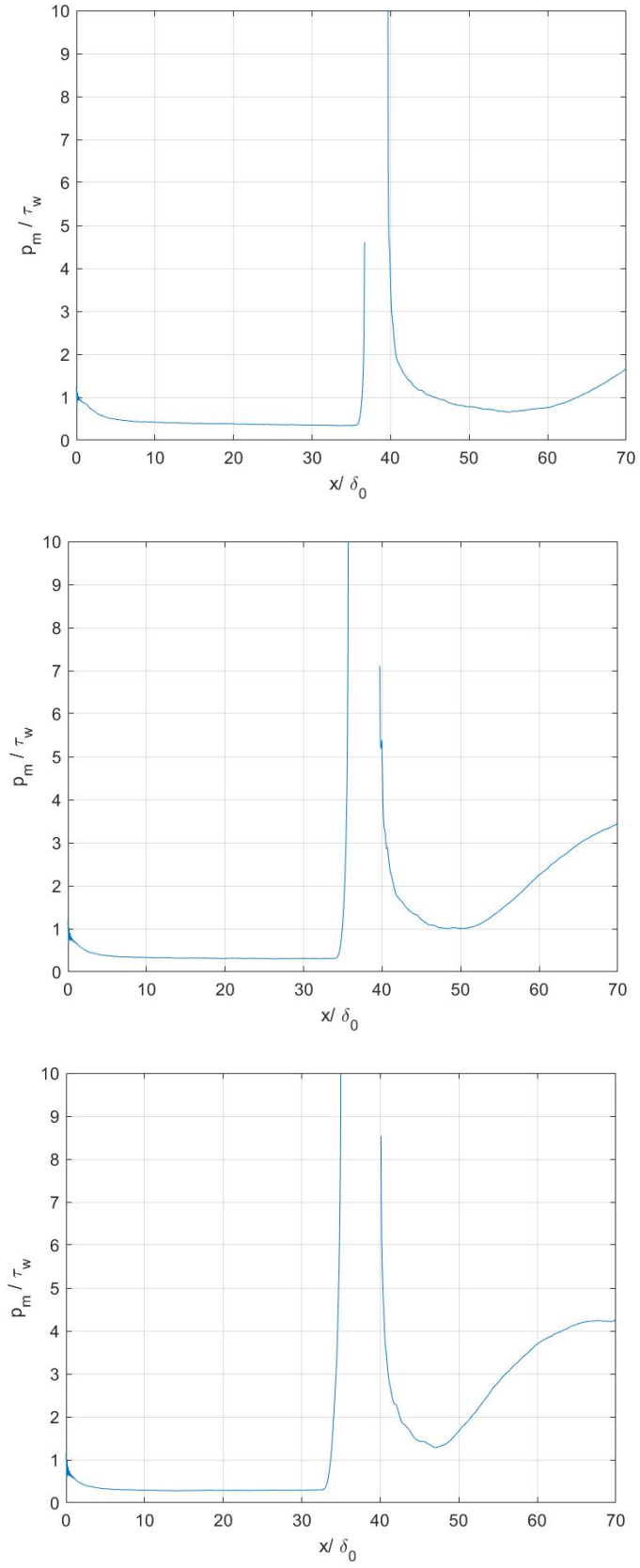



*Figure 5-32 . Mean pressure normalized with the wall shear stress for* $T_{rat} = [1 \quad 2 \quad 3]$

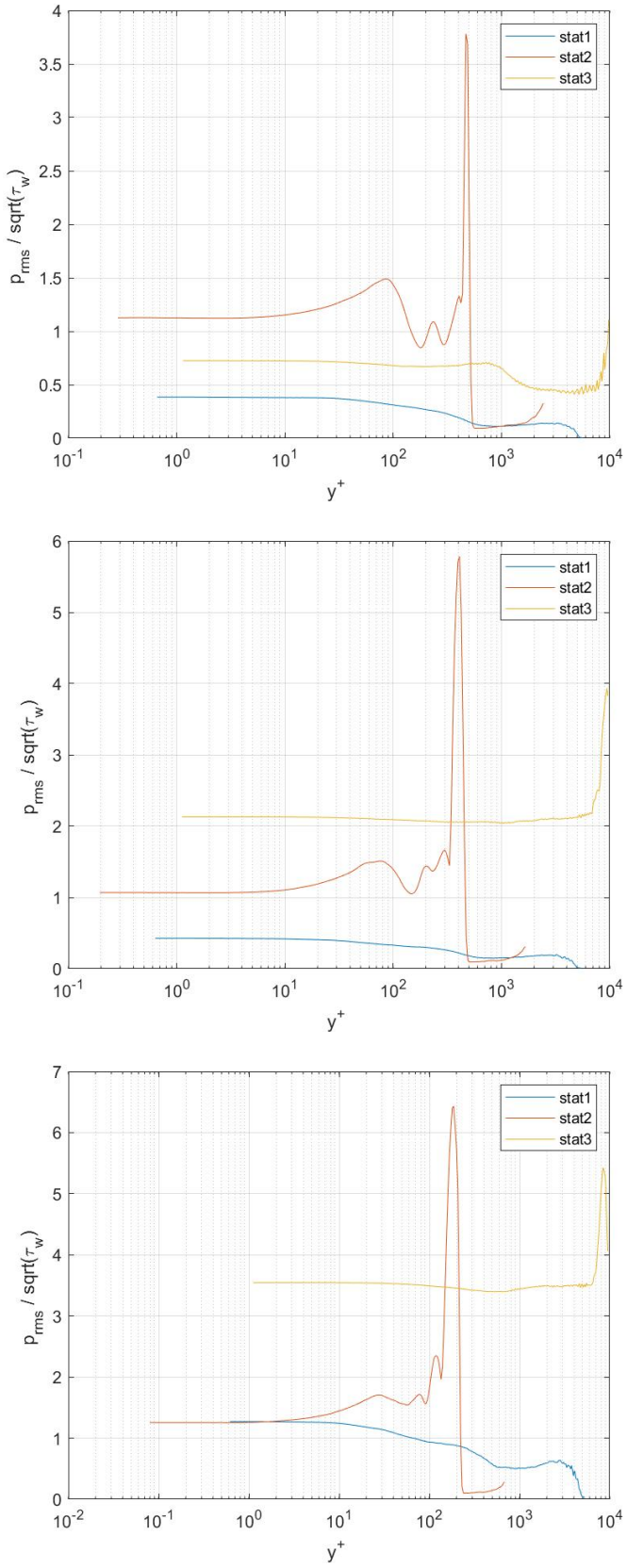



*Figure 5-33 Pressure root mean square , the pressure fluctuation after the reflected shock gets higher by increasing wall temperature.*

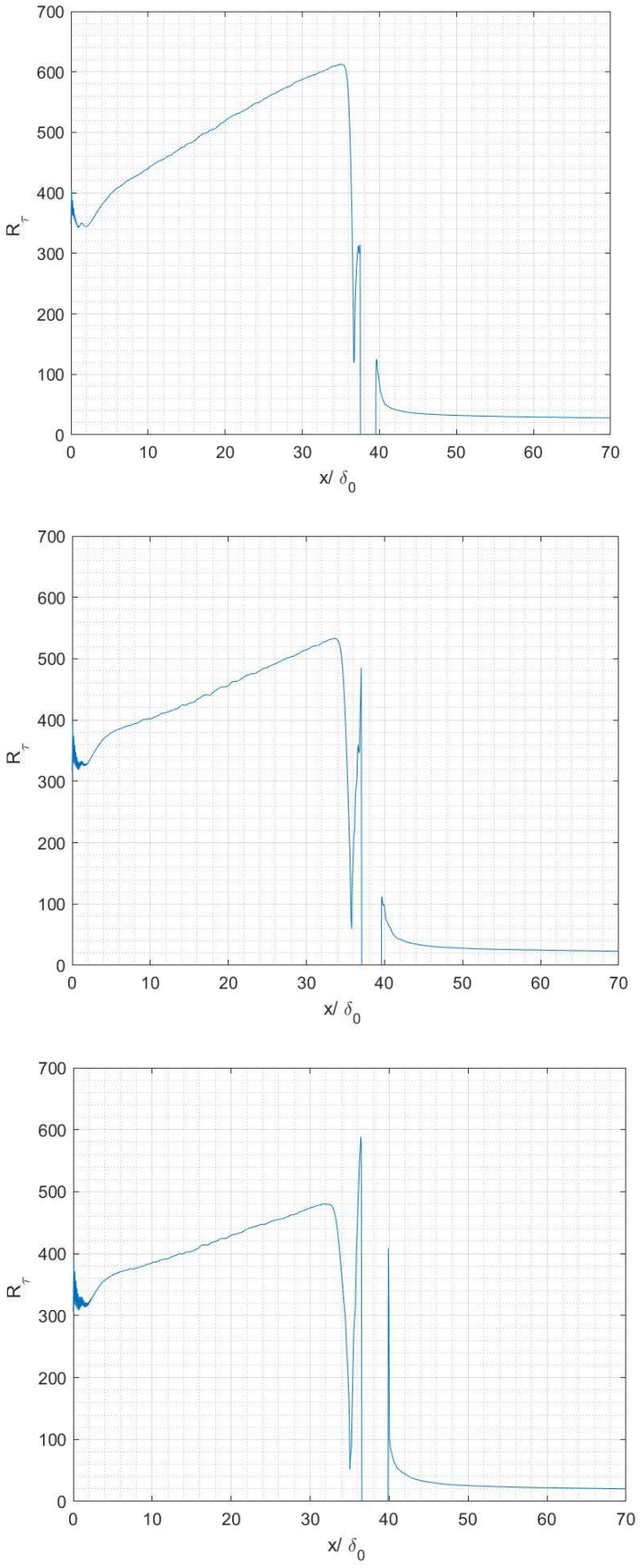



*Figure 5-34 Friction Reynolds number for* $T_{rat} = [1 \quad 2 \quad 3]$

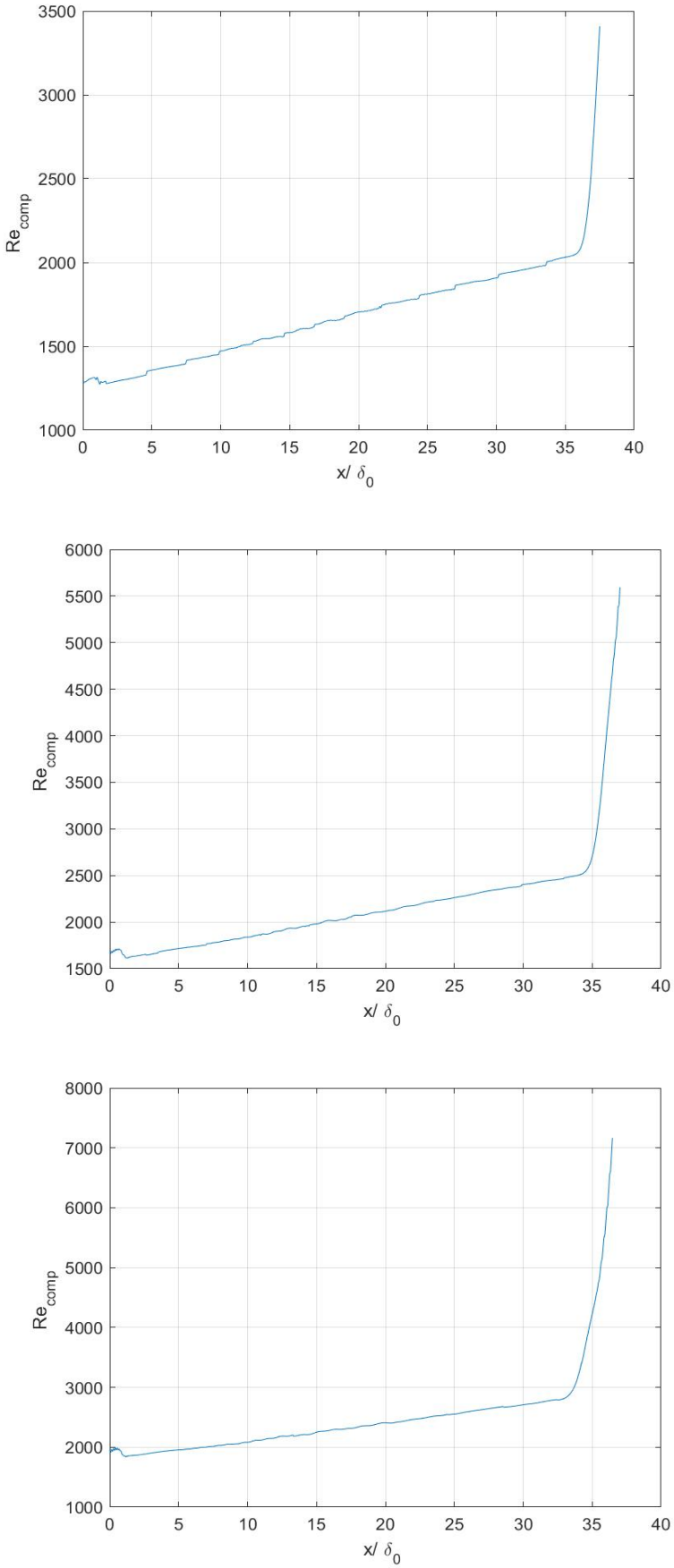



*Figure 5-35Compressible Reynolds number for* $T_{rat} = [1 \quad 2 \quad 3]$

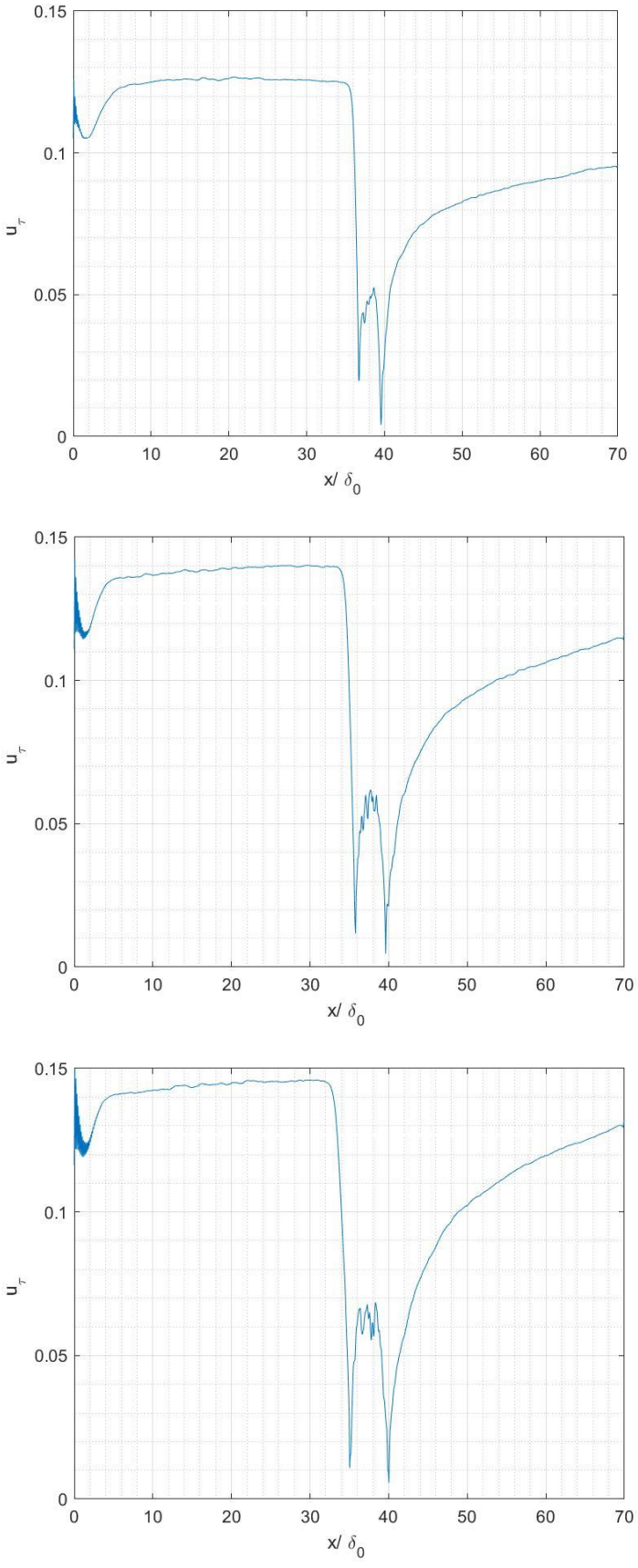



*Figure 5-36 Friction Velocity for* $T_{rat} = [1 \quad 2 \quad 3]$

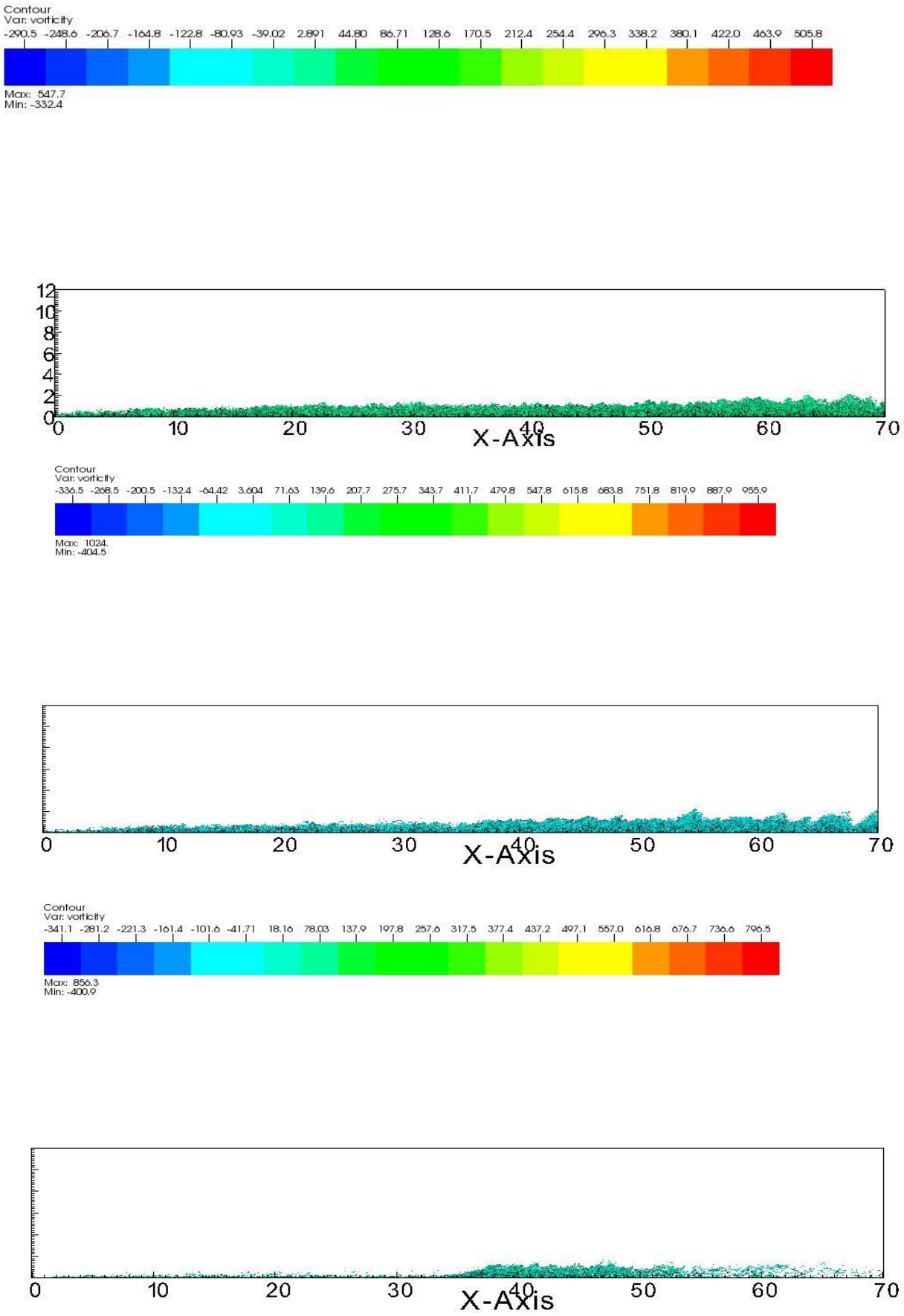



*Figure 5-37 Q-criterion applied to last .vtr file saved by the program.*

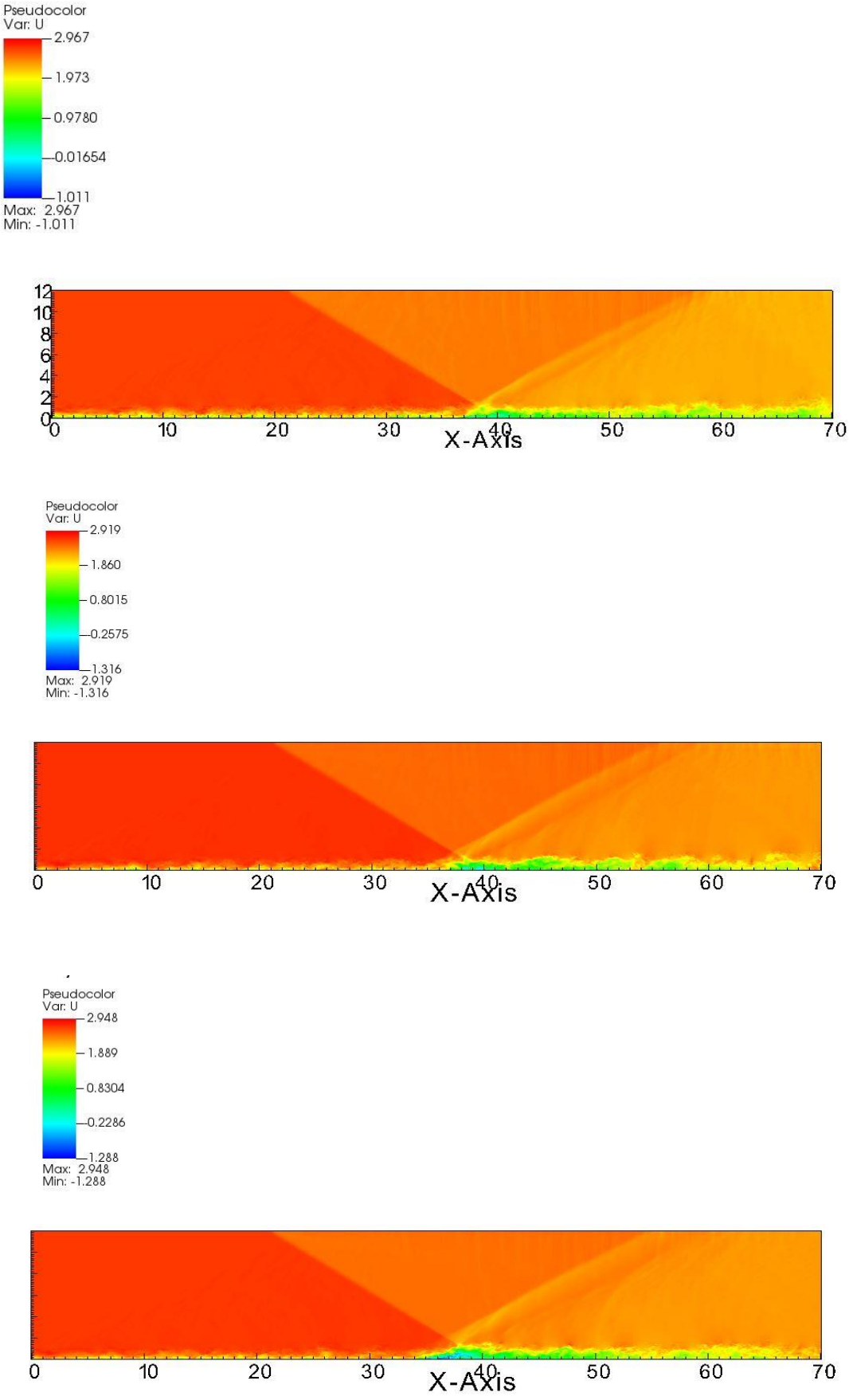



*Figure 5-38 Pseudocolor of the U velocity component of the last .vtr file saved by the program.*

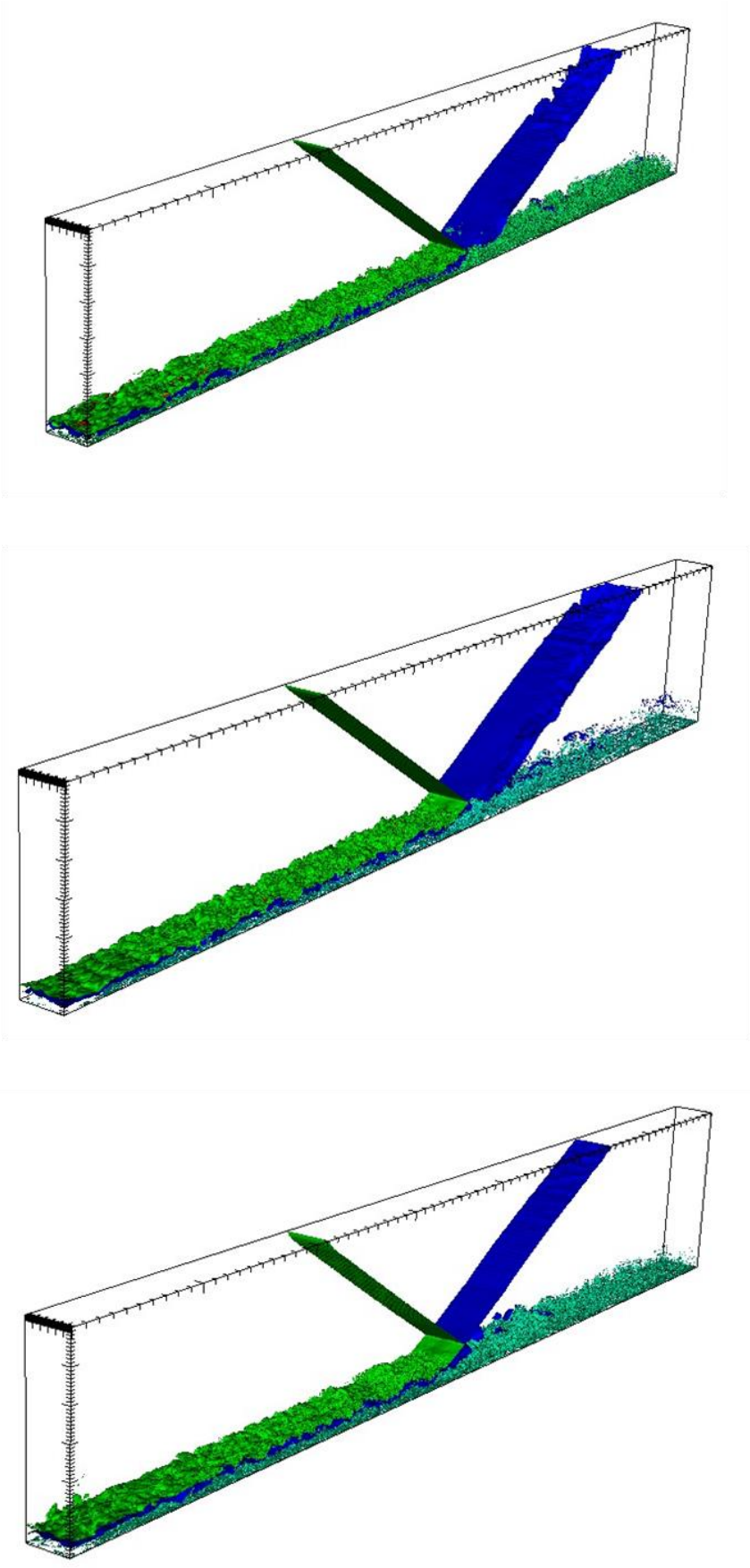


*Figure 5-39 Mach contour and q-criterion contour for* $T_{rat} = [1 \quad 2 \quad 3]$

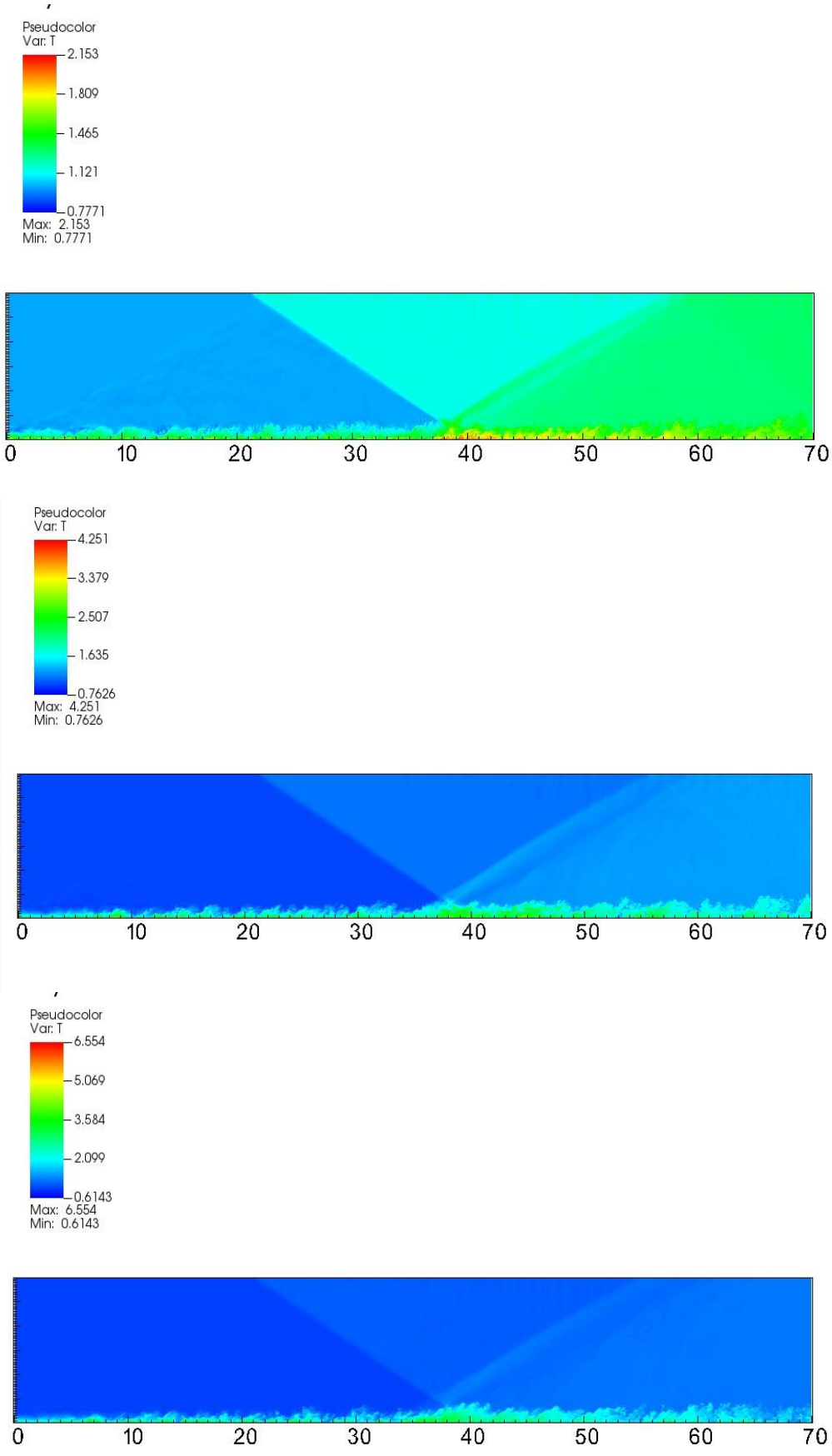



*Figure 5-40 Temperature pseudocolor for* $T_{rat} = [1 \quad 2 \quad 3]$ .

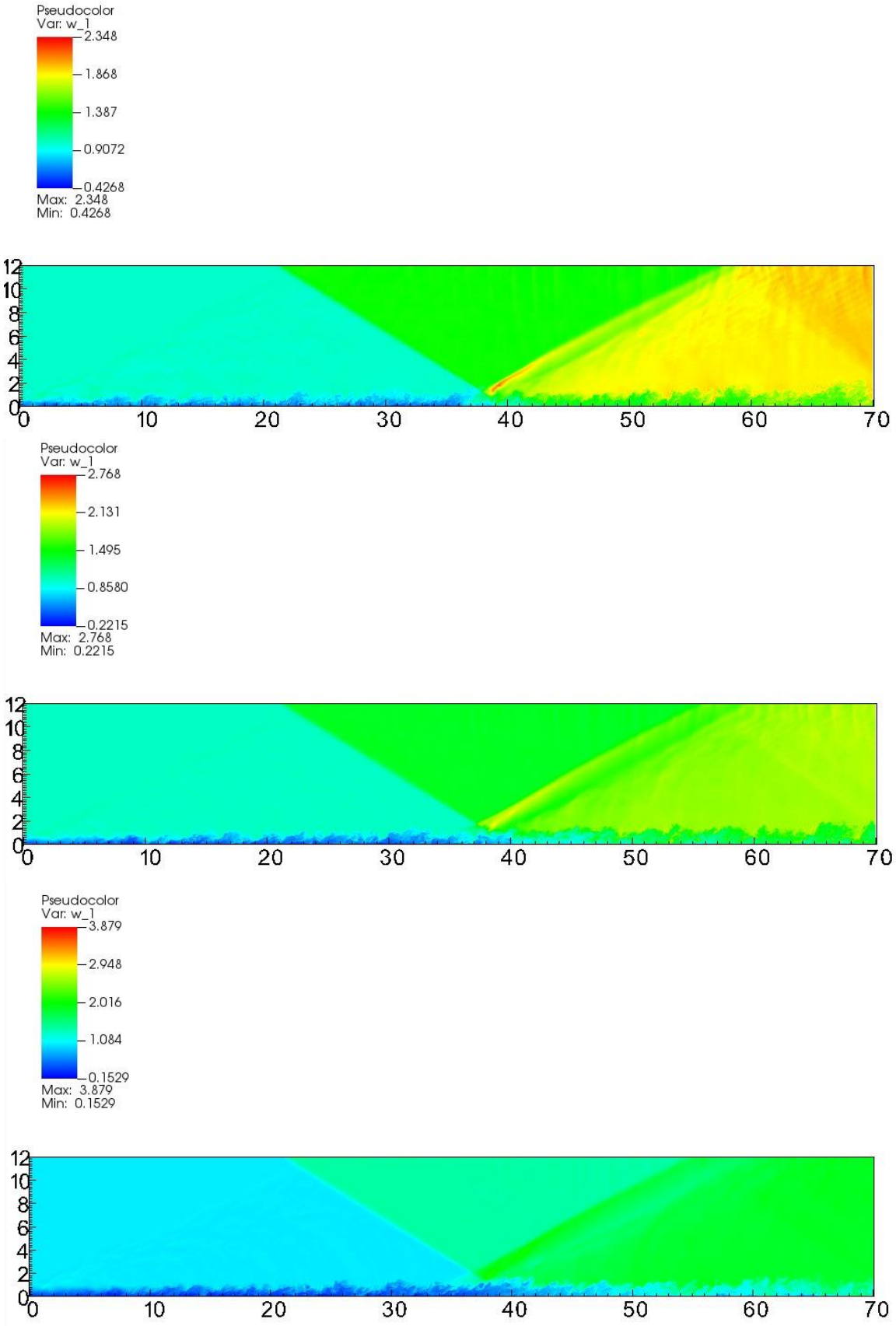



*Figure 5-41 Density pseudocolor for* $T_{rat} = [1 \quad 2 \quad 3]$

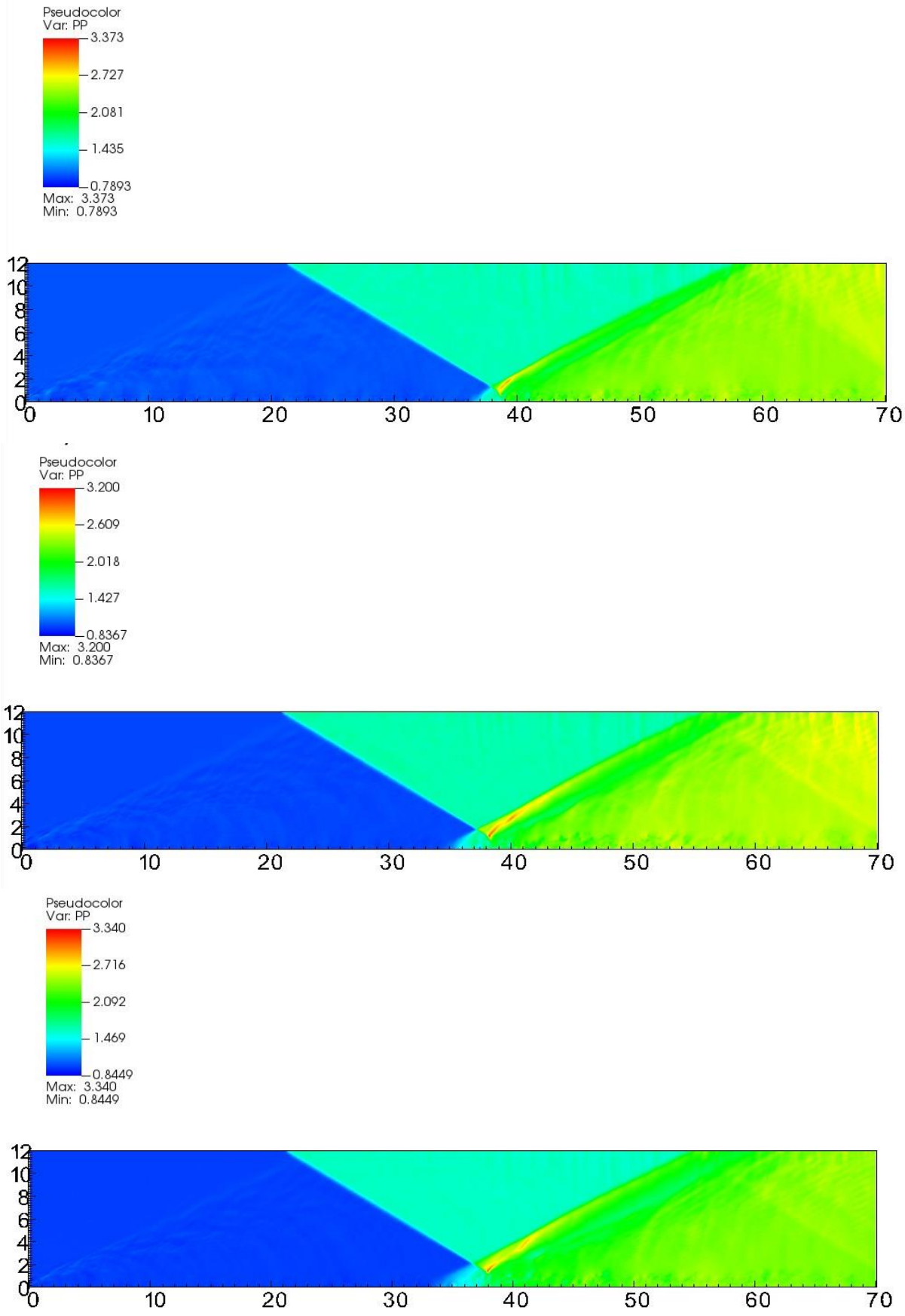



*Figure 5-42 Pressure pseudocolor for* $T_{rat} = [1 \quad 2 \quad 3]$

## 5.3 Q-Criterion applied to different grid sizes

This section is focused on how DNS simulations can capture different eddy length scales by changing the number of grid points in the three cartesian dimensions.

Simulations were carried out on a computational domain with limits rlx = 70 , rly = 12 , rlz = 3, in the three cartesian dimensions .

Mach number was equal to 2.28 , Shock wave angle is 8 deg and nominal impinging point is x = 40. The CFL number for numerical stability is 0.5 . Friction Reynolds number $Re_\tau = 475$.

- First case had a grid size of 1024x96x72 points;
- Second case had a grid size of 1024x192x72 points;
- Third case had a grid size of 2048x192x72 points;

The q-criterion was applied to the last .vtr file saved by the program , corresponding to a physical time of nearly $200\ s$ and a number of 500000 iterations.

The legends of the graphs show an increase in the range of values of the q-criterion. The CFD solver can capture turbulent eddies with smaller length scales by reducing the grid size with more points in the three cartesian dimensions. Thus, when we do not apply a simplified model for turbulence in the set of equations , such as in Direct Numerical Simulations, it is important to choose a finer mesh to capture eddies with different turbulent energies , that may be of interest for the case under investigation .

The compressible friction coefficient statistics are reported , by increasing the grid size the discrete solution converges to the real one as the truncation error reduces with a finer mesh for the computational domain .

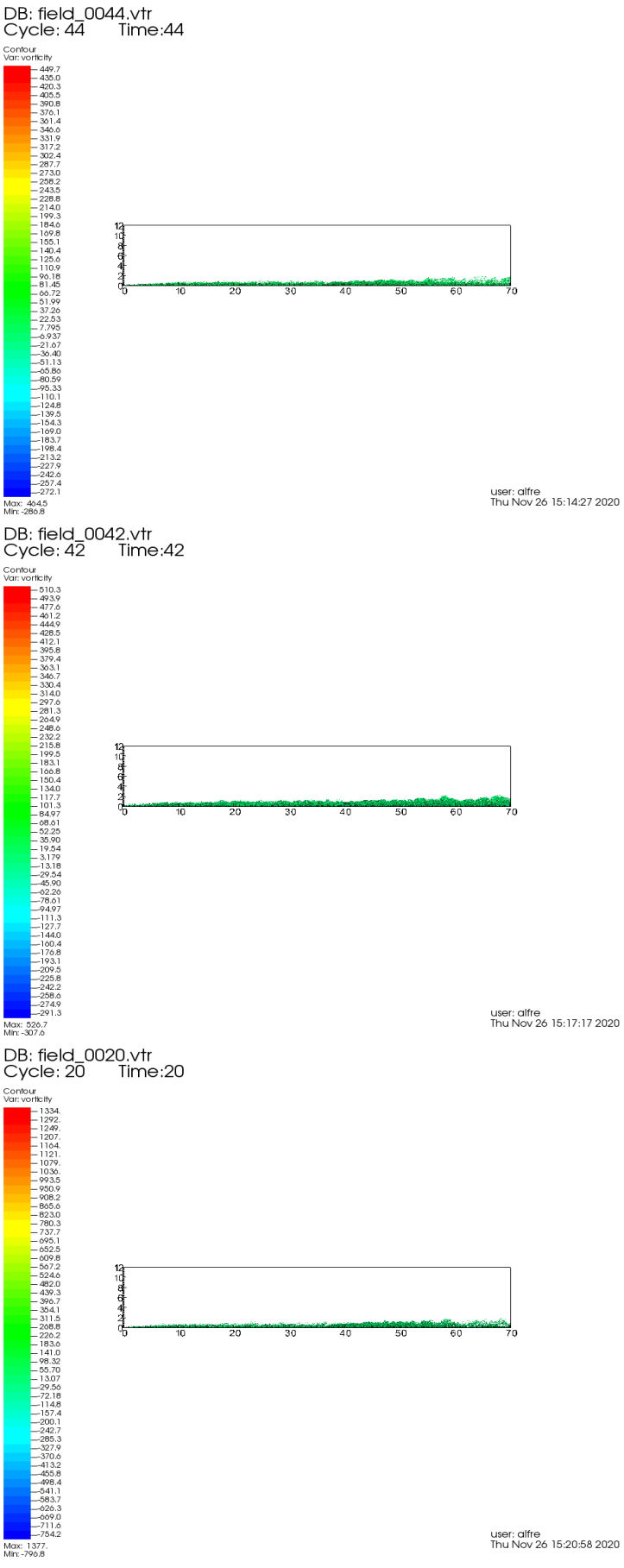



*Figure 5-43 Q-criterion applied to simulations with different grid sizes*

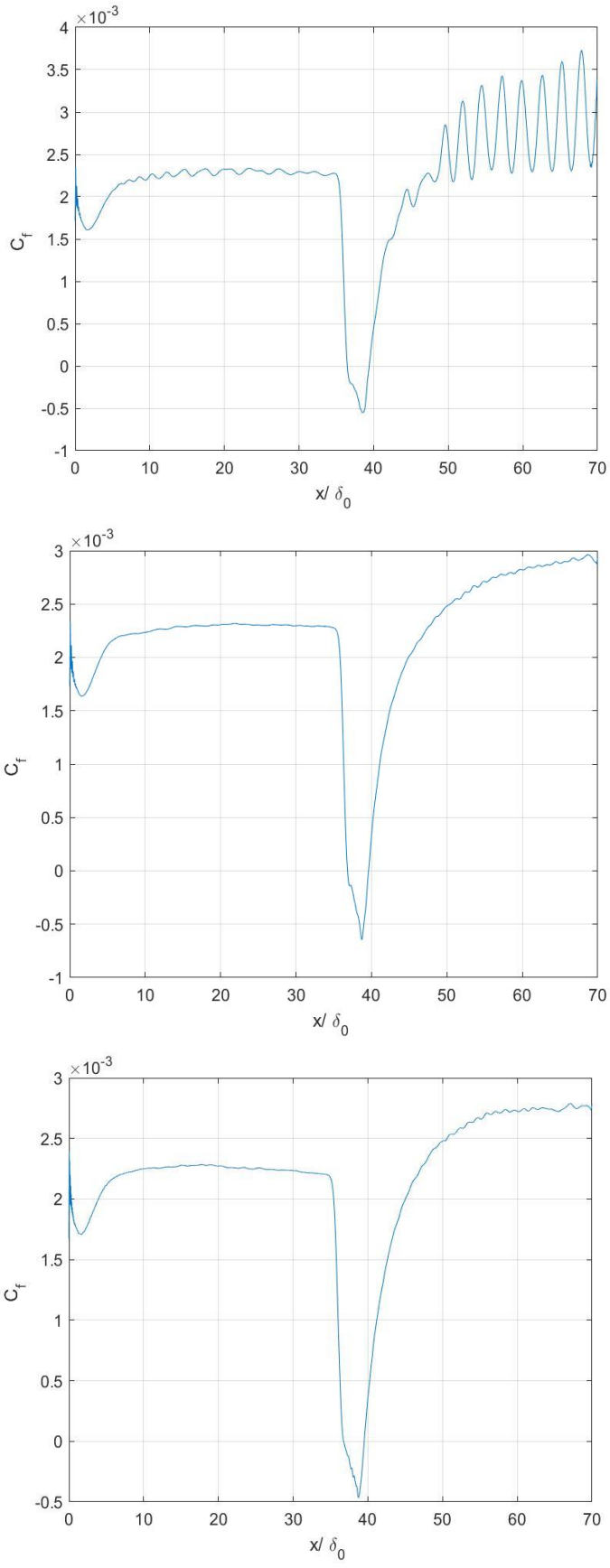



*Figure 5-44 Difference in Cf statistics with increasing grid sizes .*

# Chapter 6

# A 2D CFD Solver in modern GPU Architecture

The CFD solver of 2D Euler equations, based on a finite volume numerical scheme, has been parallelized using CUDA Fortran library for a GPU implementation in HPC@POLITO Academic Computing Center. The set of PDE we want to solve is the following , where **U** is the vector of conservative variables, **F** is the flux tensor for compressible inviscid flows, **V** is the cell volume ( in the 2D case the cell area ) , **S** is the cell delimiting surface ( in the 2D case the cell edges ) and $\bar{\boldsymbol{n}}$ is the external normal to the cell surface :

$$\frac{\partial}{\partial t}\int_V U\, dV + \int_S \bar{F}\cdot\bar{n}\, dS = 0$$

$$U = \begin{bmatrix} \rho \\ \rho\bar{q} \\ \rho E \end{bmatrix}$$

$$F = \begin{bmatrix} \rho\bar{q} \\ \rho\bar{q}\otimes\bar{q} + p\bar{I} \\ (\rho E + p)\bar{q} \end{bmatrix}$$

In the module *variabili.f90* we use *cudafor* to access GPU directives , kernels and special memory attributes . The number of grid points in x and y directions is defined in the *input.dat* file . We carried out dynamic allocation of arrays through the attribute *allocatable* in the variable declaration to avoid segmentation fault errors with high numbers of grid points.

Variables resident in the GPU memory have the *device* attribute , we can simply copy data from the CPU to the GPU and back with explicit assignment syntax ( for instance A_gpu = A ) . To improve performance we used the *pinned* attribute for variables which are frequently transferred between host and device , to put them in page-locked host memory, such as primitive variables *a, u, v, p, T, s* .

GPU read-only variables are allocated in the device *constant* memory space , which allows really fast accesses but can be overwritten only by the host CPU. Some scalar variables , such as velocity module $q$ and local time steps $\Delta t_{loc_x}$ , are allocated in local thread private memory.

Using **!$cuf** kernels we parallelize independent loops in the grid generation through the automatic CUDA Fortran mapping of loops on the GPU threads . In this way we do not need to specify the number of threads in a block and the number of blocks in the grid and thread indexes are calculated automatically. The mesh grid is algebraic and the lower boundary is represented by a sine curve defined between the two abscissas x1 and x2. For instance the y coordinate of the mesh is:

```
!$cuf kernel do(2) <<<*,*>>>
    do i=0,nc
        do j=0,mc
            y_gpu(i,j)=b_gpu(i)+(c_gpu(i)-b_gpu(i))*(1.*j)/mc
        end do
    end do
!@cuf iercuda=cudaDeviceSynchronize()
```

As GPU Kernel execution is asynchronous with respect to the host code execution , we need to call the *cudaDeviceSynchronize* routine after each *cuf* kernel to stop the CPU and wait that every previous Kernel launch is completed . After that we need to copy back to the CPU grid point matrices x and y, as well as cell centers , cell edges and edge normal components . Hence , we can print out the mesh using the CPU to write .plt files .

The same approach is used to define the initial condition, GPU versions of conservative and primitive variables are needed to compute Eulerian fluxes and to integrate in time on the device.

In the subroutine *compute_dt.f90* we need to evaluate the minimum global time step that guarantees the CFL numerical stability. However , to make it in parallel on the GPU we can loop in one cartesian dimension with multiple threads set in the other one . For each cell center , the local time step in x and y direction is :

$$\Delta t_{loc_x} = \frac{\Delta x}{(u+a)} CFL$$

$$\Delta t_{loc_y} = \frac{\Delta y}{(v+a)} CFL$$

When we have a two dimensional nested loop , we can decide to parallelize just the outer one with multiple threads with the **do(1)** syntax . In this way , each thread is going to loop on the other dimension to evaluate the minimum time step of a single slice of data . Hence , multiple threads were set in the x direction and each thread looped in the y direction to calculate local minimum time step of the vertical slice of cell centers. Finally , to evaluate minimum global time step we can copy back to the CPU memory the time steps calculated previously and use the *minval* routine on host resident variables to reduce the operation.

The main purpose of GPUs is to maximize the throughput using thousands of cores which run in parallel independent operations . Therefore, we can calculate Eulerian fluxes at the cell interfaces for internal points of the domain following local Lax-Friedrichs schemes, using device resident variables and !$cuf kernels defined previously.

$$F_{n+\frac{1}{2}} = \frac{1}{2}(F_{n+1} + F_n) - \lambda_{max}(U_{n+1} - U_n)$$

$$\lambda_{max} = max[(u+a)_n, (u+a)_{n+1}]$$

In this CUDA version of the CFD solver , we use one single MPI process and one single CPU-GPU unit of the cluster node . Hence , we do not need to implement the swapping procedure of ghost cell variables across different MPI processes, which is necessary to compute Eulerian fluxes at the boundaries of each subdomain when multiple MPI processes are used .

Instead, to compute fluxes at the boundary edges of the domain we still can parallelize the problem using one-dimensional blocks of threads . Some scalar variables local to each thread are defined using GPU primitive variables (*a, u, v, s, p* ) .

After the calculation of all Eulerian fluxes in the domain , we can perform time integration always in parallel through the use of GPU conservative variables . Moreover, we compute in parallel the primitive variables *a, u, v, p, T, s* for the next iteration.

$$U_{n,m}^{k+1} = U_{n,m}^{k} - \frac{\Delta t}{V_{n,m}} \sum \bar{F} \cdot \bar{n}\, \Delta S$$

Finally , to write the .plt files for Visit we need to copy back to the CPU the primitive variables and call the writing subroutine *WDKs_tk* . We need to remember that the copying procedure can deliver poor speed performance , so we should limit the printing procedure every $k_{out}$ iterations by setting a proper value in the input.dat file.

## 6.1 Comparison of performances

The following simulations were carried out using both the original CPU version of the code and the GPU CUDA Fortran version.

In the input.dat file we set the inlet Mach number $M_{inl} = 0.5$ and the $CFL = 0.5$ . Exit static pressure is $p_{exit} = p_{tot} / \left(1 + \frac{\gamma-1}{2} {M_{inl}}^2\right)^{\frac{\gamma}{\gamma-1}}$ and $p_{tot} = 1$ as equations are in non-dimensional form.

- We run several cases where we kept constant the number of iterations and we increased the number of grid points for the two versions.
- We evaluated performances by reporting the CPU-clock time , using subroutine *cpu_time* in the main program to estimate the time needed for the integration of equations , and the Memory usage per grid point.

*Table 8 Grid characteristics of the simulations carried out*

| Case | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| X points | 200 | 400 | 400 | 800 | 800 | 1600 | 1600 |
| Y points | 200 | 200 | 400 | 400 | 800 | 800 | 1600 |

The following results show that with a fixed number of iterations the integration time on the GPU is much lower than the CPU one with a high number of grid points. For a lower number of iterations and grid points the difference between the two versions is much smaller, as shown in the graphs where the time ratio between the CPU and GPU clock time is reported as a function of grid point ratio with the reference case ( 200x200 grid points ) .

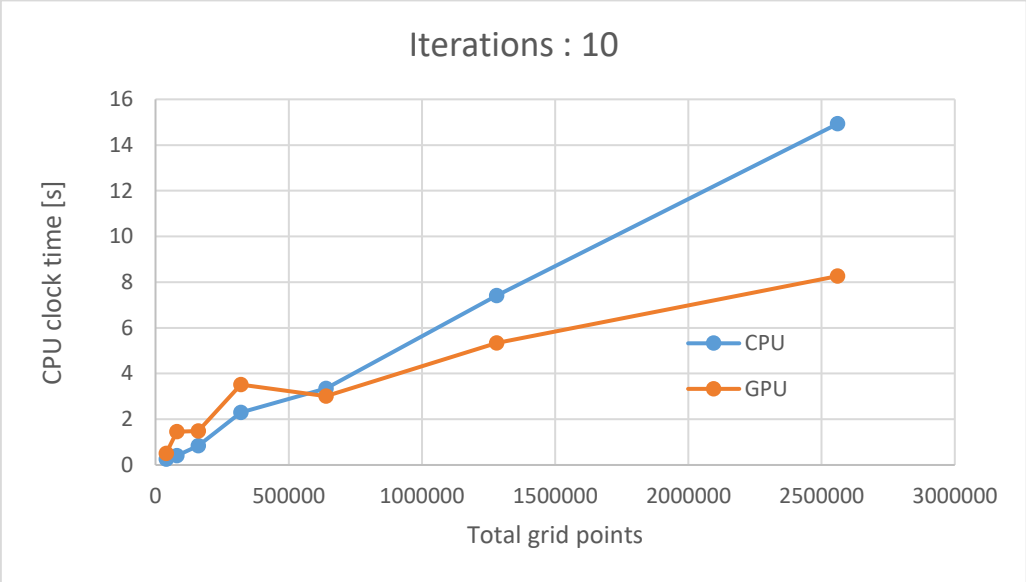



*Figure 6-1 CPU and GPU time performance at fixed 10 iterations*

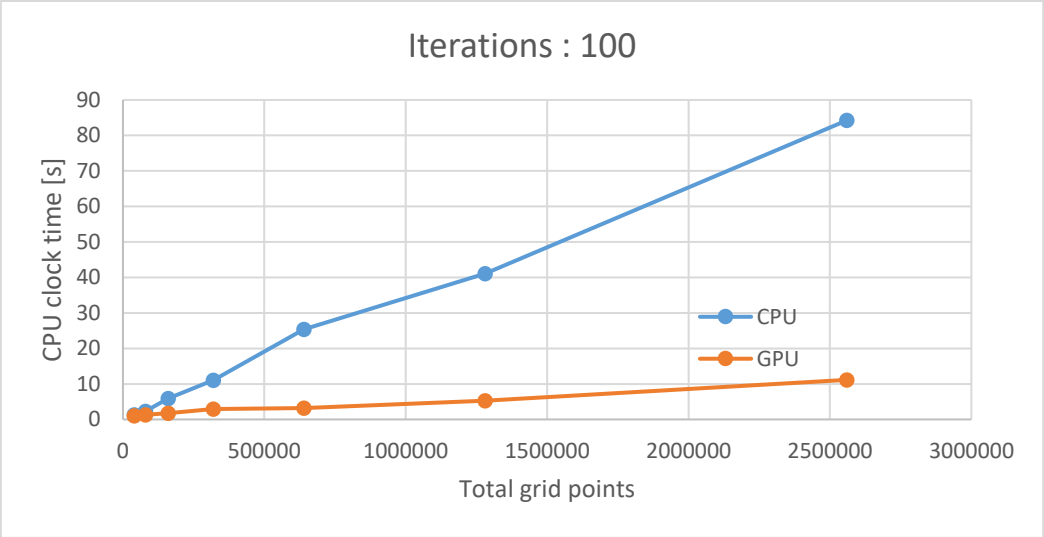



*Figure 6-2 CPU and GPU time performance at fixed 100 iterations*

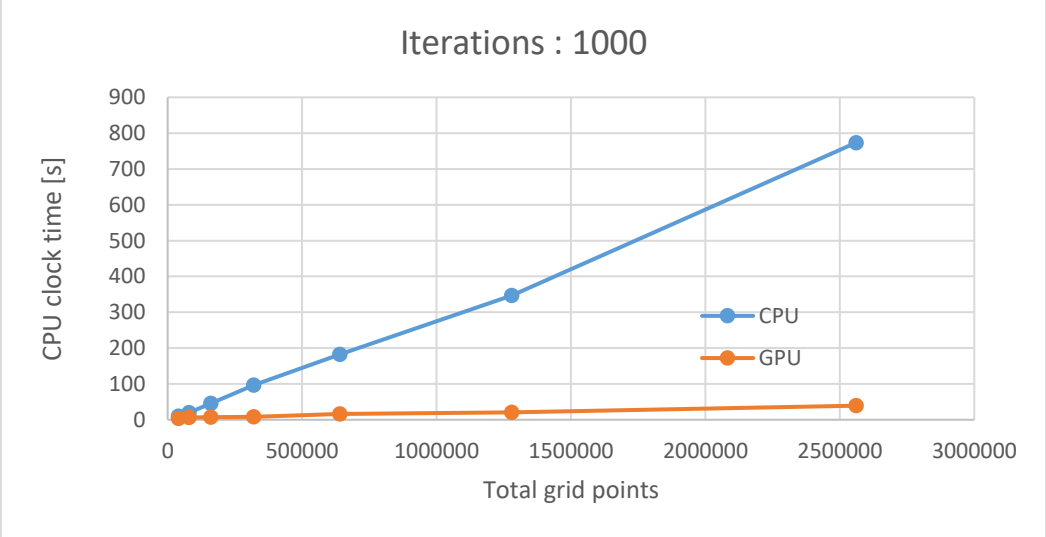



*Figure 6-3 CPU and GPU time performance at fixed 1000 iterations*

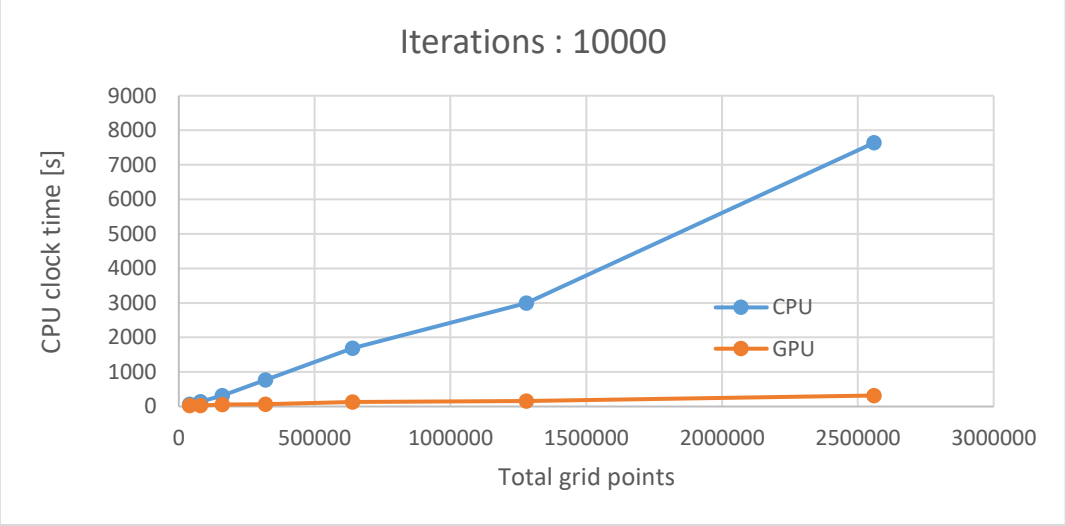



*Figure 6-4 CPU and GPU time performance at fixed 10000 iterations*

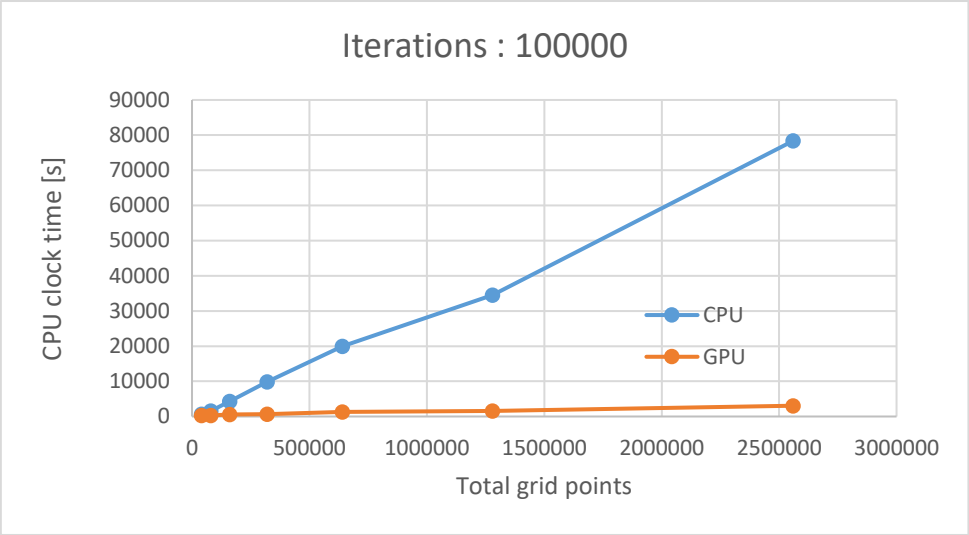



*Figure 6-5 CPU and GPU time performance at fixed 100000 iterations*

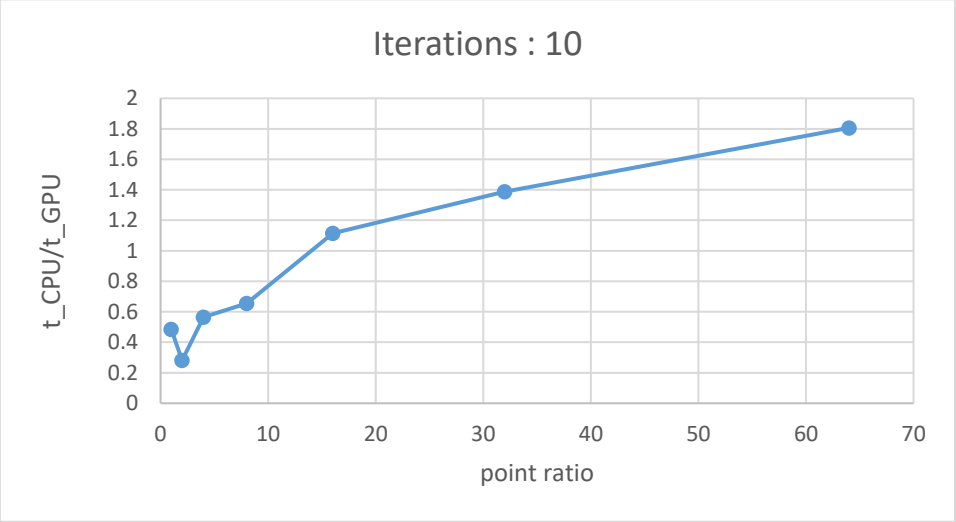



*Figure 6-6 Time ratio between CPU and GPU job clock time for a fixed number of 10 iterations*

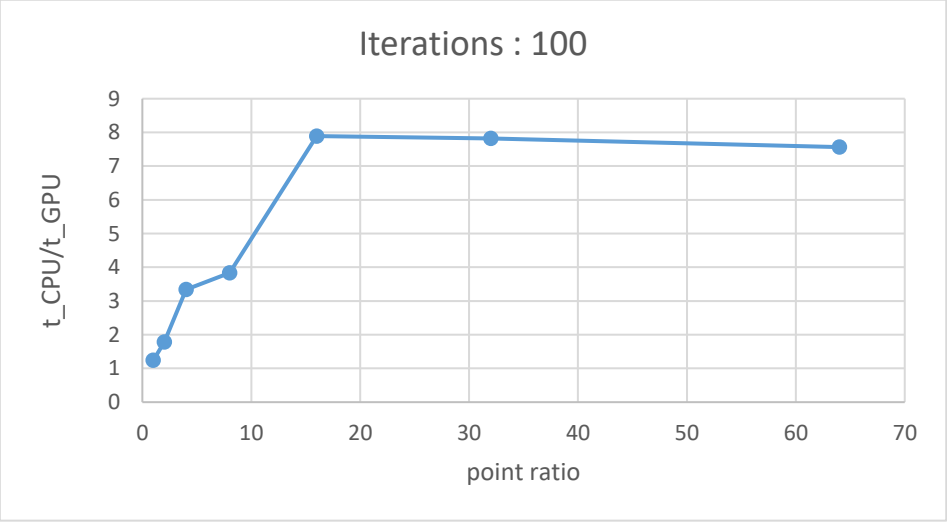



*Figure 6-7 Time ratio between CPU and GPU job clock time for a fixed number of 100 iterations*

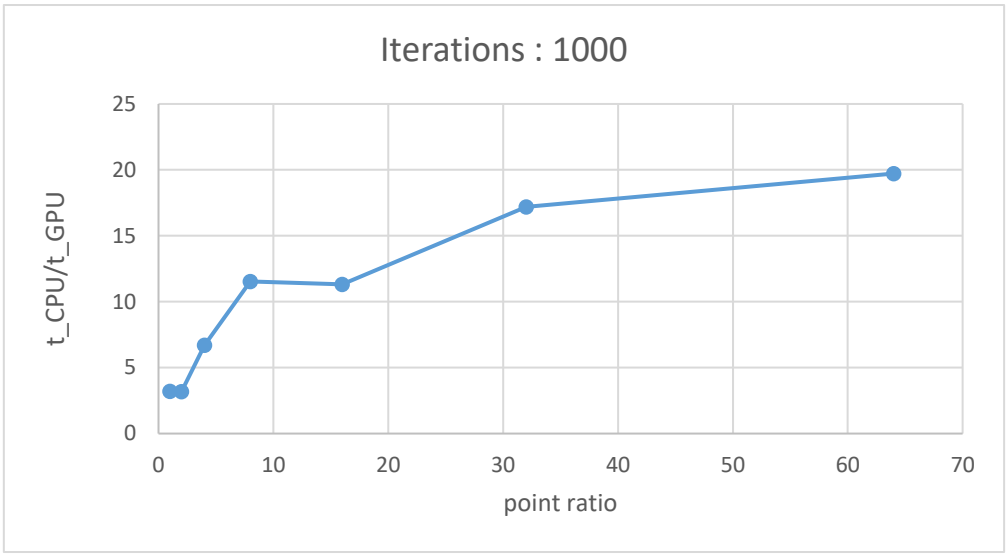



*Figure 6-8 Time ratio between CPU and GPU job clock time for a fixed number of 1000 iterations*

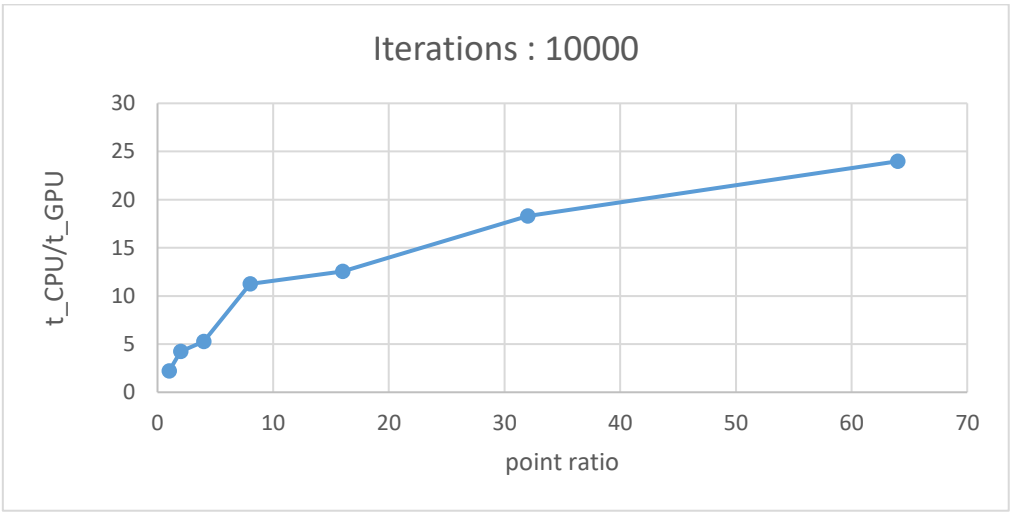



*Figure 6-9 Time ratio between CPU and GPU job clock time for a fixed number of 10000 iterations*

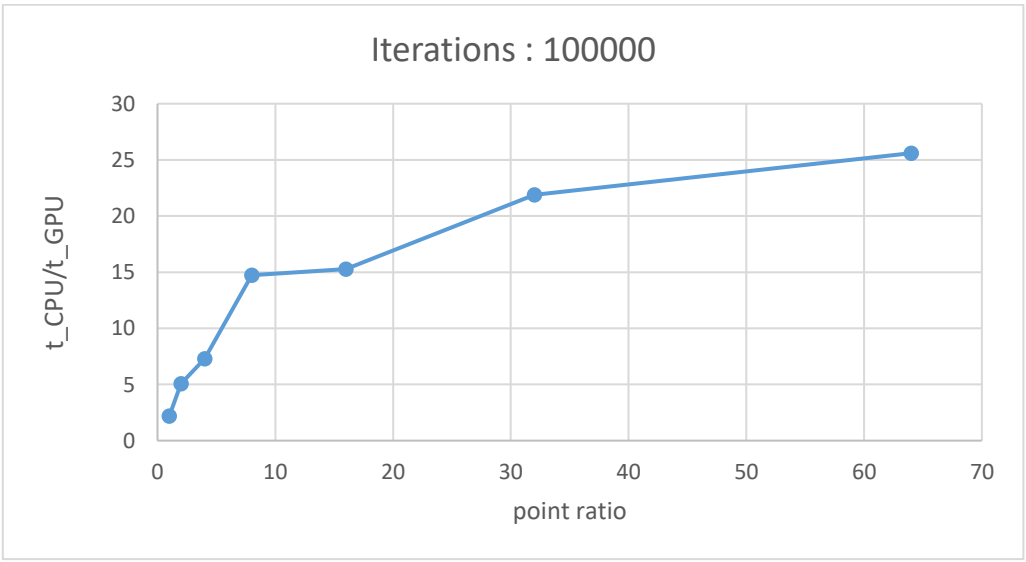



*Figure 6-10 Time ratio between CPU and GPU job clock time for a fixed number of 100000 iterations*

Through the call to the *cpu_time* subroutine in the main program , it is possible to estimate the time needed to write a single output file with the CPU for different grid sizes , the following graph shows the results :

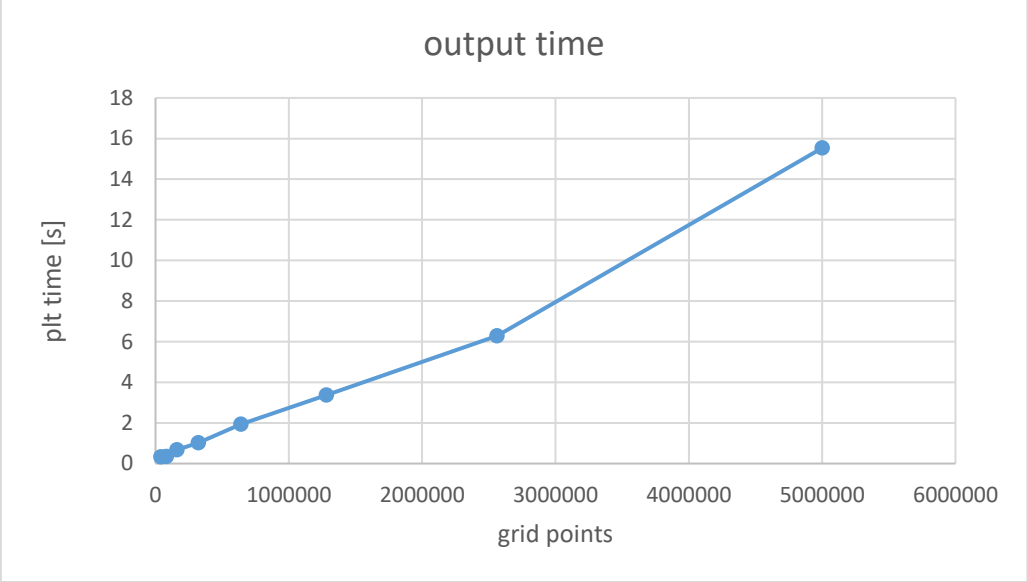



*Figure 6-11 Time needed to write a single output file for different grid sizes*

At fixed number of iterations , the GPU version of the code requires more memory per grid point than the CPU version , however the memory usage decreases exponentially by increasing the size of the grid and converges to the CPU value.

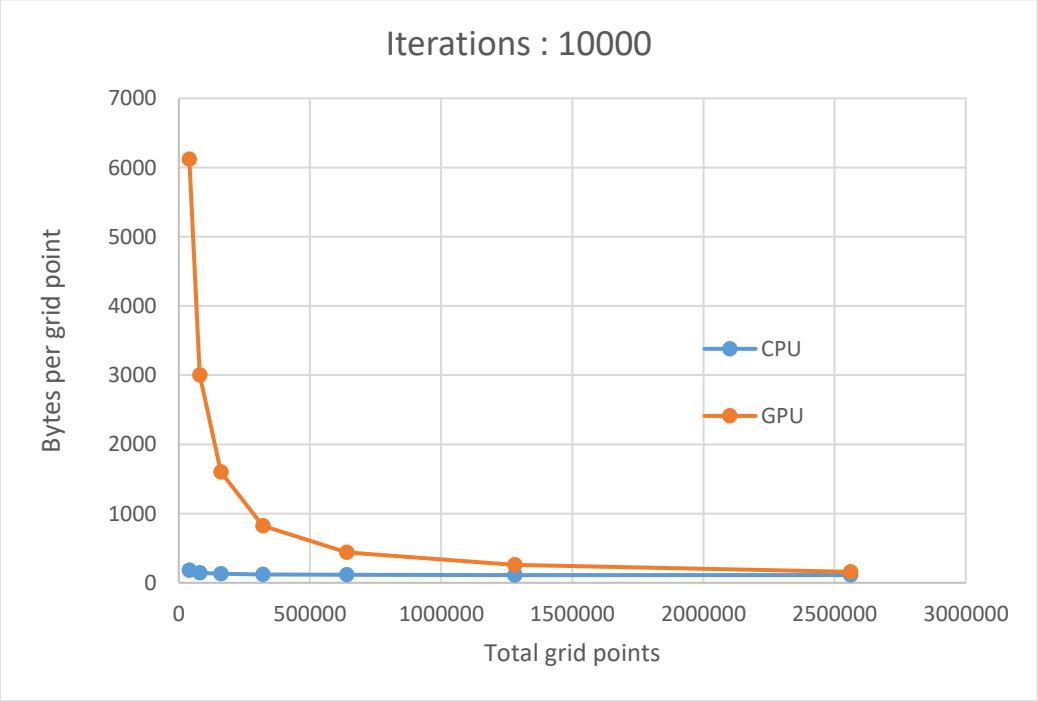



*Figure 6-12 CPU and GPU memory usage at fixed 10000 iterations*

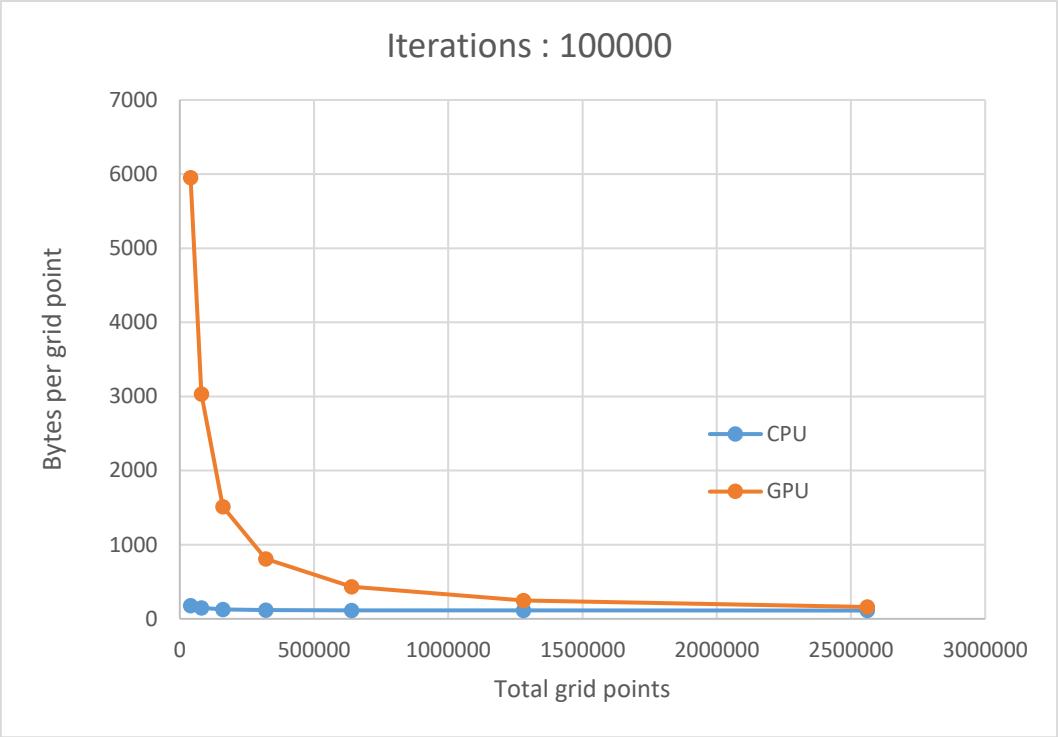



*Figure 6-13 CPU and GPU memory usage at fixed 100000 iterations*

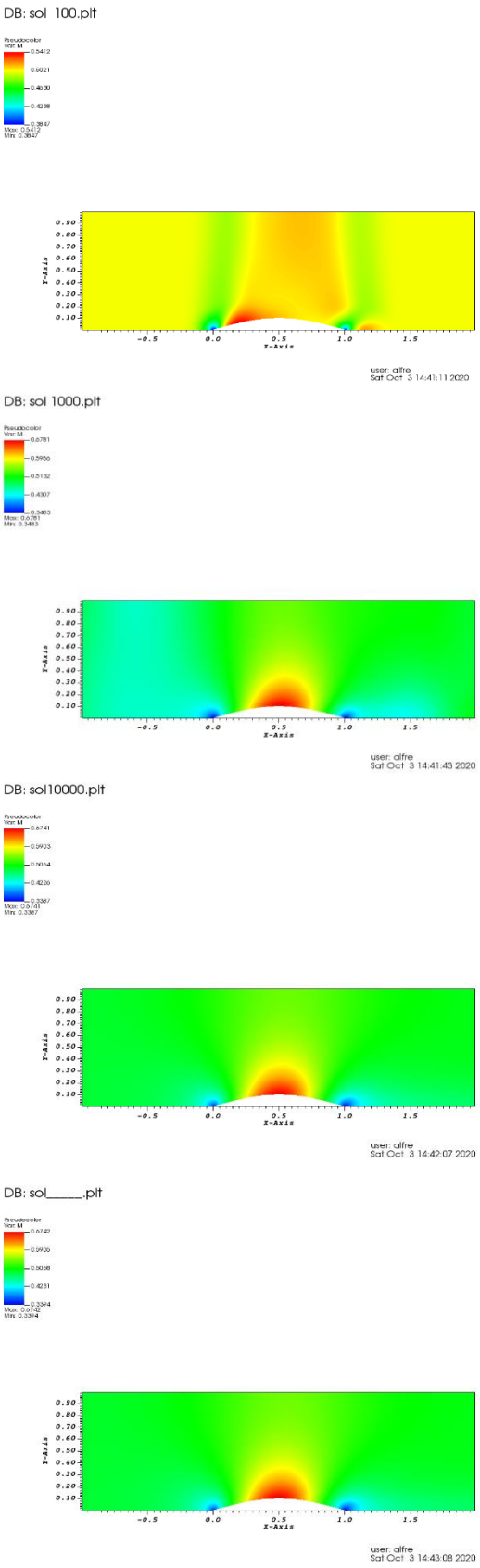



*Figure 6-14 CPU .plt files at different number of iterations . Pseudocolor of Mach number . Grid size:200x200*

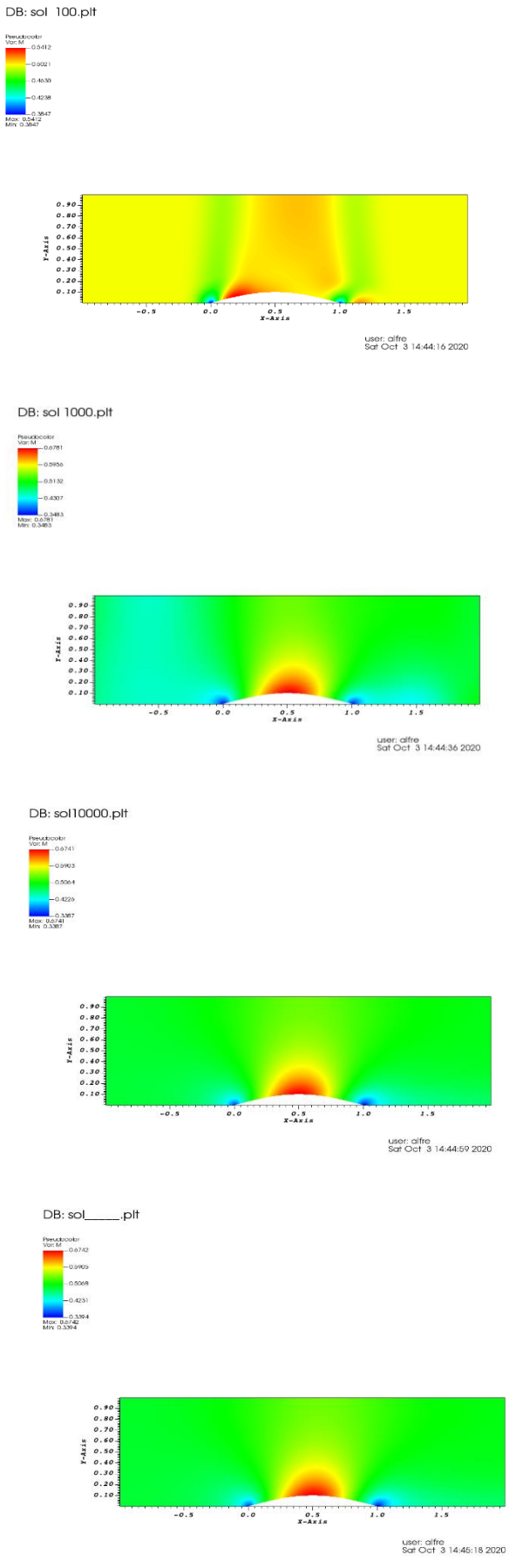



*Figure 6-15 GPU .plt files at different number of iterations. Pseudocolor of Mach number. Grid size: 200x200*

We carried out a simulation with a high number of grid points , that is 5000x1000 , using only the GPU version of the code . Total iterations are 100000 ; Job Wall clock time is: 00:57:18 ; Bytes per node are 111 . Pseudocolor of Mach number at the last iteration is reported.

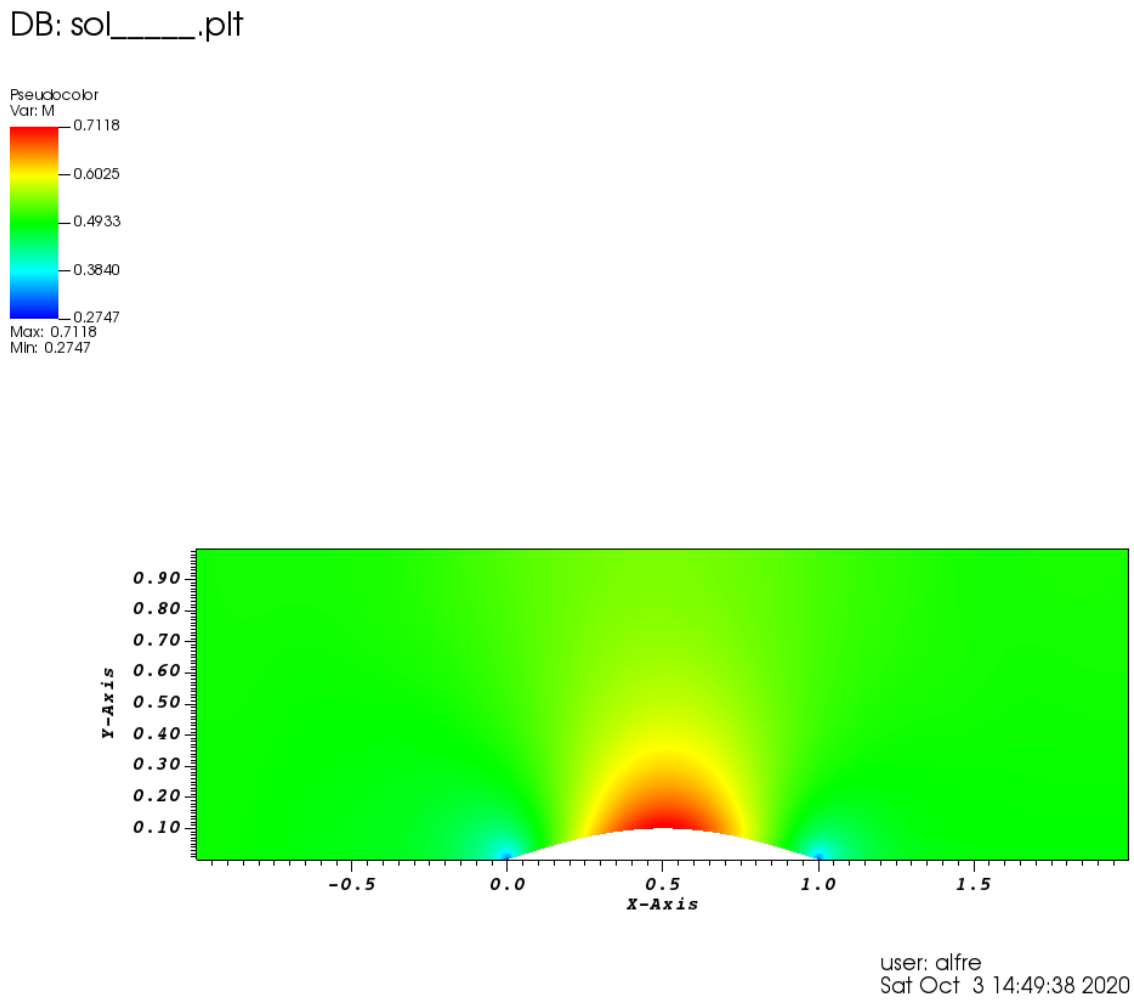



*Figure 6-16 Subsonic flow field with a grid size of 5 000 000 points*

It can be noticed that the numerical solution becomes more symmetric by increasing the number of grid points and tend to converge to the real solution of the case under investigation , this happens because the numerical viscosity needed to guarantee the CFL stability gets smaller with a finer mesh and the discrete solution converges to the real one as the truncation error reduces.

Therefore , GPUs enabled CFD solvers show a great potential to scale the grid and calculate more accurate solutions of the flow field , as the integration time with a really fine mesh made of millions of points is still reasonable to wait compared to the time needed with CPU solvers with the same grid settings .

The CFD code can solve even supersonic flow fields , to make an example a simulation with 1600x1600 grid points was carried out using one GPU and setting an inlet Mach number of 1.5 . Total

iterations are 100000 ; Job Wall clock time is 00:51:54 ; Bytes per node are 158. Pseudocolor of Mach number at the last iteration is reported.

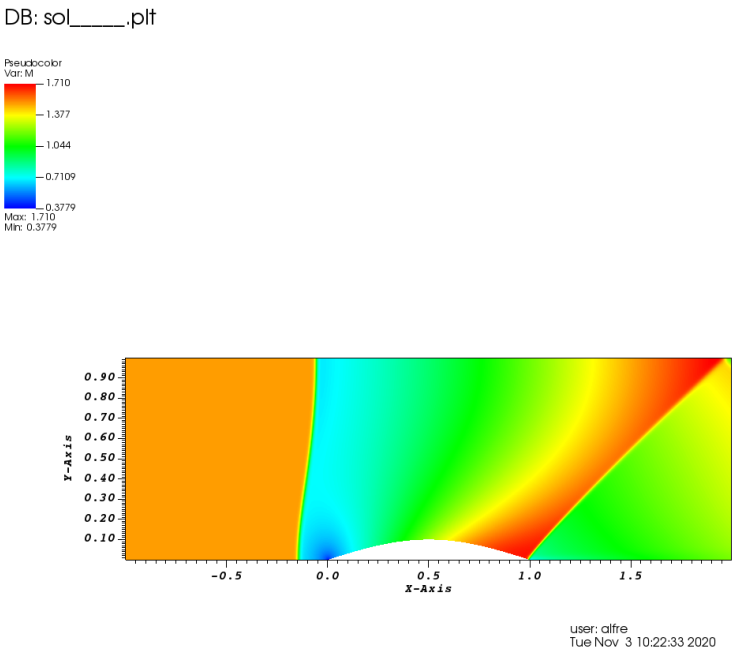



*Figure 6-17 Supersonic flow field with a grid size of 2 560 000 points*

From the previous picture it is well visible the detached normal shock close to the leading edge of the lower sine profile , afterwards the subsonic flow accelerates reaching again supersonic values until a second oblique shock wave makes it horizontal at the outlet .

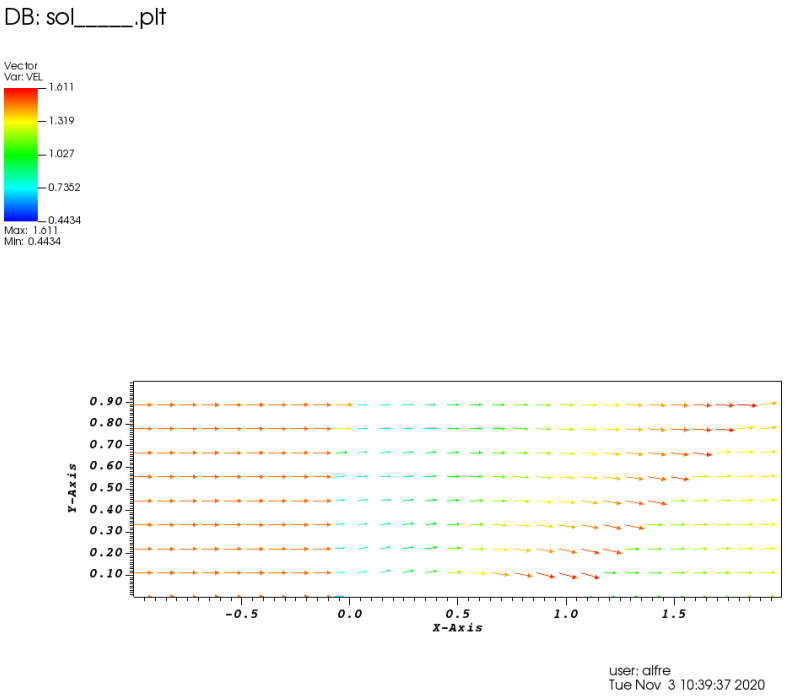



*Figure 6-18 Supersonic vector field around the selected sinusoidal geometry.*

# Chapter 7

# Conclusions

The main purpose of this master thesis was reporting the benefits of using GPU enabled CFD solvers to carry out simulations on mesh grids with tens of millions of points and to investigate the physics of compressible flows , in particular the complex interaction between an oblique shock wave and a compressible turbulent boundary layer (SBLI) .

The modern GPU architecture and the usage of cloud computing in HPC@POLITO Legion Cluster made possible great improvements in time performance for jobs requiring huge amount of computational resources , especially when four GPUs in the same cluster node were involved .

In fact , it is impossible to complete a Direct Numerical Simulation of a SBLI on a grid of more than 28 000 000 points using a single CPU or GPU on a personal computer , because the time needed for the job would be incredibly long compared to the cloud-based computation , in particular with the high number of iterations that are necessary to cut off the transitory and to collect statistics at the convergent steady state .

Moreover , as we do not apply any simplified model of turbulence in the set of Navier-Stokes equations solved by the GPU program STREAmS , we can think about capturing different scales of turbulent eddies by simply refining the mesh as much as we can , to be able to see even the smaller coherent structures in the turbulent fluctuations . This was nearly impossible in CPU-based CFD solvers , where quite often models like the Reynolds Averaged Navier-Stokes equations are necessary to compute the mean properties of turbulence in really complex cases , even though we lose the ability to see eddies with various length scales associated.

Finally , the CUDA Fortran version of the 2D CFD solver of Euler equations has great potential to complete simulations on inviscid compressible flows in a really short time . Even using a single GPU of the cluster , we can tend to converge to the real physical solution of the problem by rising with millions of points the mesh grid , to minimize the truncation error due to the spatial and time discretization of the PDE set and to reduce the numerical viscosity which is needed for CFL stability.

To sum up , if we want to calculate more accurate solutions of the flow field , waiting reasonable time to carry out the integration of equations describing the physics of our problem , we need to move to GPU enabled solvers and to cloud-based computing resources . In this way, it is possible to start the CFD analysis even from the preliminary phases of an aerospace project , to better drive the design of components and to predict performances without leaning to expensive experimental sessions .

# Bibliography


[1] M. Bernardini, D. Modesti, F. Salvadore and S. Pirozzoli, "STREAmS: a high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows," *physics.comp-ph,* 2020.

[2] NVIDIA, "CUDA C , https://docs.nvidia.com/cuda/cuda-c-programming-guide , accessed 2020-10-04," [Online].

[3] NVIDIA, "CUDA FORTRAN, https://developer.nvidia.com/cuda-fortran, accessed 2020-10-04," [Online].

[4] NVIDIA, "CUDA , http://developer.nvidia.com/cuda-zone , accessed 2020-10-04," [Online].

[5] "https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf," [Online]. [Accessed 14 10 2020].

[6] "https://www.hpc.polito.it/docs/guide-slurm-it.pdf," [Online]. [Accessed Aug 2020].

[7] OpenMPI, "https://www.open-mpi.org/video/general," [Online]. [Accessed 5 10 2020].

[8] A. Kempf, S. Wysocki and M. Pettit, "An efficient, parallel low storage implementation of Klein's turbulence generator for LES and DNS," *Computers and Fluids,* vol. 60, pp. 58-60, 2012.

[9] D. S.-K. Ting, Basics of Engineering Turbulence, Elsevier Inc., 2016.

[10] M.Klein, A. Sadiki and J.Janicka, "A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations," *J. Comput. Phys,* vol. 186 (2), pp. 652-665, 2003.

[11] T. Lund, X. Wu and D. Squires, "Generation of turbulent inflow data for spatially-developing boundary layer simulations," *J. Comp. Phys.,* vol. 140, pp. 233-258, 1998.

[12] Z.-T. Xie and I. Castro, "Efficient generation of inflow conditions for large eddy simulation of street-scale flows," *Flow, Turbulence and Combustion,* vol. 81 (3), pp. 449-470, 2008.

[13] S. Pirozzoli and M. Bernardini, "Turbulence in supersonic boundary layers at moderate Reynolds number," *J. Fluid Mech,* no. 688, pp. 120-168, 2011.

[14] "https://visit-sphinx-github-user-manual," [Online]. [Accessed 13 10 2020].

[15] K. M.Bercin, Z.-T. Xie and S. R. Turnock, "Exploration of digital-filter and forward-stepwise synthetic turbulence generators and an improvement for their skewness-kurtosis,," *Computers and Fluids,* vol. 172, pp. 443-466, 2018.

[16] K. Iwamoto, "Database for fully developed channel flow, THTLAB Internal Report (ILR-0201), Dept. Mech. Eng., Univ. Tokyo. DNS database (CH12_PG.WL7)," 2002. [Online]. Available: http://www.thtlab.t.u-tokyo.ac.jp/.