Parallel Spherical Harmonic Transforms on heterogeneous architectures (GPUs/multi-core CPUs)

Mikolaj Szydlarski, Pierre Esterie, Joel Falcou, Laura Grigori, R. Stompor

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallel Spherical Harmonic Transforms on heterogeneous architectures (GPUs/multi-core CPUs)

Mikolaj SZYDLARSKI — Pierre ESTERIE — Joel FALCOU — Laura GRIGORI — Radek STOMPOR

N° 7635

15 May 2012

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--7635--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Parallel Spherical Harmonic Transforms on heterogeneous architectures (GPUs/multi-core CPUs)

Mikolaj SZYDLARSKI [*] , Pierre ESTERIE [†] , Joel FALCOU [‡] , Laura GRIGORI [§] , Radek STOMPOR [¶]

Thème NUM — Systèmes numériques
Équipe-Projet Grand-large

Rapport de recherche n° 7635 — 15 May 2012 — 28 pages

Abstract: Spherical Harmonic Transforms (SHT) are at the heart of many scientific and practical applications ranging from climate modelling to cosmological observations. In many of these areas new, cutting-edge science goals have been recently proposed requiring simulations and analyses of experimental or observational data at very high resolutions and of unprecedented volumes. Both these aspects pose formidable challenge for the currently existing implementations of the transforms.

This paper describes parallel algorithms for computing the SHTs with two variants of intra-node parallelism appropriate for novel supercomputer architectures, multi-core processors and Graphic Processing Units (GPU) and discusses their performance tests, alone and embedded within a top-level, MPI-based parallelization layer ported from the S^2HAT library, in terms of their accuracy, overall efficiency and scalability. We show that our inverse SHTs with GeForce 400 Series GPUs equipped with latest CUDA architecture ("Fermi") outperforms the state of the art implementation for a multi-core processor executed on a current Intel Core i7-2600K. Furthermore, we show that an MPI/CUDA version of the inverse transform run on a cluster of 128 NVIDIA Tesla S1070 is as much as 3 times faster than the hybrid MPI/OpenMP version executed on the same number of quad-core processors Intel Nahalem for problem sizes motivated by our target applications. For the direct transforms, the performance is however found to be at the best comparable. Here we discuss in detail optimizations of two major steps involved in the transforms calculation, demonstrating how the overall performance efficiency can be obtained, and elucidating the sources of the dichotomy between the direct and the inverse operations.

Key-words: Spherical Harmonic Transforms, hybrid architectures, hybrid programming, OpenMP, CUDA, Multi-GPU

[*] INRIA Saclay-Île de France, F-91893 Orsay, France (mikolaj.szydlarski@inria.fr).
[†] Laboratoire de Recherche en Informatique, Bat 490, Université Paris-Sud 11, France (pierre.esterie@lri.fr).
[‡] Laboratoire de Recherche en Informatique, Bat 490, Université Paris-Sud 11, France (joel.falcou@lri.fr).
[§] INRIA Saclay-Ile de France, Bat 490, Université Paris-Sud 11, France (laura.grigori@inria.fr).
[¶] CNRS, Laboratoire Astroparticule et Cosmologie, Université Paris Diderot, France (radek@apc.univ-paris7.fr).

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Parallel Spherical Harmonic Transforms on heterogeneous architectures (GPUs/multi-core CPUs)

Résumé : Pas de résumé

Mots-clés : Pas de motclef

1 Introduction

Spherical harmonic functions define an orthonormal complete basis for signals defined on a 2-dimensional sphere. Spherical harmonic transforms are therefore common in all scientific applications where such signals are encountered. These include a number of diverse areas ranging from weather forecasts and climate modelling, through geophysics and planetology, to various applications in astrophysics and cosmology. In these contexts a direct Spherical Harmonic Transform (SHT) is used to calculate harmonic domain representations of the signals, which often possesses simpler properties and are therefore more amenable to a further investigation. An inverse SHT is then used to synthesize a sky image given its harmonic representation. Both of those are also used as means of facilitating the multiplication of huge matrices, defined on the sphere, and having a property of being diagonal in the harmonic domain. Such matrices play an important role in statistical considerations of signals, which are statistically isotropic and are among key operations involved in Monte Carlo Markov Chain sampling approaches used to study such signals, e.g. [30].

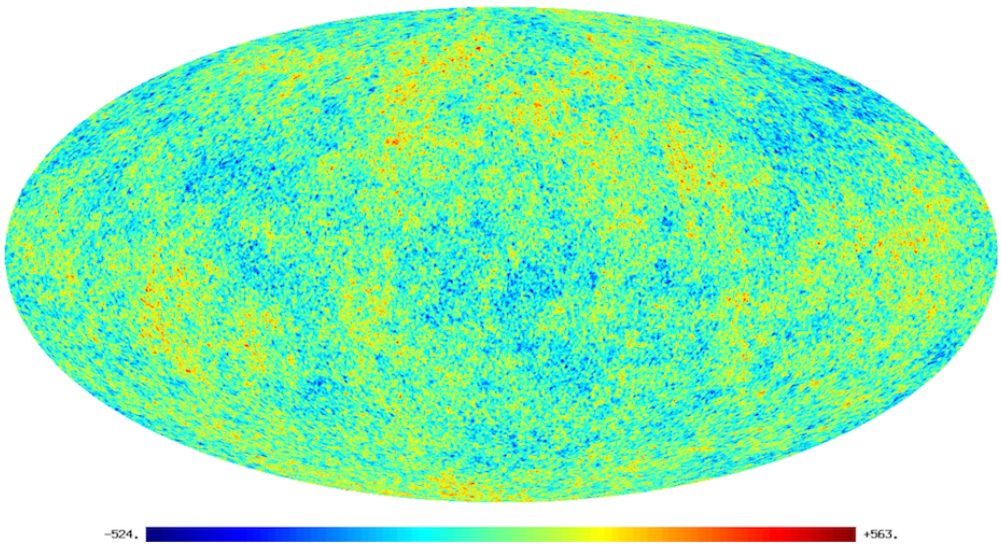


Figure 1: Sky map example synthesised using our MPI/CUDA implementation of the `alm2map` routine. The units are μK. An overall monopole term, with an amplitude of $\sim 2.7\,10^6\mu$K, has been subtracted to uncover the minute fluctuations.

The specific goal of this work is to assist simulation and analysis efforts related to the Cosmic Microwave Background (CMB) studies. CMB is an electromagnetic radiation left over after the hot and dense initial phase of the evolution of the Universe, which is popularly referred to as the Big Bang. Its observations are one of the primary ways to study the Universe. The CMB photons reaching us from afar carry an image of the Universe at the time when it had just a few percent of its current age of $\sim$ 13Gyears. This image confirms that the early Universe was nearly homogeneous and only small, 1 part in 10^5, deviations were present, as displayed in Fig 1. Those initiated the process of so-called structure formation, which eventually led to the Universe as we observe today. The studies of the statistical properties of these small deviations are one of the major pillars on which the present day cosmology is built. On the observational front the CMB anisotropies are targeted by an entire slew of operating and forthcoming experiments. These include a currently observing European satellite called Planck[1], which follows in the footsteps of two very successful American missions, COBE[2] and WMAP[3], as well as a number of balloon-borne and ground-based observatories. Owing to quickly evolving detector technology, the volumes of the data collected by these experiments have been increasing at the Moore's law rate, reaching at the present the values on order of petabytes. These new data aiming at progressively more challenging science goals not only provide, or are expected to do so, images of the sky with an unprecedented resolution, but their scientific exploitation will require intensive, high precision, and voluminous simulations, which in turn will require highly efficient novel approaches to a calculation of SHTs. For instance, the current and forthcoming balloon-borne and ground-based experiments will produce maps of the sky containing as many as $\mathcal{O}(10^5 - 10^6)$ pixels and up to $\mathcal{O}(10^6 - 10^7)$ harmonic modes. Maps from already operating Planck satellite will consist of between

[1] PLANCK: http://sci.esa.int/science-e/www/area/index.cfm? fareaid =17
[2] COBE: http://lambda.gsfc.nasa.gov/product/cobe/
[3] WMAP: http://map.gsfc.nasa.gov/

$\mathcal{O}(10^6)$ and $\mathcal{O}(10^8)$ pixels and harmonic modes. The production of high precision simulations reproducing reliably polarized properties of the CMB signal, in particular generated due to the so called gravitational lensing, requires an effective number of pixels and harmonic modes, as big as $\mathcal{O}(10^9)$. These latter effects constitute some of the most exciting new lines of research in the CMB area. The SHT implementation, which could successfully address all this needs would not only have to scale well in the range of interest, but also be sufficiently quick to be used as part of massive Monte Carlo simulations or extensive sampling solvers, which are both important steps of the scientific exploitation of the CMB data.

At this time there are a few software packages available, which permit calculations of the SHT. These include, HEALPIX [3], GLESP [2], CCSHT [1], LIBPSHT [4], and S^{2}HAT [5], which are commonly used in the CMB research, and some others such as, SPHARMONICKIT/S2KIT [6] and SPHERPACK [7]. They propose different levels of parallelism and optimization, while implementing, with an exception of two last ones, essentially the same algorithm. Among those the LIBPSHT package [23] offers typically the best performance, due to efficient code optimizations, and is considered as the state of the art implementation on serial and shared-memory platforms, at least for some of the popular sky pixelizations. Conversely, the implementation offered by the S^{2}HAT library is best adapted to the distributed memory machines, and fully parallelized and scalable with respect to both memory usage and calculation time. Moreover, it provides a convenient and flexible framework straightforwardly extensible to allow for multiple parallelization levels based on hybrid programming models. Given the specific applications considered in this work, the distributed memory parallelism seems inevitable and this is therefore the S^{2}HAT package, which we select as a starting point for this work.

The basic algorithm we use hereafter, and which is implemented in both LIBPSHT and S^{2}HAT libraries as described in the next section, scales as $\mathcal{O}(\mathcal{R}_N\,\ell_{max}^2)$, where $\mathcal{R}_N$ is the number of rings in the analyzed map and ℓ_{max} fixes the resolution and thus determines the number of modes in the harmonic domain, equal to $\simeq \ell_{max}^2$. For full sky maps, we have usually $\ell_{max} \propto \mathcal{R}_N \propto n_{pix}^{1/2}$, where n_{pix} is the total number of sky pixels, and therefore the typical complexity is $\mathcal{O}(\ell_{max}^3) \propto \mathcal{O}(n_{pix}^{3/2})$. We note that further improvements of this overall scaling are possible, for instance, if multiple identical transforms have to, or can, be done simultaneously, what could lower the numerical cost to essentially that of a single transform. In this paper we however focus on the core algorithm and leave an investigation of these potential extensions to future work.

We note that alternative algorithms have been also proposed, some of which display superior complexity. In particular, Driscoll and Healy [10] proposed a divide-and-conquer algorithm with a theoretical scaling of $\mathcal{O}(n_{pix}\,\ln^2 n_{pix})$. This approach is limited to special equidistant sphere grids and being inherently numerically unstable requires corrective measures to ensure high precision results. This in turn affects its overall performance. For instance, the software SPHARMONICKIT/S2KIT, which is the most widely used implementation of this approach, has been found a factor 3 slower than the HEALPIX transforms implementing the standard $\mathcal{O}(n_{pix}^{3/2})$ method at the intermediate resolution of $\ell_{max} \sim 1024$ [34]. Other algorithms also exist and typically involve a precomputation step of the same complexity, but often less favorable prefactors, as the standard approach, and an actual calculation of the transforms, which typically exploits either the Fast Multipole Methods, e.g., [28, 32] or matrix compression techniques, e.g., [18, 33], to bring down its overall scaling to $\mathcal{O}(n_{pix}\ln n_{pix})$ [28], $\mathcal{O}(n_{pix}\ln^2 n_{pix})$ [32, 33], or $\mathcal{O}(n_{pix}^{5/4}\ln n_{pix})$ [18]. The methods of this class typically require significant memory resources needed to store the precomputation products and are advantageous only if a sufficient number of successive transforms of the same type has to be performed in order to compensate for the precomputation costs. The most recent, and arguably satisfactory, implementation of such ideas is a package called WAVEMOTH[4] [25], which achieves a speed-up of the inverse SHT with respect to the LIBPSHT library by a factor ranging from 3 to 6 for $\ell_{max} \sim 4000$. In such a case the required extra memory is on order of 40GBytes and it depends strongly, i.e., $\propto \ell_{max}^3$, on the resolution. We also note that in some applications the need to use SHTs can be sometimes by-passed by resorting to approximate but numerically efficient means such as the Fast Fourier Transforms as for instance in the context of convolutions on the sphere as discussed in [11].

In many practical applications, as the ones driving this research, SHTs are just one of many processing steps, which need to be performed to accomplish a final task. In such a context, these are the memory high-water mark and algorithm scalability, which frequently emerge as the two most relevant requirements, as the cost of the SHT transforms is often subdominant and their complexity more favorable than that of many other typical operations. From the memory point of view, the standard algorithm, having the smallest memory footprint, remains therefore an attractive option. The important question is then whether its efficient, scalable, parallel implementation is indeed possible. Such a question is particularly pertinent

[4] WAVEMOTH: https://github.com/wavemoth/wavemoth

in the context of the heterogeneous architectures and hybrid programming models and this is the context investigated in this work.

The past work on the parallelization of SHTs includes a parallel version of the two step algorithm of Driscoll and Healy introduced by [16], the algorithm of [9] based on rephrasing the transforms as matrix operations and therefore making them effectively vectorizable, and a shared memory implementation available in the LIBPSHT package. More recently, algorithms have been developed for NVIDIA General-Purpose Graphics Processing Units (GPGPU) [26, 15]. This latter work provided a direct motivation for the investigation presented here, which describes what, to the best of our knowledge, is the first hybrid design of parallel SHT algorithms suitable for a cluster of GPU-based architectures and current high-performance multi-core processors, and involving hybrid OpenMP/MPI and MPI/CUDA programming. We find that once carefully optimized and implemented, the algorithm displays nearly perfect scaling in both cases, at least up to 128 MPI processes mapped on the same number of pairs of multi-core CPU-GPU. We find that inverse SHT with GeForce 400 Series equipped with the latest CUDA device ("Fermi") outperforms state of the art implementation for a multi-core processors executed on latest Intel Core i7-2600K, while the direct transforms in both these cases perform comparably.

This paper is organised as follows. In section 2 we introduce the algebraic background of the spherical harmonic transforms. In section 3 we describe a basic, sequential algorithm and list useful assumptions concerning a sphere pixelisation, which facilitate a number of acceleration techniques used in our approach. In following section, we introduce a detailed description of our parallel algorithms along with two variants suitable for clusters of GPUs and clusters of multi-cores. Section 5 presents results for both implementation and finally section 6 concludes the paper.

2 Algebraic background

2.1 Definitions and notations

Just as the Fourier basis facilitates a quick and efficient evaluation of convolutions in one or more dimensional spaces, the spherical harmonic basis does so but for functions defined on the surface of a sphere, i.e., depending on spherical coordinates: $\theta \in (0, \pi)$ (colatitude) and $\phi \in [0, 2\pi)$ (the angle counterclockwise about positive z-axis from the positive x-axis). We can express a band-limited approximation $\tilde{f}$ to a real-valued spherical function $f(\theta, \phi)$ $f : S^2 \to \mathbb{R}$ in terms of the spherical harmonic basis functions as

$$\tilde{f}\,(\theta, \phi) \quad = \quad \sum_{\ell=0}^{\ell_{max}} \sum_{m=-\ell}^{\ell} a_{\ell m}\, Y_{\ell m}\,(\theta, \phi)\,, \tag{1}$$

where $a_{\ell m}$ are spherical harmonic coefficients describing f in the harmonic domain, $Y_{\ell m}$ is a spherical harmonic basis function of degree ℓ and order m, and the bar over f indicates that it is merely a band-limited approximation of f with a bandwidth ℓ_{max}. The spherical harmonic coefficients are computed by taking the scalar inner product of f with the corresponding basis function, which can be expressed as the integral

$$a_{lm} = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} d\theta\, d\phi\; f(\theta, \phi) Y^{\dagger}_{lm}(\theta, \phi) \sin\theta, \tag{2}$$

where † denotes complex conjugation.

In actual applications the spherical harmonic transforms are rather used to project *grid point data* (a discretized scalar field $\boldsymbol{s}_n$) on the sphere onto the spectral modes in an *analysis* step (as displayed in equation (3)) and an inverse transform reconstructing the grid point data from the spectral information in a *synthesis* step (as displayed in equation (4)). Depending on the choice of ℓ_{max} and the grid geometry the transform $\boldsymbol{s}_n \to \boldsymbol{a}_{lm}$ may only be approximate, what is indicated by a tilde over $\boldsymbol{a}$.

$$\tilde{\boldsymbol{a}}_{\ell m} \quad = \quad \sum_{\{\boldsymbol{\theta}_n, \boldsymbol{\phi}_n\}} \boldsymbol{s}_n\,(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n)\; Y_{\ell m}\,(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n)\,, \tag{3}$$

$$\boldsymbol{s}_n\,(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n) \quad = \quad \sum_{\ell=0}^{\ell_{max}} \sum_{m=-\ell}^{\ell} \boldsymbol{a}_{\ell m}\, Y_{\ell m}\,(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n)\,. \tag{4}$$

In CMB applications, $\boldsymbol{s}_n$ is a vector of pixelized data, e.g. the brightness of incoming CMB photons, assigned to n_{pix} locations (θ_n, ϕ_n) on the sky defined as centres of suitable chosen sky pixels. The parameter ℓ_{max} defines a maximum order of the Legendre function and thus a band-limit of the field $\boldsymbol{s}_n$, and in practice it is set by the experimental resolution.

The basis functions $Y_{\ell m}(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n)$ are defined in terms of normalised associated Legendre functions $\mathcal{P}_{\ell m}$ of degree l and order m,

$$Y_{\ell m}(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n) \equiv \mathcal{P}_{\ell m}(\cos\boldsymbol{\theta}_n)\, e^{im\boldsymbol{\phi}_n}, \tag{5}$$

where we use the relation $Y_{l,-m} = (-1)^m Y^{\dagger}_{lm}$ to compute all spherical harmonics in terms of the associated Legendre functions P_{lm} with $m \geq 0$. Consequently, by separating variables following equation (5) we can reduce the computation of the spherical harmonic transform to a regular Fourier transform in the longitudinal coordinate ϕ_n followed by a projection onto the associated Legendre functions,

$$\tilde{\boldsymbol{a}}_{\ell m} = \sum_{\{\boldsymbol{\theta}_n\}} \left[\left(\sum_{\{\boldsymbol{\phi}_n\}} \boldsymbol{s}_n(\boldsymbol{\theta}_n, \boldsymbol{\phi}_n)\; e^{im\boldsymbol{\phi}_n} \right) \mathcal{P}_{\ell m}(\cos\boldsymbol{\theta}_n) \right]. \tag{6}$$

The associated Legendre functions satisfy the following recurrence (with respect to the multipole number ℓ for a fixed value of m) critical for the algorithms developed in this paper,

$$\mathcal{P}_{\ell+2,m}(x) = \beta_{\ell+2,m}\, x\, \mathcal{P}_{\ell+1,m}(x) + \frac{\beta_{\ell+2,m}}{\beta_{\ell+1,m}} \mathcal{P}_{\ell m}(x), \tag{7}$$

where

$$\beta_{\ell m} = \sqrt{\frac{4\,\ell^2 - 1}{\ell^2 - m^2}}. \tag{8}$$

The recurrence is initialised by the starting values,

$$\begin{aligned} \mathcal{P}_{mm}(x) &= \frac{1}{2^m\, m!} \sqrt{\frac{(2m+1)!}{4\pi}}\, \left(1 - x^2\right)^m \\ &\equiv \mu_m \left(1 - x^2\right)^m, \qquad (9) \\ \mathcal{P}_{m+1,m}(x) &= \beta_{\ell+1,m}\, x\, \mathcal{P}_{mm}(x). \qquad (10) \end{aligned}$$

The recurrence is numerically stable but a special care has to be taken to avoid under- or overflow for large values of ℓ_{max} [13] e.g., for increasing m the $\mathcal{P}_{mm}$ values can become extremely small such that they can no longer be represented by the binary floating-point numbers in IEEE 754[5] standard. However, since we have freedom to rescale all the values of $\mathcal{P}_{\ell m}$, we can dynamically rescale current values, while performing the recurrence, to prevent a potential overflow on a subsequent step. More precisely on each step the newly computed value of the associated Legendre function is tested and rescaled if found to be too close to the over- or underflow limits. The rescaling coefficients (e.g., in form of their logarithms) are kept track of and used to rescale back all the computed values of $\mathcal{P}_{\ell m}$ at the end as required. This scheme is based on two facts. First, the values of the associated Legendre functions calculated via the recurrence change gradually and rather slowly on each step. Second, their actual values as needed by the transforms are representable within the range of the double precision values.

The specific implementation of these ideas used in all our algorithms follows that of the S^{2}HAT software and the HEALPix package. It is based on a use of a precomputed vector of values, sampling the dynamic range of the representable double precision numbers and thus avoids any explicit computation of numerically-expensive logarithms and exponentials. This scaling vector is used to compare the values of $\mathcal{P}_{\ell m}$ computed on each step of the recurrence, and then it is use to rescale them if needed. For sake of simplicity, in this paper we assume that this is an integral part of associated Legendre function evaluation (independent of architecture) and we omit explicit description of algorithm related to rescaling. For more details we refer to [15] and documentation of the HEALPix package.

2.2 Specialized formulation

For a general grid of $\mathcal{O}(n_{pix})$ points on the sphere, a direct calculation has computational complexity $\mathcal{O}(\ell^2_{max} n_{pix})$ which is a serious limitation for problems in range of our interest. For instance, the current and forthcoming balloon-borne and ground-based experiments will produce maps of sky containing as many as $n_{pix} \in [10^5, 10^6]$ pixels and up to $\mathcal{O}(10^6 - 10^7)$ harmonic modes. Maps from already operating Planck satellite will consist of between $\mathcal{O}(10^6)$ and $\mathcal{O}(10^8)$ pixels and harmonic modes. However, the algorithm we use hereafter and which is described in the next section, scales as $\mathcal{O}(n_{pix}\, \ell_{max})$. We obtain such complexity with help of additional geometrical constraints imposed on the permissible pixelizations. These are [13],

[5] The IEEE has standardised the computer representation for binary floating-point numbers in IEEE 754. This standard is followed by almost all modern machines.

- the map pixels are arranged on $\mathcal{R}_N \approx \sqrt{n_{pix}}$ iso-latitudinal rings, where latitude of each ring is identified by a unique polar angle θ_n (for sake of simplicity hereafter we will refer to this angle by the ring index i.e., $r_n \to \theta_n$, where $n \in [0, \mathcal{R}_N]$)
- within each ring r_n, pixels must be equidistant ($\Delta\phi$ =CONST), though their number can vary from a ring to another ring.

The requirement of the iso-latitude distribution for all pixels helps to save on the number of required FLOPs as the associated Legendre function need now be calculated only once for each pixel ring. Consequently, nearly all spherical grids, which are currently used in CMB analysis (for instance HEALPix and GLESP), as well as other fields conform with such assumptions.

Taking into account these restrictions and definition of basis function $Y_{\ell m}$ (5) we can rewrite (adapted from [13]) the synthesis step from equation (4) as

$$\boldsymbol{s}\left(r_n, \boldsymbol{\phi}_n\right) = \sum_{m=-\ell_{max}}^{\ell_{max}} e^{im\phi_0}\, \boldsymbol{\Delta}_m^A\left(r_n\right), \tag{11}$$

where $\boldsymbol{\Delta}_m^A\left(r_n\right)$ is a set of functions such as

$$\boldsymbol{\Delta}_m^A\left(r_n\right) \equiv \begin{cases} \sum_{\ell=0}^{\ell_{max}} \boldsymbol{a}_{\ell 0}\, \mathcal{P}_{\ell 0}\left(\cos r_n\right), & m = 0, \\ \sum_{\ell=m}^{\ell_{max}} \boldsymbol{a}_{\ell m}\, \mathcal{P}_{\ell m}\left(\cos r_n\right), & m > 0, \\ \sum_{\ell=|m|}^{\ell_{max}} \boldsymbol{a}_{\ell |m|}^{\dagger}\, \mathcal{P}_{\ell |m|}\left(\cos r_n\right), & m < 0, \end{cases} \tag{12}$$

and respectively the analysis step(3) as

$$\tilde{\boldsymbol{a}}_{\ell m} = \sum_{n=0}^{\mathcal{R}_N} \boldsymbol{\Delta}_m^S\left(r_n\right)\, \mathcal{P}_{\ell m}\left(\cos r_n\right), \tag{13}$$

where

$$\boldsymbol{\Delta}_m^S\left(r_n\right) \equiv \sum_{\{\phi_n\}} \boldsymbol{s}\left(r_n, \boldsymbol{\phi}_n\right)\, e^{-im\phi_0}. \tag{14}$$

The $\boldsymbol{\phi}_0$ denotes the ϕ angle of the first pixel in the ring.

From above, we can see that the spectral harmonic transforms can be split into two successive computations: one calculating the associated Legendre functions, Eqs. (12) and (13), and the other performing the series of Fast Fourier Transforms (FFTs), Eqs. (11) and (14) [20]. This splitting immensely facilitates implementation, for instance, in order to evaluate Fourier transforms we can use third-party libraries devoted to this problem. However, we have to bear in mind that different libraries offer different robustness with strong dependence on length of FFT. This focus our special attention since the number of samples (pixels) per ring may vary and this strongly affects efficiency of the FFT algorithms. For instance, well-know `fftpack` library [29] used in HEALPIX [13] and LIBPSHT [23] libraries performs well only for a length N_{FFT}, whose prime decomposition only contains the factors 2, 3, and 5, while for prime number lengths its complexity degrades from $\mathcal{O}\left(N_{FFT} \log N_{FFT}\right)$ to $\mathcal{O}(N_{FFT}^2)$. For this reason we employ by default in all versions of our algorithm the `FFTW` library [12] which uses $\mathcal{O}\left(N_{FFT} \log N_{FFT}\right)$ algorithms even for prime sizes.

2.3 Time consumption

Three parameters typically determine the sizes and numerical complexity characteristic of our problem. These are a total number of pixels, n_{pix}, a number of rings of pixels in the analyzed map, $\mathcal{R}_N$, and a maximum order of the associated Legendre functions, ℓ_{max}. In particular a number of modes in the harmonic domain is $\sim \ell_{max}^2$. In well defined cases these three parameters are not completely independent. For example, in the case of full sky maps, we have usually $\ell_{max} \propto \mathcal{R}_N \propto n_{pix}^{1/2}$. This imply that recurrence step requires $\mathcal{O}\left(\mathcal{R}_N \ell_{max}^2\right)$ floating point operations (FLOPS), due to the fact that for each of $\mathcal{R}_N$ rings we have to calculate a full set, up to ℓ_{max}, of the associated Legendre functions, $\mathcal{P}_{\ell m}$, at cost $\propto \ell_{max}^2$. The Fourier transform step, as mentioned above, has then subdominant complexity $\mathcal{O}\left(\mathcal{R}_N \ell_{max} \log \ell_{max}\right)$. These theoretical expectations agree with our experimental results shown in Fig. 2 depicting a typical breakdown of the average overall time between the main steps involved in the SHTs computation as obtained for a run performed on quad-core Intel i7-2600K processor. We can observe that indeed the evaluation of the

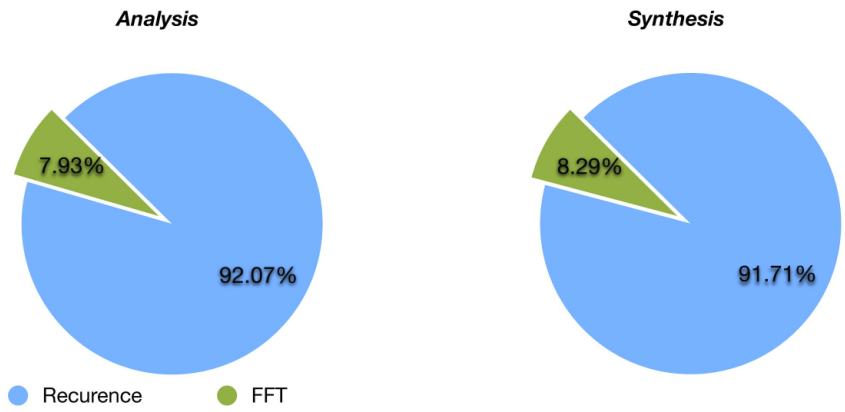


Figure 2: The overall time breakdown between the main steps of the SHTs algorithm as computed with the identical band limit ($\ell_{max} = 4096$) for quad-core Intel i7-2600K processor.

Legendre transform dominates by far over FFT in terms of time of computation. This motivated us to study the possible improvements of this part of the calculation. We explore this issue in the context of the heterogeneous architectures and hybrid programming models. In particular we introduce a parallel algorithm that is suitable for clusters of accelerators, namely multi-core processors and GPUs which we employ for an efficient evaluation of spherical harmonic transforms.

3 Basic algorithm

We proceed by developing algorithms which agree with assumptions listed in the previous section. Then, in the next section we introduce their parallel versions and we discuss possible improvements specific to the targeted architectures.

Algorithm 1 presents the computation of discrete, direct SHT which is a realisation of equation (3) for a given "bandwidth" ℓ_{max} and input data vector $\boldsymbol{s}$ obtained from the samples of the original function which we wish to transform. Respectively, algorithm 2 presents the computation of discrete, inverse transform

Algorithm 1 DIRECT TRANSFORM (EQ. 3)

Require: $\boldsymbol{s} \in \mathbb{R}_{n_{pix}}$ *values*
STEP 1 - $\boldsymbol{\Delta}_m^S(r)$ CALCULATION
for *every ring* r ***do***
for *every* $m = 0, ..., m_{max}$ ***do***
∘ *calculate* $\boldsymbol{\Delta}_m^S(r)$ *via FFT Eq.* (14)
end for(m)
end for(r)

STEP 2 - PRE-COMPUTATION
for *every* $m = 0, ..., m_{max}$ ***do***
∘ $\boldsymbol{\mu}_m$ *pre-computation Eq.* (9)
end for (m)

STEP 3 - CORE CALCULATION
for *every ring* r ***do***
for *every* $m = 0, ..., m_{max}$ ***do***
for *every* $\ell = m, ..., \ell_{max}$ ***do***
∘ *compute* $\mathcal{P}_{\ell m}$ *via Eq.* (7)
∘ *update* $\boldsymbol{a}_{\ell m}$, *Eq.* (13)
end for (ℓ)
end for (m)
end for (r)

return $\boldsymbol{a}_{\ell m} \in \mathbb{C}^{\ell \times m}$

from equation (3) which takes as input a set of complex numbers $\boldsymbol{a}_{\ell m}$, interpreted as Fourier coefficients in a spherical harmonic expansion and returns a set of sample values i.e., real data vector $\boldsymbol{s}$.

Algorithm 2 INVERSE TRANSFORM (EQ. 4)

Require: $\boldsymbol{a}_{\ell m} \in \mathbb{C}^{\ell \times m}$ *values*
STEP 1 - PRE-COMPUTATION
for *every* $m = 1, ..., m_{max}$ ***do***
∘ $\boldsymbol{\mu}_m$ *precomputation Eq.* (9)
end for *(m)*

STEP 2 - $\boldsymbol{\Delta}_m^A$ CALCULATION
for *every ring* r ***do***
for *every* $m = 0, ..., m_{max}$ ***do***
for *every* $\ell = m, ..., \ell_{max}$ ***do***
∘ *compute* $\mathcal{P}_{\ell m}$ *via Eq.* (7)
∘ *update* $\boldsymbol{\Delta}_m^A(r)$*, Eq.* (14)
end for *(ℓ)*
end for *(m)*
end for *(r)*

STEP 3 - $\boldsymbol{s}$ CALCULATION
for *every ring* r ***do***
for *every* $m = 0, ..., m_{max}$ ***do***
∘ *calculate* $\boldsymbol{s}$ *via FFT and given* $\boldsymbol{\Delta}_m^A(r)$*, Eq.* (11)
end for*(m)*
end for *(r)*

return $\boldsymbol{s} \in \mathbb{R}_{n_{pix}}$

3.1 Similarities

Both algorithms use a divide and conquer approach i.e., in this approach our problem of computing projection onto associated Legendre functions is decomposed into smaller subproblems of a similar form (rings of pixels). The subproblems are solved recursively by a further subdivision (computation of $\mathcal{P}_{\ell m}$ via recurrence with respect to the multipole number ℓ for a fixed value of m) and finally their solutions are combined to solve the original problem. Furthermore, both algorithms consist of three main steps:

- the pre-computation of the starting values of the recurrence (9) in the loop over all $m \in [0, m_{max}]$ (step 2 in Algorithm 1 and step 1 in Algorithm 2),
- the evaluation of the associated Legendre function for each ring of pixels via the recurrence from equation (7). This requires three nested loops, where the loop over ℓ is set innermost to facilitate the evaluation of the recurrence with respect to the multipole number ℓ for a fixed value of m (step 3 in Algorithm 1 and step 2 in Algorithm 2).
- step where via FFTs we either generate the cosine transform representation of the input vector $\boldsymbol{s}$ (11) (step 1 in Algorithm 1) or to complete computation in inverse algorithm an abelian FFT is performed to compute the sums (14) (step 3 in Algorithm 2).

3.2 Differences

Both algorithms are almost the same in terms of number and type of arithmetic operations required for the computation (see for instance Figure 2 or a detailed analysis in [17]). The main difference consists in the dimension of the partial results which need to be updated within the nested loop where we evaluate the associated Legendre functions (step 3 in Algorithm 1 and step 2 in Algorithm 2). In case of the direct transform, for each ring of pixels on the sphere we need to update all non-zero values of the matrix $\boldsymbol{a}_{\ell m} \in \mathbb{C}^{\ell_{max} \times m_{max}}$ by a product between the precomputed inner sums defined as $\boldsymbol{\Delta}_m^S$ (14) and associated Legendre function for each fixed m. In the case of the inverse transform, where the input is a set of complex numbers $\boldsymbol{a}_{\ell m}$, the nested loop (step 2 in Algorithm 2) is used to compute the inner sum $\boldsymbol{\Delta}_m^A$ (12), which can be obtained as a matrix-vector product

$$\boldsymbol{\Delta}_m^A = \begin{pmatrix} \Delta_m^A(r_0, m) \\ \Delta_m^A(r_1, m) \\ \vdots \\ \Delta_m^A(\mathcal{R}_N, m) \end{pmatrix} = \begin{pmatrix} \mathcal{P}_{mm}(\cos r_0) & \dots & \mathcal{P}_{\ell_{max} m}(\cos r_0) \\ \mathcal{P}_{mm}(\cos r_1) & \dots & \mathcal{P}_{\ell_m axm}(\cos r_1) \\ \vdots & \vdots & \vdots \\ \mathcal{P}_{mm}(\cos r_{\mathcal{R}_N}) & \dots & \mathcal{P}_{\ell_{max} m}(\cos r_{\mathcal{R}_N}) \end{pmatrix} \begin{pmatrix} a_{mm} \\ a_{m+1m} \\ \vdots \\ a_{\ell_{max} m} \end{pmatrix}. \quad (15)$$

As we will see further in this report, those differences may cause big contrast in terms of evaluation time and memory consumption. This effect can be additionally enhanced by the type of the architecture on which we execute our algorithm.

4 Parallel Spherical Harmonic Transforms

4.1 Top-level distributed-memory parallel framework

The top level parallelism employed in this work is adopted from the S^{2}HAT library developed by the last author. Though not new, it has not been described in detail in the literature yet and we present below its first detailed description including its performance model and analysis. The framework is based on a distributed-memory programming model and involves two computational stages separated by a single instance of a global communication involving all the processes performing the calculations. These together with a specialized, but flexible data distribution, are the major hallmarks of the framework. The two stages of the computations can be further parallelized and/or optimized and we discuss two examples of potential intra-node parallelization approaches below. We note that the framework as such does not imply what algorithm should be used by the MPI processes, e.g., [25], though of course its overall performance and scalability will depend on a specific choice.

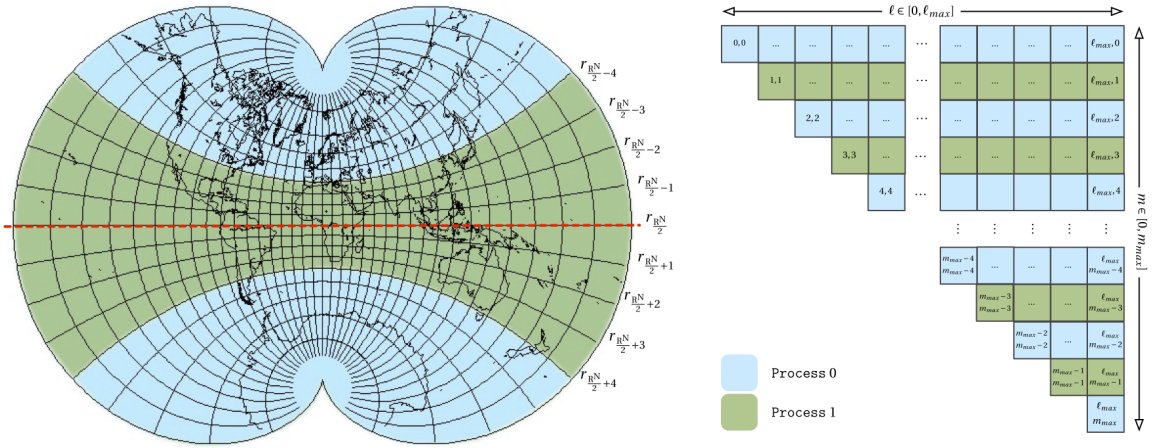
The major motivation behind such a distributed memory parallelization level comes from the volumes of the data, both maps and harmonic coefficients, to be analyzed in the applications targeted here and from the need to maintain flexibility of the routines designed to be used in massively parallel applications, where SHTs typically constitute only one step of the overall process. The latter requirement calls not only for a flexible interface and an ability to use efficiently as many MPI processes as available, but also for a low memory footprint per process.

4.1.1 Data distribution and algorithm

The simple structure of the top-level parallelism outlined above is facilitated by a data layout, which assumes that both the map $\boldsymbol{s}$ and its harmonic coefficients $\boldsymbol{a}_{\ell m}$, are distributed. The map distribution is ring-based, i.e., we assign to each process a part of the full map consisting of complete rings of pixels. To do so we make use of the assumed equatorial symmetry of the pixelization schemes and first divide all rings of one of the hemispheres (including the equatorial ring if present) into disjoint subsets made of complete, consecutive rings, and then assign to each process part of the map corresponding to one of such ring subsets together with its symmetric, counterpart from the other hemisphere. If we denote the rings mapped on a processor i as $\mathcal{R}_i$, then $\cup_{i=0}^{n_{procs}} \mathcal{R}_i = \mathcal{R}$, where $\mathcal{R}$ is the set of all rings and n_{procs} is the number of processes used in the computation. We also have $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ if $i \neq j$. An example ring distribution is depicted in Figure 3(a), where different colours mark pixels on iso-latitudinal rings mapped on two different processes.

In order to distribute harmonic domain object we divide the 2-dimensional array of the coefficients $\boldsymbol{a}_{\ell m}$ with respect to a number, m, and assign to each process a subset of values of m, $\mathcal{M}_i$, and a corresponding subset of the hamonic coefficients, $\{a_{\ell m} : m \in \mathcal{M}_i \text{ and } \ell \in [m, \ell_{max}]\}$. As with the ring subsets, the subsets of m values have to be disjoint, $\mathcal{M}_i \cap \mathcal{M}_j = \emptyset$ if $i \neq j$ and include all the values, $\cup_{i=0}^{n_{procs}} \mathcal{M}_i = \mathcal{M}$. An example of the harmonic domain distribution is depicted in Figure 3(b), where like in the case of the ring distribution we marked with different colours $a_{\ell m}$ coefficients mapped on two different processes.

This data layout allows to avoid any redundancy in terms of calculations and memory and any need for an intermediate inter-process communication, while at the same time is general enough that at least in principle can accommodate an entire slew of possibilities as driven by real applications. Any freedom in selecting a specific layout should be used to ensure a proper load balancing in terms of both memory and work, and it may, and in general will, depend on what algorithm is used by each of the MPI processes. Though perfect load balancing, memory- and work-wise, may not be always possible, in practice we have found that good compromises usually exist. In particular, the map distribution which assigns the same number of rings to each process is typically satisfactory, even if it may in principle lead to an unbalanced work and/or memory distribution. The latter is indeed the case for the HEALPIX pixelization which does not preserve the same number of samples per ring, what introduces differences in the memory usage between the processes but also in the workload. Some fine tuning could be used here to improve on both these aspects, as it is indeed done in S^{2}HAT, but even without that the differences are not critical. For the distribution of the harmonic objects we propose $\mathcal{M}_i \equiv \{m : m = i + k\, n_{procs}$ or $m = m_{max} - i - k\, n_{procs},$ where $k \in [0, m_{max}/2]\}$, so the process i stores the values of m equal to $[i, i + n_{procs}, ..., m_{max} - i - n_{procs}, m_{max} - i]$. This usually provides a good first try, which appears satisfactory in a number of cases as it will be explained in the next section.



(a) The iso-latitudinal rings distribution.

(b) Distribution of $a_{\ell m}$ coefficients.

Figure 3: Example of a distribution of the iso-latitudinal rings and the $\boldsymbol{a}_{\ell m}$ coefficients among two processes. Each colour marks elements assigned to one process. For rings visualisation we used an August's Conformal Projection of the sphere on a two-cusped epicycloid.

We note that this data layout imposes an upper limit on the number of processes which could be used, and which is given by $\min(m_{max}/2, \mathcal{R}_N/2)$. This is typically sufficiently generous not to lead to any problems, as usually we have $\mathcal{R}_N \propto m_{max} \simeq \ell_{max}$ and the limit on a number of processes is $\mathcal{O}(\ell_{max})$.

With this data distribution, Algorithms 1 and 2 require only straightforward and minor modifications to become functional parallel codes. As an example, Algorithm 3 details the parallel inverse transform, which is a parallel version of Algorithm 2. The operations listed there are to be performed by each of the

Algorithm 3 General parallel `alm2map` algorithm (Code executed by each MPI process)

Require: $\boldsymbol{a}_{\ell m} \in \mathbb{C}^{\ell \times m}$ for $m \in \mathcal{M}_i$ and all ℓ
Require: indices of rings $r \in \mathcal{R}_i$

step 1 - Pre-computation
for every $m \in \mathcal{M}_i$ **do**
  ∘ $\boldsymbol{\mu}_m$ pre-computation Eq. (9)
end for (m)

step 2 - $\boldsymbol{\Delta}_m^A$ calculation
for every ring $r \in \mathcal{R}$ **and** every $m \in \mathcal{M}_i$ **do**
  ∘ initialise the recurrence: $\mathcal{P}_{mm}$, $\mathcal{P}_{m+1,m}$ using precomputed $\boldsymbol{\mu}_m$ Eqs. (9) & (10)
  ∘ precompute recurrence coefficients, $\beta_{\ell m}$ Eq. (8)
  for every $\ell = m+2, ..., \ell_{max}$ **do**
    ∘ compute $\mathcal{P}_{\ell m}$ via the 2-point recurrence, given precomputed $\boldsymbol{\beta}_{\ell m}$, Eq. (7)
    ∘ update $\boldsymbol{\Delta}_m^A(r)$, Eq. (12)
  end for (ℓ)
end for (r) **and** (m)

global communication
  ∘ $\left\{\boldsymbol{\Delta}_m^A(r),\ m \in \mathcal{M}_i,\ \underline{\text{all}}\ r \in \mathcal{R}\right\} \overset{\texttt{MPI\_Alltoallv}}{\Longleftrightarrow} \left\{\boldsymbol{\Delta}_m^A(r),\ r \in \mathcal{R}_i,\ \underline{\text{all}}\ m \in \mathcal{M}\right\}$

step 3 - $\boldsymbol{s}_n$ calculation
for every ring $r \in \mathcal{R}_i$ **do**
  ∘ via FFT calculate $\boldsymbol{s}(r, \phi)$ for all samples ϕ of ring r, given pre-computed $\boldsymbol{\Delta}_m^A(r)$ for $\underline{\text{all}}$ m.
end for(r)

return $\boldsymbol{s}_n$ for all $(r \in \mathcal{R}_i)$

n_{proc} MPI processes involved in the computation. Like its serial counterpart, the parallel algorithm can be also split into steps. First, we precompute the starting values of the Legendre recurrence $\boldsymbol{\mu}_m$ (9), but only results for $m \in \mathcal{M}_i$ are preserved in memory. In next step, we calculate the functions $\boldsymbol{\Delta}_m^A$ using Eq. (12) for *every* ring r of the sphere, but only for m values in $\mathcal{M}_i$. Once this is done, a global (with respect to

the number of processes involved in the computation) data exchange is performed so that at the end each process has in its memory the values of $\mathbf{\Delta}_m^A(r)$ computed only for rings $r \in \mathcal{R}_i$ but for *all* m values. Finally, via FFT we synthesize partial results as in equation (11). From the point of view of the framework, steps 1 and 2 constitute the first stage of the computation, followed by the global communication, and the second computation involving here a calculation of the FFT. As mentioned earlier, though the communication is an inherent part of the framework, the two computation stages can be adapted as needed. Hereafter, we adhere to the standard SHT algorithm as used in Algorithm 3 .

4.1.2 Performance analysis

Operation count. We note that the scalability of all the major data products can be assured by adopting an appropriate distribution of the harmonic coefficients and the map. This stems from the fact that apart of the input and output objects, i.e., maps and harmonic coefficients themselves, the volumes of which are inversely proportional to a number of the MPI processes by the construction, the largest data objects derived as intermediate products, are arrays Δ_A or Δ_S. Their sizes are in turn given as $\mathcal{O}(\mathcal{R}_N\, m_{max}/n_{proc})$, and again decrease inversely proportionally with n_{procs}. These objects are computed during the first computation stage, subsequently redistributed as the result of the global communication, and used as an input for the second stage of the computation. Their total volume, as integrated over the number of processes, therefore also determines the total volume of the communication, as discussed in the next Section. As these three objects are indeed the biggest memory consumers in the routines the overall memory usage scales as $(n_{pix} + 2\, m_{max}\, \ell_{max} + \mathcal{R}_N\, m_{max})/n_{procs}$. For typical pixelization schemes, full sky maps, analyzed at its sampling limit, we have $n_{pix} \sim \ell_{max}^2$, $\ell_{max} \simeq m_{max}$, and $\mathcal{R}_N \sim n_{pix}^{1/2}$, and therefore each of these three data objects contribute nearly evenly to the total memory budget of the routines. Consequently, the actual high memory water mark is typically $\sim 50\%$ higher than the volume of the input and output data. We note that if all three parameters, $\ell_{max},\ m_{max}, \mathcal{R}_N$, are allowed to assume arbitrary values memory re-using and, in particular, in-place implementations are not straightforward, as it may not be easy or possible to fit any of these data objects in the space assigned for others. For this reason, this kind of options are not implemented in the software discussed here.

Given the assumed data layout, also the number of floating point operations (FLOPS) required for the evaluation of FFTs and associated Legendre functions scales inversely proportionally to the number of the processes. The only exception is related to the cost of the initialization of the recurrence (Eq. 9), which is performed redundantly by each MPI process to avoid an extra communication. Consequently, each process receives at least one m value on order of m_{max} and has to perform at least $\mathcal{O}(m_{max})$ operations as part of the pre-computation, equation (9). A summary of the FLOPs required in each step of the parallel SHTs algorithms is presented in Table 1, where we have assumed here that the subsets $\mathcal{R}_i$ and $\mathcal{M}_i$ contain $\simeq \mathcal{R}_N/n_{proc}$ and $\simeq m_{max}/n_{proc}$ elements, respectively.

		FLOPs
Pre-computation	Step {2,1}	$O\,(m_{max})$
Recurrence	Step {3,2}	$O\left((\mathcal{R}_N \ell_{max} \dfrac{m_{max}}{n_{proc}}\right)$
FFTs	Step {1,3}	$O\left(\dfrac{\mathcal{R}_\mathcal{N}}{n_{proc}}\, m_{max} \log m_{max}\right)$

Table 1: Number of floating point operations required in the parallel SHTs algorithm.

Communication cost. In order to determine the overall performance of the framework we have to also estimate the communication cost. The data exchange is performed with help of a single collective operation of the type all-to-all in which each process sends data to, and receives from, every other process. We use a simple model to estimate the cost of the collective communication algorithms in terms of latency and bandwidth use i.e., we assume that the time taken to send a message between any two nodes can be modelled as $\alpha + \beta\, n$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte, and n is the number of bytes transferred. However in case of the collective communication routines, different MPI libraries use different algorithms, which moreover may depend on a size of the message. For specificity, we base our analysis on MPICH [19], a widely used MPI library. Similar considerations can be performed for other libraries.

The all-to-all communication in MPICH [31] for short messages (≤ 256 kilobytes per message) uses the index algorithm by Bruck et al. [8]. It is a store-and-forward algorithm that takes $\lceil \log n \rceil$ steps at the expense of some extra data communication ($\beta\, n \log n$ instead of $\beta\, n$). For long messages and even number of processes, MPICH uses a pairwise-exchange algorithm, which takes $n_{proc} - 1$ steps. On each step, each process calculates its target process and exchanges data directly with that process i.e., data is directly communicated from source to destination, with no intermediate steps.

Note that in case of even distribution of rings r and m values, the size of message to be exchanged between each pair of processes can be defined as

$$S_{msg} := \mathcal{R}_N \frac{m_{max}}{n_{proc}} n_{\mathbb{C}}, \tag{16}$$

where $n_{\mathbb{C}}$ is the size in bytes of a complex number representation (usually $n_{\mathbb{C}} = 16$).

Finally, merging all the assumptions listed above we can derive a following formula for the total time require for a communication in our parallel SHTs algorithm.

$$T_{comm} = \begin{cases} \alpha \log n_{proc} + \beta\, S_{msg} \log n_{proc} & \text{when } S_{msg} \leq 256\,\text{kB} \\ \alpha\,(n_{proc} - 1) + \beta\, S_{msg}\ (n_{proc} - 1) & \text{when } S_{msg} > 256\,\text{kB} \end{cases} . \tag{17}$$

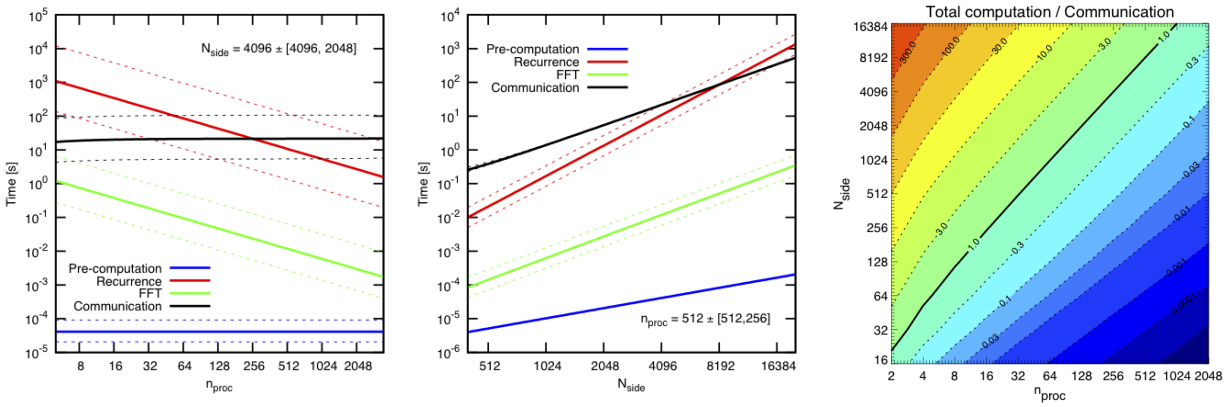


Figure 4: The two left most plots show theoretical run-times (logarithmic scale) as function of a number of processes and a fixed size of the problem, left most panel, and as a function of a resolution/size of the problem for the fixed number of processes, middle. The sizes of the problem shown in the leftmost panel with thick, solid lines, correspond to $\simeq 2\ 10^8$ pixels and corresponding harmonic modes, as given the HEALPix parameter value $N_{side} = 4096$. Thin, dashed lines show cases with 4 times fewer (more) pixels, lower and upper lines respectively. In the middle panel the number of processes is assumed to be $n_{proc} = 512$, thick, solid line, or 256 (1024), as shown by thin dashed lines. The contours in the rightmost panel display a ratio of the total computation to communication times shown as a function of a number of process and size of the problem.

In Figure 4 we illustrate theoretical runtimes depending on the number of the employed processes (leftmost panel) and on the size of the problem (middle panel). The transform parameters have been set assuming a standard full sky, fully resolved case, and the HEALPix pixelization, i.e., $\ell_{max} = m_{max} = 2\, N_{side}$, $\mathcal{R}_N = 4\, N_{side} - 1$, $n_{pix} = 12\, N_{side}^2$. Here, N_{side} is the HEALPix resolution parameter (for more details see beginning of the section 5 and [13]). For the hardware parameters, in the estimation of the communication cost we have used following [24, 27] $\alpha = 10^{-5}$, $\beta = 10^{-9}$, while for the cost of calculations we have assumed that each MPI process attains the effective performance of 10Giga operations per second, resulting in a time of $\gamma = 10^{-10}$ seconds per FLOP. The choice we have made is meant to be only indicative, yet at its face value the latter number is more typical of the MPI processes being nodes of multi-cores than just single cores, in particular given the usual efficiency of such calculations, which is usually on order of $10 - 20\%$. If less efficient MPI processes are used then the solid curves in Fig. 4 corresponding to the computation times should be shifted upwards by a relevant factor.

The dominant calculation is the recurrence needed to compute the associated Legendre functions in agreement with our earlier numerical result, Fig. 2. The computation scales nearly perfectly with the number of processed used decaying as $\propto 1/n_{proc}$ and it increases with growing size of the problem, $\propto N_{side}^2$. At the same time the communication cost is seen to be independent on a number of processes. This is because for the considered here numbers of processes the size of the message exchanged between any two

processes is always large and the communication time is thus described by the second term of the second equation of Eqs. 17. Given that the single message size, S_{msgs}, decreases linearly with a number of processes, the total communication time does not depend on it. The immediate consequence of these two observations is that both these times will be found comparable if sufficiently large number of processes is used. A number of processes at which such a transition takes place will depend on the constants entering in the calculation of Eqs. 17 and the size of the problem. This dependence of the critical number of processes on the size of the problem is shown in the rightmost panel of Fig. 4 with thick solid contour labelled 1.0. Clearly, from the perspective of the transform runtimes there is no extra gain, which could be obtained from using more processes than given by their critical value. In practice, a number of processes may need to be fixed by memory rather than computational efficiency considerations. As the memory consumption depends on the problem size, i.e., n_{pix}, in the same way as the communication volume, by increasing the number of processes in unison with the size of the problem we can ensure that the transform runs efficiently, i.e., the fraction of the total execution time spent on communicating will not change, and that it will always fit within the machines memory.

We note that some improvement in the overall cost of the communication could be expected by employing a non-blocking communication pattern applied to smaller data objects, i.e., single rows of the arrays, Δ_A and Δ_S corresponding to a single ring or m value, successively and immediately after they become available. Though such an approach could extend the parameter space in which the computation time is dominant and the overall scaling favorable, it will most likely have only a limited impact given that the same, large, total volume of the data, which needs to be communicated. The communication volume could be decreased, and thus again the parameter space with the computation dominance extended, if redundant data distributions and redundant calculations are allowed for in the spirit of the currently popular *communication avoiding* algorithms, e.g., [14]. Though these kinds of approaches are clearly of interest they are outside of the scope of this work and are left here for future research.

We also not that the conclusions may disfavour algorithms which require on the one hand big memory overhead and/or need to communicate more. Such algorithms will tend to require a larger number of processes to be used for a given size of the problem and thus will quickly get into bad scaling regime. Their application domain may be therefore rather limited. We emphasize that this general conclusion should be reevaluated case-by-case for any specific algorithms. Likewise, it is clear that the regime of the scalability of the standard SHT algorithm could be extended by decreasing the communication volume. Given that the communicated objects are dense, the latter would typically require some lossy compression techniques, we do not consider them here. Instead we focus our attention on accelerating intra-node calculations. This could improve the transforms runtimes for a small number of processes, however, if successful unavoidably will also result in lowering the critical values for the processes numbers. leading to losing the scalability earlier. Nevertheless, the overall performance of these improved transforms would never be inferior to that of the standard ones whatever a number of processes.

4.2 Exploiting intra-node parallelism

In this Section we consider two extensions of the basic parallel algorithm, Algorithm 3, each going beyond its MPI-only paradigm. Our focus hereafter is therefore on the computational tasks as performed by each of the MPI processes within the top-level framework outlined earlier. Guided by the theoretical models from the previous Section we expect that any gain in performance of a single MPI process will translate directly into analogous gain in the overall performance of the transforms at least as long as the communication time is subdominant, i.e., in a relatively broad regime of cases of practical importance. Hereafter we therefore develop implementations suitable for multi-core processors and GPUs and discuss the benefits and trade-offs they imply.

As highlighted by Fig. 4, of the three computational steps, this is the recurrence used to compute the associated Legendre functions which takes by far the dominant fraction of the overall runtime. Consequently, we will pay special attention to accelerating this part of the algorithm, while we will keep on using standard libraries to compute the FFTs. The latter will be shown to deliver a sufficient performance given our goals here. The associated Legendre function recurrence involves three nested loops and we will particularly consider the benefits and downside of their different ordering in order to attain the best possible performance.

Hereafter, for simplicity we will refer to the algorithms for direct and inverse spherical transforms by the core names of their corresponding routines i.e., `map2alm` and `alm2map` respectively.

4.2.1 Multithreaded version for multi-core processors

Since the largest and fastest computers in the world today employ both shared and distributed memory architectures, software which use only a distributed memory approaches such as MPI may not fully exploit

the shared memory underlying architecture and thus may loose some efficiency. Shared memory approaches, based on explicit multithreading or language-generated multithreading (e.g. OpenMP) are more accurate on shared memory architectures. Thus, many hybrid approaches, typically mixing MPI and OpenMP, have been proposed to better exploit clusters of of multi-core processors. In this spirit, we introduce the second level of parallelism based on multithreaded approach in the computation of SHTs.

As discussed shortly at the beginning of this section, the order of the three nested loops can be changed. For instance, the iterations over m in the outermost loop allow precomputing $\beta_{\ell m}$ Eq. (8), $\mathcal{P}_{mm}$ Eq. (9) and $\mathcal{P}_{m+1,m}$ Eq. (10) only once for each ring with extra storage of an order $\mathcal{O}(\ell_{max})$. But in this case we also need to store intermediate products which require additional $\mathcal{O}(\mathcal{R}_N \ell_{max})$ storage comparable to the size of the input and output. However modern super computers are equipped with memory banks of size in tens of GB. This fostered us for developing multithreaded versions of Algorithm 1 and 2 in which we place loop over m outermost. That is, we introduced the multithread layer via OpenMP for m loop i.e., a set of $m \in \mathcal{M}_i$ values (already assigned to i^{th} MPI process) is evenly divided into a number of subsets equal to the number of threads mapped onto each physical core available on a given multi-core processor. We denote this subset of m values as $\mathcal{M}_i^{\mathrm{T}}$ where subindex i is an i^{th} thread. In such a way each physical core executes one thread concurrently and calculates intermediate results of SHTs for its local (in respect to shared memory) subset of m. The Algorithm 4 describes the STEP 1 of Algorithm 3 in its multithreaded version.

Algorithm 4 $\mathbf{\Delta}_m^A$ CALCULATION PER THREAD

Require: all $m \in \mathcal{M}_i^T \Rightarrow \bigcup_{i=1}^{\text{Nth}} \mathcal{M}_i^T = \mathcal{M}_i$
  for all $m \in \mathcal{M}_i^T$ **do**
    ◦ initialise $\mathcal{P}_{mm}$ Eqs. (9)
    for every ring r **do**
      for every $\ell = m+2, ..., \ell_{max}$ **do**
        ◦ precompute $\beta_{\ell m}$ Eq. (8)
        ◦ compute $\mathcal{P}_{\ell m}$ via $\boldsymbol{\beta}_{\ell m}$, Eq. (7)
        ◦ update $\mathbf{\Delta}_m^A(r)$ in shared memory, Eq. (12)
      end for (ℓ)
    end for(r)
  end for(m)

Load balance between active threads

Selecting a loop over m as an outermost one is certainly the best approach from the point of view of speed optimization (see previous subsection), but such configuration leads to poor load balance. Notice that due to assumptions listed in section 2.2 the ℓ-recursion starts from $\ell = m$ (see for example Eq. 12). Therefore, the number of flops required for evaluating the recurrence from equation (9) differs for different m. For that reason, we have to carefully map subsets $\mathcal{M}_i^T$ onto active threads to ensure that the total time required for the evaluation of the problem on a given node will be equally divided by the number of available cores.

Because the lenght of the loop over ℓ decreases with growing number of m, we create subsets M_i^T by taking values from the set M_i in "max-min" fashion i.e., iteratively we take from $\mathcal{M}_i$ the maximum value and the minimal one and we assign them to a chosen thread. In the same moment we try to equilibrate the number of such pairs within each subset $\mathcal{M}_i^T$. Example of a such a mapping is depicted in Figure 5.

In next subsection, we introduce a similar multithreaded approach for accelerating the projection of associated Legendre functions using GPU.

4.2.2 SHTs on GPGPU

A CUDA (Compute Unified Device Architecture) device is a chip consisting of hundreds of simple cores, called Streaming Processors (SP) which execute instructions sequentially. Groups of such processors are encapsulated in a Streaming Multiprocessor (SM) which executes instruction in a SIMD (Single Instruction, Multiple Data) fashion. Therefore each SM manages hundreds of active threads in a cyclic pipeline in order to hide memory latency i.e., a number of threads are grouped together and scheduled as a Warp, executing the same instructions in parallel. Therefore CUDA threads are grouped together into a series of thread blocks. All threads in the thread block execute on the same SM, and can synchronise and exchange data using very fast but small (up to 48KB in size for the latest NVIDIA carts) shared memory. The shared

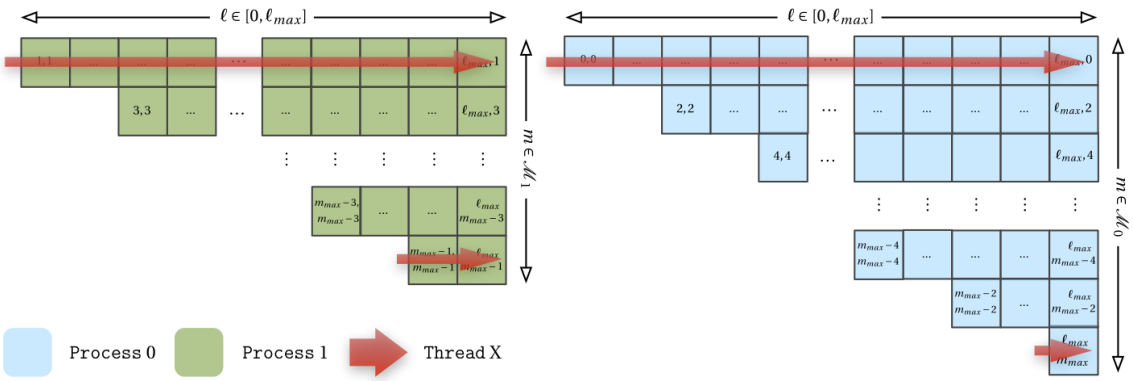


Figure 5: Example of two "min-max" pairs of m values assigned to Thread X on Process 1 (left) and 0 (right) respectively. The read arrow represents the direction and the lenght of the ℓ-recurrences (Eq. 10) evaluated by a thread for a given "min-max" pair.

memory can be used as a user-managed cache enabling higher bandwidth than direct data copy from the slow device global memory.

The restrictions related to memory (both size and speed) and the lack of synchronisation between the blocks of threads drive the structure of our algorithms devised for GPUs. For instance, placing the loop over m as outermost (like in algorithms dedicated for multi-core CPUs) is not appropriate for CUDA applications due to the memory size limitation. Consequently, the vector $\boldsymbol{\beta}_{\ell m}$ cannot be entirely stored in shared memory. Its values would need to be recomputed for each m and accessed sequentially in the ℓ-loop. A possible implementation would require re-computing the $\boldsymbol{\beta}_{\ell m}$ coefficients for each pass through the ℓ-loop, which would significantly affect the performance. Therefore, to agree with the philosophy of programming for GPUs, which consists in using fine grained parallelism and launching a very large number of threads in order to use all the available cores and hide memory latency, we set iterations over r in the outermost loop and parallelize it in such a way that a number of rings (preferably only one) is assigned to one thread. In this model each thread computes the 2-point recurrence for all *available* m-values (but we assume that GPU is a part of distributed memory system therefore) in contiguous fashion as depicted on Figure 6. This makes it straightforward to plan the computation of $\boldsymbol{\beta}_{\ell m}$ via Eq. (9) in batches, which can be cached and shared inside a thread block. Note that in case of distributed memory systems the number of m values in set $\mathcal{M}_i$ may vary for different i, while the total number of rings $\mathcal{R}_N$ stays constant. Therefore, it is easier to plan good load balance for GPU when we parallize our algorithms over r loop.

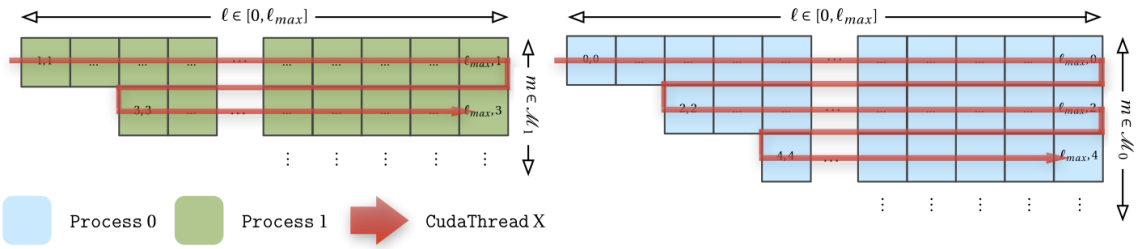


Figure 6: Schematic view of ℓ-recurrence iterations (eq. 10) evaluated by each thread on CUDA device assigned to Process 1 and 0 respectively.

Setting loop over rings outermost in our nested loop triggers two completely different approaches for memory management in direct and inverse SHTs. Therefore we dedicate the following two subsections to describe in detail the differences between `map2alm` and `alm2map` algorithms for GPUs.

`map2alm`

Mapping rings onto big number of threads, indeed facilitates the parallelization of SHTs algorithm on GPUs i.e., in such set-up each active thread perform exactly the same number and type of operations but with

different values. However, the direct transform requires that for each ring we update all non-zero values of the matrix $\boldsymbol{a}_{\ell,m} \in \mathbb{C}^{m_{max} \times \ell_{max}}$. This leads to all-reduce operation[6] between active threads. This is to say, the partial results generated by all active threads need to be summed together and written in the memory space from which we will read the final solution after all threads will be executed. Note that on CUDA, active threads can communicate only in thread block, which makes this task more difficult.

The Algorithm 5 outlines the STEP 3 of algorithm 1 adapted to CUDA. We assume however that we are working on a memory distributed system, therefore the range of m values is limited to $\mathcal{M}_i$. To ensure workaround for the high latency device memory and take advantage of the fast shared memory, we first calculate the values of $\boldsymbol{\beta}_{\ell m}$ vectors in segments, which allows us to fit them to the shared memory i.e., all threads within a block in a "collaborative" way compute in advance $\boldsymbol{\beta}_{\ell m}$ values and store them in shared memory. Next, we use those precomputed values to evaluate associated Legendre function $\mathcal{P}_{\ell m}(r)$. After, each active thread fetches from global memory a precomputed sum $\boldsymbol{\Delta}_m^S(r)$ and performs a final multiplication Eq. (13). Once it's done, we sum all those partial results within a given block of threads using adapted[7] version of parallel reduction primitive from CUDA SDK. For such locally reduced result one thread in block updates corresponding entry in global memory. This implies that $j =$ `NBT` (number of blocks of threads) threads will try to increase the value of a variable at the same space of global memory - which is referred to as a *race condition problem*, common in multithread applications. Fortunately, race conditions are easy to avoid in CUDA by using atomic operations. An atomic operation is capable of reading, modifying, and writing a value back to memory without the interference of any other threads, which guarantees that a race condition will not occur. Therefore, all final updates of final results in global memory are performed via CUDA `atomicAdd` function. In case when atomic functions are not supported on a given GPU we can easily avoid race conditions by increasing granularity of recurrence step. In this scenario we loop over m values on CPU and execute $\mathcal{M}_i$ "*smaller*" CUDA kernels which for a fixed values m and $\boldsymbol{\mu}_m$, evaluate `NBT` reduced (per block of threads) results $a^{\star}_{\ell m}$, which are next copy from device to host memory and sum together using CPU. This variant of our algorithm corresponds to the state `HYBRID_EVALUATION = true`, in listing 5.

`alm2map`

The inverse SHT also requires the reduction of partial results, but in contrary to the direct transform, we need to sum them for all ℓ as displayed in Eq. (12). With rings assigned to single threads, this leads to rather straightforward implementation. Algorithm 6 outlines the `alm2map` algorithm devised for GPUs and it is based on the version detailed in [15], which conforms with all assumptions stated in this paper.

First, we partition the vector of the $\boldsymbol{a}_{\ell m}$ coefficients in the global memory into small tiles. In this way a block of threads can fetch resulting segments in sequence and fit them in the shared memory. Next, we calculate $\boldsymbol{\beta}_{\ell m}$ values in the same way as in the `map2alm` algorithm. With all the necessary data in the fast shared memory, in following steps, each thread evaluates in parallel an associated Legendre function and updates the partial result $\boldsymbol{\Delta}_m^A$. In contrast to the direct transform, those updates are performed without a race condition problem i.e., each thread stores its temporary values in small space of the shared memory to which it has an exclusive and very fast access. Once, the loop over all ℓ is finished, the final partial result is written to the global memory.

4.3 Exploitation of hybrid architectures

Many new supercomputers are hybrid clusters i.e., they are a combination of computational nodes interconnected (usually via PCI-Express bus) with multiply GPUs. In such a way, one or more multi-core processors have access to one GPU. Facing these heterogeneities, the front end user of the spherical harmonic package will have to decide which type of architecture to choose for calculation. The three-stage structure of our algorithms allows us to freely map different steps of the algorithms onto different accelerators in hybrid system. However, the steps have to be evaluated sequentially in unchanged order. The intention of this flexibility is to facilitate selection of the most efficient library for FFTs e.g. NVIDIA provides its own library for FFT on GPUs i.e., `CUFFT` [21] which efficiency strongly depends on the version of CUDA device and the version of the routine. Therefore, on some supercomputers with highly tuned version of FFTW3 library, FFTs perform better on multi-core CPUs. Note that our algorithms require the evaluation of FFTs on complex data, which length may vary from ring to ring.

In practice, on heterogeneous systems we assign each step of the algorithm to the architecture on which it performs faster. The default assignment for both transformation is depicted in Figure 7. Note that due

[6] Reduce-type operation combines the distributed elements using predefined operation (e.g., summation, multiplication etc.) and returns the combined value in the output buffer.

[7] We omitted a part of the code which writes partial result to global memory

Algorithm 5 $\boldsymbol{a}_{\ell m}$ CALCULATION ON CPU&GPU

Require: $\boldsymbol{\Delta}_m^S$ & $\boldsymbol{\mu}_m$ vectors in GPU global memory for $m \in \mathcal{M}_i$
for every ring r assign to one GPU thread **do**
for every $m \in \mathcal{M}_i$ **do**
for every $\ell = m+2, ..., \ell_{max}$ **do**
- use precomputed or, if needed, precompute in parallel a segment of $\boldsymbol{\beta}_{\ell m}$ (8);
- compute $\mathcal{P}_{\ell m}$ via Eq. 7;
- evaluate: $a^r_{\ell m} = \boldsymbol{\Delta}_m^S(r)\,\mathcal{P}_{\ell m}(r)$
- sum all partial results $a^r_{\ell m}$ within a block of threads

$$a^\star_{\ell m} = \sum_{i=0}^{\texttt{NTPB}} (a^r_{\ell m})_i \tag{18}$$

if `HYBRID_EVALUATION` **then**
- for each block of threads with different id = `BID`, write $\boldsymbol{a}^\star_{\ell m}$ in global device memory

$$\boldsymbol{a}^{GPU}_{\ell m}[\texttt{BID}] = a^\star_{\ell m}$$

- copy partial results $\boldsymbol{a}^{GPU}_{\ell m}[\ldots]$ from device to host memory and sum together using CPU

for $i = 0 \to$ `NBT` **do**
$\boldsymbol{a}_{\ell m} = \boldsymbol{a}_{\ell m} + a^{\texttt{GPU}}_{\ell m}[i]$
end for
else
- update $\boldsymbol{a}_{\ell m}$ in global memory via atomic addition: $\boldsymbol{a}_{\ell m} = \boldsymbol{a}_{\ell m} + a^\star_{\ell m}$

end if
end for (ℓ)
end for(m)
end for(r)
return $\boldsymbol{a}_{\ell,m} \in \mathbb{C}^{m_{max} \times \ell_{max}}$

Algorithm 6 $\boldsymbol{\Delta}_m^A$ CALCULATION ON GPU

Require: $\boldsymbol{a}_{\ell,m} \in \mathbb{C}^{m_{max} \times \ell_{max}}$ matrix in GPU global memory
for every ring r assign to one GPU thread **do**
for $m \in \mathcal{M}_i$ **do**
for every $\ell = m+2, ..., \ell_{max}$ **do**
- use fetched or, if needed, fetch in parallel a segment of $\boldsymbol{a}_{\ell m}$ data;
- use precomputed or, if needed, precompute in parallel a segment of $\boldsymbol{\beta}_{\ell m}$ (8);
- compute $\mathcal{P}_{\ell m}$ via Eq. 7;
- update $\boldsymbol{\Delta}_m^A(r)$ for a given prefetched $\boldsymbol{a}_{\ell m}$ and computed $\mathcal{P}_{\ell m}$ (12);

end for (ℓ)
- save final $\boldsymbol{\Delta}_m^A(r)$ in global memory

end for(m)
end for(r)
return vector of partial results: $\boldsymbol{\Delta}_m^A \in \mathbb{C}$

to their sequential nature, we omit steps in which we evaluate the starting values of the recurrence Eq. (9) (step 2 in Algorithm 1 and step 1 in Algorithm 2), which are always computed on CPU.

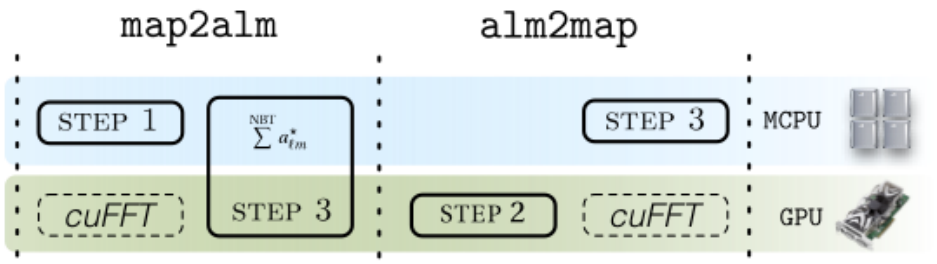


Figure 7: Default assignment of the major steps of the SHTs algorithm to one of the two architectures, multi-core processor or GPU, in a GPU/CPU heterogeneous system. Step 3 in `map2alm` algorithm covers two architectures due to optional use of CPU for reducing partial results between the execution of the CUDA kernels.

5 Numerical experiments

In order to evaluate the accuracy, efficiency and scalability of our new algorithm, we perform a series of tests with different geometry of the 2-sphere grid and different values of ℓ_{max} and m_{max}. To validate our algorithms, in each experiment we perform a pair of backward and forward SHTs starting out with a backward transform on a set of random $\boldsymbol{a}_{\ell m}^{init}$ coefficients which are uniformly distributed random numbers in the range $(-1, 1)$. To measure the accuracy of the algorithm we analyse the resulting map and we calculate the error as the difference between the final output $\boldsymbol{a}_{\ell m}^{out}$ and the initial set $\boldsymbol{a}_{\ell m}^{init}$ via the following formula,

$$\mathcal{D}_{err} = \sqrt{\frac{\sum_{\ell}\sum_{m}\left|\boldsymbol{a}_{\ell m}^{init} - \boldsymbol{a}_{\ell m}^{out}\right|^2}{\sum_{\ell}\sum_{m}\left|\boldsymbol{a}_{\ell m}^{init}\right|^2}}. \tag{19}$$

The pairs of transforms were tested on HEALPix grids with a different resolution parameter N_{side}. Accordingly to this parameter $n_{pix} = 12N_{side}^2$, $\mathcal{R}^N = 4N_{side} - 1$ where maximum lenght of the ring (equator) is equal to $4N_{side}$. Finally we also choose $\ell_{max} = m_{max} = 2 \times N_{side}$, except for some specific experiments.

5.1 Validation against other implementation

In this experiment, a set of $a_{\ell m}^{init}$ coefficients with a different value of ℓ_{max} was generated, then a pair of backward and forward transforms were performed to converted $a_{\ell m}^{init}$ to a HEALPIx map with a fixed $N_{side} = 1024$ and then back to $a_{\ell m}^{out}$. Starting with the same input array we used our both implementation (MPI/OpenMP & MPI/CUDA) and the **libpsht** library [23] to evaluate transforms. We measured the total time and the final error $\mathcal{D}_{err}$ Eq. (19).

Figure 8 depicts $\mathcal{D}_{err}$ errors for **s2hat_{mt,cuda}** and **libpsht** transforms pairs on HEALPIx grids with various ℓ_{max} and N_{side} values. Every data point is the maximum error value encountered in the pairs of transforms. As we can observe, all implementations provide almost identical quality of solutions with respect to different sizes of the problem.

5.1.1 Scaling with growing resolution and performance comparison with libpsht

To study the time scaling of the algorithm with growing resolution of the problem defined via N_{side} parameter, we perform a number of experiments with various $N_{side} = \ell_{max}/2$ parameter. Like in previous experiment we compare our results with **libpsht** package.

We choose Martin Reinecke's **libpsht** because it is to our knowledge, the fastest implementation of SHTs suitable for an astrophysical application and it is essentially a realisation of the same algorithm as the one used in our parallel algorithm. This highly optimized implementation is based on explicit multithreading and SIMD extensions (SSE and SSE2) but its functionality is limited to the shared memory systems only.

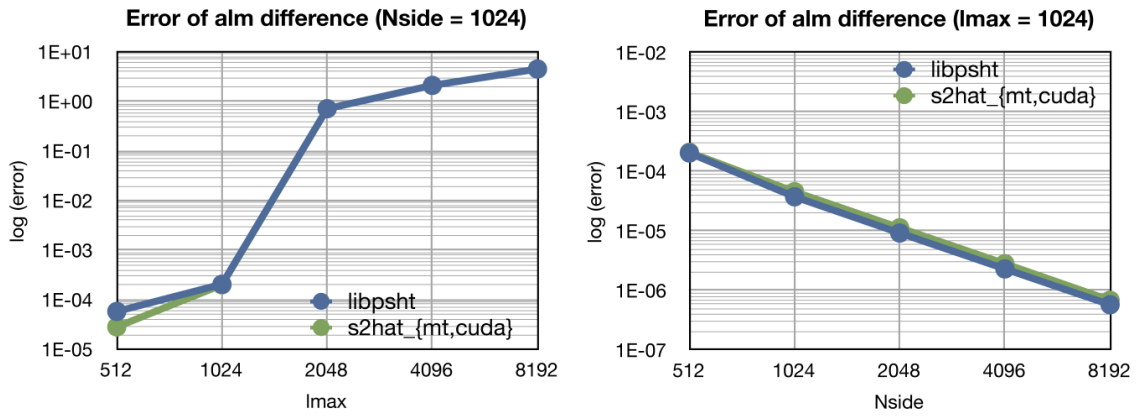


Figure 8: $\mathcal{D}_{err}$ for **s2hat_{mt,cuda}** and **libpsht** transforms pairs on HEALPIx grids with various ℓ_{max} and N_{side}. Every data point is the maximum error value encountered in transforms pairs. Loss of precision seen in the left panel for $\ell_{max} \gtrsim 2048$ and shared by both implementations is due to aliasing inherent to the adopted pixelization scheme.

CPU	*clock speed*	*GPU*	*Compilators*	*line style*
Core i7-960K	3.20 GHz	GeForce GTX 480	gcc 4.4.3 / nvcc 4.0	solid line
Core i7-2600K	3.40 GHz	GeForce GTX 460	gcc 4.4.5 / nvcc 4.0	dashed line
Xeon X5570	2.93 GHz	Tesla S1070	icc 12.1.0 / nvcc 4.0	dot line

Table 2: Different Intel multi-core processors and NVDIA GPUs used in our experiments.

Consequently, we perform all comparison experiments with only one pair of multi-core processor and GPU on three different systems listed in Table 2.

In a similar fashion as in our multithreaded version, all CPU-intensive parts of **libpsht**'s transforms algorithm are instrumented for parallel execution on multi-core computers using OpenMP directives. Therefore, in all our tests we set-up for all runs a number of OpenMP threads equal to the number of physical cores (OMP_NUM_THREADS = 4).

The range of pixelization sizes was limit by the size of the memory of a single GPU. Therefore we limit our tests to $N_{side} = 4096$. We also limit timing measurement in this experiment to the recurrence part only (step 3 in Algorithm 1 and step 2 in Algorithm 2), however in case of CUDA routines, those measurements includes also the time spend on any data movements between GPU and CPU.

The results of this experiment are shown in Figures 9 and 10. The figures depict the measured times in seconds and the number of Gflops per second in logarithmic scale. The number or flops include all the operations necessary for the purpose of the numerical stability like rescaling in case of s^{2}hat routines (see paragraph 2).

As expected, our multithreaded version is always slower in comparison to **libpsht** by a constant factor. Mainly, due to lack of SSE intrinsics in our code, which allows on some hardware to perform two or more arithmetic operations of the same kind with a single machine instruction. Especially on latest intel i7-2600K processor, SSE optimised **libpsht** implementation takes advantage of the extended width of the SIMD register[8] and Advanced Vector Extensions (AVX) which significantly increase parallelism and throughput in floating point SIMD calculations.

The other important observation is that forward and backward SHTs on CPU, with identical parameters, require almost identical time. Furthermore, performance analysis allows several deductions about performance of GPU version of SHTs algorithms.

- It is fairly evident that forward transform on CUDA with slower, not only with respect to its CPU version, but also in comparison with backward transform on the same architecture. Since the implementation of the CUDA kernel for realisation of projection onto associated Legendre polynomials is essentially the same for both directions of the transform, the performance lost is mainly due to very

[8]The width of the SIMD register in file on i7-2600K is increased from 128 bits to 256 bits in respect with previous generation of i7 processors

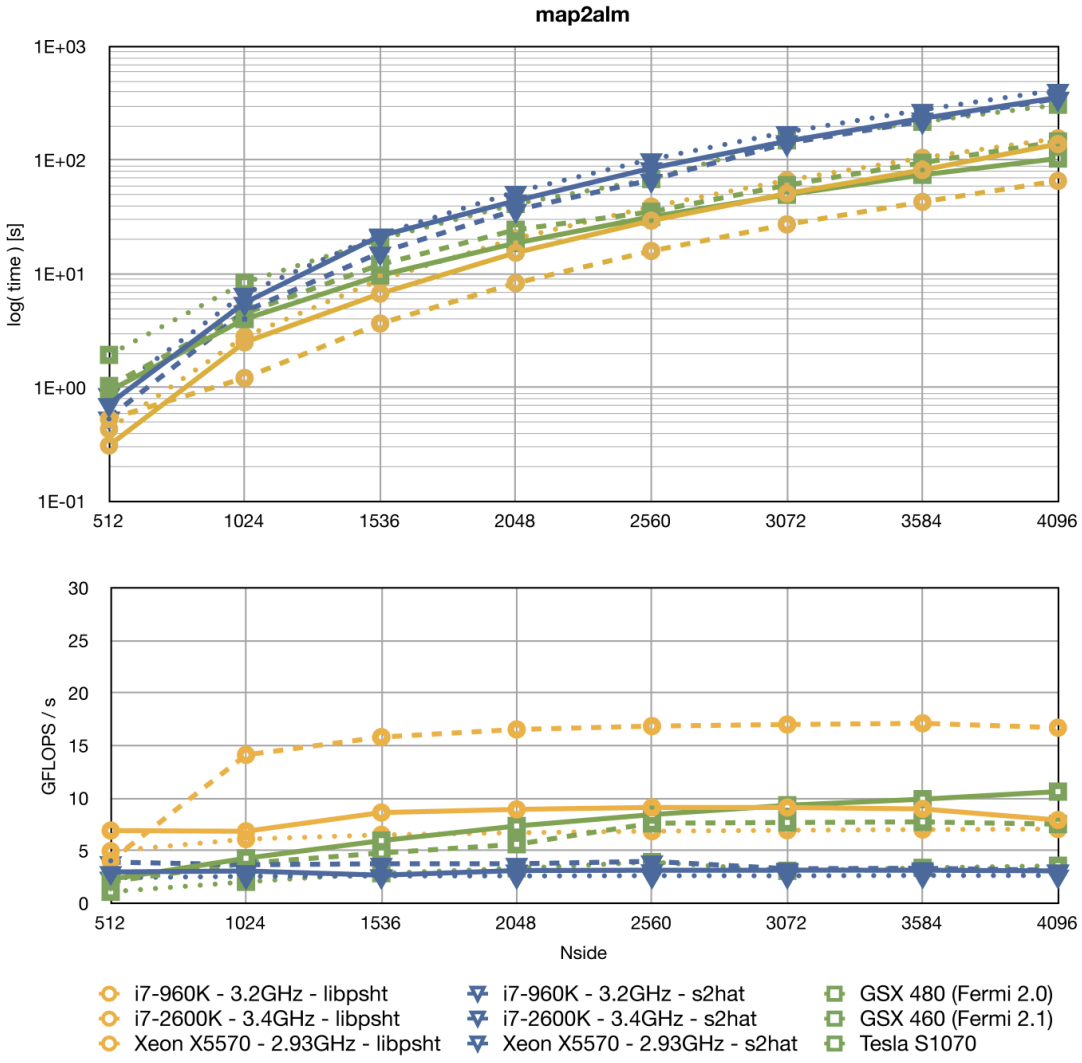


Figure 9: Benchmarks for direct SHT performed on different (see Table 2 for details) pairs of Intel multi-core processor with GPU on a HEALPIx grid. The top plot shows timings for projection on Legendre polynomials, the libpsht code (orange circles) is compared with hybrid S^{2}HAT codes (blue&green lines). In each case we use a HEALPIx grid with $N_{side} = \ell_{max}/2$. In case of CUDA routines, timings include time of data movements between CPU and GPU. Bottom pane shows number of GFLOPS per second. Note that CUDA routines loose performance in comparison to the SSE optimised lipsht due to very costly reduction of partial results. First in shared memory and then on host memory with use of CPU. Nevertheless, NVIDIA GTX 480 outperforms libpsht on Core i7-960K.

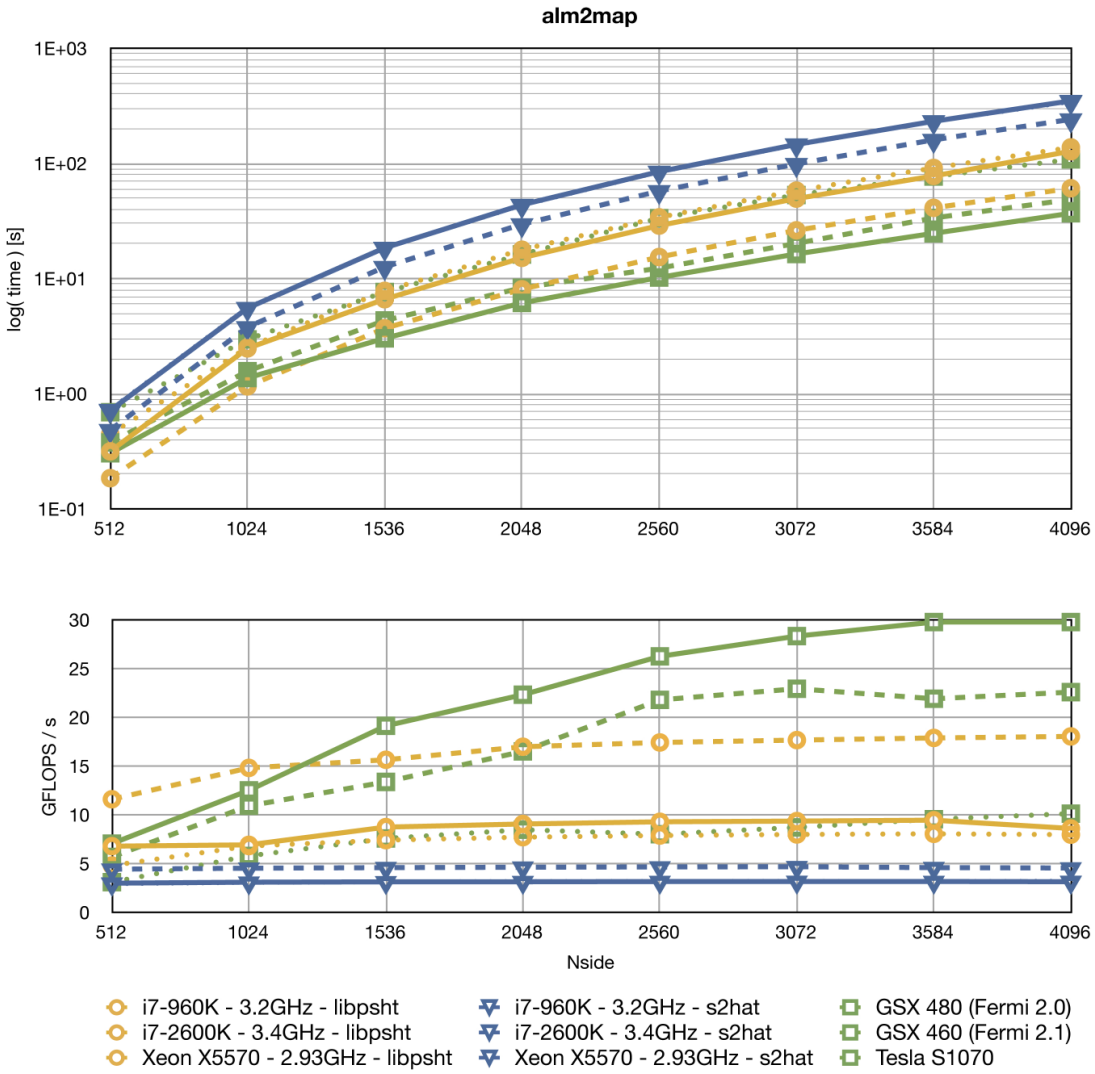


Figure 10: Benchmarks for inverse SHT performed on different (see Table 2 for details) pairs of Intel multi-core processor with GPU on a HEALPIx grid. The top pane shows timings for projection on Legendre polynomials, the libpsht code (orange circles) is compared with hybrid S^{2}HAT codes (blue&green lines). In each case we use a HEALPIx grid with $N_{side} = \ell_{max}/2$. In case of CUDA routines, timings includes time of data movements between host and device. Bottom pane shows number of GFLOPS per second. Note that GeForce 400 Series equipped with latest CUDA architecture (codename "Fermi") outperform libpsht on the latest, Intel Core i7-2600K. Bear in mind that GeForce family is not design for professional use therefore their double-precision throughput has been reduced to 25% of the full design.

expensive reduction of partial results. First in shared memory and then on slow global memory or on host memory with use of CPU.

- Thanks to the latest CUDA architecture (codename "Fermi") GeForce 400 Series of GPUs by far dominate over old generation Tesla S1070. Note that, unlike Tesla series, GeForce family is not designed for professional use therefore their double-precision throughput has been reduced to 25% of the full design. Nevertheless, those two GPUs (GTX 460 & GTX 480) outperform **libpsht** on the latest intel i7 processor in the evaluation inverse transform.

- CUDA version seems to favour cases with big N_{side} which indicates usage of many threads (as we explained in §4.2.2, each ring of the sphere is assigned to one CUDA thread) i.e., both GPU routines gain performance with growing number of sky-pixels (thus, also rings).

5.2 Scaling with number of multi-core processors and GPUs

The main purpose of using parallel SHTs is to exhibit data parallelism i.e., to allow us perform SHTs in big resolutions by distributing our input and output data across different computing nodes to be processed in parallel. A good parallel algorithm should scale with a size of the problem and a number of computing nodes. To measure those two quantities we perform a series of experiments with different values of N_{side} and a number of MPI processes spanned on hybrid pairs of multi-core CPU-GPU. For all experiments in this section, we set $\ell_{max} = 2 \times N_{side}$.

All experiments in this section were executed on CCRT's super computer, one of the first hybrid cluster built, based on Intel and Nvidia processors. It has 1068 nodes dedicated for computation, each node is composed of 2 Intel Nehalem quad-cores (2.93 GHz) and 24 GB of memory. One part of the system has a Multi-GPU support with 48 Nvidia Tesla S1070 servers, 4 Tesla cards with 4GB of memory and 960 processing units are available for each server. Two computation nodes are connected to one Tesla server by the PCI-Express bus, therefore 4 processors handle 4 Tesla cards. The high performance interconnect is based on Infiniband DDR. This system provides a peak performance of 100 Tflops for the Intel part and 200 Tflops for the GPU part. Intel compiler version 12.0.3 and NVIDIA Cuda compiler 4.0 were used for compilation. Furthermore, we span MPI processes in our experiment in such a way that exactly one MPI is assigned to one pair of multi-core processor and GPU (4 MPI processes per one Tesla server). To increase intra-node parallelism we set-up for all runs a number of OpenMP threads equal to the number of physical cores of Nehalem CPU (OMP_NUM_THREADS = 4).

Figure 11 shows maximum overall time measured on one of 128 CPU/GPUs involves in computation of pair of SHTs in different size of the map. As we can observe, our both implementations scale well with respect to the size of the problem.

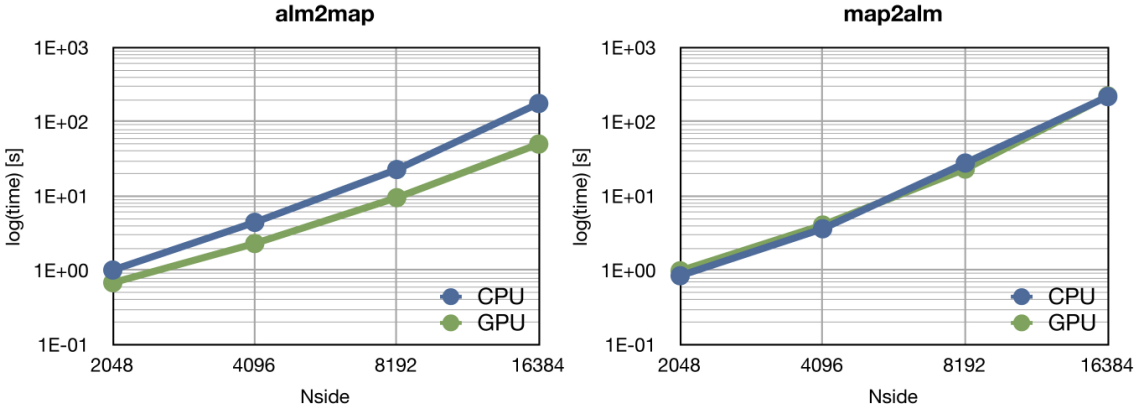


Figure 11: Maximum elapsed wall clock time for a pair of transforms with various N_{side} and $\ell_{max} = 2 \times N_{side}$. Experiment evaluated with 128 MPI processes spanned on the same number of pairs Intel Nehalem quad-cores and Nvidia Tesla S1070.

Figures 12 and 13 depict the runtime distribution over main steps of the algorithm multiplied by the number of MPI processes. In both versions, all steps scale almost perfectly (at least up to 128 MPI processes). As expected, only the communication cost grows slowly with number of processes. Note, that according to our prediction (see §2.3), the evaluation of the projection onto associated Legendre polynomials dominates

by far over the remaining steps and it de facto defines the overall time. On contrary, the time spend on memory transfers between host and the GPU device never exceed $1s$, and deviation from perfect scalability visible on plots can be explained by random errors of `MPI_Wtime()` function used in our experiments for time measurement (due to its limited resolution, `MPI_Wtime()` rerun trustable results only for elapsed times greater then $1s$).

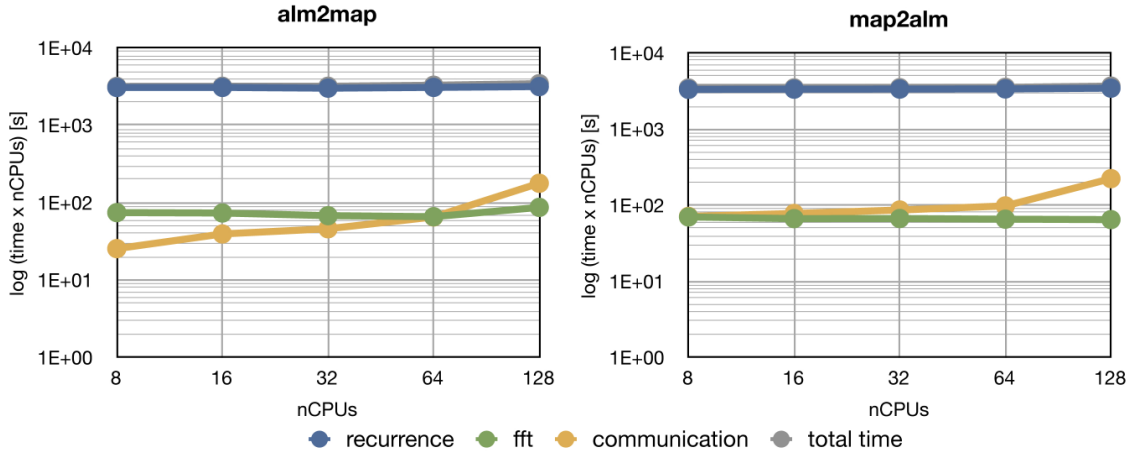


Figure 12: Maximum elapsed wall clock time distribution over main steps of the algorithm for the hybrid MPI/OpenMP version of the **alm2map_mt** and **map2alm_mt** routines. Results for experiment with $N_{side} = 16384$ and $\ell_{max} = 2 \times N_{side}$.

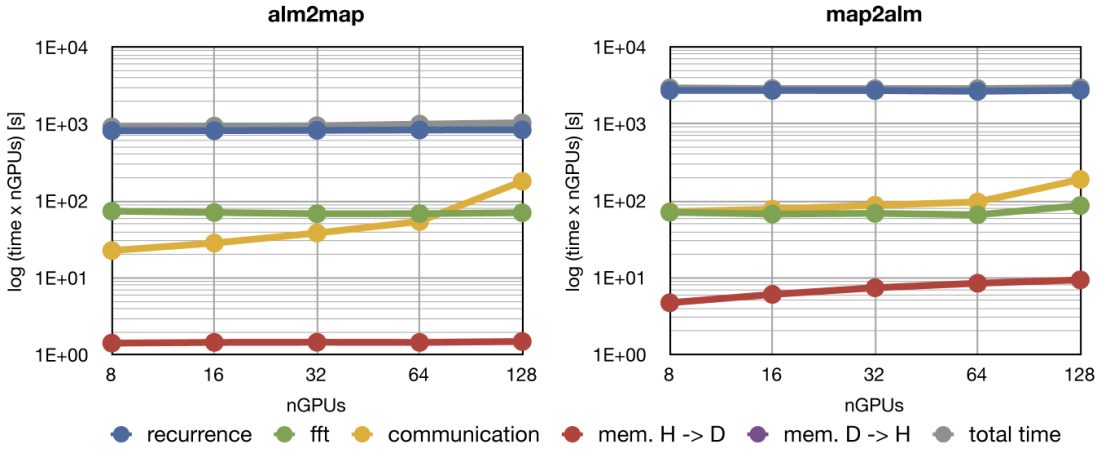


Figure 13: Maximum elapsed wall clock time distribution over steps of the algorithm for the hybrid MPI/CUDA version of the **alm2map_cuda** and **map2alm_cuda** routines. Results for experiment with $N_{side} = 16384$ and $\ell_{max} = 2 \times N_{side}$.

In both implementations we assign the evaluation of FFTs to the CPU due to its slightly better performance in comparison to CUFFT also available on Titane (see Fig. 14).

5.3 SHTs accuracy on CUDA

Both x86 and CUDA architecture used in our experiments respect the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985). IEEE 754 standardizes how arithmetic results should be approximated in floating point. Whenever working with inexact results, programming decisions can affect accuracy. It is important to consider many aspects of floating point behaviour in order to achieve the highest performance with the precision required for any specific application. This is especially true in a heterogeneous computing environment where operations are perform on different types of hardware.

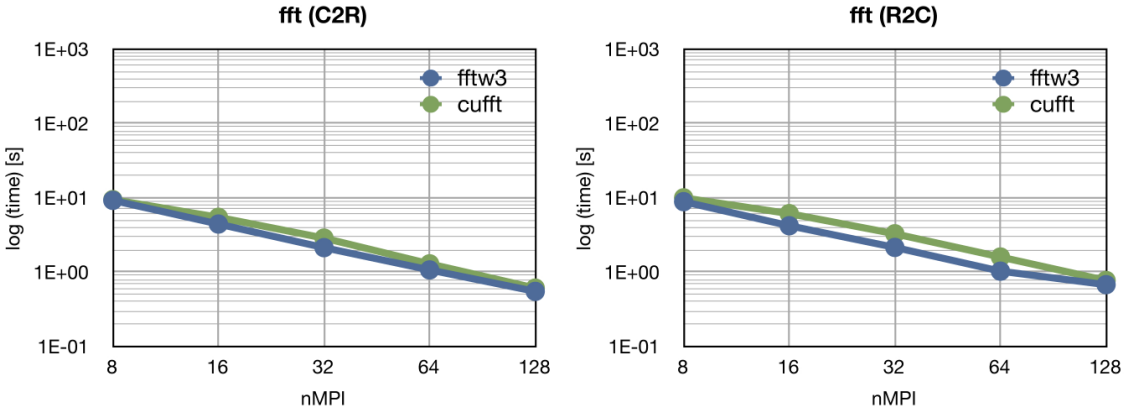


Figure 14: The overall difference between the FFTs evaluation on Titane supercomputer via cuFFT on GPU and Intel MKL Fourier Transform wrapped to FFTW3 interface. Values correspond to experiment with $N_{side} = 16384$ and $\ell_{max} = 2 \times N_{side}$.

Finally, if we compare overall time between CPU and GPU version, we find out that implementation of **alm2map** for CUDA is in all our big test cases, in average $\times 3.3$ faster then multithreaded version. In contrary, **map2alm** due to time consuming reduction operations remains slower then its counterpart dedicated for CPUs. Average speed-up of hybrid implementation MPI/CUDA with respect to hybrid MPI/OpenMP version is depicted in Figure 15

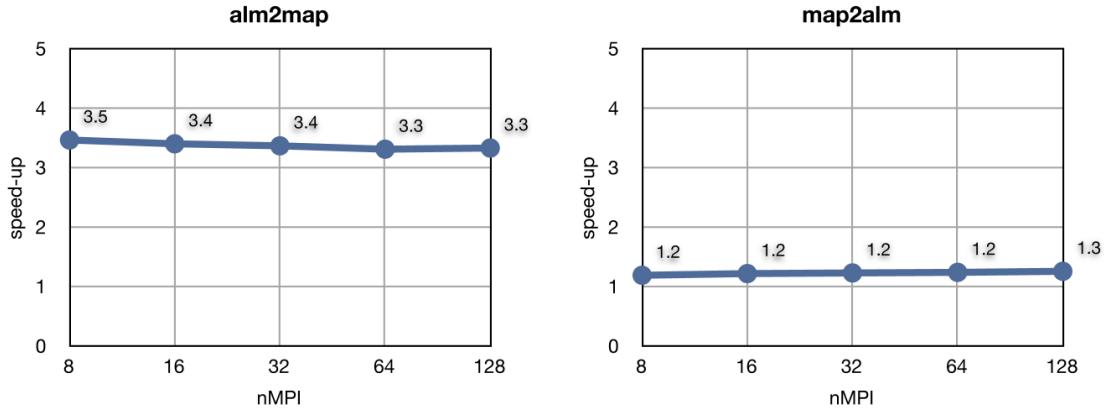
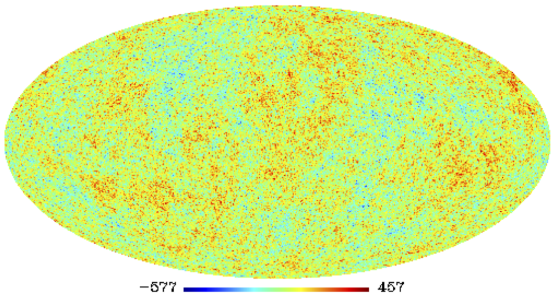
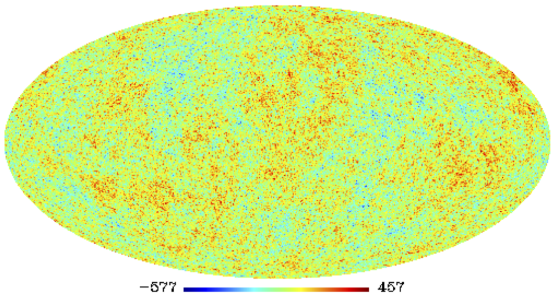


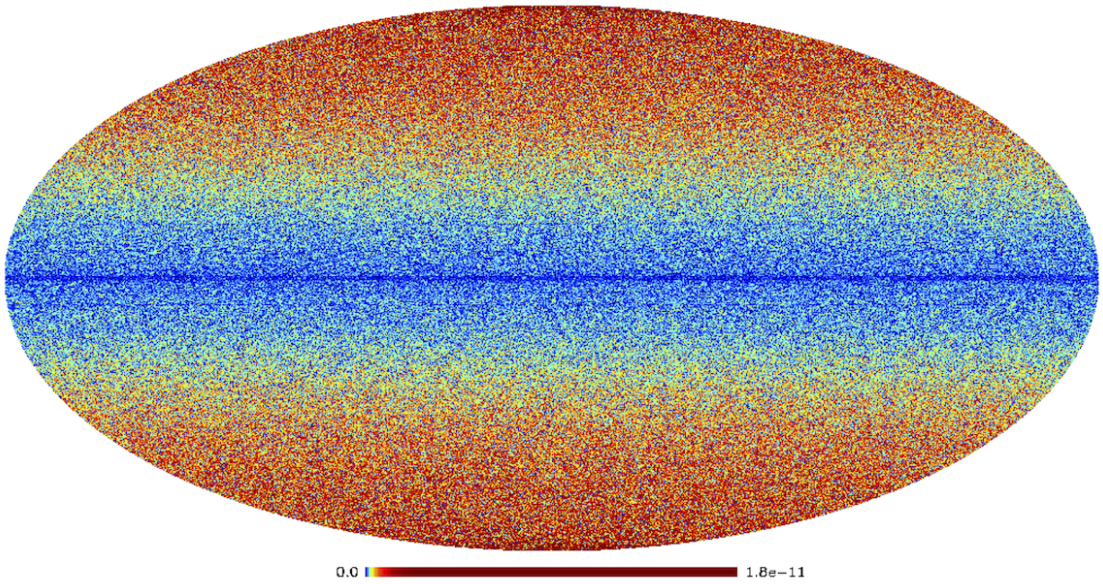
Figure 15: Average speed-up of hybrid implementation MPI/CUDA with respect to hybrid MPI/OpenMP version for different number of MPI processes.



(a) Reference map evaluated on CPU



(b) Map generated on Nvidia Tesla S1070 GPU



(c) Absolute difference between maps generated on CPU and Nvidia GPU.

Figure 16: CMB maps generated on two different architectures and their absolute difference.

The IEEE 754 standard defines a handful of basic arithmetic operations like add, subtract, multiply, divide and square root. The results of these operations are guaranteed to be the same for all implementations of the standard, for a given format and rounding mode. However, IEEE standard do not define accuracy of math functions like sinus, cosines or logarithm, which depends on implementation. For instance CUDA math library includes all the math functions listed in the C99 standard plus some additional useful functions [22]. These functions have been tuned for a reasonable compromise between performance and accuracy which may differ from their equivalent on x86 architectures. For that reason, we performed series of experiment in which we converted the same random set $a_{\ell m}^{init}$ (see beginning of this section) to a HEALPIx map on two different architecture, namely NVIDIA GPU and x86 processor, and we calculate an absolute difference of those two maps (see the example result in Figure 16). In all our tests the maximum difference between two sky-pixels never exceeded value of $1.0 \cdot 10^{-11}$, which indicates a very good agreement between those two architectures.

6 Conclusion and future work

This paper describes a parallel algorithm for computing the SHTs with two variants of intra-node parallelism specific for given architectures i.e., multi-core processors and GPUs. We tested both versions in terms of accuracy, overall efficiency and scalability. We showed that our inverse SHTs with GeForce 400 Series

equipped with latest CUDA architecture (codename "Fermi") outperform state of the art implementation for multi-core processor executed on latest Intel Core i7-2600K.

For solving big problems, a cluster of NVIDIA Tesla S1070 can accelerate overall execution of inverse transform by as much as 3 times with respect to the hybrid MPI/OpenMP version executed on the same number of quad-core processors Intel Nahalem. We also showed that this nontrivial algorithm exhibits some difficulties in full adaptation to CUDA architecture i.e., a direct transform is not as well adapted to GPUs as an inverse transform, which impacts significantly overall performance. Nevertheless, direct transform with NVIDIA GTX 480 outperformed libpsht on Core i7-960K.

The current implementation still leaves some space for future optimizations. Therefore, as a future work, we may add SSE intrinsics (or/and Advanced Vector Extensions (AVX) for latest x86 architectures) in order to introduce low level vectorization on CPUs. From the algorithm side, it would be crucial to enhance the set of our routines to support transforms with spin quantities, which are important part of a CMB science.

7 Acknowledgments

This work has been supported in part by French National Research Agency (ANR) through COSINUS program (project MIDAS no. ANR-09-COSI-009). The HPC resources were provided by GENCI- [CCRT] (grants 2011-066647 and 2012-066647) in France and by the NERSC in the US, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Some of the results in this paper have been derived using the HEALPix package [13].

References

[1] CCSHT.

[2] GLESP.

[3] HEALPIX.

[4] LIBPSHT.

[5] S^{2}HAT.

[6] SPHARMONICKIT/S2KIT.

[7] SPHERPACK.

[8] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8:1143–1156, 1997.

[9] J. B. Drake, P. Worley, and E. D'Azevedo. Algorithm 888: Spherical harmonic transform algorithms. *ACM Trans. Math. Softw.*, 35:23:1–23:23, October 2008.

[10] J. R. Driscoll and D. M. Healy. Computing fourier transforms and convolutions on the 2-sphere. *Advances in Applied Mathematics*, 15(2):202 – 250, 1994.

[11] F. Elsner and B. D. Wandelt. ARKCoS: artifact-suppressed accelerated radial kernel convolution on the sphere. *Astronomy and Astrophysics*, 532:A35, Aug. 2011.

[12] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[13] K. M. Górski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *ApJ*, 622:759–771, Apr. 2005.

[14] L. Grigori, J. Demmel, and H. Xiang. Communication avoiding gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 29. IEEE Press, 2008.

[15] I. Hupca, J. Falcou, L. Grigori, and R. Stompor. Spherical harmonic transform with gpus. Technical Report 7409, INRIA, 2010.

[16] M. Inda, R. Bisseling, and D. Maslen. On the efficient parallel computation of legendre transforms. *SIAM Journal on Scientific Computing*, 23(1):271–303, 2001.

[17] R. W. Melton and L. M. Wills. An analysis of the spectral transfrom operations in climat and weather models. *SIAM J. Sci. Comput.*, 31(1):167–188, 2008.

[18] M. Mohlenkamp. A fast transform for spherical harmonics. *Journal of Fourier Analysis and Applications*, 5(2):159–184, 1999.

[19] MPICH2. *MPICH – A portable implementation of MPI*, 2011.

[20] P. F. Muciaccia, P. Natoli, and N. Vittorio. Fast Spherical Harmonic Analysis: A Quick Algorithm for Generating and/or Inverting Full-Sky, High-Resolution Cosmic Microwave Background Anisotropy Maps. *ApJ*, 488:L63+, Oct. 1997.

[21] Nvidia. *CUDA CUFFT Library*, 2010.

[22] Nvidia. *NVIDIA CUDA Programming Guide*, 2010.

[23] M. Reinecke. Libpsht - algorithms for efficient spherical harmonic transforms. *Astronomy and Astrophysics*, 526:A108, Feb. 2011.

[24] R. Rugina and K. Schauser. Predicting the running times of parallel programs by simulation. *Parallel Processing Symposium, International*, 0:0654, 1998.

[25] D. S. Seljebotn. Wavemoth-Fast Spherical Harmonic Transforms by Butterfly Matrix Compression. *Astrophysical Journal Supplement Series*, 199:5, Mar. 2012.

[26] V. Soman. Accelerating spherical harmonic transforms on the nvidia gpu. Technical report, Department of Electrical Engineering, University of Wisconsin, 2009.

[27] M. Sottile, V. Chandu, and D. Bader. Performance analysis of parallel programs via message-passing graph traversal. *Parallel and Distributed Processing Symposium, International*, 0:64, 2006.

[28] R. Suda and M. Takami. A fast spherical harmonics transform algorithm. *Math. Comput.*, 71:703–715, April 2002.

[29] P. N. Swarztrauber. *Vectorizing the FFTs, in Parallel Computations.* Academic Press, 1982.

[30] M. Tanner. *Tools for statistical inference. Observed data and data augmentation methods.* Springer Verlag, 1992.

[31] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *Int'l Journal of High Performance Computing Applications*, 19:49–66, 2005.

[32] M. Tygert. Fast algorithms for spherical harmonic expansions, ii. *Journal of Computational Physics*, 227(8):4260 – 4279, 2008.

[33] M. Tygert. Fast algorithms for spherical harmonic expansions, iii. *Journal of Computational Physics*, 229(18):6181–6192, 2010.

[34] Y. Wiaux, L. Jacques, P. Vielva, and P. Vandergheynst. Fast Directional Correlation on the Sphere with Steerable Filters. *ApJ*, 652:820–832, Nov. 2006.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399