INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING, *SERIES B*
Volume 4, Number 4, Pages 394–412



# GPU COMPUTING FOR MESHFREE PARTICLE METHOD

M. PANCHATCHARAM[⋆,⩘], S. SUNDAR[⋆], V. VETRIVEL[⋆], A. KLAR[⩘], AND S. TIWARI[⩘]

**Abstract.** Graphics Processing Units (GPUs), originally developed for computer games, now provide computational power for scientific applications. A study on the comparison of computational speed-up and efficiency of a GPU with a CPU for the Finite Pointset Method (FPM), which is a numerical tool in Computational Fluid Dynamics (CFD) is presented. As FPM is based on the point cloud, it is so expensive when the number of particles are in millions. We have demonstrated the application of the FPM using a single-GPU (Nvidia Tesla M2050) and Intel CPU (Dual Xeon). Importance of the GPU is realized by the FPM since GPU yields a computational speed-up of 70× for the Poisson equation with various boundary conditions.

**Key words.** Finite Pointset Method(FPM), CUDA, GPU, Bi-CGSTAB.

## 1. Introduction

Nowadays, computational methods and related hardware are really inseparable. The hardware architecture progress leads the numerical methods that can be used with a reasonable computational cost. To increase the computational ability of CPU, a large and expensive cache is integrated and a many-core design has been employed [10]. The small scale problems of CFD can be solved on a PC of multi-core CPU and shared-memory parallel programming. For large scale problem, a PC with few cores cannot offer enough computational capability, and a cluster with many CPUs (or cores) is needed. Nevertheless, the memory bottleneck which appears in the form of bandwidth limitation and fetching latency, has restricted the performance of the many-core systems. In the meantime, Graphics Processing Units (GPUs), having recently turned into general-purpose programmable units, can provide a different solution to the memory access problem.

Initially driven by the gamer market, GPUs recently became suitable for high performance computing applications. A GPU is a multi-threaded, many core processors which was originally developed for graphics processing. However, in recent years, the so-called General Purpose GPU (GPGPU) has been used widely for computation in different fields because it has a high computing ability and a relatively low cost. The main advantage of GPUs is their ability to perform significantly more floating point operations (FLOPS) per unit time than a CPU. One of the market leaders, NVIDIA, developed a parallel computation architecture called CUDA (Compute Unified Device Architecture) [7]. CUDA is an extension of C language which allows us to program the NVIDIA GPUs in an easy way. Other than NVIDIA, there are several ways to realize the GPGPU computing: Computer Graphics with OpenGL [6], OpenCL [12], Stream (ATI Corporation) [1]. But according to Du et al. [5], at this moment CUDA is more efficient on the GPU than OpenCL. Hence, in our study, the NVIDIA GPU with CUDA platform is chosen because CUDA is used in a variety of different fields of scientific computation such as graphics, biology, linear algebra, PDE solvers and computational physics. Especially in

Received by the editors October 31, 2012 and, in revised form, August 31, 2013.
2000 *Mathematics Subject Classification.* 65Y05, 65Y20, 35Q30, 76D05.
This research was supported by DAAD.

the applications of computational mathematics and computational physics, it obtained surprising speed-up. Harada et al. [8] obtained 28× speed-up on GPU for Smoothed Particle Hydrodynamics (SPH) solver using 262,144 particles. Rossinelli et al. [18] present a GPU-accelerated solver for simulations of bluff body flows in 2D using a re-meshed vortex particle method and 30× speed-up was obtained. Knibbe et al. [9] implemented a Krylov solver on GPU with a 30× speed-up. Luo et al. [11] obtained 71× speed-up on GPU for 1D shock tube problem using CESE (Conservation Element and Solution Element) method.

The Finite Pointset method (FPM) [20] is a mesh free method for solving fluid dynamic equations. It is a Lagrangian particle method, in which computational domain is filled with a finite number of particles. Particles move with fluid velocities and they carry the fluid quantities, like the density, the velocity, the pressure and so on. Similarly, boundaries are also approximated by a finite number of boundary particles and boundary conditions are prescribed for them. Since it is a Lagrangian particle method, the distribution of particles can be arbitrary. The FPM has more advantages on fluid flow problems with moving boundary in time, where the remeshing is not required. The disadvantage of the FPM is the constant particle management at each time step of a flow problem, otherwise, it may lead to inaccuracy or instability. Drum et al. [4], Tiwari and Kuhnert [21] have applied the FPM for various fluid flow problems. Our main goal is to solve the incompressible Navier-Stokes equations by the FPM in GPU, however, we restrict ourselves to solve the Poisson equation, since the pressure Poisson equation is the essential part of solving the incompressible Navier-Stokes equations. Therefore, in this paper, we focus on the iterative solvers for the Poisson equation in two dimensions on GPU using CUDA.

The particles generated in CPU and transferred to the GPU for neighbor search computations. In this paper, we implemented the non-recursive merge-sort algorithm to sort the neighbors.

Finally, the computation in CPU produces a system of equations, where the matrix is sparse and has no blocks. This system is solved in GPU using the Bi-CGSTAB solver from *cusp* library.

The rest of the paper is organized as follows: Section 2 is the overview of the Finite Pointset method. In section 3, key facts of CUDA are provided. The specific aspects of the GPU implementation of Krylov methods such as Bi-CGSTAB using *cusp* library and non-recursive merge-sort algorithm are considered in detail in Section 4. Section 5 will represent the numerical results for some benchmark problems and analyze the computational performance. Finally conclusions are drawn in Section 6.

## 2. The Finite Pointset Method

In this paper, we restrict ourselves for solving the Poisson equation in $\bar{\Omega}(\subset R^2)$. Let us consider the following boundary value problem

$$\Delta u = f \quad \text{in} \quad \Omega \tag{1}$$

$$a_0 u + b_0 \frac{\partial u}{\partial \vec{n}} = g \quad \text{on} \quad \partial\Omega, \tag{2}$$

where $a_0, b_0, f$ and $g$ are given functions and $\vec{n}$ is the unit normal on boundary pointing inside the domain. Here, $(a_0, b_0) = (1, 0)$ denotes the Dirichlet boundary

value problem, whereas $(a_0, b_0) = (0, 1)$ denotes the Neumann boundary value problem. In the case of a mixed boundary value problem, the boundary conditions on $\partial\Omega = \partial\Omega_1 \cup \partial\Omega_2$ are given by

$$u = g_1 \quad \text{on} \quad \partial\Omega_1 \quad \text{and} \quad \frac{\partial u}{\partial \vec{n}} = g_2 \quad \text{on} \quad \partial\Omega_2, \tag{3}$$

where $g_1$ and $g_2$ are given functions.

We approximate $\bar{\Omega} = \Omega \cup \partial\Omega$ by a finite number of points with position $\vec{x_i} = (x_i, y_i),\ i = 1, 2, ..., N$. We indicate the interior as well as boundary particles by assigning flags. Let $u$ be a function in $C^2(\Omega)$ and $u_i = u(\vec{x_i})$ for $i = 1, 2, ..., N$. Let us approximate the function $u(\vec{x})$ at $\vec{x} = (x, y)$ in terms of the values of a set of neighboring points. In order to limit the number of neighbor points, we associate a weight function $w = w(\vec{x_i} - \vec{x}; h)$ with a small compact support, where $h$ denotes the size of the support. We consider the Gaussian weight function of the following form:

$$w(\vec{x_i} - \vec{x}; h) = \begin{cases} exp(-\alpha \frac{\left\|\vec{x_i} - \vec{x}\right\|^2}{h^2}) & \text{if } \frac{\left\|\vec{x_i} - \vec{x}\right\|}{h} \leq 1 \\ 0 & \text{else} \end{cases} \tag{4}$$

where $\alpha$ is a positive constant. We considered $\alpha = 6.25$ in this paper. The size of $h$ defines a set of neighboring particles around $\vec{x}$. Let $P(\vec{x}, h) = \{\vec{x_i} : i = 1, 2, ..., m\}$ be the set of neighboring points in $\vec{x}$ in a ball of radius $h$. For consistency reasons, there should be at least 6 particles and they should be neither on the same line nor on a circle. Otherwise matrix $M$ in eq. (8) becomes singular.

Consider Taylor's expansion of $u(x_i, y_i)$ around $(x, y)$

$$\begin{aligned} u_i = u(x_i, y_i) =& u(x, y) + u_x(x, y)dx_i + u_y(x, y)dy_i \\ &+ u_{xx}(x, y)\frac{{dx_i}^2}{2} + u_{xy}(x, y)dx_i dy_i + u_{yy}(x, y)\frac{{dy_i}^2}{2} + e_i, \end{aligned} \tag{5}$$

where $e_i$ is the error in the Taylor's series expansion at the point $\vec{x_i}$ and $dx_i = x_i - x, dy_i = y_i - y$. The unknowns $u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}$ at $(x, y)$ in (5) are computed by minimizing the error $e_i$ for $i = 1, 2, ..., m$. Here, we use the weighted least squares method [3]. Our task is to obtain the value of $u$, not its derivatives. Therefore, we add (1) as a constraint to (5) for the interior particles and (1) and (2) as constraints to (5) for the boundary particles. The system of equations (5) along with the constraints can be rewritten as

$$\vec{e} = M\vec{a} - \vec{b}, \tag{6}$$

$$\text{where } M = \begin{pmatrix} 1 & dx_1 & dy_1 & \frac{dx_1^2}{2} & dx_1 dy_1 & \frac{dy_1^2}{2} \\ 1 & dx_2 & dy_2 & \frac{dx_2^2}{2} & dx_2 dy_2 & \frac{dy_2^2}{2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & dx_m & dy_m & \frac{dx_m^2}{2} & dx_m dy_m & \frac{dy_m^2}{2} \\ 0 & 0 & 0 & 1 & 0 & 1 \\ a_0 & b_0 n_x & b_0 n_y & 0 & 0 & 0 \end{pmatrix},$$

$\vec{a} = [u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}]^T$, $\vec{b} = [u_1, u_2, ..., u_m, f, g]^T$ and $\vec{e} = [-e_1, -e_2, ..., -e_m, -e_{m+1}, -e_{m+2}]^T$, $n_x$ and $n_y$ are the $x$ and $y$ components of the unit normal vector $\vec{n}$ on the boundary at $\vec{x}$. Therefore, we have $m + 2$ equations for 6 unknowns.

For $m > 6$, this system is over-determined with respect to the 6 unknowns. Let's compute $u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}$ at $\vec{x}$ minimizing the quadratic form

$$J = \sum_{i=1}^{m} w_i {e_i}^2 + (u_{xx} + u_{yy} - f)^2 + (a_0 u + b_0 \frac{\partial u}{\partial \vec{n}} - g)^2 \tag{7}$$

The above equations can be expressed in the form

$$J = (M\vec{a} - \vec{b})^T W (M\vec{a} - \vec{b})$$

where

$$W = \begin{pmatrix} w_1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & w_2 & \cdots & 0 & & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & w_m & 0 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

The minimization of $J$ formally yields

$$\vec{a} = (M^T W M)^{-1} (M^T W) \vec{b}. \tag{8}$$

The matrices $M$ and $W$ and the vector $\vec{b}$ given above are for boundary particles. For interior particles the last row of the matrices $M$ and $W$ and the last component of the vector $\vec{b}$ are omitted. After performing the multiplications and computing inverse, (8) can be written as follows:

$$\begin{pmatrix} u \\ u_x \\ u_y \\ u_{xx} \\ u_{xy} \\ u_{yy} \end{pmatrix} = \begin{pmatrix} \zeta_{11} & \zeta_{12} & \zeta_{13} & \zeta_{14} & \zeta_{15} & \zeta_{16} \\ \zeta_{21} & \zeta_{22} & \zeta_{23} & \zeta_{24} & \zeta_{25} & \zeta_{26} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \zeta_{61} & \zeta_{62} & \zeta_{63} & \zeta_{64} & \zeta_{65} & \zeta_{66} \end{pmatrix} \begin{pmatrix} \sum_{i=1}^{m} w_i u_i + a_0 g \\ \sum_{i=1}^{m} w_i dx_i u_i + b_0 n_x g \\ \sum_{i=1}^{m} w_i dy_i u_i + b_0 n_y g \\ \sum_{i=1}^{m} w_i \frac{dx_i^2}{2} u_i + f \\ \sum_{i=1}^{m} w_i dx_i dy_i u_i \\ \sum_{i=1}^{m} w_i \frac{dy_i^2}{2} u_i + f \end{pmatrix} \tag{9}$$

Our goal is to find the value of $u$. Equating the first component of the vector $\vec{a}$ and the value of right hand side in (8), gives

$$u = \sum_{i=1}^{m} \beta_i u_i + r \tag{10}$$

where $r$ has no $u_i$ components. Note that we have calculated the value of $u$ locally for a single particle in the domain $\Omega$. For each particle, one has to invert a $6 \times 6$ matrix. If we do the above process for all particles in the domain, we get a sparse linear system of equations. Hence, we have a system

$$AU = R, \tag{11}$$

where

$$A = \begin{pmatrix} * & * & \cdots & * & 0 & \cdots & * & 0 \\ * & \cdots & * & * & 0 & \cdots & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & * & \cdots & 0 & * & * \end{pmatrix}, U = (u_1, u_2, ...., u_N)^T, R = (r_1, r_2, ..., r_N)^T.$$

In the matrix $A$, we have non-zero entries denoted by $'*'$ corresponding to the neighbor indices. Since the matrix $A$ is sparse, we stored them in CSR format [14]. Properties of the matrix $A$, such as positive definiteness, condition number and so on, are under open research topic. In order to solve the linear system (11), we employed different iterative linear solvers such as Gauss-Siedel, SOR, Bi-CGSTAB, etc. However, we found Bi-CGSTAB is relatively faster compare to other iterative solvers. Hence, in this paper, we discuss Bi-CGSTAB algorithm alone.

We store $U$ in $U^{old}$ and use them as the input for the next iterations. After each iterations, we have two values, $U$ and $U^{old}$. The iteration is stopped if the error satisfies

$$\frac{\sum_{i=1}^{N} |u_i - u_i^{old}|}{\sum_{i=1}^{N} |u_i|} < \epsilon_1, \tag{12}$$

where $\epsilon_1$ depends on the problem and $h$. In this paper, we considered $h$ as constant and it is equal to 3 times the initial particle spacing. If we increase the distance between points, $h$ becomes larger and hence the rate of convergence increases.

In the next section, we describe CUDA programming and its application on FPM.

## 3. CUDA

Before we implement the FPM with the help of GPU, we introduce some basic facts about GPU and CUDA [7].

We begin with Fermi based GPU architecture[17], which is given in Fig. 1. Initially, Fermi GPUs were implemented with 3 billion transistors and they had

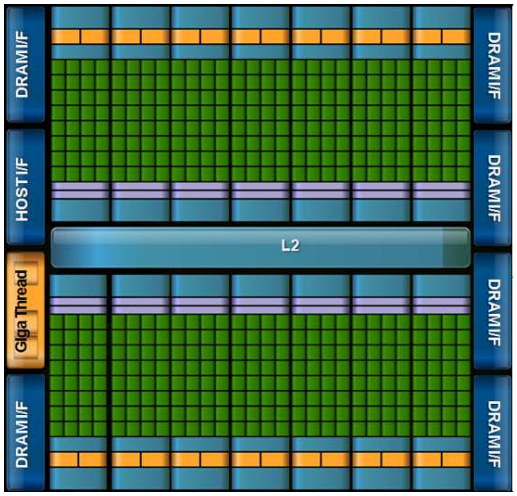



FIGURE 1. Fermi architecture. Source: NVIDIA

512 CUDA cores. These 512 CUDA cores are organized in 16 SMs (Streaming Multiprocessors) of 32 cores each. In this paper, we have used NVIDIA Tesla

M2050, which has 14 SMs of 32 cores each. In Fig. 1, SMs are denoted in blue color rectangular strip, where as green squares residing inside of each SM are called as cores or CUDA cores. The most fundamental execution resources on the chip are cores. The darker blue block which lies on the sides of the diagram are alled as 64-bit memory interfaces. All 14 SMs are located around common L2 cache where as registe s and L1 cache are given in blue and orange portions respectively at the bottom and top.

Each Fermi streaming multiprocessor has four separate special function units (SFU) which execute mathematical functions such as sine, cosine, square root, etc. Also, a 64 KB shared memory/L1 cache resides in each SM. From Fig. 2, one can

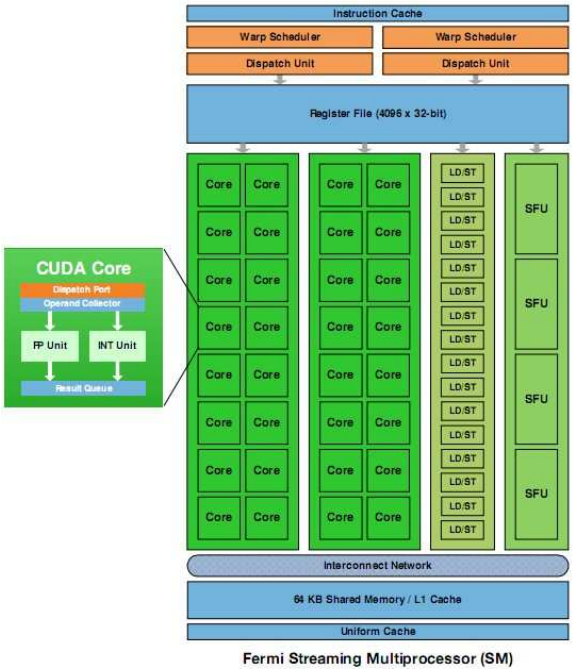



FIGURE 2. Fermi Streaming multiprocessor. Source: NVIDIA

observe that each CUDA core has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).

The basic elements of CUDA memory is given in Fig. 3. A *Thread* on GPU is a basic element of the data to be processed. A *Warp* in CUDA is a group of 32 threads, which is the minimum size of the data processed in SIMD (Single Instruction Multiple Data) fashion by a CUDA multiprocessor. *Blocks* Contain 64 to 512 threads and these blocks are put together in *Grids*, which is the top of the thread hierarchy. The advantage of grouping is that the number of blocks in a grid apply a kernel to a large quantity of threads in a single call. CUDA runtime executes the block sequentially if the hardware has few resources, whereas it executes in parallel if the hardware has a very large number of resources. CPU and GPU are referred as *host* and *device* respectively. In a grid, blocks are managed in x-, y- and z-directions. The dimensions of a grid are given by the variable gridDim, whose components are **gridDim.x, gridDim.y** and **gridDim.z**. If gridDim.z is set to 1, blocks in a grid is actually managed in a 2D way. The dimensions of a block are given by the variable blockDim, whose components are given as **blockDim.x,**

**blockDim.y** and **blockDim.z**. The block index is given by blockIdx **(blockIdx.x, blockIdx.y and blockIdx.z)**, so that threads are managed in a 3D way in a block and the thread index is given by threadIdx **(threadIdx.x, threadIdx.y, threadIdx.z)**. When calling a CUDA kernel, the number of blocks in a grid and threads in a block should be announced by using **dim3** declaration.

Threads creation, scheduling, and management are performed entirely in hardware. To manage the large population of threads efficiently, the GPU employs SIMT (Single Instruction Multiple Threads) architecture in which the threads of a block are executed in groups of 32 called warps. A warp executes a single instruction at a time across all its threads. The threads of a warp are free to follow their own execution path and all such execution divergence is handled automatically in hardware. The threads of a warp are free to use arbitrary addresses when accessing off-chip memory with load operations. Accessing scattered locations results in memory divergence. It requires the processor to perform one memory transaction per thread. If the locations, to be accessed, are close together, then the per-thread operations can be coalesced for better memory efficiency.

Each SM has a *shared memory* which is used for communications between threads within blocks. This memory area provides a way for threads in the same block to communicate. Since there is an important restriction that all the threads in a given block have to be executed by the same multiprocessor, two threads from different blocks cannot communicate during their execution.

The shared memory of a GPU has the life cycle of the block of threads, while the global memory has the life cycle of the executed program. A thread executes on the device has only access to the device's DRAM and on-chip memory through registers (Read-Write (RW)), local memory (RW), shared memory (RW), global memory (RW), constant memory (Read only (R)), and texture memory (R) as illustrated in Fig. 3. The global, constant, and texture memory spaces can be read from or written to by the host and are persistent across kernel calls by the same application. The hardware and memory details are explained in detail in [15, 16].

From the software point of view, CUDA is an extension of the C language. A CUDA program basically consists of CPU code and at least one kernel, i.e. a void returning function to be executed by the GPU. The keyword **__global__** indicates the kernel function which is called by the CPU and executed by the GPU, where as **__device__** indicates the function is called and executed only by the GPU. Finally, **__host__** keyword is for the function to be called and exectued by CPU only. Note that the functions **__global__** and **__device__** should not be recursive and they cannot have a variable number of arguments. A variable with the keyword **__shared__** indicates that it will be stored in the shared memory of SM.

As we explained earlier, the number of blocks in a grid and threads in a block should be announced by using **dim3** declaration, while calling CUDA kernel. The variable dim3 should be included as a parameter as follows:
$dim3\ gridDim(i, j, k)$;
$dim3\ blockDim(p, q, r)$;
$kernelfunction <<< gridDim, blockDim >>> (a, b, c)$;
where $i, j$ and $k$ are the number of blocks in $x, y$ and $z$ directions in grid. $p, q$ and $r$ are the number of threads in $x, y$ and $z$ directions in a block. $a, b$ and $c$ are the parameters of the kernel. The CUDA function call differ from C function call only by the part $<<< gridDim, blockDim >>>$. This kernel is executed on GPU and called from CPU. This kernel function should be declared with the

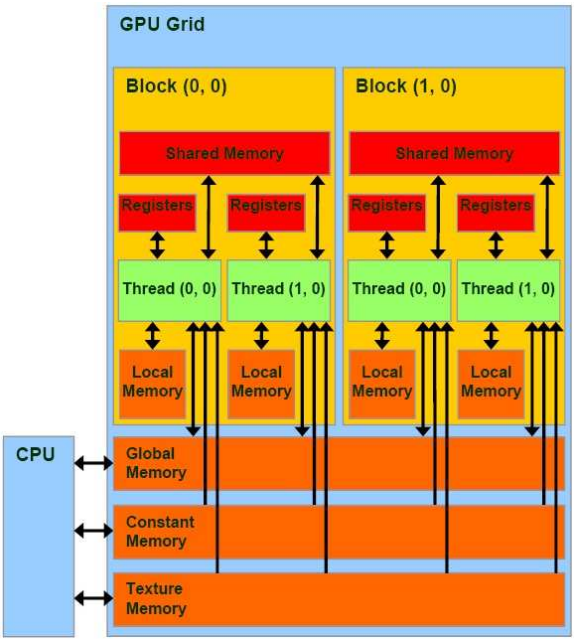



FIGURE 3. CUDA memory model

keyword **__global__**. The CUDA API essentially comprises functions for memory manipulation in VRAM: cudaMalloc to allocate memory, cudaFree to free it and cudaMemcpy to copy data between RAM and VRAM and vice-versa.

We will end this section by explaining how a CUDA program is compiled. Compiling is done in several levels. In the first level, the code dedicated to CPU is extracted from the file and passed to the standard compiler. In the next level, the code dedicated to the GPU is converted into an intermediate language PTX which is like an assembler. Finally, the last level translates this intermediate language into commands that are specific to the GPU and encapsulates them in binary form which is executable.

Next, let us discuss about the Bi-CGSTAB algorithm and CUDA programming for Bi-CGSTAB.

## 4. Bi-CGSTAB

Consider the system $AU = R$, which we generated in section 2. Let us look at the implementation of the Bi-CGSTAB on the GPU to find the solution of the system. The Bi-CGSTAB follows the algorithm described in [22]. It is a Krylov subspace method. The Bi-CGSTAB algorithm on the GPU has been already developed by NVIDIA [13]. The Bi-CGSTAB algorithm for a sparse matrix on the GPU is available in the *cusp* library [14]. Since our matrix $A$ is sparse, we have employed the *cusp* library to solve the system $AU = R$ without any preconditioners.

**4.1. Scalar Product and Vector Operations.** The scalar product of two vectors each with $N$ components is realized in three steps as explained in Algorithm 1. At first, each thread in a block accesses each component of both vectors, multiplies them point-wise and stores them in the shared memory. Once the computation is done in all threads (**__syncthreads()** ), we do the partial sum inside the block in the next step. At the final step, we use an atomic function called *atomicadd*() to

compute the final sum. Initially, CUDA atomic functions were designed for integers only, in the recent versions, we can use atomic functions for float variables also. Hence, we implemented atomic functions in our computations. In case of vector addition or subtraction, each thread computes the addition or subtraction of each component of the vector.

**Algorithm 1** Scalar Product and Reduction

```
__global__ void reduction(x, y, z)
__shared__ int temp[THREADSPERBLOCK]
i ← threadIdx.x + blockIdx.x * blockDim.x
if i < N then
    temp[threadIdx.x] ← x[i] * y[i]
__syncthreads()
end if
if 0 = threadIdx.x then
    sum ← 0
    for i = 0 to THREADSPERBLOCK do
        sum ← sum + temp[i]
    end for
    atomicAdd(z, sum)
end if
```

Apart from Bi-CGSTAB, we have implemented CUDA to sort the neighbor list which is explained in the next section.

**Algorithm 2** Merge-sort

```
__global__ mergesort(list1,list2,N)
__shared__ temp[]
x ← threadIdx.x
temp[x] ← list1[x]
__syncthreads()
k ← 1
while k < N do
    i ← 1
    while i + k ≤ N do
        u ← i + k
        if u > N then
            u ← N + 1
        end if
call mergesortgpu(temp,list2,i,i+k, u)
        i ← i + 2k
    end while
__syncthreads()
    list1[x] ← temp[x]
end while
```

**4.2. Neighbor Search and Merge-Sort.** The most important part of the meshfree method is searching neighbors of particles and sorting them. In the present study, we employed GPU to neighbor search using merge-sort algorithm. At first

we initialize the particles in the CPU. After initialization, we construct a voxel data structure which contains the computational domain. The voxels form a regular grid of squares in 2D with side length $h$. For each voxel, the indices of points are contained in it. Moreover, for each point it is known, in which voxel it is contained and which are the points inside the ball of radius h. Now, we establish three types of lists. The first list requires to loop over all particles and computing voxel [4]. This is of complexity $O(N)$. The second list can be obtained from the first list by sorting with respect to the voxel indices. It is of complexity $O(NlogN)$ and we have used the efficient sorting algorithm developed by Satish et al [19].

The merge sort procedure consists of three steps:

(1) Divide the input into p equal-sized tiles.
(2) Sort all p tiles in parallel with p thread blocks.
(3) Merge all p sorted tiles.

Since we assume the input sequence is a contiguous array of records, the division performed in Step (1) is trivial. For sorting individual data tiles in Step (2), we use an implementation of Batchers odd-even merge sort [2]. For sorting $t$ values on chip with a t-thread block, we have found the Batcher sorting networks to be substantially faster than either radix sort or quicksort. We use the odd-even merge sort, rather than the more common bitonic sort, because our experiments show that it is roughly 5-10% faster in practice. All the real work of merge sort occurs in the merging process of Step (3), which we accomplish with a pair-wise merge tree of log $p$ depth. At each level of the merge tree, we merge pairs of corresponding odd and even subsequences. This is obviously an inherently parallel process. However, the number of pairs to be merged decreases geometrically. This coarse-grained parallelism is insufficient to fully utilize massively parallel architectures. Our primary focus, therefore, is on designing a process for pairwise merging that will expose substantial fine grained parallelism.

Using existing CUDA examples, we created a host function which passes two input arrays, the first being list of numbers to be sorted and the second is the destination array. Both of these arrays were first created and initialized in host memory. Then using `cudaMalloc` and `cudaMemcpy`, they were shuttled onto the GPU's dedicated memory. This merge function is called inside the host function. The merge function is a **__device__** function which performs the actual merge-sort algorithm on the GPU. GPU algorithm for merge-sort is given in Algorithms 2 and 3.

The second list obtained from GPU gives direct access to the voxel, where this particle is contained in. Now all particles in the voxel and its neighboring voxels are tested for being inside the ball. The indices of the particles in these voxels are given by the first list. Finally, the neighborhood information is saved in the third list. As each particle requires matrix inversion as given in eq. (8), employing each thread to compute matrix inversion for each particle reduced the computation time. Algorithm 4 gives the pseudo code to launch and execute GPU kernels.

In the next section,we present numerical results for the Poisson equation with Dirichlet, Neumann and mixed boundary conditions.

## 5. Numerical Results

In this section, we compute some benchmark problems. A comparative study on the numerical tool Finite Pointset Method in CPU and GPU are presented.

**Algorithm 3** Merge-sort inside GPU

```
__device__ mergesortgpu(list1,list2,m,n,p)
i ← m, j ← n, k ← p
while i < n & j < p do
    if list1[i] ≤ list2[j] then
        list2[k] ← list1[i], i ← i + 1
    else
        list2[k] ← list1[j], j ← j + 1
    end if
    k ← k + 1
end while
while i < n do
    list2[k] ← list1[i], i ← i + 1, k ← k + 1
end while
while j < p do
    list2[k] ← list1[j], j ← j + 1, k ← k + 1
end while
for k = 1 → u − 1 do
    list1[k] ← list2[k]
end for
```

**Algorithm 4** Pseudo Code

1. Initialize particles in GPU

2. Transfer Data from GPU to CPU (cudaMemcpy)

3. Launch neighbor search kernel, merge-sorting kernel and copy sorted list to CPU (cudaMemcpy)

4. Local matrix inversion in GPU

5. Generate matrix in CPU

6. Transfer the date from CPU to GPU (cudaMemcpy)

7. Use CUSP library to find the solution of the linear system

8. Update coordinates of particles (cudaMemcpy device to host).

**5.1. Case I.** Now let's consider the two dimensional Poisson equation with Dirichlet boundary conditions.

$$\begin{aligned} \Delta u &= 4 \text{ on } \Omega = [0,1] \times [0,1] \\ u &= x^2 + y^2 \text{ on } \partial\Omega, \end{aligned} \tag{13}$$

where the exact solution is given by $u(x, y) = x^2 + y^2$.

In Fig. 4 we have plotted the exact solution and the numerical solutions obtained from CPU as well as GPU. One can observe that the numerical solutions and exact solution have a very good agreement. The total number of particles is equal to 1681. Here, we have generated the particles in a lattice of distance 0.025 and $h = 3 \times dx$.

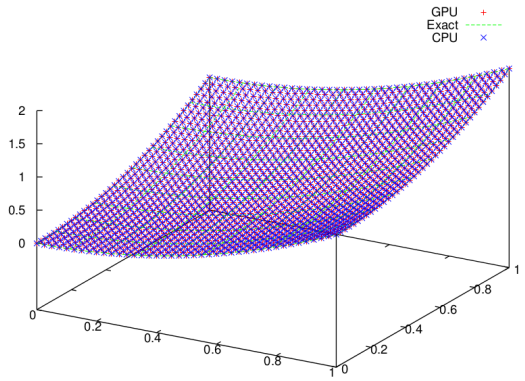



FIGURE 4. Solution of Dirichlet boundary value problem

**5.2. Case II.** Next, we deal with the two dimensional Poisson equation with Neumann boundary conditions.

$$\begin{aligned} \Delta u &= -u \text{ on } \Omega = [0,1] \times [0,1], \\ \frac{\partial u}{\partial n}(0,y) = -sin(y), \frac{\partial u}{\partial n}(1,y) &= sin(y), 0 \le y \le 1 \\ \frac{\partial u}{\partial n}(x,0) = -cos(x), \frac{\partial u}{\partial n}(x,1) &= cos(x), 0 \le x \le 1 \end{aligned} \tag{14}$$

The exact solution is given by $u(x,y) = sin(x) + cos(y)$.

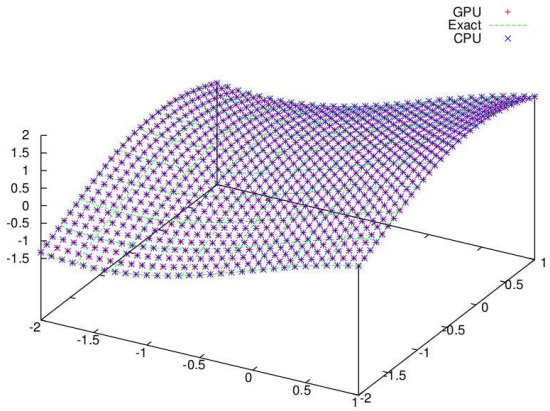



FIGURE 5. Solution of Neumann boundary value problem

In Fig. 5, we have plotted the exact solution together with the numerical solutions obtained from the CPU and the GPU. In this case also, the numerical solutions match with the exact solution. The number of particles is same as in the case I.

**5.3. Case III.** In this case the two dimensional Poisson equation with Mixed boundary conditions is discussed.

$$\begin{aligned} \Delta u &= -8\pi^2 sin(2\pi x) sin(2\pi y) \text{ on } \Omega = [0,1] \times [0,1] \\ u(0,y) &= u(1,y) = 1 \text{ on } 0 \le y \le 1 \text{ and} \\ \frac{\partial u}{\partial n}(x,0) = -2\pi sin(2\pi x), \frac{\partial u}{\partial n}(x,1) &= 2\pi sin(2\pi x) \text{ for } 0 < x < 1 \end{aligned} \tag{15}$$

The exact solution is given by $u(x, y) = sin(2\pi x)sin(2\pi y) + 1$.

In Fig. 6, we have plotted the exact solution along with numerical solutions

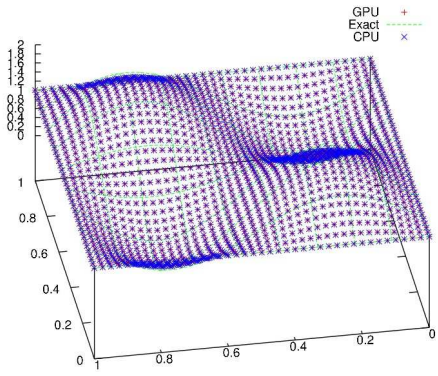



FIGURE 6. Solution of Neumann boundary value problem

obtained from the CPU and the GPU. We can see the matching of the numerical solutions with the exact solution. In this case also, the total number of particles is same as in the case I.

In Fig. 7, we plot the numerical convergence of the scheme. We have plotted the relative error ($\frac{\|u_\epsilon - u\|}{\|u\|}$) against the number of particles, where $u_\epsilon$ denotes the numerical solution and $u$ denotes the exact solution.

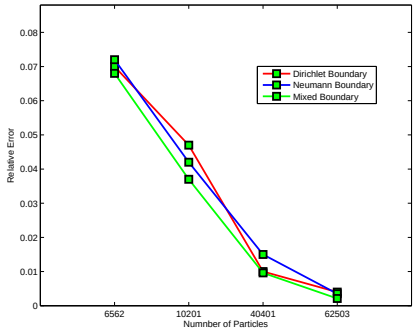



FIGURE 7. Relative error for FPM in GPU

**5.4. GPU Performance.** To test the acceleration performance, an acceleration ratio (speed-up) $\gamma$ is defined as

$$\gamma = \frac{t_{CPU}}{t_{GPU}} \tag{16}$$

where the total processing time on the CPU, $t_{CPU}$, comprises only the time of main loop executed while the total processing time on the GPU, $t_{GPU}$, includes additional time of transferring data between Host and Device in the interest of fairness. $\gamma$ first rises as the number of particles increases. There are two reasons for the changes in $\gamma$. First, if the number of particles is less, the time spent on data transferring between Host and Device takes up a considerable proportion of the total

processing time of the GPU. As the number of particles increases, the proportion decreases rapidly. Second, only after all blocks in a kernel executed the next kernel can be launched in the GPU. The time spent on kernel launching can be roughly considered as a fixed cost. Using more blocks means a reduction in the percentage of kernel launching time, which implies that with a less number of particles and fewer blocks, the pipeline for the GPU has a lower usage, while a more number of particles gains a higher performance because the pipeline approaches higher usage.

TABLE 1. Key facts of NVIDIA Tesla M2050

| | Nvidia Tesla M2050 |
|---|---|
| Number of streaming Processors | 448 |
| Size of Memory | 3 GB |
| Clock rate | 1.15 GHz |
| Double precision peak value | 515 GFlops |
| Single precision peak value | 1030 GFlops |
| | Dual Intel Xeon 5660 |
| Number of cores | 6 |
| Clock rate | 2.8 GHz |
| RAM | 32 GB |
| DDR | InfiniBand |

In most situation, a GBytes size Device memory is enough to contain millions of particles which can support a normal 2D or 3D problem. Thus, we can carry out the computation entirely on the GPU without data transfer between Host and Device for most problems. As for the situation with much larger data size in which the Device memory is not enough (if there is only one GPU), our method can hold such an acceleration ratio with using of stream. The data is divided into smaller parts and processed part by part using asynchronous copy. In our present study, the most important problem is to reduce the time spent on data transfer between GPUs. Setting a relatively large buffer memory is an effective way. In this case, the data exchange frequency will rapidly decrease and the bandwidth between GPUs can be used effectively. The bandwidth between Host and Device is another problem to be considered. The data can be packaged on GPU first, then transport them. In this way, the large bandwidth of Device memory and PCIe (Peripheral Component Interconnect Express) can be fully used.

In the initial days, results in GPU computing discussed on performance on the run-times on one GPU and one CPU core. This comparison of course was not fair. Instead, it is more reasonable to compare one CPU socket with one GPU. For example, four cores CPU is compared with the performance of one GPU. But, even this type of comparison may be questionable if it is not clearly stated which CPU and GPU are compared. From the CPU perspective, one might benchmark dual-/quad-/hexa- or even octo-core CPUs which of course are all different in their performance characteristics, but are installed in one socket. Moreover, there exist several generations and classes of GPUs on the market. We therefore want to advocate the concepts of performance per dollar and performance per Watt. In the first approach, we compare similar priced hardware. To this end, it is still unclear if the price of two full systems or just the hardware price of GPUs and CPUs should be opposed. On one hand, one always has to buy a CPU if one wants to use a GPU.

On the other hand, one can buy a quite cheap CPU (up to an order of magnitude less expensive) and apply it just as a controller for the GPU.

A perfectly fair system-to-system comparison could only be done using a mixed GPU and multi-core CPU code. To shorten the discussion, we focus here only on the direct hardware price comparison of one GPU and appropriate CPUs. We believe that this performance per dollar analysis allows us to have a quit realistic look at the real commercial advantage of GPU computations.

The other metric for performance should be performance per Watt. That is, we measure and compare the required power consumptions for a given simulation task. We expect this approach to become more and more important especially in connection with the Green IT discussion. Beside of the obvious environmental advantages of smaller carbon footprints for large data centers, power consumption is clearly related to energy costs. This is why large companies will be more and more interested to acquire power efficient hardware. Furthermore, ongoing Exascale projects address power consumption as a key issue for the next generation of computing clusters since the current processor technology will not scale due to power limitations.

In this present study, we have executed the program on NVIDIA Tesla M2050. Specifications of NVIDIA Tesla M2050 is given in Table 1.

The test cases use a maximum of 62500 particles as an example. Actual peak values are calculated by clock cycles in the kernel function and are given in the Table 2 together with efficiencies. The efficiencies vary in a small range, which means the kernel function has almost the same performance on different GPUs.

TABLE 2. Double Precision Peak values and efficiencies

| | NVIDIA Tesla M2050 |
|---|---|
| Actual(GFLOPS) | 103 |
| Theory(GFLOPS) | 515 |
| Efficiency | 20% |

In this paper, we have applied GPU techniques basically for 7 different parts of the FPM, namely, the linear system in eq. (11), local matrix inversion in eq. (8), neighbor searching, merge-sorting, particles initialization and scalar products. The total speed-up distribution for the FPM is given in Fig. 8. We computed the time to finish each kernel in GPU and similarly the computation time to execute the same task in the CPU. Summing up GPU kernel execution time and CPU execution of the same task has been calculated. The ratio of each kernel execution time (may be multiple times) and the total kernel execution time is given as the individual contribution of each kernel. The acceleration ratio of the GPU scheme is also provided, as shown in Fig. 9.

From the Fig. 9(b), one can observe the efficiency of the GPU, which is 70 times faster than CPU for 62503 particles. As we discussed earlier, for the comparison purpose, we kept all the cores but one as idle in CPU, so that one can use a single core CPU to control the GPU. However, we present in Fig. 9(a) the acceleration ratio for the same simulation by using all the 6 cores in the CPU.

We finally discuss the performance per Watt results for our code on the different in-house hardware platforms. To do that, we run the Poisson equation for 62503 particles in the full 6-core CPU system and on the NVIDIA Tesla M2050 GPU without ECC. During the simulation, we attach a power measuring tool to each

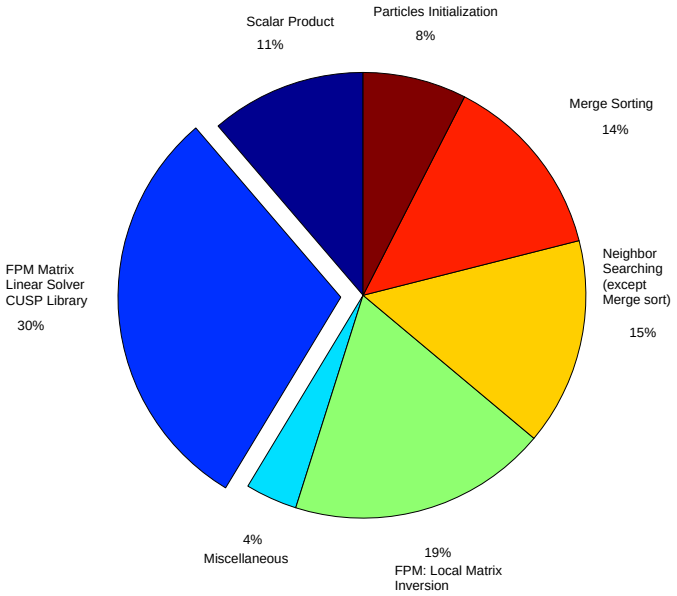


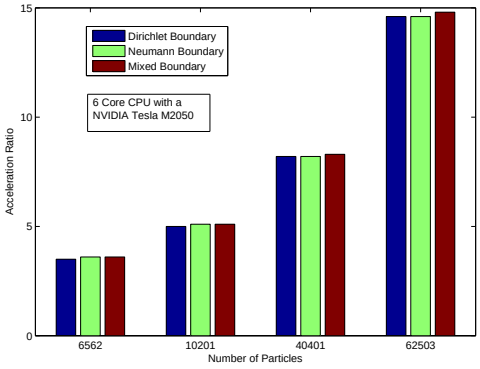

FIGURE 8. Speedup distribution in FPM

of the machines and note the power consumption. The resulting numbers are presented in Fig. 10. The maximum power consumption is required by the CPU system. In contrast, the results of the GPU cluster are already significantly better. The numbers for the NVIDIA card is quite impressive. It is by a factor of 2 more power efficient for calculations without ECC. These results are a clear indicator for the superiority of GPU computations. This may be important for large computing centers which are looking for energy efficient systems.

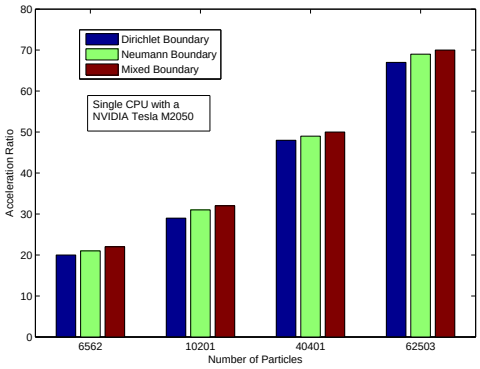
## 6. Conclusion

Implementation of Bi-CGSTAB in the Finite Pointset Method on a GPU is performed for the solution of the Poisson equation in three different cases. In the CPU implementation which is compared with the GPU code, this linear system is solved efficiently by using Bi-CGSTAB, whereas the GPU code utilizes the *cusp* library to solve the same linear system, resulting a faster but relatively efficient computation. Therefore, as the number of particles increases, allocated time for solutions of the linear system increases remarkably for the GPU implementation. In this context, 45% of the operating time is consumed to solve the linear system. This shows that an improved solver for the solution of such relatively a large (more than 1 million particles) linear system would further increase speed-up of the GPU implementation over the CPU implementation. Currently, speed-ups between 60× - 70× are achieved on GPU compared to CPU computations.

It is concluded that, the FPM for the solution of the Poisson equation can achieve excellent speed-up when implemented on a GPU. Hence, when we solve the incompressible Navier Stokes equations by the FPM using GPU, we can save the computational time efficiently. As they required repeated solution of the Poisson




(a) GPU acceleration ratio for single CPU(6 cores) vs N-VIDIA Tesla M2050




(b) GPU acceleration ratio for single core CPU vs NVIDI-A Tesla M2050

FIGURE 9. GPU acceleration ratio

equation in each time step, further speed-up is possible with an efficient GPU implementation.

## Acknowledgments

We would like to thank DAAD for the financial support.

## References

[1] Bayoumi A., Chu A., Hanafy A., Harrell P., Refai-Ahmed G., *Scientific and engineering computing using ATI stream technology*, Computing in Science & Engineering, 11:92-97, 2009.

[2] K. E. Batcher, *Sorting networks and their applications,* in Proc. AFIPS Spring Joint Computer Conference, vol. 32, pp. 307C314, 1968.

[3] Björck A., *Numerical Methods for Least Squares Problems*, SIAM. ISBN 978-0-898713-60-2, 1996.

[4] Drum C., Tiwari S., Kuhnert J., Bart H.J., *Finite pointset method for simulation of the liquid-liquid flow field in an extractor*, Computers and Chemical Engineering., 32:2946-2957, 2008.

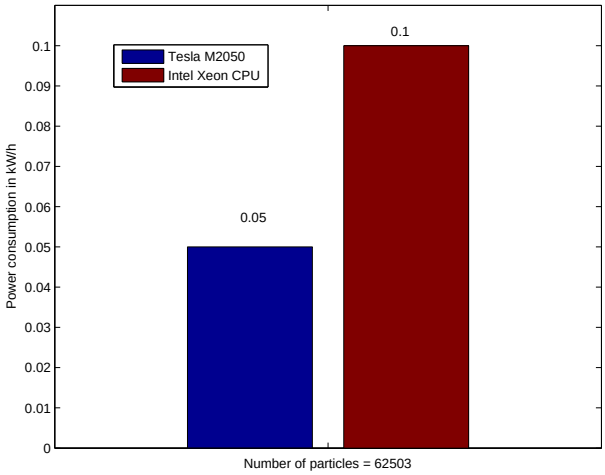



FIGURE 10. The measured power consumption figure shows that Telsa GPU is about a factor of 2.0 more power-efficient compared to standard CPU

[5] Du P., Luszczek P., Dongarra J., *OpenCL evaluation for numerical linear algebra library development*, Symposium on Applications Accelerators in High Performance Computing, SAAHPC'10, 2010.
[6] Göddeke D., *GPGPU-basic math tutorial*, Ergebnisberichte des Instituts fur Angewandte Mathematik. Nummer 300.
[7] Greg R., Massimiliano F., *A CUDA Fortran Implementation of BWAVES*, September,2010.
[8] Harada T., Koshizuka S., Kawaguchi Y., *Smoothed particle hydrodynamics on GPUs*, Proceeding of the Spring Conference on Computer Graphics, 235-241, 2007.
[9] Knibbe H., Osterlee C. W., Vuik C., *GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method*, Journal of Computational and Applied Mathematics, 236:281-293, 2011.
[10] Kuznik F., Obrecht C., Rusaouen G., Roux JJ., *LBM based flow simulation using GPU computing processor*, Comp. Math Appl. 59:2380-2392,2010.
[11] Luo X., Qin F., Ran W., *GPU accelerated CESE method for 1D shock tube problems*, Journal of Computational Physics, 230:8797-8812, 2011.
[12] Munshi A., *OpenCL: Parallel Computing on the GPU and CPU*, SIGGRAPH, Tutorial.
[13] Nathan B., Michael G., *Efficient Sparse Matrix-Vector Multiplication on CUDA*,NVIDIA Technical Report NVR-2008-004, December 2008.
[14] Nathan B., Michale G., *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*,2012, http://cusp-library.googlecode.com.
[15] NVIDIA Inc., *NVIDIA CUDA Programming Guide Version 4.0*, June, 2011. http://developer. nvidia.com/nvidia-gpu-computing-documentation.
[16] NVIDIA Inc., *NVIDIA CUDA Best Practices Guide, version 4.0*, May, 2011. http://developer. nvidia.com/nvidia-gpu-computing-documentation.
[17] NVIDIA Inc., *White Paper NVIDIA's Next Generation CUDA compute architecture, Fermi*, 2009.
[18] Rossinelli D., Bergdorf M., Cottet G., Koumoutsakos P., *GPU accelerated simulations of bluff body flows using vortex particle methods*, Journal of Computational Physics, 229:3316-3333, 2010.
[19] Satish N., Harris M., Garland M*Designing efficient sorting algorithms for manycore GPUs*, Proc. 23rd IEEE International Parallel and Distributed processing symposium, May 2009.

[20] Tiwari S., Kuhnert J., *Finite pointset method based on the projection method for simulations of the incompressible Navier-Stokes equations*, Meshfree methods for partial differential equations (Bonn, 2001), 373?387, Lect. Notes Comput. Sci. Eng., 26, Springer, Berlin, 2003.
[21] Tiwari S., Kuhnert J., *Modeling of two phase flows with surface tension by finite pointset method (FPM)*, Journal of Computational and Applied Mathematics, 203: 376-386, 2007.
[22] Van der Vorst, H. A. *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems*, SIAM Journal on Scientific and Statistical Computing 13 (2): 631C644, 1992.

⋆ Department of Mathematics, IIT Madras, Chennai - 600 036, India
*E-mail*: mpanch13114@gmail.com

⩘ Fachbereich Mathematik, TU Kaiserslautern Kaiserslautern - 67663, Germany