A CUDA-based Implementation of an improved SPH method on GPU

L. Antonelli[a,c], E. Francomano[b,c], F. Gregoretti[a,c]

[a] *Institute for High Performance Computing and Networking of CNR - via Pietro Castellino, 111 - Napoli*

[b] *University of Palermo, Department of Engineering - Viale delle Scienze, Ed. 6 - 90128 Palermo*

[c] *Istituto Nazionale di Alta Matematica "Francesco Severi", Research group GNCS, Piazzale Aldo Moro 5, 00185 Rome, Italy*

Abstract

We present a CUDA-based parallel implementation on GPU architecture of a modified version of the Smoothed Particle Hydrodynamics (SPH) method. This modified formulation exploits a strategy based on the Taylor series expansion, which simultaneously improves the approximation of a function and its derivatives with respect to the standard formulation. The improvement in accuracy comes at the cost of an additional computational effort. The computational demand becomes increasingly crucial as problem size increases but can be addressed by employing fast summations in a parallel computational scheme. The experimental analysis showed that our parallel implementation significantly reduces the runtime, when compared to the CPU-based implementation.

Keywords:
Smoothed Particle Hydrodynamics, Fast Gauss Transform, Graphics Processing Unit.

1. Introduction

Smoothed Particle Hydrodynamics (SPH) method, introduced independently by Lucy [1], and Gingold and Monaghan [2], can be regarded as the oldest among the modern mesh-free particle methods. The most important advantage of these methods is that the mesh generation is not required during the discretization of the domain, as is the case with the grid based methods. This allows the simulations of real problems with complex geometry or containing discontinuities, singularities or with a non linear behavior. Moreover, computations with high dimensional data can be considered too. In recent years these methods have received a strong interest emerging as valid computational alternatives in numerous problems, from different areas of science and engineering, that require the numerical solution of integral equations or PDEs with different boundary conditions [3, 4, 5, 6, 7]. Applications can be found in geodesy and mapping [8], geoscience [9], metereology [10], computer graphics [11], signal and image processing [12], computational finance [13, 14], learning theory [15, 16, 17], biomathematics [18, 19, 20]. Many of these applications involve function approximation or need derivative estimation at any data location with any known data distribution in the problem domain. In SPH, the state of a system is represented by integral approximation on a set of arbitrarily scattered data which interact with each other within the range controlled by a smoothing function. Since its introduction in the 1970's, Smoothed Particle Hydrodynamics has been widely applied, permitting a straightforward handling of very large deformations [21] such as happens in high energy phenomena i.e. explosion, high velocity impact, and penetrations. We address the reader for further study about variations of the SPH methods and new applications to [22, 23] and references therein. The SPH has been successfully adopted for electromagnetics (EM) transients simulations too [5, 24, 25, 26] and better results have been gained dealing with an improved formulation of SPH based on the Taylor series expansion adopting the Gaussian as kernel function [26]. The corrections turned out to be suitable, providing significant mesh-free estimates of the electric and magnetic field components. With the aim to treat real applications in an accurate and fast fashion, here we present some preliminary investigations of the improved method on GPU referring to various test functions with different data sets. Although originally designed to process computer graphic, the GPUs evolved into a highly parallel, multithreaded, manycore processor giving rise to the general-purpose GPU computing. In fact, many scientific and numerical algorithms have building blocks with inherent massive parallelism, that can benefit from GPU acceleration [27, 28]. GPU-accelerated algorithms still run on the CPU but offload these parallel building blocks on GPUs taking advantage of its computational horsepower. A fully parallel implementation of the improved SPH on GPU and its experimental analysis is described throughout the work. This work completes and extends the one presented in [29] where only one task of the improved SPH algorithm was implemented on the GPU. The paper is organized as follows. In the Section 2 the fundamental computational tasks of the improved SPH are provided. In Section 3 a detailed GPU implementation of the method is presented. Section 4 provides the experimental results referring to two different hardware configurations with two different GPU architectures (Volta [30] and Turing [31]). Finally, in Section 6 the conclusions and the future work are given.

2. The method: standard and improved formulation

In this section the theoretical basis of the method are shortly described in order to introduce the fundamental computational tasks for the CUDA-based implementation. An approximation to a multivariate function $f : \Omega \subset \Re^d \rightarrow \Re$ with $d \geq 1$ is generated as:

$$< f_h(\mathbf{x}) >= \int_\Omega f(\boldsymbol{\xi})\mathsf{K}(\mathbf{x}, \boldsymbol{\xi}; h)d\Omega, \tag{1}$$

where $\mathsf{K}(\mathbf{x}, \boldsymbol{\xi}; h)$ is the kernel function, $\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(d)}) \in \Omega$ and $\boldsymbol{\xi} = (\xi^{(1)}, \xi^{(2)}, \dots, \xi^{(d)}) \in \Omega$ are the evaluation and source points respectively, and $h \in \Re^+$ measures the influence of K. Using a discrete representation of the computational domain by N d-dimensional source points, $\Xi = \{\boldsymbol{\xi}_j\}_{j=1}^N$, each associated with a subdomain $\Omega_j \subset \Omega$, the so-called *particle approximation* of (1) is defined as

$$f_h(\mathbf{x}) = \sum_{j=1}^{N} f(\boldsymbol{\xi}_j)\mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h)d\Omega_j, \tag{2}$$

where $d\Omega_j$ is the measure of Ω_j. From now on we will indicate with $\boldsymbol{X}$ the set of the evaluation points, where $\#(\boldsymbol{X})=M$. The formula (2) can lose accuracy, e.g., when irregular point distributions are considered.

In order to increase the accuracy of the standard method up to the order $k+1$, we consider the k-th order Taylor expansion of $f(\boldsymbol{\xi})$, with f a function sufficiently smooth:

$$f(\boldsymbol{\xi}) = \sum_{|\alpha| \leq k} \frac{1}{\alpha!}(\boldsymbol{\xi} - \mathbf{x})^\alpha \mathcal{D}^{(\alpha)} f(\mathbf{x}) + \mathcal{O}(h^{k+1}), \tag{3}$$

where $\alpha = (\alpha^{(1)}, \alpha^{(2)}, \dots, \alpha^{(d)}) \in \mathbb{N}^d$ is a multi-index, $|\alpha| = \sum_{i=1}^d \alpha^{(i)}$, $\alpha! = \prod_{i=1}^d (\alpha^{(i)})!$, $\mathbf{y}^\alpha = (y^{(1)})^{\alpha^{(1)}} \cdot (y^{(2)})^{\alpha^{(2)}} \cdot \dots \cdot (y^{(d)})^{\alpha^{(d)}}$, and $\mathcal{D}^{(\alpha)} = \frac{\partial^{|\alpha|}}{(\partial x^{(1)})^{\alpha^{(1)}} \dots (\partial x^{(d)})^{\alpha^{(d)}}}$.

By multiplying (3) by the kernel function and its derivatives up to the k-th order, integrating over Ω and adopting (2), an approximation of the function f and its derivatives till the order k, at each evaluation point $\mathbf{x}$, are simultaneously provided by solving the linear system [32]:

$$\mathbf{A}_{\mathbf{x}}^{(k)} \mathbf{c}_{\mathbf{x}}^{(k)} = \mathbf{b}_{\mathbf{x}}^{(k)}, \tag{4}$$

where

$$\mathbf{A}_{\mathbf{x}}^{(k)} = \begin{pmatrix} \sum_{j=1}^{N} \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j & \dots & \frac{1}{k!} \sum_{j=1}^{N} (\xi_j^{(d)} - x^{(d)})^k \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^{N} \mathcal{D}^{(k)} \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j & \dots & \frac{1}{k!} \sum_{j=1}^{N} (\xi_j^{(d)} - x^{(d)})^k \mathcal{D}^{(k)} \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j \end{pmatrix},$$

$$\mathbf{c}_{\mathbf{x}}^{(k)} = \begin{pmatrix} f(\mathbf{x}) \\ \vdots \\ \mathcal{D}^{(k)} f(\mathbf{x}) \end{pmatrix}, \quad \mathbf{b}_{\mathbf{x}}^{(k)} = \begin{pmatrix} \sum_{j=1}^{N} f(\boldsymbol{\xi}_j) \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j \\ \vdots \\ \sum_{j=1}^{N} f(\boldsymbol{\xi}_j) \mathcal{D}^{(k)} \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j \end{pmatrix}.$$

The size m of the system (4) increases as both the data dimension and the order of accuracy increase, i.e. $m = (d+k)!/(d!\,k!)$. As described in [32], the construction of the coefficient matrix and the rhs vector of the system (4), and its solution at each evaluation point, raise the computational cost of the method. When the Gaussian function, infinitely differentiable and smooth even for high order derivatives, is adopted, such as in electromagnetics transients simulations [26], this further computational effort can be controlled by the use of fast summation methods. In fact, being the derivative of a Gaussian kernel a Hermite polynomial times a Gaussian kernel, the elements of $\mathbf{A}_{\mathbf{x}}^{(k)}$ are obtained through the computation of weighted Gaussian sums of this form:

$$\begin{aligned}\sum_{j=1}^{N}(\xi_j^{(r)} - x^{(r)})^{\beta}\mathsf{K}(\mathbf{x},\boldsymbol{\xi}_j;h)d\Omega_j &= \\ = \sum_{j=1}^{N}\sum_{n=0}^{\beta}(-1)^n\binom{\beta}{n}(\xi_j^{(r)})^n(x^{(r)})^{\beta-n}\mathsf{K}(\mathbf{x},\boldsymbol{\xi}_j;h)d\Omega_j &= \\ = \sum_{n=0}^{\beta}(-1)^n\binom{\beta}{n}(x^{(r)})^{\beta-n}\left[\sum_{j=1}^{N}(\xi_j^{(r)})^n\mathsf{K}(\mathbf{x},\boldsymbol{\xi}_j;h)d\Omega_j\right]&,\end{aligned} \tag{5}$$

where $\beta = 0,\ldots,2k$, $r = 1,\ldots,d$; similarly the elements of $\mathbf{b}_{\mathbf{x}}^{(k)}$ are obtained through the computation of weighted Gaussian sums of this form:

$$\begin{aligned}\sum_{j=1}^{N}(\xi_j^{(r)} - x^{(r)})^{\gamma}f(\boldsymbol{\xi}_j)\mathsf{K}(\mathbf{x},\boldsymbol{\xi}_j;h)d\Omega_j &= \\ = \sum_{n=0}^{\gamma}(-1)^n\binom{\gamma}{n}(x^{(r)})^{\gamma-n}\left[\sum_{j=1}^{N}(\xi_j^{(r)})^n f(\boldsymbol{\xi}_j)\mathsf{K}(\mathbf{x},\boldsymbol{\xi}_j;h)d\Omega_j\right]&,\end{aligned} \tag{6}$$

where $\gamma = 0,\ldots,k$, $r = 1,\ldots,d$. Therefore, the construction of the systems (4) requires the evaluation of the Gaussian transforms in the square brackets, that we denote as:

$$G_l(\mathbf{x}) = \sum_{j=1}^{N} w_l(\boldsymbol{\xi}_j)\mathsf{K}(\mathbf{x},\boldsymbol{\xi}_j;h), \quad l = 1,\ldots,L. \tag{7}$$

Once this evaluation has been carried out, the elements of $\mathbf{A}_{\mathbf{x}}^{(k)}$ and $\mathbf{b}_{\mathbf{x}}^{(k)}$ can be computed simultaneously, for all the evaluation points, through a linear combination of the vectors containing the transforms $G_l(\mathbf{x})$ and the powers of a specific coordinate vector of $\mathbf{x}$, as highlighted in (5) and (6) respectively. In summary, the improved SPH method can be structured into the five computational tasks (a)-(e) described in Algorithm 1.

Following this task organization, we developed a serial version [29] of the improved SPH algorithm in C++, that served as reference for validating the results of the parallel implementation.

Algorithm 1: Improved SPH

```
Input: $d$, $k$, $N$, $M$, $f$
Output: $\mathbf{c}_{\mathbf{x}}^{(k)}, \forall \mathbf{x} \in \boldsymbol{X}$
(a) generate:
    $\boldsymbol{\Xi} \leftarrow N$ $d$-dimensional source points
    $\boldsymbol{X} \leftarrow M$ $d$-dimensional evaluation points
(b) compute the weights in (7): $w_l(\boldsymbol{\xi}_j)$, $\forall \boldsymbol{\xi}_j \in \boldsymbol{\Xi}$, $l = 1, \ldots, L$
(c) evaluate the Gauss transforms (7): $G_l(\mathbf{x})$, $\forall \mathbf{x} \in \boldsymbol{X}$, $l = 1, \ldots, L$
(d) compute the matrices $\mathbf{A}_{\mathbf{x}}^{(k)}$ and the rhs vectors $\mathbf{b}_{\mathbf{x}}^{(k)}$ in (4), $\forall \mathbf{x} \in \boldsymbol{X}$
(e) solve (4): $\mathbf{A}_{\mathbf{x}}^{(k)} \mathbf{c}_{\mathbf{x}}^{(k)} = \mathbf{b}_{\mathbf{x}}^{(k)}$ , $\forall \mathbf{x} \in \boldsymbol{X}$
```

Task (a)
In our implementation, in the scope of task (a), Ω was set equal to $[0,1]^d$ and three distributions of N source points were considered: uniform d-dimensional mesh, Halton [33] and Sobol' [34] d-dimensional sequences. The Halton and Sobol' points were generated by using the C++ code available from [35] with $O(dN)$ FLOPs. The M evaluation points were distributed on a uniform mesh over Ω requiring $O(dM)$ FLOPs.

Task (b)
The weights $w_l(\boldsymbol{\xi}_j), l = 1, \ldots, L$ of the Gaussian transforms (7) required in (5) and (6), can be computed according to the graded lexicographic order of the d-dimensional monomial $\boldsymbol{\xi}_j$ as follows

$$w_l(\boldsymbol{\xi}_j) = \rho_j (\boldsymbol{\xi}_j^{\alpha})_s, \qquad |\alpha| = \beta = 0, \ldots, 2k, \quad s = 1, \ldots, \binom{d+\beta-1}{\beta} \tag{8}$$

$$w_l(\boldsymbol{\xi}_j) = \rho_j (\boldsymbol{\xi}_j^{\alpha})_t f(\boldsymbol{\xi}_j), \qquad |\alpha| = \gamma = 0, \ldots k, \quad t = 1, \ldots, \binom{d+\gamma-1}{\gamma}. \tag{9}$$

In (8) and (9), $\rho_j = \frac{1}{\pi h^2} d\Omega_j$, $f(\boldsymbol{\xi}_j)$ is the function to be approximated and evaluated in $\boldsymbol{\xi}_j$, while, according to the previous multi-index notation, $(\boldsymbol{\xi}_j^{\alpha})_s$ and $(\boldsymbol{\xi}_j^{\alpha})_t$ are the s-th and the t-th monomial of $\boldsymbol{\xi}_j$, with total degree β and γ respectively. The total number of $w_l(\boldsymbol{\xi}_j)$, i.e. the number of Gaussian transforms required for the construction of $\mathbf{A}_{\mathbf{x}}^{(k)}$, $\forall \mathbf{x} \in \boldsymbol{X}$ is

$$L_{\mathbf{A}_{\mathbf{x}}^{(k)}} = \sum_{\beta=0}^{2k} \binom{d+\beta-1}{\beta} = \frac{(d+2k)!}{d!(2k)!}, \tag{10}$$

while the total number of $w_l(\boldsymbol{\xi}_j)$, i.e. the number of Gaussian transforms required for the construction of $\mathbf{b}_{\mathbf{x}}^{(k)}, \forall \mathbf{x} \in \boldsymbol{X}$ is

$$L_{\mathbf{b}_{\mathbf{x}}^{(k)}} = \sum_{\gamma=0}^{k} \binom{d+\gamma-1}{\gamma} = \frac{(d+k)!}{d!k!}. \tag{11}$$

Note that $L_{\mathbf{b}_{\mathbf{x}}^{(k)}}$ is equal to m, while $L_{\mathbf{A}_{\mathbf{x}}^{(k)}} < m^2$. Therefore $L = L_{\mathbf{A}_{\mathbf{x}}^{(k)}} + L_{\mathbf{b}_{\mathbf{x}}^{(k)}} < m^2 + m$. $L_{\mathbf{A}_{\mathbf{x}}^{(k)}}$ and $L_{\mathbf{b}_{\mathbf{x}}^{(k)}}$ increase as both the data dimension d and the order of accuracy k increase, as shown in Table 1 for common values of k and d.

Accuracy order $(k+1)$		data dimension $d=1$	$d=2$	$d=3$
$k=0$	$L_{\mathbf{A}_{\mathbf{x}}^{(0)}}$	1	1	1
	$L_{\mathbf{b}_{\mathbf{x}}^{(0)}}$	1	1	1
$k=1$	$L_{\mathbf{A}_{\mathbf{x}}^{(1)}}$	3	6	10
	$L_{\mathbf{b}_{\mathbf{x}}^{(1)}}$	2	3	4
$k=2$	$L_{\mathbf{A}_{\mathbf{x}}^{(2)}}$	5	15	35
	$L_{\mathbf{b}_{\mathbf{x}}^{(2)}}$	3	6	10

Table 1: Number of Gauss transforms to be evaluated by the improved SPH for common values of k and d.

A sketch of the algorithm to compute the weights (8) and (9) with $O(LdN)$ FLOPs is described in the Algorithm 2.

Algorithm 2: Improved SPH - task (b)

```
Input: $d$, $k$, $N$, $\boldsymbol{\Xi}$, $h$, $\mathbf{d\Omega}$, $f$
Output: $\mathbf{w}(\boldsymbol{\xi}_j)$, $\forall \boldsymbol{\xi}_j \in \boldsymbol{\Xi}$
$l := 0$
% compute weights $w_l(\boldsymbol{\xi}_j)$ in (8)
for $\beta \leftarrow 0$ to $2k$ do
    % number of monomials of total degree $\beta$
    $s := \binom{d+\beta-1}{\beta}$
    for $n \leftarrow 1$ to $s$ do
        $l := l + 1$
        % compute the multi-index $\alpha$ according to the graded
          lexicographic order
        $\alpha := (\alpha^{(1)}, \alpha^{(2)}, \dots, \alpha^{(d)})$ s.t. $|\alpha| = \sum_{i=1}^{d} \alpha^{(i)} = \beta$
        for $j \leftarrow 1$ to $N$ do
            $\rho_j := \frac{1}{\pi h^2} d\Omega_j$
            $w_l(\boldsymbol{\xi}_j) := \rho_j (\boldsymbol{\xi}_j^{\alpha})_s$
        end
    end
end
% compute weights $w_l(\boldsymbol{\xi}_j)$ in (9)
for $\gamma \leftarrow 0$ to $k$ do
    % number of monomials of total degree $\gamma$
    $t := \binom{d+\gamma-1}{\gamma}$
    for $n \leftarrow 1$ to $t$ do
        $l := l + 1$
        % compute the multi-index $\alpha$ according to the graded
          lexicographic order
        $\alpha := (\alpha^{(1)}, \alpha^{(2)}, \dots, \alpha^{(d)})$ s.t. $|\alpha| = \sum_{i=1}^{d} \alpha^{(i)} = \gamma$
        for $j \leftarrow 1$ to $N$ do
            $\rho_j := \frac{1}{\pi h^2} d\Omega_j$
            $w_l(\boldsymbol{\xi}_j) := \rho_j (\boldsymbol{\xi}_j^{\alpha})_t f(\boldsymbol{\xi}_j)$
        end
    end
end
```

Task (c)

Task(c) is unquestionably the most computationally intensive task (see Table 2). Several approaches exist to accelerate the Gaussian kernel summations [36, 37, 38]. These approaches evaluate the weighted Gaussian sum, using special data structures and approximation techniques, breaking down the computational complexity from $O(LMN)$ to $O(L(M+N))$. `figtree` [39], that we used in our serial implementation, integrates different computational strategies and automatically chooses the fastest method for a given data and desired accuracy.

Task (d)

We note that for a given data dimension d the matrix $\mathbf{A}_{\mathbf{x}}^{(k)}$ and the vector $\mathbf{b}_{\mathbf{x}}^{(k)}$ contain respectively, all the matrices $\mathbf{A}_{\mathbf{x}}^{(p)}$ and the vectors $\mathbf{b}_{\mathbf{x}}^{(p)}$ with $p < k$ as described in the scheme below

$$\left[\begin{matrix}\left[\begin{matrix}\left[\begin{matrix}\left[\mathbf{A}_{\mathbf{x}}^{(0)}\right] & \\ & \mathbf{A}_{\mathbf{x}}^{(1)}\end{matrix}\right] & \\ & \mathbf{A}_{\mathbf{x}}^{(2)}\end{matrix}\right] & & \\ & \ddots & \\ & & \mathbf{A}_{\mathbf{x}}^{(k)}\end{matrix}\right]; \quad \left[\begin{matrix}\left[\begin{matrix}\left[\begin{matrix}\left[\mathbf{b}_{\mathbf{x}}^{(0)}\right] \\ \mathbf{b}_{\mathbf{x}}^{(1)}\end{matrix}\right] \\ \mathbf{b}_{\mathbf{x}}^{(2)}\end{matrix}\right] \\ \vdots \\ \mathbf{b}_{\mathbf{x}}^{(k)}\end{matrix}\right]. \qquad (12)$$

For the sake of simplicity, we give the details for this task with $d = 2$ and $k = 2$.

Recalling the (5) and setting

$$(\mathbf{q}^{(r)})^{\beta} := \sum_{j=1}^{N} (\xi_j^{(r)} - x^{(r)})^{\beta} \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j,$$

with $\beta = 0, \ldots, 2k$ and $r = 1, 2$, the computation of the M coefficient matrices, for each evaluation point $\mathbf{x} = (x^{(1)}, x^{(2)})$, requires linear combinations between the M-dimensional vectors $\mathbf{G}_l, l = 1, \ldots, L_{\mathbf{A}_{\mathbf{x}}^{(2)}}$ and the powers of the coordinate vectors of all evaluation points. Specifically, being $\boldsymbol{X}^{(r)}, r = 1, 2$, the r-th coordinate vector of all evaluation points, the required linear combinations are:

$$
\begin{aligned}
\tilde{\mathbf{A}}_1 &:= (\mathbf{q}^{(1)})^0 = (\mathbf{q}^{(2)})^0 = \mathbf{G}_1\\
\tilde{\mathbf{A}}_2 &:= (\mathbf{q}^{(1)})^1 = \mathbf{G}_2 - \boldsymbol{X}^{(1)}\mathbf{G}_1\\
\tilde{\mathbf{A}}_3 &:= (\mathbf{q}^{(2)})^1 = \mathbf{G}_3 - \boldsymbol{X}^{(2)}\mathbf{G}_1\\
\tilde{\mathbf{A}}_4 &:= (\mathbf{q}^{(1)})^2 = \mathbf{G}_4 - 2\boldsymbol{X}^{(1)}\mathbf{G}_2 + (\boldsymbol{X}^{(1)})^2\mathbf{G}_1\\
\tilde{\mathbf{A}}_5 &:= (\mathbf{q}^{(1)})(\mathbf{q}^{(2)}) = \mathbf{G}_5 - \boldsymbol{X}^{(1)}\mathbf{G}_3 - \boldsymbol{X}^{(2)}\mathbf{G}_2 + \boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_1\\
\tilde{\mathbf{A}}_6 &:= (\mathbf{q}^{(2)})^2 = \mathbf{G}_6 - 2\boldsymbol{X}^{(2)}\mathbf{G}_3 + (\boldsymbol{X}^{(2)})^2\mathbf{G}_1\\
\tilde{\mathbf{A}}_7 &:= (\mathbf{q}^{(1)})^3 = \mathbf{G}_7 - 3\boldsymbol{X}^{(1)}\mathbf{G}_4 + 3(\boldsymbol{X}^{(1)})^2\mathbf{G}_2 - (\boldsymbol{X}^{(1)})^3\mathbf{G}_1\\
\tilde{\mathbf{A}}_8 &:= (\mathbf{q}^{(1)})^2(\mathbf{q}^{(2)}) = \mathbf{G}_8 - \boldsymbol{X}^{(2)}\mathbf{G}_4 - 2\boldsymbol{X}^{(1)}\mathbf{G}_5 + 2\boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_2 + (\boldsymbol{X}^{(1)})^2\mathbf{G}_3 +\\
&\quad - (\boldsymbol{X}^{(1)})^2\boldsymbol{X}^{(2)}\mathbf{G}_1\\
\tilde{\mathbf{A}}_9 &:= (\mathbf{q}^{(1)})(\mathbf{q}^{(2)})^2 = \mathbf{G}_9 - \boldsymbol{X}^{(1)}\mathbf{G}_6 - 2\boldsymbol{X}^{(2)}\mathbf{G}_5 + 2\boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_3 + (\boldsymbol{X}^{(2)})^2\mathbf{G}_2 +\\
&\quad - \boldsymbol{X}^{(1)}(\boldsymbol{X}^{(2)})^2\mathbf{G}_1\\
\tilde{\mathbf{A}}_{10} &:= (\mathbf{q}^{(2)})^3 = \mathbf{G}_{10} - 3\boldsymbol{X}^{(2)}\mathbf{G}_6 + 3(\boldsymbol{X}^{(2)})^2\mathbf{G}_3 - (\boldsymbol{X}^{(2)})^3\mathbf{G}_1\\
\tilde{\mathbf{A}}_{11} &:= (\mathbf{q}^{(1)})^4 = \mathbf{G}_{11} - 4\boldsymbol{X}^{(1)}\mathbf{G}_7 + 6(\boldsymbol{X}^{(1)})^2\mathbf{G}_4 - 4(\boldsymbol{X}^{(1)})^3\mathbf{G}_2 + (\boldsymbol{X}^{(1)})^4\mathbf{G}_1\\
\tilde{\mathbf{A}}_{12} &:= (\mathbf{q}^{(1)})^3(\mathbf{q}^{(2)}) = \mathbf{G}_{12} - 3\boldsymbol{X}^{(1)}\mathbf{G}_8 - \boldsymbol{X}^{(2)}\mathbf{G}_7 + 3(\boldsymbol{X}^{(1)})^2\mathbf{G}_5 + 3\boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_4 +\\
&\quad - 3(\boldsymbol{X}^{(1)})^2\boldsymbol{X}^{(2)}\mathbf{G}_2 - (\boldsymbol{X}^{(1)})^3\mathbf{G}_3 - (\boldsymbol{X}^{(1)})^3\boldsymbol{X}^{(2)}\mathbf{G}_1\\
\tilde{\mathbf{A}}_{13} &:= (\mathbf{q}^{(1)})^2(\mathbf{q}^{(2)})^2 = \mathbf{G}_{13} - 2\boldsymbol{X}^{(1)}\mathbf{G}_9 + (\boldsymbol{X}^{(1)})^2\mathbf{G}_6 + 4\boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_5 - 2\boldsymbol{X}^{(2)}\mathbf{G}_8 +\\
&\quad - 2(\boldsymbol{X}^{(1)})^2\boldsymbol{X}^{(2)}\mathbf{G}_3 + (\boldsymbol{X}^{(2)})^2\mathbf{G}_4 - 2\boldsymbol{X}^{(1)}(\boldsymbol{X}^{(2)})^2\mathbf{G}_2 + (\boldsymbol{X}^{(1)})^2(\boldsymbol{X}^{(2)})^2\mathbf{G}_1\\
\tilde{\mathbf{A}}_{14} &:= (\mathbf{q}^{(1)})^1(\mathbf{q}^{(2)})^3 = \mathbf{G}_{14} - \boldsymbol{X}^{(1)}\mathbf{G}_{10} - 3\boldsymbol{X}^{(2)}\mathbf{G}_9 + 3\boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_6 + 3(\boldsymbol{X}^{(2)})^2\mathbf{G}_5 +\\
&\quad - 3\boldsymbol{X}^{(1)}(\boldsymbol{X}^{(2)})^2\mathbf{G}_3 - (\boldsymbol{X}^{(2)})^3\mathbf{G}_2 + \boldsymbol{X}^{(1)}(\boldsymbol{X}^{(2)})^3\mathbf{G}_1\\
\tilde{\mathbf{A}}_{15} &:= (\mathbf{q}^{(2)})^4 = \mathbf{G}_{15} - 4\boldsymbol{X}^{(2)}\mathbf{G}_{10} + 6(\boldsymbol{X}^{(2)})^2\mathbf{G}_6 - 4(\boldsymbol{X}^{(2)})^3\mathbf{G}_3 + (\boldsymbol{X}^{(2)})^4\mathbf{G}_1.
\end{aligned}
\tag{13}
$$

In a similar way, recalling the (6) and setting

$$
(\mathbf{p}^{(r)})^\gamma := \sum_{j=1}^{N} f(\boldsymbol{\xi}_j)(\xi_j^{(r)} - x^{(r)})^\beta \mathsf{K}(\mathbf{x}, \boldsymbol{\xi}_j; h) d\Omega_j,
$$

with $\gamma = 0, \ldots, k$ and $r = 1, 2$, the linear combinations required by the computation of the M rhs vectors, for each evaluation point $\mathbf{x} = (x^{(1)}, x^{(2)})$, involve the vectors $\mathbf{G}_l, l = L_{\mathbf{A}_\mathbf{x}^{(2)}} + 1, \ldots, L_{\mathbf{b}_\mathbf{x}^{(2)}}$ and the powers of the coordinate vectors of all evaluation points, as follows

$$\begin{aligned}
\tilde{\mathbf{b}}_1 &:= (\mathbf{p}^{(1)})^0 = (\mathbf{p}^{(2)})^0 = \mathbf{G}_{16} \\
\tilde{\mathbf{b}}_2 &:= (\mathbf{p}^{(1)})^1 = \mathbf{G}_{17} - \boldsymbol{X}^{(1)}\mathbf{G}_{16} \\
\tilde{\mathbf{b}}_3 &:= (\mathbf{p}^{(2)})^1 = \mathbf{G}_{18} - \boldsymbol{X}^{(2)}\mathbf{G}_{16} \\
\tilde{\mathbf{b}}_4 &:= (\mathbf{p}^{(1)})^2 = \mathbf{G}_{19} - 2\boldsymbol{X}^{(1)}\mathbf{G}_{17} + (\boldsymbol{X}^{(1)})^2\mathbf{G}_{16} \\
\tilde{\mathbf{b}}_5 &:= (\mathbf{p}^{(1)})(\mathbf{p}^{(2)}) = \mathbf{G}_{20} - \boldsymbol{X}^{(1)}\mathbf{G}_{17} - \boldsymbol{X}^{(2)}\mathbf{G}_{18} + \boldsymbol{X}^{(1)}\boldsymbol{X}^{(2)}\mathbf{G}_{16} \\
\tilde{\mathbf{b}}_6 &:= (\mathbf{p}^{(2)})^2 = \mathbf{G}_{21} - 2\boldsymbol{X}^{(2)}\mathbf{G}_{18} + (\boldsymbol{X}^{(2)})^2\mathbf{G}_{16}.
\end{aligned} \tag{14}$$

Note that in (13) and (14) all the operations are intended component-wise. The algorithm for the computation of matrix $\mathbf{A}$ containing all the $\mathbf{A}_{\mathbf{x}}^{(k)}$, and of the vector $\mathbf{b}$ containing all the $\mathbf{b}_{\mathbf{x}}^{(k)}$, is shown in Algorithm 3. Note that in Algorithm 3 each $\tilde{\mathbf{A}}_l$ is used to compute more entries of the $\mathbf{A}_{\mathbf{x}}^{(k)}$.

Task (e)
Since linear systems (4) are dense and unstructured they can be solved by using common linear algebra solvers. In fact, in our serial implementation, task (e) was implemented by using the LAPACK routines DGETRF and DGETRS from the auto-tuning ATLAS library [40]; DGETRF computes the LU factorization of $\mathbf{A}_{\mathbf{x}}^{(k)}$ with $O(M(m^3/3))$ FLOPs, while DGETRS performs the corresponding triangular solves with $O(M(m^2/2))$ FLOPs.

In Table 2 we summarize the overall computational tasks along with the number of floating point operations required.

3. GPU implementation of the improved SPH

We developed a GPU-based approach to accelerate the improved SPH algorithm. Its scheme is presented in Figure 1.

Three kernels were implemented and are actually essential for efficient parallel execution of the improved SPH algorithm: `BuildGtWeights` to compute the set of weights used for the Gauss transforms computation, `EvaluateGt` to evaluate the Gauss transforms and `BuildSystems` to compute the matrices and rhs vectors. Instead, for the solution of the systems, we made use of the batched dense linear algebra kernels available in the NVIDIA cuBLAS library as already described in [29]. In particular, the factorization of the matrices was performed by using the cuBLAS function `cublasDgetrfBatched` and the solution of the triangular systems by using `cublasDgetrsBatched`.

Algorithm 3: Improved SPH - task (d)

```
Input: $d$, $k$, $M$, $\boldsymbol{X}$, $h$, $\mathbf{d\Omega}$, $\mathbf{G}$
Output: $\mathbf{A}$, $\mathbf{b}$
% compute size of the system (4)
$m := \binom{d+k}{k}$
Compute the vectors $\tilde{\mathbf{A}}_j$, $j = 1, \ldots, L_{\mathbf{A}_{\mathbf{x}}^{(k)}}$ and $\tilde{\mathbf{b}}_j$, $j = 1, \ldots, L_{\mathbf{b}_{\mathbf{x}}^{(k)}}$ as described in
  (13) and (14);
for $i \leftarrow 1$ to $M$ do
    % building the $i$-th coefficient matrix
    $ii = i * m^2$;

    $A_{11}(ii) = \tilde{A}_1(i)$;    $A_{12}(ii) = \tilde{A}_2(i)$;    $A_{13}(ii) = \tilde{A}_3(i)$;
    $A_{21}(ii) = \tilde{A}_2(i)$;    $A_{22}(ii) = \tilde{A}_4(i)$;    $A_{23}(ii) = \tilde{A}_5(i)$;
    $A_{31}(ii) = \tilde{A}_3(i)$;    $A_{32}(ii) = A_{23}(ii)$;    $A_{33}(ii) = \tilde{A}_6(i)$;
    $A_{41}(ii) = -\tilde{A}_1(i) + \frac{2}{h^2}\tilde{A}_4(i)$;    $A_{42}(ii) = -\tilde{A}_2(i) + \frac{2}{h^2}\tilde{A}_7(i)$;    $A_{43}(ii) = -\tilde{A}_3(i) + \frac{2}{h^2}\tilde{A}_8(i)$;
    $A_{51}(ii) = \tilde{A}_5(i)$;    $A_{52}(ii) = \tilde{A}_8(i)$;    $A_{53}(ii) = \tilde{A}_9(i)$;
    $A_{61}(ii) = -\tilde{A}_1(i) + \frac{2}{h^2}\tilde{A}_6(i)$;    $A_{62}(ii) = -\tilde{A}_2(i) + \frac{2}{h^2}\tilde{A}_9(i)$;    $A_{63}(ii) = -\tilde{A}_3(i) + \frac{2}{h^2}\tilde{A}_{10}(i)$;

    $A_{14}(ii) = \frac{1}{2}\tilde{A}_4(i)$;    $A_{15}(ii) = \tilde{A}_5(i)$;    $A_{16}(ii) = \frac{1}{2}\tilde{A}_6(i)$;
    $A_{24}(ii) = \tilde{A}_7(i)$;    $A_{25}(ii) = \tilde{A}_8(i)$;    $A_{26}(ii) = \tilde{A}_9(i)$;
    $A_{34}(ii) = \frac{1}{2}\tilde{A}_8(i)$;    $A_{35}(ii) = \tilde{A}_9(i)$;    $A_{36}(ii) = \frac{1}{2}\tilde{A}_{10}(i)$;
    $A_{44}(ii) = \frac{1}{2}(-\tilde{A}_4(i) + \frac{2}{h^2}\tilde{A}_{11}(i))$;    $A_{45}(ii) = -\tilde{A}_5(i) + \frac{2}{h^2}\tilde{A}_{12}(i)$;    $A_{46}(ii) = \frac{1}{2}(-\tilde{A}_6(i) + \frac{2}{h^2}\tilde{A}_{13}(i))$;
    $A_{54}(ii) = \frac{1}{2}\tilde{A}_{12}(i)$;    $A_{55}(ii) = \tilde{A}_{13}(i)$;    $A_{56}(ii) = \frac{1}{2}\tilde{A}_{14}(i)$;
    $A_{64}(ii) = \frac{1}{2}(-\tilde{A}_4(i) + \frac{2}{h^2}\tilde{A}_{13}(i))$;    $A_{65}(ii) = -\tilde{A}_5(i) + \frac{2}{h^2}\tilde{A}_{14}(i)$;    $A_{66}(ii) = \frac{1}{2}(-\tilde{A}_6(i) + \frac{2}{h^2}\tilde{A}_{15}(i))$;

    % building the $i$-th rhs vector
    $ii = i * m$;
    $b_1(ii) = \tilde{b}_1(i)$;
    $b_2(ii) = \tilde{b}_2(i)$;
    $b_3(ii) = \tilde{b}_3(i)$;
    $b_4(ii) = \tilde{b}_1(i) + \frac{2}{h^2}\tilde{b}_4(i)$;
    $b_5(ii) = \tilde{b}_5(i)$
    $b_6(ii) = \tilde{b}_1(i) + \frac{2}{h^2}\tilde{b}_6(i)$;
end
```

Tasks of the improved SPH	
(a) generate data points	$O(dN + dM)$
(b) compute the weights	$O(LdN)$
(c) evaluate the Gauss transforms	$O(L(M + N))$
(d) compute $\mathbf{A}_{\mathbf{x}}^{(k)}$ and $\mathbf{b}_{\mathbf{x}}^{(k)}$ $, \forall \mathbf{x} \in \boldsymbol{X}$	$O(LdM)$
(e) solve linear systems $\mathbf{A}_{\mathbf{x}}^{(k)} \mathbf{c}_{\mathbf{x}}{}^{(k)} = \mathbf{b}_{\mathbf{x}}^{(k)}$ $, \forall \mathbf{x} \in \boldsymbol{X}$	$O(M(m^3/3 + m^2/2))$

Table 2: Number of floating point operations for each task of the improved SPH. In (b) FLOPs estimate doesn't take into account the complexity of function evaluations.

The scheme 1 highlights only the data transferred from CPU to GPU and vice versa. All the other data relevant to the computation is only allocated on the GPU (**d_w** to store the weights, **d_G** to store the Gaussian transforms and **d_A**, **d_b** to store the system matrices and rhs vectors). We stored source and evaluation data points in contiguous linear arrays in column-major order i.e. data points vectors are split up by coordinates, stored with a spacing of N and M respectively. Thus, accesses of the same coordinate from threads per block (*tpb*) data points to global memory are coalesced (`BuildGtWeights` and `BuildSystems`) and transfer of the same coordinate from *tpb* data points from global memory to shared memory are coalesced (`EvaluateGt`).

Being Algorithm 2 and Algorithm 3 naturally parallel, `BuildGtWeights` and `BuildSystems` implements those algorithms organizing parallel execution by assigning to each thread one source point and one evaluation point respectively. We set *tpb* for them as 512. We didn't experiment too much with their block size being those task computational complexities neglectable, in practise, compared to the one of the Gauss transforms evaluation (see Table 2). Direct evaluation of Gaussian transforms would have an high benefit from GPU implementation. Despite the expected high speedups obtained by a GPU-based approach, the asymptotic dependence on data size is still $O(NM)$. Therefore, a linear approach like `figtree` will anyway, outperform the GPU implementation at some point. Moreover, this latter one will be likely always outperformed by a parallel implementation of an approximation approach. In all the conducted experiments with accuracy ϵ set to 10^{-6}, `figtree` always automatically selected the direct evaluation using tree data structure (*direct+tree* method) [39] that was therefore our choice for the parallel implementation.

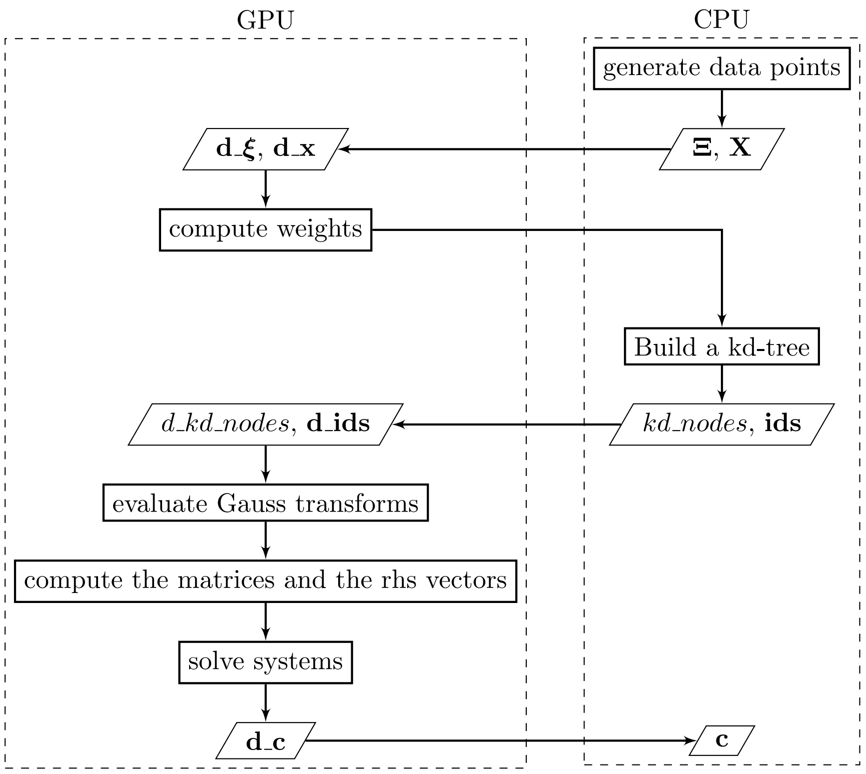

Figure 1: Flowchart of the improved SPH algorithm implementation on GPU

Thus, in this section we give details about our implementation of the *direct+tree* method implemented in `figtree`. This method computes the Gauss transforms by summing only the source points within a certain distance from the evaluation point whose contribution is at least ϵ. At this purpose, a tree data structure (a kd-tree) is used allowing efficient fixed-distance nearest neighbor search ('k' Nearest Neighbors search algorithm [41]).

Our parallel implementation uses a fixed number of nearest neighbor source points ensuring equal computational load on GPU multiprocessors. Conducted experimental tests showed that 32 was a number of neighbor sources enough to guarantee the same accuracy as obtained with `figtree` implementation, with ϵ set to 10^{-6}.

The GPU implementation of the *direct+tree* method is summarized in Algorithm 4. Recursive nature of traditional kd-tree based Nearest Neighbors algorithm makes it not suitable to be implemented on the GPU. Therefore, in our implementation, first, we construct the kd-tree on the source points set on the host (`BuildKDTree`), then we transfer it to the device to allow nearest neighbor search of sources, for different evaluation points, to be executed in parallel.

We used the kd-tree data structure implementation of [42]: a balanced static kd-tree, with height bounded to $\lceil log_2(N) \rceil$, built by setting the cutting plane through the median point of each sub-tree, stored as a left-balanced binary array. Child pointers are directly computed through the index relationship between the array elements (given a node at index i, its left and right children are found at $2i$ and $2i+1$ respectively). This results in a fully minimal kd-tree (kd_nodes), where each kd-node contains just the original points rearranged into a left-balanced binary tree order. We also store a remapping array of size $O(N)$ (**ids**) for converting rearranged node indices back into the original point indices to obtain the final search results.

In our implementation we adopted a parallelization approach analogous to the one used by [43], where the direct method for kernel summation was employed: a chunk of evaluation points is stored in the shared-memory by all threads in a thread-block and each thread is responsible to evaluate a particular evaluation point. Each thread block then finds (`KNNSearch`) the 32 nearest neighbors source points to that evaluation point. The resulting nearest neighbors distances are stored in a fixed size array, in order to get the compiler to likely put it in the register file, which is much faster than local memory. The weights corresponding to each nearest neighbor source point are also stored in a fixed size array. Since it is good to have a number of threads per block that is a multiple of 32, as that is the warp size, and since we wanted all threads in a block to jointly process the entire neighborhood of a single evaluation point, the CUDA thread block size was set to 32. Finally each thread updates the evaluation sum at each evaluation point. We use a local variable for the evaluation sum. Once all the nearest neighbors source points are evaluated, the sum in the register is written back to global memory.

The `KNNSearch` function, that implements the 'k' Nearest Neighbors algorithm, uses a search stack and a trim optimization to avoid unproductive search paths. The neighbors are tracked by a closest heap data structure. Search stack is declared as fixed size data structure, therefore, to prevent overflowing the stack, the length of any kd-tree search path had to be bounded. Bounding the maximum height of the kd-tree accomplishes to this need. For more details about `BuildKDTree` and `KNNSearch` see [42].

We used double precision arithmetic throughout all the computations except for kd-tree data structure and `KNNSearch` whose algorithmic correctness didn't require it.

4. Results and Discussion

We performed numerical experiments with $d = 2$ and $k = 1, 2$, using the four test functions reported in Table 3.

Table 3: functions used in the numerical experiments.

$f_a(x^{(1)}, x^{(2)}) =$	$16\, x^{(1)} x^{(2)} (1 - x^{(1)})(1 - x^{(2)})$
$f_b(x^{(1)}, x^{(2)}) =$	$\tanh \frac{1}{9}(9(x^{(2)} - x^{(1)}) + 1)$
$f_c(x^{(1)}, x^{(2)}) =$	$\frac{1.25 + \cos(5.4\, x^{(2)})}{6 + 6(3\, x^{(1)} - 1)^2}$
$f_d(x^{(1)}, x^{(2)}) =$	$\frac{1}{3}\texttt{exp}\left(-\frac{81}{16}\left(\left(x^{(1)} - \frac{1}{2}\right)^2 + \left(x^{(2)} - \frac{1}{2}\right)^2\right)\right)$

For each distribution of source points as described in Section 2, we set $N = (2^n + 1)^2$ and $h = 1/2^n$, with $n = 4, \ldots, 12$, and $M = (\sqrt{N} + 1)^2$.

The experiments were run on two different hardware configurations: System a) Intel Xeon Gold 5215 2.50GHz CPU with 92 GB of RAM and NVIDIA Quadro RTX600 GPU with 64 cores in 72 streaming multiprocessors (SMs) running at 1.77 GHz; system b) Intel Xeon Gold 5218 2.30GHz with 192 GB of RAM and NVIDIA V100 NVLINK GPU with 64 cores in 80 SMs running at 1.53 GHz. Both systems have the Linux CentOS 7.6 operating system, the GNU 4.8.5 C++ compiler and CUDA 10.1.

On both hardware configurations, we executed the parallel code on the GPU and compared the execution time with the one obtained by the serial C++ version running on one core of the CPU.

Figure 2 and Figure 3 show, for each distribution of source points, the runtime in milliseconds for the execution on the GPU and CPU, as well as the speedup achieved. For each number of source points N and evaluation points $M = (\sqrt{N} + 1)^2$, the time is the average over the runs for all the test functions.

cpu_vs_gpu_Nfixed_uniform_quadrortx6000_2-eps-converted-to.pdf cpu_vs_gpu_Mfixed_uniform_quadrortx6000_2-eps-conv

cpu_vs_gpu_Nfixed_halton_quadrortx6000_2-eps-converted-to.pdf cpu_vs_gpu_Mfixed_halton_quadrortx6000_2-eps-conve

cpu_vs_gpu_Nfixed_sobol_quadrortx6000_2-eps-converted-to.pdf cpu_vs_gpu_Mfixed_sobol_quadrortx6000_2-eps-conver

Figure 2: GPU vs CPU performance comparison with N fixed and varying M and with M fixed and varying N on System a).

With our parallel implementation, we are able to evaluate one million of points in one second, achieving a maximum error accuracy in approximating the function of about 10^{-6} and a maximum error accuracy in approximating the derivative of the function of about 10^{-3}. Overall, we obtained the same results in terms of accuracy as those obtained with the serial implementations (MATLAB, C++ [29]).

Figure 2 shows the efficiency results obtained on configuration a). We see that the speedup of the GPU over the CPU goes up to 80. Figure 3 shows the efficiency results obtained on configuration b). Here the speedup of the GPU over the CPU goes up to 90. Better speedup results on configuration b) are due to the slightly greater number of its GPU SMs compared to configuration a). On both configurations, we can observe a better speedup when sources are not uniformly distributed. Moreover, the lower speedup for smaller values of N and M is due to a poor performance of the batched linear systems solving that almost nullify the gain obtained in the Gaussian transforms evaluation task. Anyway, the GPU-based proposal exhibits an increasing speed-up as M increases with N fixed.

The efficiency results confirm what we asserted in our previous work [29]:

that the improved SPH method could benefit from an implementation on GPU.

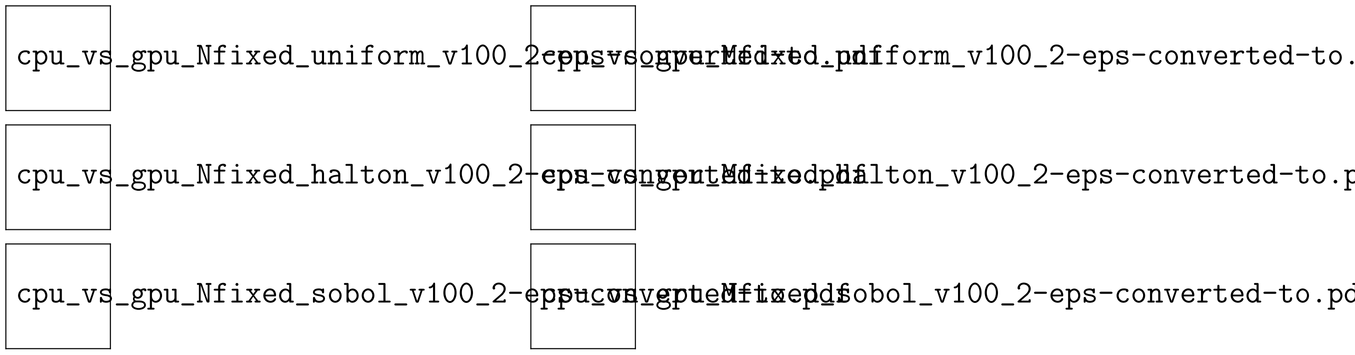
cpu_vs_gpu_Nfixed_uniform_v100_2-eps-converted-to.pdf
cpu_vs_gpu_Mfixed_uniform_v100_2-eps-converted-to.
cpu_vs_gpu_Nfixed_halton_v100_2-eps-converted-to.pdf
cpu_vs_gpu_Mfixed_halton_v100_2-eps-converted-to.p
cpu_vs_gpu_Nfixed_sobol_v100_2-eps-converted-to.pdf
cpu_vs_gpu_Mfixed_sobol_v100_2-eps-converted-to.pd

Figure 3: GPU vs CPU performance comparison with N fixed and varying M and with M fixed and varying N on System b).

5. Conclusions

This paper presents a CUDA-based parallel implementation on GPU of an improved SPH method. The experimental analysis demonstrates the significant acceleration of the GPU implementation compared to its CPU counterpart. It provides a speed up of the execution time of up to 90 times and can evaluate one million of points in one second, achieving a good accuracy order, in approximating the function and its derivatives, with various uniform and non uniform source point sets and different test functions. Dealing with EM problems derivative estimates, at any data location with irregularly data spaced in the problem domain, are frequently required. Satisfactory accuracy results and computational advantages of the proposed method, encourage its use as a building block for implementing parallel applications, capable of solving very large real EM problems, on GPU-based architectures.

6. Acknowledgments

This research has been carried out within RITA (Rete ITaliana di Approssimazione) and was partially supported by the Research INdAM – GNCS Project 2020, *“Multivariate approximation and functional equations for numerical modeling”*.
The authors are grateful to Daniela di Serafino for valuable suggestions and comments on the work presented here.

Algorithm 4: direct+tree on GPU

```
Input: $d, N, M, \Xi, \mathbf{d\_\xi}, h, \mathbf{d\_w}, \mathbf{d\_x}$
Output: $\mathbf{d\_G}$
Sets up the thread blocks for the GPU and the number of nearest
  neighbour ($k, BLOCK\_SIZE$);
Allocates host and device memory for the $k$d-tree structure and
  remapping array;
BuildKDTree($d, \Xi, kd\_nodes$);
Transfers the kd-tree structure and remapping array onto the GPU
  ($d\_kd\_nodes$ and $\mathbf{d\_ids}$);
for $w \leftarrow 1$ to $L$ do
    Invokes the parallel GPU EvaluateGt() kernel to evaluate the
      Gauss transforms;
end

Procedure EvaluateGt()
    $T_i \leftarrow (blockIdx.x * blockDim.x + threadIdx.x)$;
    $i \leftarrow threadIdx.x$;
    declare evalutation sum local variable $gt\_val$, fixed size arrays Dist
      and w respectively for NN distances and weights;
    $gt\_val \leftarrow 0.0$;
    $hSquare \leftarrow h * h$;
    $x\_shared\_x_i \leftarrow d\_x_{T_i}$;
    $x\_shared\_y_i \leftarrow d\_x_{M+T_i}$;
    for $n \leftarrow 1$ to $\lceil k/BLOCK\_SIZE \rceil$ do
        KNNsearch($\mathbf{Dist}, \mathbf{w}, \mathbf{d\_\xi}, \mathbf{d\_w}, k, x\_shared\_x_i,$
          $x\_shared\_y_i, d\_kd\_nodes, \mathbf{d\_ids}, N, w$);
        for $e \leftarrow 1$ to $k$ do
            $gt\_val \leftarrow gt\_val + w_e *$ exp($-Dist_e/hSquare$);
        end
    end
    $d\_G_{w*M+T_i} \leftarrow gt\_val$;
```

References

[1] L. B. Lucy, A numerical approach to the testing of the fission hypothesis., The Astronomical Journal 82 (1977) 1013–1024.

[2] R. A. Gingold, J. J. Monaghan, Smoothed particle hydrodynamics: theory and application to non-spherical stars., Monthly Notices of the Royal Astronomical Society 181 (1977) 375–389.

[3] E. J. Kansa, Y. C. Hon, Circumventing the ill-conditioning problem with multiquadric radial basis functions: Applications to elliptic partial differential equations (1998).

[4] E. Kansa, Exact explicit time integration of hyperbolic partial differential equations with mesh free radial basis functions, Engineering analysis with boundary elements 31 (7) (2007) 577–585.

[5] G. Ala, E. Francomano, An improved smoothed particle electromagnetics method in 3d time domain simulations, Int. J. Numer. Model. 25 (4) (2012) 325–337.

[6] G. Ala, G. E. Fasshauer, E. Francomano, S. Ganci, M. J. McCourt, An augmented MFS approach for brain activity reconstruction, Mathematics and Computers in Simulation 141 (2017) 3 – 15.

[7] X. Chen. J.H. Jung, Matrix Stability of Multiquadric Radial Basis Function Methods for Hyperbolic Equations with Uniform Centers, Journal of Scientific Computing 51 (2012) 683—702.

[8] R. Hardy, Theory and applications of the multiquadric-biharmonic method 20 years of discovery 1968–1988, Computers & Mathematics with Applications 19 (8) (1990) 163 – 208.

[9] Z.-L. Wang, J. Wang, R. Shen, The application of a meshless method to consolidation analysis of saturated soils with anisotropic damage, Computers & Geosciences 34 (7) (2008) 849 – 859.

[10] D. W. Pepper, J. Waters, A local meshless method for approximating 3d wind fields, Journal of Applied Meteorology and Climatology 55 (1) (2016) 163–172.

[11] J. Belinha, Meshless methods in biomechanics-Bone tissue remodelling analysis, Lecture Notes in Computational Vision and Biomechanoics 16, Springer, 2014.

[12] J. Shawe-Taylor, N. Cristianini, Kernel Methods for Pattern Analysis, Cambridge University Press, 2004.

[13] D.J. Duffy, Finite Difference Methods in Financial Engineering: A Partial Differential Approach, 2006.

[14] Y.-C. Hon, X. zhong Mao, A radial basis function method for solving options pricing model, Financial Engineering 8 (1998) 31–49.

[15] B. Scholkopf, A. J. Smola, Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond, MIT Press, Cambridge, MA, USA, 2001.

[16] C.E. Rasmussen, C. Williams, Gaussian Processes for Machine Learning, MIT Press, Cambridge, Massachussetts, 2006.

[17] I. Steinwart, A. Christmann, Support Vector Machines, Information Science and Statistics, Springer, New York, 2008.

[18] E. Francomano, F. M. Hilker, M. Paliaga, E. Venturino, An efficient method to reconstruct invariant manifolds of saddle points, Dolomites Research Notes on Approximation 10 (2017) 25–30.

[19] E. Francomano, F. M. Hilker, M. Paliaga, E. Venturino, Separatrix reconstruction to identify tipping points in an eco-epidemiological model, Applied Mathematics and Computation 318 (2018) 80 – 91, recent Trends in Numerical Computations: Theory and Algorithms.

[20] E. Francomano, M. Paliaga, Detecting tri-stability of 3d models with complex attractors via meshfree reconstruction of invariant manifolds of saddle points, Mathematical Methods in the Applied Sciences 41 (17) (2018) 7450–7458.

[21] S. Li, W.K. Liu, Meshfree Particle Methods, Springer-Verlag, Berlin, 2007.

[22] M. B. Liu, G. R. Liu, Smoothed particle hydrodynamics (sph): an overview and recent developments, Archives of Computational Methods in Engineering 17 (1) (2010) 25–76.

[23] J. S. Chen, T. Belytschko, Meshless and Meshfree Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 886–894.

[24] G. Ala, G. Di Blasi, E. Francomano, A numerical meshless particle method in solving the magnetoencephalography forward problem, Int. J. Numer. Model. 25 (5–6) (2012) 428–440.

[25] G. Ala, E. Francomano, A multi-sphere particle numerical model for non-invasive investigations of neuronal human brain activity, Progress in Electromagnetics Research Letters 36 (2013) 143–153.

[26] G. Ala, E. Francomano, Numerical investigations of an implicit leapfrog time-domain meshless method, J. Sci. Comput. 62 (3) (2015) 898–912-

[27] R. Couturier: Designing Scientific Applications on GPUs, Chapman & Hall/CRC (2013).

[28] V. Kindratenko: Numerical Computations with GPUs, Springer Publishing Company, Incorporated (2014).

[29] L. Antonelli, D. di Serafino, E. Francomano, F. Gregoretti, M. Paliaga, Towards an efficient implementation of an accurate sph method, in: Y. D. Sergeyev, D. E. Kvasov (Eds.), Numerical Computations: Theory and Algorithms, Springer International Publishing, Cham, 2020, pp. 3–10.

[30] NVIDIA, NVIDIA Volta Architecture Whitepaper. http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[31] NVIDIA, NVIDIA Turing Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[32] E. Francomano, M. Paliaga, Highlighting numerical insights of an efficient SPH method, Applied Mathematics and Computation 339 (C) (2018) 899–915.

[33] J. H. Halton, On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals, Numer. Math. 2 (1) (1960) 84–90.

[34] I. Sobol', On the distribution of points in a cube and the approximate evaluation of integrals, USSR Computational Mathematics and Mathematical Physics 7 (4) (1967) 86 – 112.

[35] Burkhardt, J.: https://github.com/cenit/jburkardt/tree/master/halton and https://cenit/jburkardt/tree/master/sobol.

[36] L. Greengard, J. Strain, The Fast Gauss Transform, SIAM J. Sci. Stat. Comput. 12 (1) (1991) 79–94.

[37] D. Lee, A. Gray, A. Moore, Dual-tree Fast Gauss Transforms, in: Proceedings of the 18th International Conference on Neural Information Processing Systems, NIPS'05, MIT Press, Cambridge, MA, USA, 2005, p. 747–754-

[38] V. C. Raykar, R. Duraiswami, The improved fast Gauss transform with applications to machine learning, Large Scale Kernel Machines (2007) 175–201.

[39] V. I. Morariu, B. V. Srinivasan, V. C. Raykar, R. Duraiswami, L. S. Davis, Automatic online tuning for fast gaussian summation, in: D. Koller, D. Schuurmans, Y. Bengio, L. Bottou (Eds.), Advances in Neural Information Processing Systems 21, Curran Associates, Inc., 2009, pp. 1113–1120. See also https://github.com/vmorariu/figtree.

[40] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimizations of software and the ATLAS project, Parallel Computing 27 (1) (2001) 3 – 35, new Trends in High Performance Computing.

[41] N. S. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, The American Statistician 46 (3) (1992) 175–185.

[42] Brown, S.: Case Studies on Optimizing Algorithms for GPU Architectures. Chapel Hill, NC: University of North Carolina at Chapel Hill Graduate School. (2015). https://doi.org/10.17615/jybr-f558

[43] B. Srinivasan, Q. Hu, R. Duraiswami, Gpuml: Graphical processors for speeding up kernel machines, Workshop on High Performance Analytics-Algorithms, Implementations, and Applications (2010).