**The Journal of Supercomputing manuscript No.**
(will be inserted by the editor)

# IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods. Image denoising as case study

**Ruben Laso · José C. Cabaleiro · Francisco F. Rivera · M. Carmen Muñiz · José A. Álvarez-Dios**

Received: date / Accepted: date

**Abstract** Iterative and time-step methods are spread far and wide in several mathematics and physics domains. At the same time, modern computers include multicore CPUs along with GPUs, so it is important to use all their computing capabilities for their efficient use. Aiming to improve performance of this kind of numerical methods, we introduce in this work a new heterogeneous parallelism CPU+GPU scheme which we call IHP. This new scheme has the advantage of being self-balanced and able to dynamically distribute the workload between CPU and GPU according to their performance on the fly. Also, it can be used with several contending technologies, like CUDA and OpenCL for GPUs or OpenMP and Intel TBB for CPUs. As a case in point, we analyse an image denoising problem based on time-step diffusion methods for brightness and chromaticity. Results show execution significant improvements in execution time using this scheme, with a minimal overhead.

**Keywords** heterogeneous parallelism, iterative methods, time-step methods, image processing, finite differences

## 1 Introduction

Heterogeneous parallelism, defined as the combination of both CPU and GPU capabilities to solve a problem, has become a hot topic in high performance computing (HPC) as performance requirements are becoming more and more demanding. Scientific software, mostly based on finite element and finite difference methods, is a common playground for researchers to explore new par-

Ruben Laso · José C. Cabaleiro · Francisco F. Rivera
CiTIUS, Universidade de Santiago de Compostela, Spain,
E-mail: {r.laso,jc.cabaleiro,ff.rivera}@usc.es

M. Carmen Muñiz · José A. Álvarez-Dios
Dep. Matemática Aplicada, Universidade de Santiago de Compostela, Spain,
E-mail: {mcarmen.muniz,joseantonio.alvarez.dios}@usc.es

allel and optimisation techniques and recently released hardware. The use of graphics processing units (GPUs) has widely spread, alongside graphics computation, as they provide a good performance with lower energy consumption. This tendency has grown and evolved into what is known as general purpose computing on GPUs (GPGPU).

Two main technologies and languages have raised as standards for GPGPU, CUDA [16] for NVIDIA devices and OpenCL [23] as an industry standard for general multi- and manycore processors. With time, CUDA has held as the *de facto* standard for NVIDIA GPUs due to the strong tie between software and hardware, usually offering better performance than OpenCL.

Although GPUs can outperform CPUs in highly parallel problems, using heterogeneous parallelism can help to exploit all computational capabilities of modern computers, improving performance. This is specially important when double precision registers are used, scenario in which GPUs struggle more than CPUs in keeping their performance since, traditionally, GPUs have been working with single precision operands.

GPGPU and heterogeneous parallelism have been used in multiple problems and applications in mathematics domain. To mention but a few, in the works of Komatitsch et al. [9] and Cecka et al. [1], finite element methods are accelerated using GPUs, while Markall et al. [12] focus on both GPU and CPU. Papadrakakis et al. [19] introduced a heterogeneous approach applied to domain decomposition methods. An implementation of real-time 3D fluid dynamics based on lattice Boltzmann simulations running on GPUs was introduced in [8]. Relatedly, Feichtinger et al. [3] introduced a heterogeneous scheme for this kind of simulations. Finite difference methods have also been considered in the literature, like in the work of Micikevicius et al. [13], where a parallel implementation for 3D finite difference is proposed. Also, Shams and Sadeghi [22] deal with the optimisation of finite difference time-domain problems on heterogeneous clusters with remarkable results.

Concerning general libraries for heterogeneous computing, some important contributions should be cited like LogFit [15, 25], used in this work to compare our proposal, which combines Intel TBB [21] and OpenCL, thus providing a general purpose library. Another interesting proposal has been introduced in [7], named Concord, focusing on regular problems. We will use this library to contrast it to our scheme. Also, Maat [20] introduces an interesting framework for simplifying heterogeneous computing, but this project has been discontinued. HPL [26] exploits heterogeneous parallelism using a language embedded in C++ and using OpenCL. Finally, Mittal and Vetter [14] present a very extensive survey about heterogeneous computing in several domains.

In this work we present a high performance working scheme using heterogeneous parallelisation, which adjusts and balances the workload dynamically according to CPU and GPU performance in execution time. Our proposal is specifically devoted to time-step or iterative methods. This kind of methods are widely used in several mathematics and physics domains, so we consider it is worth to develop a heterogeneous parallel scheme that specifically fits them. We have used an image denoising problem as case study, as it is an easy-to-

understand problem which can exemplify the principles of our heterogeneous strategy. Note that our proposal is adaptable to other iterative or time step methods, not only to those related in the image processing domain.

## 2 IHP: Iterative Heterogeneous Parallelism

This work introduces the IHP (Iterative Heterogeneous Parallelism) working scheme. The objective of IHP is to use all the computational capabilities of a computer, sharing out the work between CPU and GPU. It supports multiple technologies like CUDA and OpenCL for GPU kernels, and OpenMP [2] or Intel TBB for CPU computations. This scheme is designed to be used for iterative or time-step methods, which are very common in several domains of physics and mathematics. This kind of methods follows the structure shown in Algorithm 1.

**Algorithm 1** Iterative or time-step methods.

**Input:** Generic function $\mathbf{u}(x, y, t)$.
Generic function $\mathbf{f}(\mathbf{u}, t)$.
Function $\mathbf{s}(\mathbf{u})$ used in the stopping criterion.
Parameter $k_{\max}$ defining the maximum number of iterations or time-steps.
**Output:** Solution $\mathbf{u}(x, y, t_k)$.

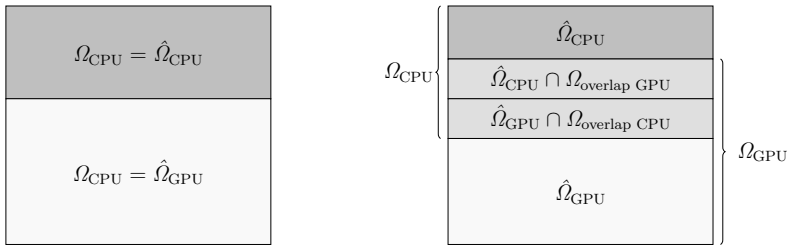
1: **for** $k = 1$ **to** $k_{\max}$ **do**
2: $\quad \mathbf{u}(x, y, t_k) = \mathbf{f}(\mathbf{u}(x, y, t_{k-1}), t_{k-1})$
3: $\quad$ **if** $\mathbf{s}(\mathbf{u}(x, y, t_k))$ meets stopping criterion **then**
4: $\quad\quad$ **return** $\mathbf{u}(x, y, t_k)$
5: **return** $\mathbf{u}(x, y, t_k)$

IHP splits the full problem domain $\Omega$ between the CPU and the GPU. Formally, two domains, $\Omega_{\text{CPU}}$ and $\Omega_{\text{GPU}}$, are defined such that $\Omega = \Omega_{\text{CPU}} \cup \Omega_{\text{GPU}}$. Additionally, another two domains, $\hat{\Omega}_{\text{CPU}}, \hat{\Omega}_{\text{GPU}} \subseteq \Omega$, are defined such that $\hat{\Omega}_{\text{GPU}} := \Omega \setminus \hat{\Omega}_{\text{CPU}}$, ruled by a given parameter $\alpha \in [0, 1]$ representing the amount of work to be done by the CPU. For example, if $\alpha = 0.3$, $\hat{\Omega}_{\text{CPU}}$ will contain a 30% of the total domain and $\hat{\Omega}_{\text{GPU}}$ the remaining 70% of $\Omega$.

Note that approximations are generally implemented differently at the boundaries, with higher numerical error, so overlap regions can be included to reduce this errors. These overlap regions are defined as $\Omega_{\text{overlap CPU}} \subset \hat{\Omega}_{\text{GPU}}$ and $\Omega_{\text{overlap GPU}} \subset \hat{\Omega}_{\text{CPU}}$. Therefore, the domains computed by CPU and GPU are $\Omega_{\text{CPU}} = \hat{\Omega}_{\text{CPU}} \cup \Omega_{\text{overlap CPU}}$ and $\Omega_{\text{GPU}} = \hat{\Omega}_{\text{GPU}} \cup \Omega_{\text{overlap GPU}}$, respectively. Figure 1 shows a simple example of a possible domain split.

For the initial value of $\alpha$, a relation between theoretical peak performances of CPU and GPU, $P_{\text{CPU}}$ and $P_{\text{GPU}}$, respectively, is proposed such that

$$\alpha = \frac{P_{\text{CPU}}}{P_{\text{GPU}} + P_{\text{CPU}}}. \tag{1}$$




(a) Without overlap. (b) With overlap.

Fig. 1: Representation of the domain division between CPU and GPU.

This initial guess for $\alpha$ is likely not optimal, a recalculation being necessary according to CPU and GPU performances on the fly. Considering that tasks are based on iterative or time-step methods, it is possible to establish a dynamic mechanism to improve the workload balance.

The main idea of our proposal is to compute chunks of iterations and establish a new value for $\alpha$ according to

$$\alpha = \frac{t_{\text{GPU}}}{t_{\text{CPU}} + t_{\text{GPU}}}, \tag{2}$$

where $t_{\text{CPU}}$ and $t_{\text{GPU}}$ are the time required by the CPU and the GPU, respectively, to complete their computations for the chunk. Note that IHP considers a linear performance model, where computation time grows up linearly with the amount of work. Also, it should be mentioned that these execution times include expensive communications between CPU and GPU.

Under ideal conditions and a constant workload through time, $\alpha$ will converge fast to an optimal value. Although, in real world, workload may vary or other processes may be assigned to the CPU or the GPU, affecting performance, and $\alpha$ consequently. Usually, these situations do not carry drastic changes in performance, so $\alpha$ should slightly oscillate around the optimal value.

Some details about precision and possible induced errors of this algorithm will be discussed. For some problems, like diffusion methods, results in a point of the domain affect the rest of points. Splitting the domain in subdomains cancels the influence of the points of each subdomain over the rest, which carries a certain error that increases with the number of iterations.

To recover this "influence" and reduce the induced errors of the domain split, two options are available. On one hand, by overlapping regions we could retrieve part of the mentioned influence, so that the bigger the region, the more precise the obtained results, but the drawback lies in more expensive communications between CPUs and GPU. On the other hand, smaller chunks would help us synchronise the results of each subdomain. In the limit case, with chunks of 1 iteration, results would be exactly the same as those obtained with a traditional parallelisation scheme, with the downside of more communications between CPU and GPU being performed. Depending on the

problem, and its sensitiveness to numerical errors, a trade-off between these two techniques should be found.

Algorithm 2 shows the pseudocode of our proposal. Note that steps 6 and 7 should be computed simultaneously, so CPU and GPU should make their respective computations concurrently.

**Algorithm 2** Heterogeneous parallelism for iterative problems.

**Input:** Initial value problem functions $\mathbf{u}(x, y, t)$ and $\mathbf{f}(\mathbf{u}, t)$.
Function $\mathbf{s}(\mathbf{u})$ for evaluating stopping criterion.
Parameter $k_{\max}$ defining the maximum number of iterations.
Initial value for parameter $\alpha$.
Minimum and maximum number of iterations per chunk $c_{\max}$ and $c_{\min}$.
**Output:** Function $\mathbf{u}(x, y, t_k)$ corresponding to the result.

1: $i = 0, k = 1$
2: **while** $k < k_{\max}$ **do**
3: Define $\Omega_{\text{CPU}}$ and $\Omega_{\text{GPU}}$ such that $\text{card}(\Omega_{\text{CPU}}) \approx \alpha\, \text{card}(\Omega_{\text{GPU}})$
4: $c_{\text{chunk}} = \min(2^i c_{\min}, c_{\max})$
5: **for** $c = 0$ **to** $c_{\text{chunk}}$ **do**
6: $\mathbf{u}(x, y, t_{k+c}) = \mathbf{f}(\mathbf{u}(x, y, t_{k+c-1}), t_{k+c-1}), \quad (x, y) \in \Omega_{\text{CPU}}$
7: $\mathbf{u}(x, y, t_{k+c}) = \mathbf{f}(\mathbf{u}(x, y, t_{k+c-1}), t_{k+c-1}), \quad (x, y) \in \Omega_{\text{GPU}}$
8: $k = k + c_{\text{chunk}}$
9: $i = i + 1$
10: **if** $\mathbf{s}(\mathbf{u}(x, y, t_k))$ meets stopping criterion **then**
11: **return** $\mathbf{u}(x, y, t_k)$
12: $\alpha = \frac{t_{\text{CPU}}}{t_{\text{GPU}} + t_{\text{CPU}}}$
13: **return** $\mathbf{u}(x, y, t_k)$

Some functions have to be implemented in order to use IHP.

- `allocate_memory()`: function to allocate the memory on the device.
- `deallocate_memory()`: function to deallocate the reserved memory in function `allocate_memory()`.
- `send_to_GPU()`: code to copy data from host memory to device memory.
- `send_to_CPU()`: code to copy data from device memory to host memory.
- `CPU_operator()`: code for host computations involving $\Omega_{\text{CPU}}$.
- `GPU_operator()`: code for device computations involving $\Omega_{\text{GPU}}$.
- `stopping_criteria()`: code for function $\mathbf{s}(\mathbf{u})$ to check if the stopping criteria is met.

Overlap is taken into account by the library itself during computation and communication related functions. Although, user has to decide how $\Omega_{\text{CPU}}$ and $\Omega_{\text{GPU}}$ are defined, as IHP intends to be a general purpose library, not just focused on image processing.

Source code for IHP and simple example codes can be found in [11], also code used in this paper can be found in [10].

## 3 Image denoising

In this section, the mathematical problem in the frame of image denoising is introduced. In particular, this work uses diffusion methods, like the ones proposed in [24], as they provide an easy-to-understand example and a good case study for the heterogeneous parallelism scheme introduced in this paper.

Any digital image is described as a function $\mathbf{I}(x, y) : \mathbb{R}^2 \to \mathbb{R}^c$, where $c$ corresponds to the number of channels (typically $c = 3$ for the common RGB –Red Green Blue– colour representation). Also, the image domain $\Omega = (0, L_x) \times (0, L_y) \subset \mathbb{R}^2$ is defined, where $L_x$ and $L_y$ are the dimensions of the image in the $x$ and $y$ axis, respectively.

Diffusion methods proposed in [24] are a set of coupled partial differential equations (PDEs), where a digital image is considered a combination of brightness (magnitude) and chromaticity (direction). So, an image is defined as $\mathbf{I}(x, y, t) = \mathbb{R}^2 \times [0, \tau) \to \mathbb{R}^c$, where $(x, y) \in \Omega$ are the spatial coordinates and $t$ represents time. Therefore, any image can be defined as the combination of its magnitude, $M(x, y, t) : \mathbb{R}^2 \times [0, \tau) \to \mathbb{R}$, and its unit direction, $\mathbf{D}(x, y, t) : \mathbb{R}^2 \times [0, \tau) \to \mathbb{R}^c$. These functions can be written as:

$$M(x, y, t) = \left( \sum_{i=1}^{c} I_i(x, y, t)^2 \right)^{\frac{1}{2}} = \|\mathbf{I}(x, y, t)\|, \tag{3}$$

$$\mathbf{D}(x, y, t) = \frac{1}{M(x, y, t)} \mathbf{I}(x, y, t). \tag{4}$$

In the denoising method, certain diffusion flows are applied to brightness and chromaticity, respectively, so the image evolves through time.

### 3.1 Brightness diffusion flows

For the brightness, there are two types of diffusion flows. First, the common Laplacian flow, based on the heat equation,

$$\frac{\partial M}{\partial t}(x, y, t) = M_{xx}(x, y, t) + M_{yy}(x, y, t) = \Delta M(x, y, t), \tag{5}$$

where the subscripts denote partial derivatives. As this is an isotropic flow, the behaviour will be the same on every direction, and as a consequence, it is not capable of preserving the edges of the objects of the image. Second, the more refined anisotropic flow, that is introduced as (see [24] and references therein)

$$\frac{\partial M}{\partial t}(x, y, t) = \frac{\left( M_{xx} M_y^2 - 2 M_x M_y M_{xy} + M_{yy} M_x^2 \right)^{\frac{1}{3}}}{1 + \|\nabla M\|}, \tag{6}$$

where the anisotropic behaviour is introduced by the term $1/(1 + \|\nabla M\|)$. In the edges of objects in the image, the norm of the gradient will grow up, stopping diffusion in these areas.

### 3.2 Chromaticity diffusion

The chromaticity is defined in (4) and represents an unit vector field that defines the direction of the colour for a given pixel. The $i$-th component of $\mathbf{D}(x, y, t)$ is denoted as $D_i(x, y, t) : \mathbb{R}^2 \times [0, \tau) \to \mathbb{R}$, with $i = 1, \ldots, c$, and indicates how much that vector is drawn to a specific colour (red, green or blue in RGB codification).

Diffusion flows for chromaticity come from solutions of Problem 1 (see [24]).

**Problem 1** Find $\mathbf{D}(x, y, t) : \mathbb{R}^3 \to \mathbb{R}^c$, such that minimises:

$$\min_{\mathbf{D}:\mathbb{R}^3 \to \mathbb{R}^c} \int_{\Omega} \|\nabla \mathbf{D}(x, y, t)\|^p \, \mathrm{d}x \, \mathrm{d}y, \quad t \in [0, \tau), p \geq 1, \tag{7}$$

subject to $\|\mathbf{D}(x, y, t)\| = 1, \quad \forall (x, y) \in \Omega, t \in [0, \tau)$.

It can be proved that the gradient descent flow of (7) has the form

$$\frac{\partial D_i}{\partial t} = \operatorname{div}\left(\|\nabla \mathbf{D}\|^{p-2} \nabla D_i\right) + D_i \|\nabla \mathbf{D}\|^p, \quad 1 \leq i \leq c.$$

In this work, only two values of $p$ are considered. The first one is $p = 2$, in which the isotropic diffusion flow is obtained by

$$\frac{\partial D_i}{\partial t} = \Delta D_i + D_i \|\nabla \mathbf{D}\|^2, \quad 1 \leq i \leq c. \tag{8}$$

The second value is $p = 1$, obtaining the anisotropic flow for the chromaticity:

$$\frac{\partial D_i}{\partial t} = \operatorname{div}\left(\|\nabla \mathbf{D}\|^{-1} \nabla D_i\right) + D_i \|\nabla \mathbf{D}\|, \quad 1 \leq i \leq c. \tag{9}$$

### 3.3 Discretisation

Taking image structure into account, an uniform mesh with $N+2$ points in the $x$ axis and $M+2$ in the $y$ axis is proposed, so the distance between two points in the mesh for each axis is $h = h_x = L_x/(N+1) = h_y = L_y/(M+1) = 1$. This mesh is denoted as $\Omega_h$ and it is defined as the set of nodes

$$\Omega_h = \{(x_i, y_i), 1 \leq i \leq N, 1 \leq j \leq M\},$$

and its boundary domain is

$$\partial\Omega_h = \{(0, y_j), (L_x, y_j), 0 \leq j \leq M+1\} \cup \{(x_i, 0), (x_i, L_y), 0 \leq i \leq N+1\}.$$

Therefore, domain $\Omega$ is represented by the set of nodes $(x_i, y_j) = (ih_x, jh_y)$, $i = 0, \ldots, N+1$, $j = 0, \ldots, M+1$.

Additionally, a discretisation of time is required in these kind of time-step methods, so a time step $\delta$ is defined, such that $k$-th instant is $t_k = k\delta$ for $k = 0, \ldots, k_{\max}$.

Finally, in this paper, the following notation is used

$$(x_i, y_j, t_k) = (ih_x, jh_y, k\delta)\,, \quad i = 1, \ldots, N, j = 1, \ldots, M, k = 1, \ldots, k_{\max},$$
$$\mathbf{u}\,(x_i, y_j, t_k) \simeq \mathbf{u}_{i,j,k}, \quad i = 1, \ldots, N, j = 1, \ldots, M, k = 1, \ldots, k_{\max},$$

for any given function $\mathbf{u}(x, y, t)$.

Concerning initial conditions, it is considered that $u(x, y, 0) = M(x, y, 0)$ and $\mathbf{u}(x, y, 0) = \mathbf{D}(x, y, 0)$ for brightness and chromaticity, respectively. For boundary conditions $\mathbf{u}(x, y, 0) = \mathbf{0}$ for every $(x, y) \in \partial\Omega_h$.

### 3.4 Numerical solution

For computing the numerical solution of the partial differential equations (5), (6), (8), and (9), finite difference schemes are used for the spatial derivatives and an Explicit Euler method for temporal derivatives. For first order spatial derivatives, centred differences are used whenever is possible. Note that, in the points $(x_i, y_j)$ with $i = \{1, N\}$ or $j = \{1, M\}$, a different approximation should be implemented, by using forward differences when $i = 1$ or $j = 1$ and backward differences when $i = N$ or $j = M$. Second order spatial derivatives are computed using centred differences in all cases, considering that $\mathbf{u}_{i,j,k} = 0$ if $(x_i, y_j) \notin \Omega_h$.

Temporal derivatives are computed using an Explicit Euler scheme,

$$\begin{cases} \mathbf{u}(x, y, t_0) = \mathbf{u}(x, y, 0), \\ \mathbf{u}(x, y, t_k) = \mathbf{u}(x, y, t_{k-1}) + \delta\mathbf{f}(\mathbf{u}(x, y, t_{k-1}), t_{k-1}), \quad k = 1, \ldots, k_{\max}, \end{cases}$$

where, $\mathbf{u}(x, y, t)$ denotes brightness or chromaticity function and $\mathbf{f}(\mathbf{u}, t)$ represents the corresponding isotropic or anisotropic flow function.

Additionally, the integral in (7) provides an estimation of the amount of noise of an image, and it can be considered as an energy measure. Then, the following function is defined:

$$E(\mathbf{u}) = \int_{\Omega} \|\nabla\mathbf{u}(x, y, t)\|^p \,\mathrm{d}x\,\mathrm{d}y, \tag{10}$$

hereinafter referred to as the image energy. As the noise alters the values of the pixels, the norm of the gradient increases and, accordingly, the energy. In this work, a general value of $p = 2$ has been used in (10), preserving the case with $p = 1$ for the anisotropic flow for chromaticity. Note that $p = 2$ makes a simplification possible, where the square root of the Euclidean norm and the quadratic power cancel each other, saving a large number of computations.

Furthermore, computing (10) involves a discretisation, so the integral of $\|\nabla\mathbf{u}(x, y, t)\|^p$ is approximated in each rectangle of $\Omega_h$ considering the trapezium rule in each iterated integral.

The values of the energy of the image are used as a stopping criterion, so the diffusion flow stops once the following condition is fulfilled

$$\frac{E(\mathbf{u}(x,y,t_k))}{E(\mathbf{u}(x,y,t_0))} \leq \rho, \quad k \in [0, k_{\max}], \rho \in (0,1).$$

Therefore, the final time $\tau$ is $t_k$ for the first $t_k$ that fulfils this condition. As the energy value is only used as a guide to stop the method, a maximum precision is not really needed, while a low computation time is preferable. Additionally, in order to save computation time, it is not necessary to compute the energy in every iteration, as it holds

$$\lim_{t\to\infty} E(\mathbf{u}(x,y,t)) = 0,$$

decreasing fast when $t$ is close to 0, but slowing down as $t$ increases.

### 3.5 Definitions of IHP domains

As mentioned in section 2, IHP splits the domain $\Omega$ in several subdomains in order to balance the workload and reduce numerical errors. In the case of image denoising, the split of the domain is almost straightforward, as it can be divided by rows. Formally, the subdomains of $\Omega_h$ are defined like

$$\begin{aligned}
\hat{\Omega}_{\text{CPU}} &= \{(x_i, y_j), 1 \leq i \leq N, 1 \leq j \leq r_{\text{split}}\}, \\
\Omega_{\text{overlap CPU}} &= \{(x_i, y_j), 1 \leq i \leq N, r_{\text{split}} < j \leq r_{\text{split}} + r_{\text{overlap}}\}, \\
\hat{\Omega}_{\text{GPU}} &= \{(x_i, y_j), 1 \leq i \leq N, r_{\text{split}} < j \leq M\}, \\
\Omega_{\text{overlap GPU}} &= \{(x_i, y_j), 1 \leq i \leq N, r_{\text{split}} - r_{\text{overlap}} \leq j \leq r_{\text{split}}\},
\end{aligned}$$

where $r_{\text{split}} = \lfloor \alpha M \rfloor$ denotes the row where the domain is split, and $r_{\text{overlap}}$ denotes the number of rows that define the overlap region.

Concerning the overlap region, the anisotropic flow for chromaticity is the diffusion flow that needs more points in the calculations. For computing the divergence of the points in the $i$-th row, the gradients of the points in the rows $i-1$ and $i+1$ are needed. These gradients are computed with the values of the points in the rows $i-2$, $i$, and $i$, $i+2$ respectively. Given that, $r_{\text{overlap}} = 2$.

## 4 Results

To validate the proposed methodology, several tests have been performed in a system with an Intel Core i5 7600 [6], with 4 cores and using 4 threads, and a NVIDIA GTX 1050 Ti [18], with 768 NVIDIA CUDA cores, 16 GB of DDR3 memory with Ubuntu 16.04.01 (Linux kernel 4.15.0). As our proposal may be used with several technologies, we show results for two specific implementations, one using CUDA and OpenMP, and other using OpenCL alongside Intel TBB. Compilers GCC 7.4.0 [4] and NVCC V10.0.130 [17] were used, with

the highest optimisation options enabled. In order to compare our proposal, tests have been replicated using the general purpose heterogeneous libraries LogFit and Concord, both using Intel TBB and OpenCL.

Concerning diffusion methods parameters, 1000 iterations have been computed, for both brightness and chromaticity, with $c_{\min} = 5$ and $c_{\max} = 50$. According to (1), initially, $\alpha = 0.03$. Also, the overlap region in our experiments is two rows in the tests. This is the smallest overlap region size that allows to use centred differences for the approximations in our implementation.

Results in this section come from the average of 5 independent executions of the image denoising algorithm with an image of $4000 \times 6016$ pixels.

Tables 1 and 2 show the results of IHP, using CUDA and OpenMP, compared to the CUDA only implementation.

Table 1: Execution times (in seconds) of CUDA only and IHP implementations using single precision.

| | | Brightness | | Chromaticity | |
|---|---|---|---|---|---|
| | | Isotropic | Anisotropic | Isotropic | Anisotropic |
| CUDA | | **4.29** | **5.61** | **12.54** | **38.51** |
| IHP | CPU | 3.32 | 5.73 | 10.75 | 36.43 |
| | GPU | 3.34 | 5.61 | 10.81 | 36.52 |
| | Copy | 0.02 | 0.02 | 0.05 | 0.04 |
| | Total | **3.45** | **5.82** | **11.11** | **36.72** |
| Improvement | | **19.54%** | **-3.66%** | **11.42%** | **4.64%** |

Table 2: Execution times (in seconds) of CUDA only and IHP implementations using double precision.

| | | Brightness | | Chromaticity | |
|---|---|---|---|---|---|
| | | Isotropic | Anisotropic | Isotropic | Anisotropic |
| CUDA | | **4.84** | **30.10** | **48.62** | **151.17** |
| IHP | CPU | 4.41 | 27.70 | 31.14 | 117.90 |
| | GPU | 4.15 | 27.68 | 31.52 | 118.59 |
| | Copy | 0.04 | 0.03 | 0.10 | 0.10 |
| | Total | **4.29** | **28.13** | **31.94** | **119.36** |
| Improvement | | **11.36%** | **6.55%** | **34.29%** | **21.03%** |

IHP succeeds in balancing the workload with low overhead, improving execution times as a consequence. When using single precision operation, where generally GPUs outperform CPUs, IHP improves up to 54.50% execution times. Concerning double precision operations, GPU struggle more than CPU to keep performance. As consequence, our heterogeneous approach gives a higher amount of work to the CPU, resulting in a improvement up to a 39.8%.

The cases where anisotropic diffusion flows are used are specially interesting as the values of $\alpha$ go close to zero. In these scenarios, the CPU can only handle between 2% to 4% of the total workload in the same time that the GPU handles the remaining 98% to 96%, respectively. The worst result comes in the computation of brightness with the anisotropic diffusion flow in single precision, where $\alpha$ tends to $\alpha = 0.02$ and the use of IHP only causes a degradation in performance of 0.2 seconds. According to this result, we can say that IHP introduces a minimal overhead. Similar improvements have been achieved with OpenCL beside OpenMP implementations.

Tables 3 and 4 show the results of the comparison of IHP (using OpenCL and Intel TBB) with LogFit and Concord.

Table 3: Execution times (in seconds) of OpenCL only, LogFit, Concord and IHP implementations using single precision.

| | | Brightness | | Chromaticity | |
|---|---|---|---|---|---|
| | | Isotropic | Anisotropic | Isotropic | Anisotropic |
| OpenCL | | **14.00** | **18.42** | **40.70** | **81.56** |
| LogFit | CPU | 9.01 | 25.81 | 54.62 | 112.85 |
| | GPU | 6.67 | 23.29 | 36.70 | 97.81 |
| | Copy | 12.02 | 21.60 | 59.74 | 61.49 |
| | Total | **22.90** | **47.82** | **100.13** | **171.79** |
| Concord | CPU | 12.98 | 127.48 | 70.42 | 154.78 |
| | GPU | 3.32 | 4.04 | 8.74 | 20.62 |
| | Copy | 7.37 | 3.73 | 15.61 | 14.68 |
| | Total | **16.65** | **131.04** | **79.34** | **165.58** |
| IHP | CPU | 5.78 | 17.12 | 32.73 | 68.44 |
| | GPU | 6.12 | 16.65 | 32.38 | 69.30 |
| | Copy | 0.18 | 0.23 | 0.30 | 0.26 |
| | Total | **6.37** | **17.84** | **33.99** | **70.15** |

The main differences between IHP and the other libraries lay in two details. First, IHP requires less data swapping between CPU and GPU. While LogFit and Concord spend several seconds in the best cases, IHP just spends up to 0.62 seconds in copy operations. This behaviour was expected as LogFit and Concord are general purpose libraries, while IHP is specifically designed to address iterative problems. Second, a much more accurate workload balance is found with our proposal, so devices spend less time idle. The difference between CPU and GPU execution time reaches the 5% in the worst case for IHP. For LogFit and Concord, non optimal workload balances cause big differences between execution times, leaving GPU computational resources unused for long periods. This is specially critical in Concord, that does not change the workload balance after profiling phase, while LogFit continuously tries to find the optimal balance using a mechanism divided in several phases.

Figure 2 shows the evolution of parameter $\alpha$ throughout the execution for IHP, LogFit, and Concord. Despite $\alpha$ is a parameter of IHP, it can be

Table 4: Execution times (in seconds) of OpenCL only, LogFit, Concord and IHP implementations using double precision.

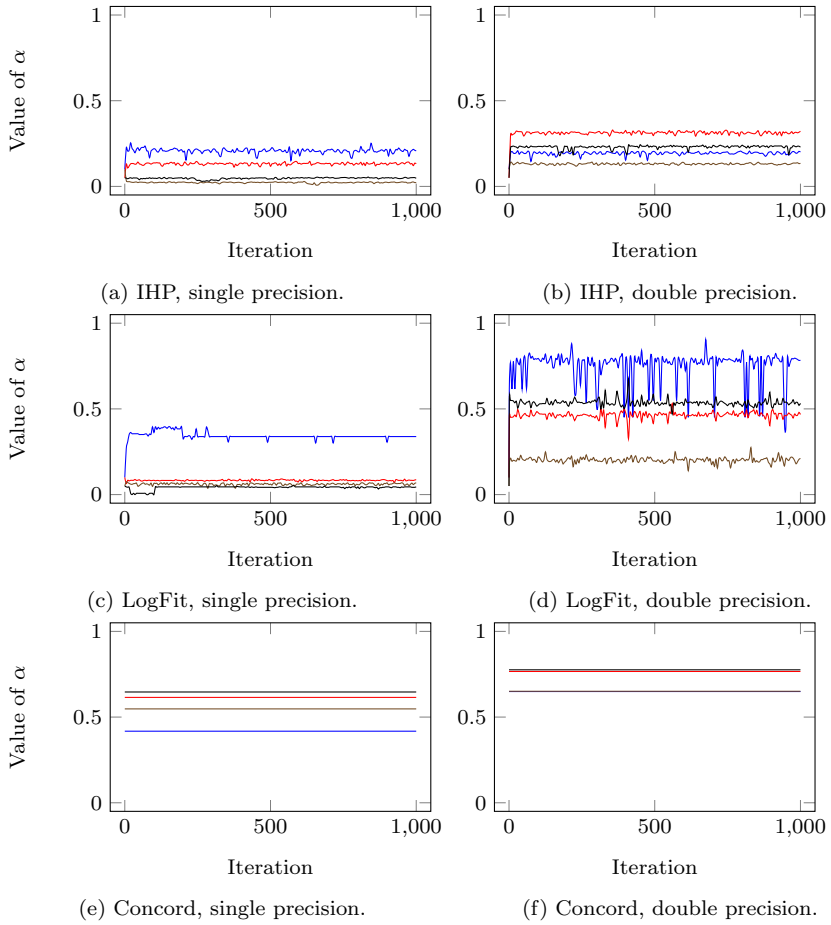
| | | Brightness | | Chromaticity | |
|---|---|---|---|---|---|
| | | Isotropic | Anisotropic | Isotropic | Anisotropic |
| OpenCL | | **16.39** | **30.78** | **57.37** | **125.54** |
| LogFit | CPU | 21.19 | 49.68 | 113.09 | 182.83 |
| | GPU | 4.21 | 22.99 | 42.55 | 100.50 |
| | Copy | 18.10 | 43.21 | 96.26 | 128.69 |
| | Total | **29.63** | **77.07** | **153.08** | **254.95** |
| Concord | CPU | 28.58 | 156.51 | 272.46 | 1058.29 |
| | GPU | 8.35 | 24.31 | 41.92 | 83.55 |
| | Copy | 27.35 | 29.71 | 91.35 | 85.51 |
| | Total | **41.67** | **172.33** | **291.51** | **1103.94** |
| IHP | CPU | 9.16 | 27.65 | 44.76 | 107.59 |
| | GPU | 9.54 | 27.79 | 45.58 | 106.03 |
| | Copy | 0.48 | 0.52 | 0.62 | 0.62 |
| | Total | **9.86** | **28.46** | **46.77** | **112.96** |

considered for LogFit and Concord too as the amount of work computed by CPU. The evolution of the differences of execution times for computing a chunk of work for the CPU and GPU in IHP is shown in Figure 3. These data have been obtained with chunks of five iterations. In figures 2 and 3, blue lines represent the value of $\alpha$ and $T_{\text{CPU}} - T_{\text{GPU}}$, respectively, in the isotropic flow for brightness; red lines, isotropic flow for chromaticity; brown lines, anisotropic flow for brightness; and black lines, anisotropic flow for chromaticity.

IHP and LogFit methods converge to a value for the parameter $\alpha$ in few iterations and keep it stable from very early on with single precision operands. Using double precision, only IHP has a stable behaviour. As mentioned before, Concord does not change value of $\alpha$ after the initial profile phase. For IHP, it should be noted that differences between CPU and GPU execution times are around $10^{-2}$ seconds, being considerably higher only in the first iterations, where an optimal value of $\alpha$ has not yet been found.

Table 5 shows discrepancies of the solutions obtained by IHP and the code that only uses GPU. In this table, the following metrics are shown: the structural similarity index (SSIM) [27], the peak signal to noise ratio (PSNR) [5], and mean square error (MSE).

Results show that this scheme produces a low discrepancy on the computations. Moreover, anisotropic diffusion flow shows a higher error because, for the finite difference approximations, computations of each pixel require the information from more neighbours, and the error builds up. Also, note that with larger rows, a higher error is potentially introduced, as more pixels are computed differently near the split zones. Even in these situations, the discrepancies between IHP implementation and the one that only uses CUDA could be considered almost negligible.




(a) IHP, single precision.

(b) IHP, double precision.

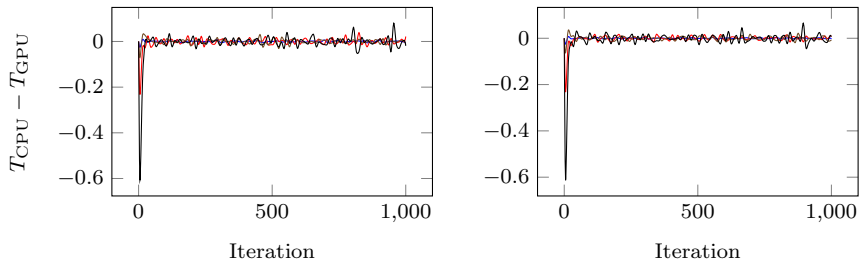
(c) LogFit, single precision.

(d) LogFit, double precision.

(e) Concord, single precision.

(f) Concord, double precision.

Fig. 2: Evolution of $\alpha$.




(a) Single precision.

(b) Double precision.

Fig. 3: Difference of execution times (seconds) between CPU and GPU in IHP.

Table 5: Comparison between GPU only and IHP solutions.

| Image (dimensions) | Isotropic Diffusion | | | Anisotropic Diffusion | | |
|---|---|---|---|---|---|---|
| | SSIM | PSNR | MSE | SSIM | PSNR | MSE |
| `firenze.jpg` (4000 × 6016) | 0.9932 | 44.0214 | 2.4714 | 0.9934 | 42.0288 | 4.0757 |
| `monarch.bmp` (768 × 512) | 0.9998 | 51.8008 | 0.4295 | 0.9991 | 46.0728 | 1.6062 |
| `peppers.tiff` (512 × 512) | 0.9999 | 53.0970 | 0.3187 | 0.9992 | 43.9128 | 2.6412 |
| `lena.jpg` (512 × 512) | 0.9998 | 52.6988 | 0.3493 | 0.9994 | 45.6138 | 1.7852 |
| `baboon.bmp` (500 × 480) | 0.9993 | 50.7597 | 0.5459 | 0.9989 | 46.1121 | 1.5917 |

As discussed in Section 2, several options are available to reduce even more numerical errors, like reducing the size of the chunks or increasing the number of rows in the overlap region, which would have a low impact in the image denoising problem. Figure 4 show the performance overhead induced by copy operations for different chunk sizes, decreasing almost linearly. In this problem, numerical errors related with chunk size are negligible (less than 0.1% for chunk size of 200) and mostly masked by rounding errors in the conversion from real values to integer values when storing the images.

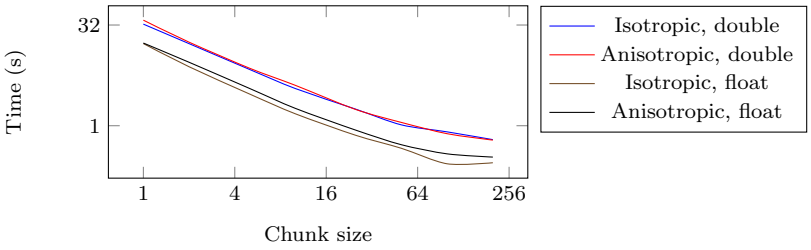



Fig. 4: Time spent (in seconds) in memory copy operations in IHP.

## 5 Conclusions

In this work, a new parallel working scheme for iterative or time-step methods has been introduced, IHP. This scheme balances the workload automatically in runtime according to CPU and GPU performance.

The proposal has been tested using an image denoising algorithm that uses finite differences and iterative diffusion flows for denoising the brightness and chromaticity, thus being a compute intensive application.

Results show that execution time improvements when using IHP are consistent and noticeable even when the difference of performance between CPU and GPU is large. In fact, an improvement up to 39% has been achieved. In the worst scenario, execution time has been increased in 0.2 seconds, noticing a negligible overhead in cases where CPU could only handle a 2% of the

workload. In contrast, LogFit and Concord were not able to improve execution times, mainly due to expensive memory transactions between CPU and GPU.

The discrepancy in the results induced by IHP parallel scheme has proved to be very low, with an structural similarity over the 99.9%, PSNR over 46 dB and low values for the MSE in most of the tests. Additionally, IHP can be easily modified to reduce even further numerical errors using some of the ideas discussed in this work, like defining appropriate sizes for overlap regions.

Concerning future work, we propose to test the IHP approach in other domains like electromagnetism, fluid dynamics, etc., that could be interesting, alongside other numerical techniques like finite element method.

**Acknowledgements** This work has received financial support from the Ministerio de Economía, Industria y Competitividad within the project TIN2016-76373-P. It was also funded by the Consellería de Cultura, Educación e Ordenación Universitaria of Xunta de Galicia (accr. 2019-2022, ED431G2019/04 and reference competitive group 2019-2021, ED431C 2018/19). Thanks to Rafael Asenjo and Dpt. of Computer Architecture of Universidad de Málaga for providing us the source code of LogFit and their help.

## References

1. Cecka, C., Lew, A.J., Darve, E.: Assembly of finite element methods on graphics processors. International journal for numerical methods in engineering **85**(5), 640–669 (2011)
2. Dagum, L., Menon, R.: Openmp: An industry-standard api for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). DOI 10.1109/99.660313. URL `https://doi.org/10.1109/99.660313`
3. Feichtinger, C., Habich, J., Köstler, H., Rüde, U., Aoki, T.: Performance modeling and analysis of heterogeneous lattice boltzmann simulations on cpu–gpu clusters. Parallel Computing **46**, 1 – 13 (2015). DOI https://doi.org/10.1016/j.parco.2014.12.003. URL `http://www.sciencedirect.com/science/article/pii/S0167819114001446`
4. GCC Developer Community: GCC, the GNU Compiler Collection. `https://gcc.gnu.org/` (2017). Accessed: 2018-12-15
5. Hore, A., Ziou, D.: Image quality metrics: Psnr vs. ssim. In: Proceedings of the 2010 20th International Conference on Pattern Recognition, ICPR '10, pp. 2366–2369. IEEE Computer Society, IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/ICPR.2010.579. URL `https://doi.org/10.1109/ICPR.2010.579`
6. Intel Corporation: Intel core i5-7600 processor. `https://ark.intel.com/products/97150/Intel-Core-i5-7600-Processor-6M-Cache-up-to-4_10-GHz` (2017). Accessed: 2019-01-08
7. Kaleem, R., Barik, R., Shpeisman, T., Hu, C., Lewis, B.T., Pingali, K.: Adaptive heterogeneous scheduling for integrated gpus. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 151–162 (2014). DOI 10.1145/2628071.2628088
8. Khan, M.A.I., Delbosc, N., Noakes, C.J., Summers, J.: Real-time flow simulation of indoor environments using lattice boltzmann method. Building Simulation **8**(4), 405–414 (2015). DOI 10.1007/s12273-015-0232-9. URL `https://doi.org/10.1007/s12273-015-0232-9`
9. Komatitsch, D., Michéa, D., Erlebacher, G.: Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda. Journal of Parallel and Distributed Computing **69**(5), 451 – 460 (2009). DOI https://doi.org/10.1016/j.jpdc.2009.01.006. URL `http://www.sciencedirect.com/science/article/pii/S0743731509000069`
10. Laso, R., Cabaleiro, J.C., Rivera, F.F., Muñiz, M.C., Álvarez-Dios, J.A.: Diffusion Methods for image denoising using IHP. `https://gitlab.citius.usc.es/ruben.laso/diffusion-methods-ihp-opencl` (2019)

11. Laso, R., Cabaleiro, J.C., Rivera, F.F., Muñiz, M.C., Álvarez-Dios, J.A.: IHP: Iterative Heterogeneous Parallelism. `https://gitlab.citius.usc.es/ruben.laso/ihp` (2019)
12. Markall, G., Slemmer, A., Ham, D., Kelly, P., Cantwell, C., Sherwin, S.: Finite element assembly strategies on multi-core and many-core architectures. International Journal for Numerical Methods in Fluids **71**(1), 80–97 (2013)
13. Micikevicius, P.: 3d finite difference computation on gpus using cuda. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, pp. 79–84. ACM, ACM, New York, NY, USA (2009). DOI 10.1145/1513895.1513905. URL `http://doi.acm.org/10.1145/1513895.1513905`
14. Mittal, S., Vetter, J.S.: A survey of cpu-gpu heterogeneous computing techniques. ACM Comput. Surv. **47**(4), 69:1–69:35 (2015). DOI 10.1145/2788396. URL `http://doi.acm.org/10.1145/2788396`
15. Navarro, A., Corbera, F., Rodriguez, A., Vilches, A., Asenjo, R.: Heterogeneous `parallel_for` template for cpu–gpu chips. International Journal of Parallel Programming **47**(2), 213–233 (2019). DOI 10.1007/s10766-018-0555-0. URL `https://doi.org/10.1007/s10766-018-0555-0`
16. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue **6**(2), 40–53 (2008). DOI 10.1145/1365490.1365500. URL `http://doi.acm.org/10.1145/1365490.1365500`
17. NVIDIA Corporation: CUDA Compiler driver NVCC, Reference Guide. `https://docs.nvidia.com/pdf/CUDA_Compiler_Driver_NVCC.pdf` (2018). Accessed: 2019-05-07
18. NVIDIA Corporation: Geforce gtx 1050 ti. `https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1050-ti/specifications` (2018). Accessed: 2019-01-08
19. Papadrakakis, M., Stavroulakis, G., Karatarakis, A.: A new era in scientific computing: Domain decomposition methods in hybrid cpu–gpu architectures. Computer Methods in Applied Mechanics and Engineering **200**(13), 1490 – 1508 (2011). DOI https://doi.org/10.1016/j.cma.2011.01.013. URL `http://www.sciencedirect.com/science/article/pii/S0045782511000235`
20. Pérez, B., Bosque, J.L., Beivide, R.: Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, GPGPU '16, pp. 42–51. ACM, New York, NY, USA (2016). DOI 10.1145/2884045.2884051. URL `http://doi.acm.org/10.1145/2884045.2884051`
21. Pheatt, C.: Intel threading building blocks. J. Comput. Sci. Coll. **23**(4), 298–298 (2008). URL `http://dl.acm.org/citation.cfm?id=1352079.1352134`
22. Shams, R., Sadeghi, P.: On optimization of finite-difference time-domain (fdtd) computation on heterogeneous and gpu clusters. Journal of Parallel and Distributed Computing **71**(4), 584 – 593 (2011). DOI https://doi.org/10.1016/j.jpdc.2010.10.011. URL `http://www.sciencedirect.com/science/article/pii/S0743731510002091`
23. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science Engineering **12**(3), 66–73 (2010). DOI 10.1109/MCSE.2010.69
24. Tang, B., Sapiro, G., Caselles, V.: Color image enhancement via chromaticity diffusion. IEEE Transactions on Image Processing **10**(5), 701–707 (2001). DOI 10.1109/83.918563
25. Vilches, A., Asenjo, R., Navarro, A., Corbera, F., Gran, R., Garzarán, M.: Adaptive partitioning for irregular applications on heterogeneous cpu-gpu chips. Procedia Computer Science **51**, 140 – 149 (2015). DOI https://doi.org/10.1016/j.procs.2015.05.213. URL `http://www.sciencedirect.com/science/article/pii/S1877050915010212`. International Conference On Computational Science, ICCS 2015
26. Viñas, M., Bozkus, Z., Fraguela, B.B.: Exploiting heterogeneous parallelism with the heterogeneous programming library. Journal of Parallel and Distributed Computing **73**(12), 1627 – 1638 (2013). DOI https://doi.org/10.1016/j.jpdc.2013.07.013. URL `http://www.sciencedirect.com/science/article/pii/S0743731513001512`. Heterogeneity in Parallel and Distributed Computing
27. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P., et al.: Image quality assessment: from error visibility to structural similarity. IEEE Transactions on Image Processing **13**(4), 600–612 (2004). DOI 10.1109/TIP.2003.819861