INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Spherical harmonic transform on heterogeneous architectures using hybrid programming

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

N° 7635

15 April 2011

Thème NUM

Rapport de recherche

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

Spherical harmonic transform on heterogeneous architectures using hybrid programming

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

Thème NUM — Systèmes numériques
Équipe-Projet Grand-large

Rapport de recherche n° 7635 — 15 April 2011 — 14 pages

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

This paper describes a multi CPU-GPUs implementation of an inverse SHT, based on hybrid programming combining MPI and CUDA, and discusses its tests as motivated by these forthcoming applications.

We present performance comparisons of the multi GPU version and a hybrid, MPI/OpenMP version of the same transform. We find that one NVIDIA Tesla S1070 can accelerate overall execution time of the SHT by as much as 3 times with respect to the MPI/OpenMP version executed on one quad-core processor (Intel Nehalem 2.93 GHz) and, owing to very good scalability of both versions, 128 Tesla cards perform as good as 256 twelve-core processor (AMD Opteron 2.1 GHz).

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).

Spherical harmonic transform on heterogeneous architectures using hybrid programming

Résumé : Les transformations en harmoniques sphériques (SHT) sont au cœur de nombreuses applications scientifiques et pratiques allant de la modélisation du climat aux observations cosmologiques. Ces domaines nécessitent des simulations et des analyses de données expérimentales engendrant des larges volumes de données. Ceci représente un défi important pour les implémentations actuelles des transformations en harmoniques sphériques.

Ce papier décrit la mise en œuvre multi CPU-GPU d'une SHT inverse, basée sur une programmation hybride, combinant MPI et CUDA. Nous comparons les performances de la version multi GPU par rapport à une version hybride MPI / OpenMP de la même transformation. Nous constatons qu'une NVIDIA Tesla S1070 peut exécuter la SHT 3 fois plus rapidement que la version MPI / OpenMP exécutée sur un processeur quad-core (Intel Nehalem cadencé à 2,93 GHz) . De plus, en raison d'un très bon passage à l'échelle des deux versions, 128 cartes Tesla donnent d'aussi bonnes performances que 256 processeurs à 12 coeurs (AMD Op te ron 2,1 GHz).

Mots-clés : transformations en harmoniques sphériques, Multi-GPU, CUDA, OpenMP, architectures hybrides, programmation hybride

1 Introduction

Spherical harmonic function define a orthonormal complete basis for signals defined on a 2-dimensional sphere. Spherical harmonic transforms are therefore common in all fields of science, where such signals are encountered. These include a number of diverse areas ranging from weather forecasts and climate modeling, through geophysics and planetology, to various applications in astrophysics and cosmology. In these contexts a direct 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 a multiplication of huge matrices, defined on the sphere, and having properties to be 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., [13].

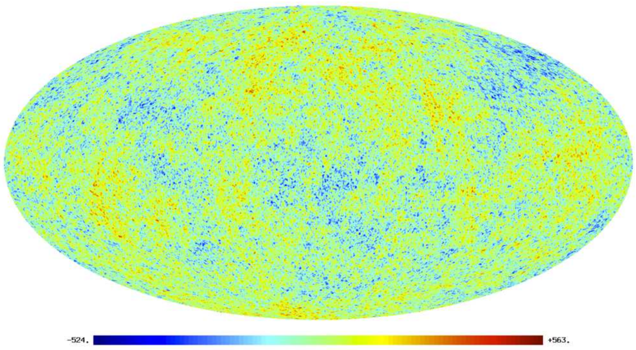


Figure 1: Sky map example synthesized using our MPI/CUDA implementation of the `alm2map` routine. The units are μK. An overall monopole term, with an amplitude of $\sim 2.7\,10^6\mu$K, has been subtracted to uncover the minute fluctuations.

The specific goal of this work is to assist simulation and analysis efforts related 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 $\sim$ 13Gyrs. This image confirms that the early Universe was nearly homogeneous and only small, 1 part in 10^5, deviations were present, e.g., 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 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 calculation of SHTs. For instance, the current and forthcoming balloon-borne and ground-based experiments will produce maps of 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 $\mathcal{O}(10^6)$ 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

[1] PLANCK: http://www.esa.int/SPECIALS/Planck/index.html
[2] COBE: http://lambda.gsfc.nasa.gov/product/cobe/
[3] WMAP: http://map.gsfc.nasa.gov/

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[4], GLESP[5], CCSHT[6], LIBPSHT[7], and S^2HAT[8], which are commonly used in the CMB research, and some others such as, SPHARMONICKIT/S2KIT[9] and SPHERPACK[10]. They propose different levels of parallelism and optimization, while, with an exception of two last ones, implement essentially the same algorithm. To date only its implementation in the S^2HAT library is fully parallel using MPI and scalable with respect to both memory usage and execution time. This is therefore the implementation, which we select as a starting point for this work.

The basic algorithm we use hereafter and which is described in the next Section, scales as $\mathcal{O}(\mathcal{R}_N\, \ell_{max}^2)$, where $\mathcal{R}_N$ is a number of rings (see next section) in the analyzed map and ℓ_{max} fixes the resolution and thus determines a 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 a total number of pixels, and therefore the typical complexity is $\mathcal{O}(\ell_{max}^3) \propto \mathcal{O}(n_{pix}^{3/2})$.

The algorithm lends itself however to further extensions in particular allowing to tackle efficiently cases when many similar transforms need to be performed. For instance, if the transforms need to be done in a sequence and a sufficient memory storage is available, the required calculations can be split in a precomputation step with the full complexity as above, but with an actual transform scaling then merely linearly with a number of pixels, n_{pix}. Similarly, if multiple identical transforms have to, or can, be done simultaneously, then the numerical cost is essentially as that of a single transform. In this paper we focus on the core algorithm and leave an investigation of these potential extensions to future work.

We note that alternative algorithms have been proposed, which display either better complexity, e.g., $\mathcal{O}(n_{pix}\, \ln n_{pix})$ [14] , $\mathcal{O}(n_{pix}\, \ln^2 n_{pix})$ [4], or comparable complexity but with lower memory requirements, e.g., $\mathcal{O}(n_{pix}\, \ln n_{pix})$ for the transform itself and $\mathcal{O}(n_{pix}^{3/2})$ for precomputations [11]. We point out however that unlike the standard $n_{pix}^{3/2}$ approach those algorithms may not be straightforwardly generalizable to the cases of so-called spin-weighted spherical harmonic transforms [6], which are, for instance, necessary for describing the polarized properties of the fields on a sphere, and thus of special practical interest. Moreover, at least in some of the cases, e.g., [14], the stability of the fast approaches is also unclear for the resolutions of interest from the point of view of the considered applications. The last but not least, their favorable scaling in the range of problem sizes of interest here is typically offset by significant pre-factors, rendering the final gains, in terms of the user time, rather minor, if any. Consequently, in spite of its inferior theoretical scaling, the standard algorithm appears not only to be more robust, and amenable to efficient parallelization, but also more versatile. It therefore provides a natural, first choice to be implemented and studied in the context of actual, real-life problems. Whether it will meet successfully all the anticipated, application-driven challenges, will depend on our ability to develop its efficient, scalable, parallel implementation.

In this work 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 CPUs and CPU/GPUs and is based on hybrid OpenMP/MPI and MPI/CUDA implementations. We find that once carefully optimized and implemented, the algorithm displays nearly perfect scaling in both these cases, at least up to 128 MPI processes, and that a single GPU card, NVIDIA Tesla S1070, is as efficient as 3 quad-core CPU processors, Intel Nehalem 2.93GHz. These results have been derived based on test cases concocted to correspond to data sizes driven by applications in the CMB area.

This paper is organized as follows. In section 2 we introduce the algebraic background of the spherical harmonic transforms. Next, in section 3 we introduce a detailed description of our parallel algorithm along with two variants suitable for clusters of GPUs and clusters of multi-cores. Section 4 presents results for both implementation. Section 5 concludes the paper.

[4] HEALPIX: http://healpix.jpl.nasa.gov/
[5] GLESP http://www.glesp.nbi.dk/
[6] CCSHT http://crd.lbl.gov/∼cmc/ccSHTlib/doc/
[7] LIBPSHT: http://www.mpa-garching.mpg.de/∼martin/ libpsht/doc/libpsht/index.html
[8] S2HAT: http://www.apc.univ-paris7.fr/∼radek/s2hat.html
[9] S2KIT: http://www.cs.dartmouth.edu/∼geelong/sphere/
[10] SPHEREPACK: http://www.cisl.ucar.edu/css/software/spherepack/

2 Algebraic background

In the actual applications the SHT is used either to represent a discretized scalar field $\boldsymbol{s}_n$ in terms of its harmonic (spectral) modes – an operation called a direct SHT or an *analysis* step,

$$\boldsymbol{a}_{\ell m} \quad = \quad \sum_{\{\theta,\phi\}} \boldsymbol{s}_n\left(\theta,\phi\right) Y_{\ell m}\left(\theta_n,\phi_n\right), \tag{1}$$

or to reconstruct the field, $\boldsymbol{s}_n$, from its harmonic representation via an inverse SHT referred to also as a *synthesis* step,

$$\boldsymbol{s}_n\left(\theta_n,\phi_n\right) \quad = \quad \sum_{\ell=0}^{\ell_{max}} \sum_{m=-\ell}^{\ell} \boldsymbol{a}_{\ell m}\, Y_{\ell m}\left(\theta_n,\phi_n\right). \tag{2}$$

where $Y_{\ell m}\left(\theta,\phi\right)$ denotes the spherical harmonic i.e., a product of a trigonometric function and an associated Legendre function $\mathcal{P}_{\ell m}$, (3).

$$Y_{\ell m}\left(\theta,\phi\right) \equiv \mathcal{P}_{\ell m}\left(\cos\theta\right) e^{im\phi}. \tag{3}$$

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 centers of suitable chosen sky pixels. ℓ_{max} defines a maximum order of the Legendre function and thus a band-limit of the field $\boldsymbol{s}$ and in practice it is set by the experimental resolution.

In this paper we will focus on synthesis step, to which we will also refer to as an `alm2map` operation, hereafter. This can be done without loss of generality as the analysis step involves essentially identical set of operations. Our goal here is then to develop an efficient and scalable (in respect to number of processors and GPUs) implementation of equation (2). Hereafter, we restrict our considerations to some specific discretizations of the sphere, in which the pixels are arranged on $\mathcal{R}_N \approx \sqrt{n_{pix}}$ iso-latitudinal rings, where latitude of each ring is identify by a unique polar angle θ_n. Moreover, within each ring, pixels must be equidistant ($\Delta\phi =$CONST), though their number can vary from a ring to a ring. For practical reasons, nearly all spherical grids, which are currently use in CMB analysis as well as other fields conform with such assumptions.

Taking into account these restrictions, and omitting the index n for shortness, we can rewrite the synthesis step [7] as

$$\boldsymbol{s}\left(\theta,\phi\right) = \sum_{m=-\ell_{max}}^{\ell_{max}} e^{im\phi}\, \boldsymbol{\Delta}_m\left(\theta\right). \tag{4}$$

where $\boldsymbol{\Delta}_m\left(\theta\right)$ is a set of functions such as:

$$\boldsymbol{\Delta}_m\left(\theta\right) \equiv \begin{cases} \sum_{\ell=0}^{\ell_{max}} \boldsymbol{a}_{\ell 0}\, \mathcal{P}_{\ell 0}\left(\cos\theta\right), & m=0; \\ \sum_{\ell=m}^{\ell_{max}} \boldsymbol{a}_{\ell m}\, \mathcal{P}_{\ell m}\left(\cos\theta\right), & m>0; \\ \sum_{\ell=|m|}^{\ell_{max}} \boldsymbol{a}^{\dagger}_{\ell|m|}\, \mathcal{P}_{\ell|m|}\left(\cos\theta\right), & m<0, \end{cases} \tag{5}$$

Hence, the spectral harmonics transform can be split into two successive computations: one calculating the associate Legendre transform, Eq. (5) , and the other performing the inverse Fourier transform, Eq. (4). The former 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 associate Legendre functions, $\mathcal{P}_{\ell m}$, at cost $\propto \ell_{max}^2$. The Fourrier transform step has 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 inverse SHT computation as obtained for a run performed on a cluster of GPUs (see §4.3 for details). We can observe that indeed evaluation of Legendre transform by far dominates over FFT in terms of time of computation. At the core of the SHT is a quick and accurate computation of spherical harmonics, $Y_{\ell m}\left(\theta,\phi\right)$. This can be reduced to an evaluation of the associated Legendre functions via 2-point recurrence [1] with respect to the multipole number ℓ for a fixed value of m,

$$\mathcal{P}_{\ell+2,m}\left(x\right) = \beta_{\ell+2,m} x\, \mathcal{P}_{\ell+1,m}\left(x\right) + \frac{\beta_{\ell+2,m}}{\beta_{\ell+1,m}} \mathcal{P}_{\ell m}\left(x\right) \tag{6}$$

where

$$\beta_{\ell m} = \sqrt{\frac{4\,\ell^2-1}{\ell^2-m^2}}. \tag{7}$$

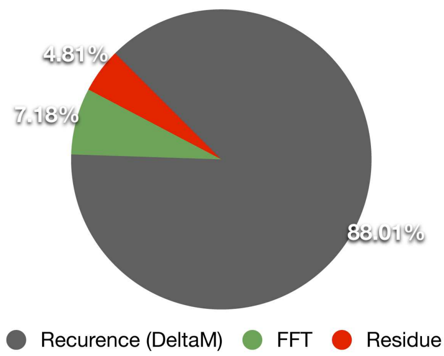


Figure 2: The overall time breakdown between the main steps of the inverse SHT algorithm as computed for a typical cluster of GPUs. See §4.3 for details.

The recurrence is initialized by the starting values,

$$\begin{aligned} \mathcal{P}_{mm}(x) &= \frac{1}{2^m\, m!}\sqrt{\frac{(2m+1)!}{4\pi}}\,\left(1-x^2\right)^m \\ &\equiv \mu_m\,\left(1-x^2\right)^m \qquad (8) \\ \mathcal{P}_{m+1,m}(x) &= \beta_{\ell+1,m}\, x\, \mathcal{P}_{mm}(x)\,. \qquad (9) \end{aligned}$$

The recurrence is numerically stable but a special care has to be taken to avoid under- or overflow for large values of ℓ_{max} [7].

3 Algorithm

This section presents a parallel algorithm for inverse spherical harmonic transform suitable for heterogeneous architectures. First, the main structure of the algorithm is outlined. Then, different parallelization and implementation techniques are discussed, depending on the target architecture.

Algorithm 1 outlines a straightforward implementation of the inverse spherical harmonic transform, that is the computation associated with equations (4) and (5). It consists of three steps. First, the starting values

Algorithm 1 BASIC `alm2map` ALGORITHM

Require: $\boldsymbol{a}_{\ell m}$ *values*
STEP 0 - PRE COMPUTATION
∘ $\boldsymbol{\mu}_m$ *pre computation Eq.* (8)

STEP 1 - $\boldsymbol{\Delta}_m$ 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. 6;*
      ∘ *update* $\boldsymbol{\Delta}_m(r)$*, given input* $\boldsymbol{a}_{\ell m}$ *and computed* $\mathcal{P}_{\ell m}$*, Eq. 5;*
    end for *(*ℓ*)*
  end for *(*m*)*
  STEP 2 - $\boldsymbol{s}$ CALCULATION
  ∘ *calculate* $\boldsymbol{s}$ *via FFT and given* $\boldsymbol{\Delta}_m(r)$ *pre-calculated for all* m*;*
end for *(*r*)*

return $\boldsymbol{s}$

of the recurrence (8) are computed, and then used in the evaluation of Legendre functions (6) in the second step. Once it is done, the partial results are synthesised via FFTs i.e., the final sum (4) is evaluated.

In the following we describe our parallel algorithm which is suitable for machines formed by nodes of multicore processors and possibly accelerators. To exploit such an architecture, our algorithm has two levels of parallelism. A first level aims at distributing the data and the computation among nodes. A second level focuses on exploiting parallelism inside each node.

At the first level, we address an important problem of the spherical harmonic transform computation, which is the very large size of the input and output data. For a big ℓ_{max} (usually $\ell_{max} \geq 2048$) the precomputed, input array $\boldsymbol{a}_{\ell m}$, whose size is proportional to $\mathcal{R}_N \ell_{max}^2$, becomes so large that it no longer fits into a single machine main memory. Hence, the algorithm needs to be memory scalable. Therefore in our approach the top level parallelism is driven by the data layout. It is presented in Algorithm 2, which is based on MPI and is composed of three steps. The algorithm divides the 2-dimensional $\boldsymbol{a}_{\ell m}$ array by assigning to a process $p_i = (m \mod n_{procs})$ a subset of all coefficients $\boldsymbol{a}_{\ell m}$. That is, we create subsets of m values: $\mathcal{M}_i$. Therefore $\cup_{i=0}^{n_{procs}} \mathcal{M}_i = \mathcal{M}$, where $\mathcal{M}$ is a set of all m and n_{procs} is a number of MPI processes used in the computation. In the same fashion we associate to each process a set of *observed* rings $r^O \in \mathcal{R}_i$ i.e., we assign to each process a subset of θ_n to ensure that equation (4) will be evaluated in parallel (after appropriate data exchange between processes).

Algorithm 2 General parallel `alm2map` algorithm (Code executed by each MPI process)

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

step 0 - Pre computation
∘ $\boldsymbol{\mu}_m$ pre computation Eq. (8)

step 1 - $\boldsymbol{\Delta}_m$ calculation
for every ring 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. (8) & (9)
∘ precompute recurrence coefficients, $\beta_{\ell m}$ Eq. (7)
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. (6)
end for (ℓ)
end for (r) **and** (m)

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

step 2 - $\boldsymbol{s}$ calculation
for every ring $r^O \in \mathcal{R}_i$ **do**
∘ using FFT calculate $\boldsymbol{s}(r^O, \phi)$ for all samples $\phi \in r^O$, given pre-computed $\boldsymbol{\Delta}_m(r^O)$ for all m.
end for(r^O)

return s_n for all $(r^O \in \mathcal{R}_i)$

The algorithm starts by precomputing starting values of Legendre recurrence $\boldsymbol{\mu}_m$ (8), but only results for $m \in \mathcal{M}_i$ are preserved. Next the *heterogeneous resources* (e.g., multi core processor or CUDA dedicated device) associated with one MPI process (or node) calculate using Eq.(5) the $\boldsymbol{\Delta}_m$ functions for *every* ring r of the sphere and $m \in \mathcal{M}_i$. Once this is done, the global (with respect to the number of processes involved in the computation) data exchange is performed so that at the end each process has access in its memory to $\boldsymbol{\Delta}_m$ functions calculated for all their observed rings $r^O \in \mathcal{R}_i$ and *all* m values. Finally via FFT we synthesize partial results (4). As it can be seen, with our approach for data distribution, the last two steps of the algorithm are highly parallel and the memory required to store the intermediate products of $\boldsymbol{\Delta}_m$ is in the order of $\mathcal{O}(n_{pix}/n_{procs})$. Therefore it is comparable to that used to store the input and output objects in their distributed form. Also like the latter they decrease as the number of employed processes increases, preserving the overall memory-scalability of the algorithm.

3.1 Exploiting intra-node parallelism for `alm2map` computation

In the following we describe the second level of parallelism of our approach. As it can be seen from Algorithm 2, the implementation of `alm2map` is composed of three nested loops which iterate over rings $r \in \mathcal{R}_N$, l and m, respectively. The overall memory requirement, efficiency of the algorithm, and possible acceleration techniques depend on the order in which the loops are nested. For instance ℓ-recurrence required by the structure of $\boldsymbol{\Delta}_m$ (5) enforces making the loop over ℓ an innermost one. The order of r and m loops can be interchanged depending on the architecture on which we execute the code. In the following we exploit this flexibility in our hybrid implementation of the algorithm.

Multithreaded version - `alm2map_cmt`

In the case of a node formed by multicore processors, the second level of parallelism exploits further multithreaded based parallelism. As discussed previously, the order of the three nested loops can be changed. For instance on CPUs, the iterations over m in the outermost loop allow precomputing $\beta_{\ell m}$ (7), $\mathcal{P}_{mm}$ (8)

and $\mathcal{P}_{m+1,m}$ (9) 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. However modern super computer are equipped with memory banks of size even up to 64 GB (e.g., Hopper from NERSC has 32 GB DDR3 1333 MHz memory per node). This fostered us for developing multithreaded version of algorithm 2 in which we place loop over m outermost. That is, we introduced the multithread layer via `OpenMP API` for m loop in the algorithm 1 i.e., a set of $m \in \mathcal{M}_i$ (input of `alm2map` function executed by i^{th} MPI process) is evenly divided into a number of subsets equal to the number of threads mapped one to each physical cores available on a given multi-core processor. We denote this subset of m values as $\mathcal{M}_i^{\text{T}}$ where subindex i is an i^{th} thread. In such a way each physical core executes one thread concurrently and calculates $\mathbf{\Delta}_m$ values for its local (in respect to shared memory) subset of m. The algorithm 3.1 describes the STEP 1 in its multithreaded version.

Algorithm 3 $\mathbf{\Delta}_m$ 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**
    for every ring r **do**
      ∘ initialise $\mathcal{P}_{mm}$ and $\mathcal{P}_{m+1,m}$ Eqs. (8) & (9)
      ∘ precompute $\beta_{\ell m}$ Eq. (7)
      for every $\ell = m+2, ..., \ell_{max}$ **do**
        ∘ compute $\mathcal{P}_{\ell m}$ via $\boldsymbol{\beta}_{\ell m}$, Eq. (6)
      end for (ℓ)
    end for(r)
  end for(m)

CUDA version- `alm2map_cuda`

NVIDIA CUDA (Compute Unified Device Architecture) is a general purpose parallel computing architecture with a new parallel programming model and instruction set that takes advantage of the parallel compute engine in NVIDIA GPUs. CUDA comes with its own software environment that allows developers to use C as a high-level programming language. However other programming languages and interfaces are also supported.

A CUDA-enabled device is a multi core chip consisting of hundreds of simple cores, called Streaming Processors (SP) which executes 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 synchronize and exchange data using very fast but small (up to 48KB in size for the latest NVIDIA carts) shared memory. The shared 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 with memory and lack of synchronizations between the blocks of threads drive the structure of our algorithm devised for GPUs. For instance, placing loop over m outermost (like in algorithm 3.1) is not appropriate for CUDA applications due to the memory size limitation. Consequently, the size of 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 consist 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 assign to one thread. In this model each thread computes the 2-point recurrence for all *available* m-values and it processes in parallel $\boldsymbol{a}_{\ell m}$ values for the same m and ℓ values. This makes it straightforward to plan the computation of $\boldsymbol{\beta}_{\ell m}$ (8) in batches, which can be cached and shared inside a thread block. We can now then outline algorithm 4 based on the first version of `alm2map` devised for GPU as detailed in [8] and which conforms with all these limitations.

Contrary to the CPU version the algorithm 4 has two new distinguishable steps (1.1 and 1.2). They ensure workaround for the high latency device memory and take advantage of the fast shared memory.

Algorithm 4 $\mathbf{\Delta}_m$ CALCULATION ON GPU

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**
      STEP 1.1 - use precomputed or, if needed, precompute in parallel a segment of $\boldsymbol{\beta}_{\ell m}$ (7);
      STEP 1.2 - use fetched or, if needed, fetch in parallel a segment of $\boldsymbol{a}_{\ell m}$ data;
      STEP 1.3 - compute $\mathcal{P}_{\ell m}$ via Eq. 6;
      STEP 1.4 - update $\mathbf{\Delta}_m(r)$ for a given prefetched $\boldsymbol{a}_{\ell m}$ and computed $\mathcal{P}_{\ell m}$ (5);
    end for (ℓ)
  end for(m)
end for(r)

Specifically, in STEP 1.1 we calculate the values of $\boldsymbol{\beta}_{\ell m}$ vectors in segments, which allow us to fit them to the shared memory, while step 1.2 ensures that a supply of $\boldsymbol{a}_{\ell m}$ values for the 2-point recurrence is kept in a bank of the fast shared memory. This two stages allow a more continuous operation of the floating point units by decreasing the memory wait time. The drawback of this technique is that it introduces two additional parameters. the size of the segments for $\boldsymbol{a}_{\ell m}$ data and $\boldsymbol{\beta}_{\ell m}$ values, which need to be setup by a user for a given architecture and size of input data. In [8] the authors attempted to find optimal values by first executing a series of experiments with different combinations of the parameters and then selecting the best combination for a given data size, i.e., a number of rings. However, we find that from the practical point of view this approach is not justified. Instead we use the data sets collected from experiments performed by [8] to find a relation between the final execution time as a function different sizes of segments and use an analysis of variance (ANOVA) [2, 3] to determine the parameters, which have the strongest impact on overall performance. From this we have deduced the following general dependencies, which we use in order to setup our algorithm for a given data set made of $\mathcal{R}$ rings,

$$\begin{aligned} \mathtt{NTPB} &= 128 \\ \mathtt{NB} &= \mathcal{R}/\mathtt{NTPB} \\ \boldsymbol{\beta}_{\ell m}\mathrm{CS} &= 2 \times \mathtt{NTPB} \\ \boldsymbol{a}_{\ell m}\mathrm{CS} &= 2 \times \mathtt{NTPB} \end{aligned} \tag{10}$$

Where `NTPB` is the number of threads per block, `NB` is a number of blocks. Therefore $\boldsymbol{\beta}_{\ell m}$CS and $\boldsymbol{a}_{\ell m}$CS are sizes of the segments for $\boldsymbol{\beta}_{\ell m}$ and $\boldsymbol{a}_{\ell m}$ respectively.

Heterogeneity & FFT

With the two new versions of `alm2map` function we are able to fully exploit potential of modern architectures like multi core processors or GPUs. However, many new super computers are hybrid clusters i.e., they are a combination of the computational nodes interconnected (usually via PCI-Express bus) with the multi GPUs server. In such a way one or more multi-core processors has access to one GPU. Facing these heterogeneities the front end user of the spherical harmonic package will have to decide, which version of synthesis function to use, i.e., which type of architecture to choose to accelerate STEP 1 of algorithm 2. However FFTs (STEP 2) also need a special attention, this is because the number of samples per ring may vary and this strongly affects efficiency of the FFT algorithms. For instance, well-know `fftpack` library [12] used in HEALPIX [7] and LIBPSHT [10] 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, N_{FFT}, its complexity degrades to $\mathcal{O}(N_{FFT}^2)$. Here, we employ in all versions of `alm2map` the `FFTW` library [5]. Because of its efficiency `FFTW` is very popular and almost all super computers have their own compilation, tuned for a maximum efficiency on a given architecture. NVIDIA provides its own library for FFT i.e., `CUFFT` [9], API of which is modeled after FFTW. Therefore, it was straightforward to employ it to the `alm2map_cuda` function to accelerate STEP 2 in our algorithm for GPU.

In practice, on the heterogeneous systems we assign each step of algorithm 2 to the architecture, on which it performs better. The default assignment for two variants of `alm2map` function is depicted in Figure 3.

4 Experimental results

In this section, we present the results of our experiments on two different architectures. First, we introduce the hardware specifications of the machines we used. This point is necessary to explain the algorithm

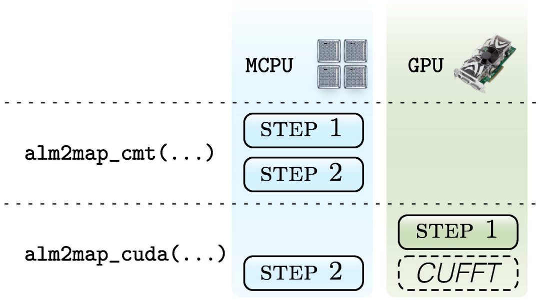


Figure 3: Default assignment of two major steps of Algorithm 2 to one of the two architectures, multi-core processor or GPU, in a GPU/CPU heterogeneous system.

behaviour on different machines. Then we show and explain the resulting curves for the synthesis step we focused on.

4.1 Architectures

The code was tested on two different machines.

- The first system is Titane[11], 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, so 4 processors handle 4 Tesla cards. The high performance interconnect is based on Infiniband DDR and the compiler suite is from Intel. In 2010, Titane get an extension of 432 nodes with also 2 Intel Nehalem quad-cores and 24 GB of memory per nodes. This system provides a peak performance of 100 Tflops for the Intel part and 200 Tflops for the GPU part.
- The second machine is Hopper-II[12], based on a Cray XE6 system. With 6392 nodes composed of 2 twelve-cores from AMD (2.1 GHz), this cluster has a peak performance of 1.288 Pflops. The memory is not homogeneous, 6008 nodes have 32 GB of memory and 384 nodes have access to 64 GB of memory. Gemini is the high performance interconnect used, it comes from Cray. Three compilers are available, Portland, Cray and GNU.

Now, we will focus on the results we obtained on the two previous architectures. First, the outcomes of the two types of parallelization (Multithreading and GPUs) introduced in section 3.1 will be describe separately. In order to test the scalability in respect to different input sizes we performed experiments with different numbers of MPI processes (up to 128) deployed on different architecture. Therefore, for each architecture and input data size we depicted the runtime distribution and the overall GFLOP per second. In the last part, we will compare the behaviour of the algorithm on the two clusters.

4.2 Hybrid MPI and OpenMP

On Hopper-II and Titane, multi-core architectures allow us to take advantage of two nested levels of parallelism. With the Intel Nehalem processors of Titane, four threads can handle the computation of $\boldsymbol{\Delta}_m$ (STEP 1). One thread is mapped to one physical core and one MPI process is assigned to one processor. The results shown in Fig. 4(a) present the execution time of each step. The work sharing over the MPI level and over the m loop gives a linear speedup for the computation of STEP 1. The massive data parallelism character of the algorithm provides a perfect scalability in terms of flops (figure 4(b)). The GLOBAL COMMUNICATION can be omitted because of its low overhead as shown previously in Fig. 2. As FFTW is the most popular library for manipulating Fourier transformations, it was chosen to have a portable code. In the case of Titane, for computing the STEP 2 we take advantage of the Intel optimisations in the Math Kernel Library (MKL) wrapped to FFTW routines. This Titane hybrid version is taken as a reference.

In Figs. 6(a) and 6(b) we show that the same hybrid version deployed on Hopper-II confirms the performances gain on STEP 1 if we use 24 threads to distribute the m loop. The threads are bound to the 24 physical cores of one node (one MPI process per node), a number which given that each node shares 32GB

[11] http://www-ccrt.cea.fr/fr/moyen_de_calcul/titane.htm
[12] http://www.nersc.gov/nusers/systems/hopper2/

of memory fulfills the data layout requirements. Compared to Titane, the STEP 2 is slower, this difference is due to the FFTW implementation available on the cluster. In the contrast to Titane, there is no vendor compiler available on Hopper-II, the GNU compiler suite is used and vendor specific optimizations are not available. The GLOBAL COMMUNICATION step shows the instability of the Hopper-II interconnect system, this behavior can be explained by the multiple jobs running at the same time in the same job queue.

If we just look at the parallelization of the STEP 1 (Fig. 5), the implementation on Hopper-II scales like on Titane due to the high data parallelism of the application. The hybrid approach for the parallelization of the recurrence ($\mathbf{\Delta}_m$) by using multithreading provides a linear speedup on multi-core architectures and scales in terms of data size and number of nodes.

4.3 Hybrid Multi-GPU

The next experiment is dedicated to the multi-GPU version and was executed exclusively on Titane. The CUDA 3.2 SDK version had been used and the entire application was compiled by the Intel compiler suite. For this test case, one MPI process was assigned to exactly one Nehalem processor in order to use one Tesla card per process. We assign the evaluation of STEP 2 to the CPU due to its better performance in comparison to CUFFT also available on Titane (see Fig. 4.3). Figures 8(a) and 8(b) display the overall time distribution and GFLOPS. As expected, the GFLOPS increase with the data size and strongly show the scalability of the GPU version. The GLOBAL COMMUNICATION stays stable on Titane and as describe in the previous section, STEP 2 has a small impact on the overall time execution due to the Intel specific optimizations and very good efficiency. We do not show the time overhead to send the data to the cards because this time is irrelevant compared to the computation time of $\mathbf{\Delta}_m$. In the next section, we will focus on comparing the two hybrid versions of the code and give an overview of the architectural advantages regarding the input data size.

4.4 Multi-GPU versus MPI and OpenMP

The comparison of the two different parallelizations is depicted on Figs. 9(a) and 9(b). Only the computation time of $\mathbf{\Delta}_m$ is plotted to extract comparable results for the most demanding (in respect to GFLOPS) step of our general algorithm.

• First, we focus on small input data sizes and only one MPI process (Fig. 9(a)). The histogram shows that one GPU performs as good as 24 threads on Hopper-II. Titane reference version (with 6 times less threads) is 3 times slower than the Hopper's one. The ratio can be explain by the Nehalem's higher frequency in comparison to Opteron. The code was also compiled with the Intel suite on Titane.

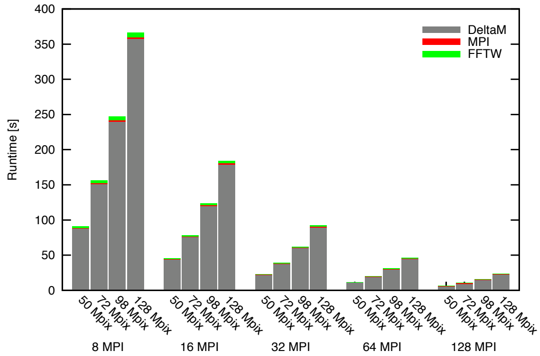
• Figure 9(b) shows the same comparison but with different data sets and number of MPI processes i.e., bigger input sizes are used and for each architecture we deployed 128 MPI processes. We observe that due to the very good scalability, the same relation between speedups are kept. That is to say, the 128 GPUs act like 128 Hopper's nodes with 24 threads mapped on the 2 AMD processors while the Titane reference version remains 3 times slower.

5 Conclusions

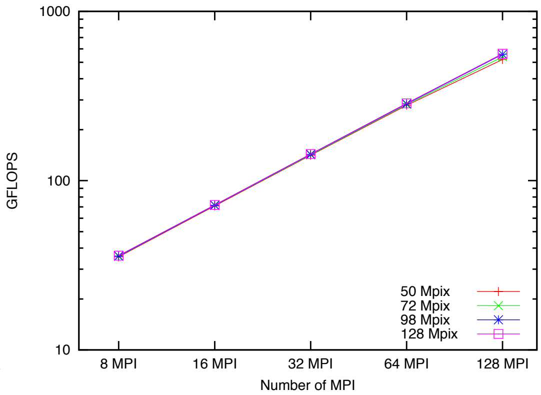
This paper describes a parallel algorithm for computing the inverse spherical harmonic transform with two variants specific for givens architectures i.e., multi-core processor and GPU. We tested both versions in terms of overall efficiency and scalability. We showed that one NVIDIA Tesla S1070 can accelerate overall execution time of the SHT by as much as 3 times with respect to the MPI/OpenMP version executed on one quad-core processor Intel Nahalem 2.93 GHz and it performs as good as 24 AMD 2.1 GHz Opteron cores. Owning to very good scalability this ratio is kept when increasing the number of MPI processes mapped to a given architecture.

6 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] (grant 2011-066647) 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.



(a) Algorithm scalability



(b) Overall algorithm scalability

Figure 4: Runtime distribution over steps of the algorithm and overall GLOPS for the multithreaded version of the `alm2map` transform tested on Titane with 4 OpenMP threads.

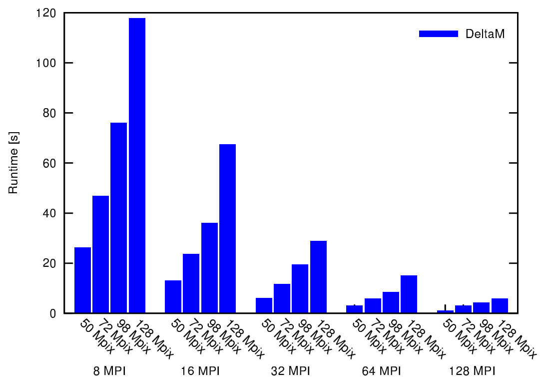
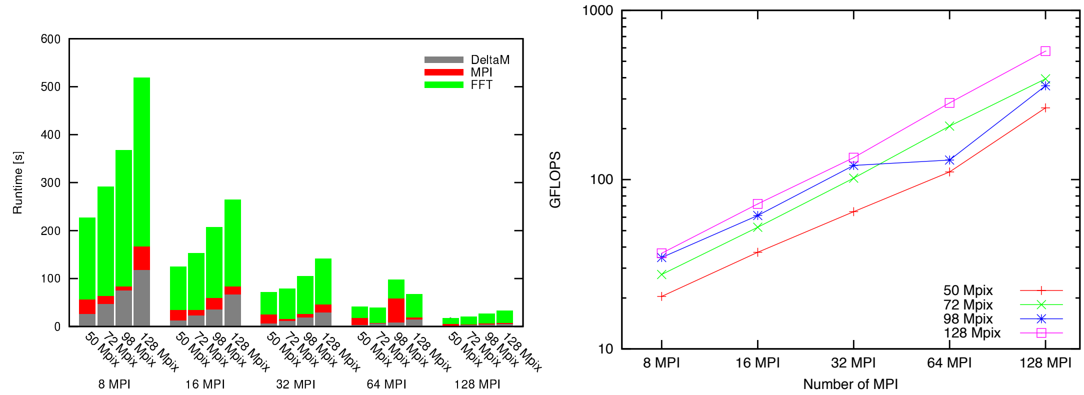


Figure 5: Runtime for the computation of STEP 1 ($\mathbf{\Delta}_m$) on Hopper-II.



(a) Algorithm scalability (b) Overall algorithm scalability

Figure 6: Runtime distribution over steps of the algorithm and overall GLOPS for the multithreaded version of the `alm2map` transform tested on Hopper-II with 24 OpenMP threads.

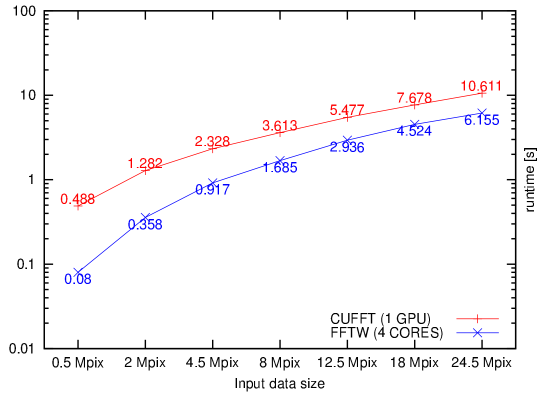
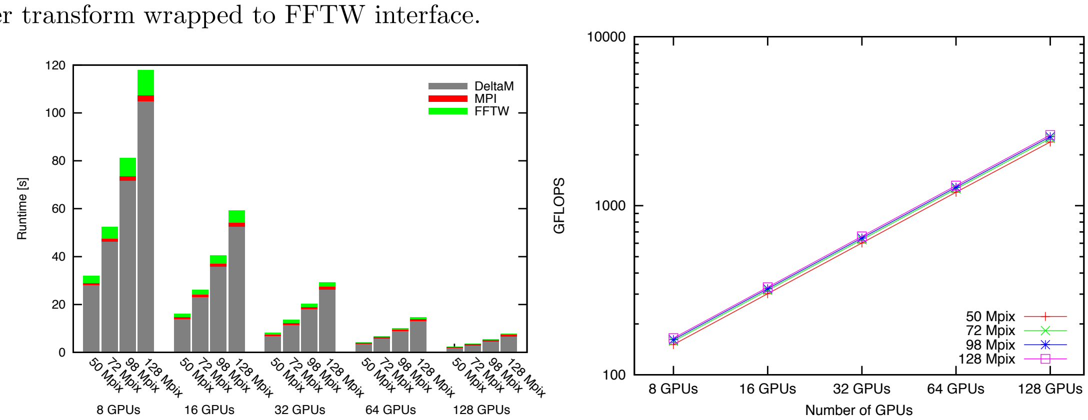


Figure 7: The overall time difference between the FFTs evaluation via CUFFT on GPU and Intel MKL Fourier transform wrapped to FFTW interface.



(a) Algorithm scalability (b) Overall algorithm scalability

Figure 8: Runtime distribution over steps of the algorithm and overall GLOPS for the multi-GPU version of the `alm2map` transform tested on Titane.

References

[1] G. B. Arfken and H. J. Weber. *Mathematical methods for physicists 6th ed.* Academic Press, 2005.

[2] R. A. Bailey and Q. Mary. *Design of Comparative Experiments*. University of London, 2008.

[3] D. R. Cox and N. M. Reid. *The theory of design of experiments*. Chapman & Hall-CRC, 2000.

[4] 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.

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

[6] J. N. Goldberg, A. J. Macfarlane, E. T. Newman, F. Rohrlich, and E. C. G. Sudarshan. Spin-s Spherical Harmonics and D. *Journal of Mathematical Physics*, 8:2155–2161, 1967.

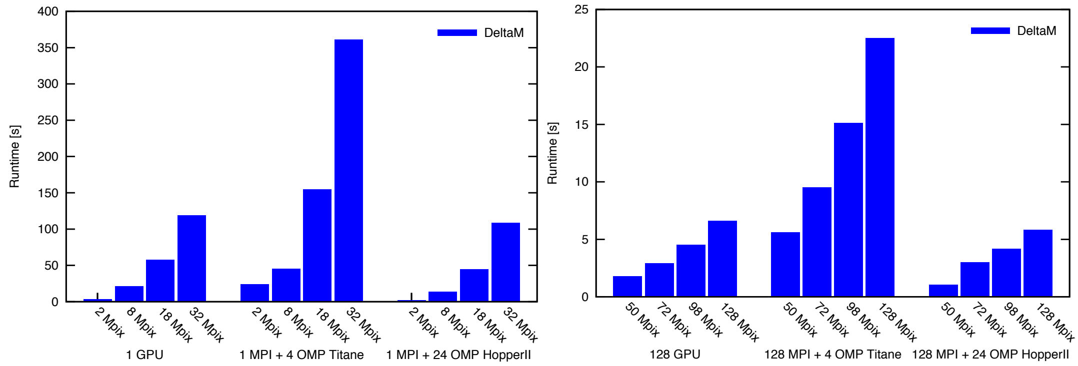


Figure 9: Comparison between the three versions of the `alm2map` transform, tested for various data sizes on Titane and Hopper-II for 1 and 128 MPI processes.

[7] 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, 2005.

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

[9] Nvidia. Cuda cufft library, 2010.

[10] Reinecke, M. Libpsht - algorithms for efficient spherical harmonic transforms. *A&A*, 526:A108, 2011.

[11] R. Suda and M. Takami. A fast spherical harmonics transform algorithm. *Mathematic of Computation*, 71(238):703–715, 2001.

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

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

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

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