Spherical harmonic transform with GPUs

Ioan Ovidiu Hupca, Joel Falcou, Laura Grigori, Radek Stompor

To cite this version:

Ioan Ovidiu Hupca, Joel Falcou, Laura Grigori, Radek Stompor. Spherical harmonic transform with GPUs. [Research Report] 2010, pp.20. inria-00522937v1

HAL Id: inria-00522937
https://inria.hal.science/inria-00522937v1

Submitted on 3 Oct 2010 (v1), last revised 6 Oct 2010 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Spherical harmonic transform with GPUs

Ioan Ovidiu HUPCA — Joel FALCOU — Laura GRIGORI — Radek STOMPOR

N° ????

Octobre 2010

Thème NUM

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

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SACLAY – ÎLE-DE-FRANCE

Spherical harmonic transform with GPUs

Ioan Ovidiu HUPCA * , Joel FALCOU † , Laura GRIGORI ‡ , Radek STOMPOR §

Thème NUM — Systèmes numériques
Équipe-Projet Grand-large

Rapport de recherche n° ???? — Octobre 2010 — 17 pages

Abstract: We describe an algorithm for computing an inverse spherical harmonic transform suitable for graphic processing units (GPU). We use CUDA and base our implementation on a FORTRAN90 routine included in a publicly available parallel package, S^{2}HAT. We focus our attention on the two major sequential steps involved in the transforms computation, retaining the efficient parallel framework of the original code. We detail optimization techniques used to enhance the performance of the CUDA-based code and contrast them with those implemented in the FORTRAN90 version. We also present performance comparisons of a single CPU plus GPU unit with the S^{2}HAT code running on either a single or 4 processors. In particular we find that use of the latest generation of GPUs, such as NVIDIA GF100 (Fermi), can accelerate the spherical harmonic transforms by as much as 18 times with respect to S^{2}HAT executed on one core, and by as much as 5.5 with respect to S^{2}HAT on 4 cores, with the overall performance being limited by the Fast Fourier transforms.

The work presented here has been performed in the context of the Cosmic Microwave Background simulations and analysis. However, we expect that the developed software will be of more general interest and applicability.

Key-words: transformée inverse en harmoniques sphériques, NVIDIA CUDA, GPU, CUDA

* INRIA Saclay-Ile de France, Bat 490, Université Paris-Sud 11, France (ioanovidiu.hupca@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

Transformée en harmoniques sphériques sur GPUs

Résumé : Nous décrivons un algorithme de calcul d'une transformée inverse en harmoniques sphériques approprié pour des processeurs graphiques. Nous utilisons CUDA et la mise en oeuvre utilise une routine du logiciel S^{2}HAT écrit en Fortran90. Nous nous concentrons sur deux étapes séquentielles de ce calcul, tout en conservant le cadre parallèle efficace du code original.

Le travail présenté ici a été réalisé dans le cadre de la simulation et de l'analyse du fond difus cosmologique. Cependant, nous estimons que les techniques présentées dans ce papier sont d'un intérêt et d'une applicabilité plus générale.

Mots-clés : keywords in french

1 Introduction

Spherical harmonic transforms are ubiquitous in diverse areas of science and practical applications, which need to deal with data distributed on a sphere. In particular, they are heavily used in various areas of cosmology, such as studies of the cosmic microwave background (CMB) radiation and its anisotropies, which have been our main motivations for this work. CMB is an electromagnetic radiation left over after the hot and very dense stage of early evolution of our Universe. The CMB measurements allow us to look back directly at the Universe when its age was only a small fraction ($\sim 3\%$) of its current one ($\sim$ 13Gyrs), and indirectly to learn about its status as far back as to $\sim 10^{-35}$sec after its nominal beginning (so called Big Bang). Not surprisingly, the CMB measurements play a vital role in the present-day cosmology and have been a driving force behind turning it into a high precision, data-driven science it is today.

The CMB radiation is nearly isotropic but minute deviations, on order of 1 part in 10^5, were first theoretically predicted and later detected. These so-called anisotropies encode the information about the Universe, its past and composition, and their detection and characterization has the major target of the CMB observations since the moment of its discovery in 1965. Over the time progressively more sophisticated and advanced observational apparata have been designed and deployed in search for their more subtle and taletelling characteristics. These include three major CMB satellites – American: Cosmic Microwave Background Explorer (COBE)[13], Wilkinson Microwave Anisotropy Probe (WMAP) [2], and European Planck[1] – and a few dozen of ground-based and balloon-borne projects. Some of these are operating at this time, including WMAP and Planck, and more are planned for the near and medium-term future, including potential new satellite missions, considered currently in Europe, US, and Japan.

The CMB detector technology has been improving quickly over the past decade, propelling a continuous increase of a number of detectors per experiment at the rate reminiscent of the Moore's law. This in turn has been driving a similar increase of the CMB data sets sizes, posing a formidable challenge for the CMB data analysis. This challenge can be only met if efficient numerical algorithms and the latest computer hardware are employed to match the data size increase by a concurrent increase in our processing capability. Spherical harmonic transforms are some of the most fundamental tools used in the CMB data processing. This is because the CMB signal is naturally a function of the observational direction and thus can be adequately described as a field defined on a sphere. The spherical harmonic functions are a suitable basis to represent and manipulate such fields. The spherical harmonic transforms involve a decomposition of the signals defined on the sphere into a set of harmonic coefficients (i.e., a *direct* spherical harmonic transform) as well as synthesis of the sky signal given a set of harmonic expansion coefficients (i.e, an *inverse* spherical harmonic transform). The latter is for instance a key step in massive Monte Carlo simulations used in the CMB data processing. As they usually require a very high resolution and precision, synthesis operations, referred hereafter as `alm2map` transforms, are particularly time and resources consuming. In this work we therefore focus specifically on `alm2map` transforms, and discuss their implementation on the NVIDIA GPU architecture within the CUDA framework. Similar techniques to these described here should be sufficient for an efficient implementation in this context of the direct `map2alm` transforms. We leave those for the future work. We note that the spherical harmonic transforms are commonly used beyond cosmology, for example, in geophysics, oceanography, or planetology and for all of which the implementation described here should be directly relevant.

There are several packages available implementing the spherical harmonic transforms with HEALPIX[2], S²HAT[3], GLESP[4], CCSHT[5], particularly popular in the CMB research. Out of these, we have selected S²HAT (Scalable Spherical Harmonic Transform) as the starting point for this research and a reference for performance comparisons. While all these packages implement similar numerical algorithms, in particular S²HAT originated from the HEALPIX routines, only S²HAT is not strictly tied to any specific sky pixelization or discretization schemes. S²HAT is written in FORTRAN90, fully parallelized using MPI, and shows memory scalability, good speedup and load-balance over a wide range of considered problems.

Our primary final target are however heterogeneous, multi-processor systems made of multiple CPUs, each accompanied by a respective GPU. As the first step towards achieving this goal we focus on porting the two main, serial steps in the calculation of the transforms onto GPUs and retain the data distribution layout and communication structure of the original MPI code. Consequently, when run on a multi-processor/multi-GPU platform our code employs MPI calls to distribute the data and workload over all the CPUs, which then send them to their respective GPUs, where the bulk of the computation is performed. The MPI communication pattern and work distribution inherited from the S²HAT package have been both demonstrated

[1] http://sci.esa.int/science-e/www/area/index.cfm?fareaid=17
[2] HEALPIX: http://healpix.jpl.nasa.gov/
[3] S²HAT: http://www.apc.univ-paris7.fr/~radek/s2hat.html
[4] GLESP: http://www.glesp.nbi.dk/
[5] ccSHT: http://crd.lbl.gov/~cmc/ccSHTlib/doc/

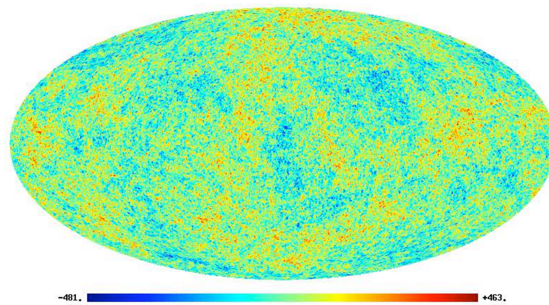


Figure 1: An example output of the CUDA `alm2map` routine the implementation of which is described in this paper. The figure shows a simulated picture of the sky as seen in the microwave band of electromagnetic radiation. The units are arbitrary.

to scale well in terms of memory load and execution time, and therefore we anticipate that the overall performance gain due to the speed of the GPUs will translate directly to an analogous total speed-up of the complete code. The proper end-to-end performance evaluation of the code on heterogeneous systems is the object of our on-going work. The performance tests presented in this paper focus specifically on the benefits due to GPUs and thus consider only single CPU/GPU cases. In addition this paper provides a first detailed description of the MPI parallelization of the efficient, scalable spherical harmonic transforms as implemented in S^{2}HAT. Fig. 1 provides an example result produced with our CUDA code. (An interested reader can compare this figure with the actual observations produced by the WMAP satellite and published in [2]).

The paper is organized as follows. In section 2 we introduce the spherical harmonic transforms and outline their MPI-based implementation. After a brief introduction to the CUDA programming model in section 3, we present a detailed description of our modified algorithm suitable for GPUs in section 4 and associated optimizations in section 5. Section 6 present some comparison results of both implementations. Section 7 concludes the paper.

2 Spherical harmonic transforms

2.1 Algebraic background

For a real, scalar field, $\boldsymbol{s}$, defined on the $\mathcal{S}^2$-sphere the pair of spherical harmonic transforms is defined as follows,

$$\boldsymbol{a}_{\ell m} = \sum_{\{\theta_p,\phi_p\}} \boldsymbol{s}\left(\theta_p,\phi_p\right) Y_{\ell m}\left(\theta_p,\phi_p\right), \tag{1}$$

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

Here the coefficients $\boldsymbol{a}_{\ell m}$ define a harmonic representation of the field $\boldsymbol{s}$, $Y_{\ell m}$ stands for a spherical harmonic, and (θ,ϕ) denote standard spherical coordinates. As in all practical application the field is pixelized or sampled on a discrete set of points, we have replaced the continuous integral in Eq. 1 by a discretized summation, which now goes over all the pixels on the sky. The upper limit, ℓ_{max} in Eq. 2 defines the band-limit of the field $\boldsymbol{s}$ and is considered to be finite. In the CMB application it is usually determined by an experiment resolution and its typical values are $\ell_{max} = \mathcal{O}\left(10^3 - 10^4\right)$. Hereafter, we will focus on the second of these two transforms and refer to it as the `alm2map` transform. Its objective is to reconstruct, or synthesize, the field, $\boldsymbol{s}$, from its harmonic coefficients $\boldsymbol{a}_{\ell m}$ on a grid of points p. (Hereafter, we will drop the index p for shortness.)

The spherical harmonics are defined as,

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

where *renormalized* associated Legendre functions, $\mathcal{P}_{\ell m}\left(\cos\theta\right)$ are solutions of the Hemholtz equations, e.g., [1], and their normalization is selected to ensure that $Y_{\ell m}$ constitute an orthonormal basis on the sphere.

$$\boldsymbol{s}\left(\theta,\phi\right) = \sum_{\ell=0}^{\ell_{max}} \sum_{m=-\ell}^{\ell} \boldsymbol{a}_{\ell m}\, Y_{\ell m}\left(\theta,\phi\right)$$

$$= \sum_{\ell=0}^{\ell_{max}} \boldsymbol{a}_{\ell 0}\, \mathcal{P}_{\ell 0}\left(\cos\theta\right) + \sum_{m=1}^{\ell_{max}} e^{im\phi} \sum_{\ell=m}^{\ell_{max}} \boldsymbol{a}_{\ell m}\, \mathcal{P}_{\ell m}\left(\cos\theta\right) + \sum_{m=1}^{\ell_{max}} e^{-im\phi} \sum_{\ell=m}^{\ell_{max}} a^{\dagger}_{\ell m}\, \mathcal{P}_{\ell m}\left(\cos\theta\right), \quad (4)$$

where we use the fact that,

$$\mathcal{P}_{\ell m}\left(\cos\theta\right) = (-1)^m\, \mathcal{P}_{\ell(-m)}\left(\cos\theta\right) \quad (5)$$

$$\boldsymbol{a}_{\ell m} = (-1)^m\, \boldsymbol{a}^{\dagger}_{\ell(-m)}. \quad (6)$$

The latter explicitly assumes that the map, $\boldsymbol{s}$, is real and a dagger denotes a complex conjugation. We can introduce now a set of functions, $\boldsymbol{\Delta}_m\left(\theta\right)$, such as,

$$\boldsymbol{\Delta}_m\left(\theta\right) \equiv \begin{cases} \sum_{\ell=0}^{\ell_{max}} \boldsymbol{a}_{\ell 0}\, \mathcal{P}_{\ell 0}\left(\cos\theta\right), & m = 0; \\ \sum_{\ell=m}^{\ell_{max}} \boldsymbol{a}_{\ell m}\, \mathcal{P}_{\ell m}\left(\cos\theta\right), & m > 0; \\ \sum_{\ell=|m|}^{\ell_{max}} \boldsymbol{a}^{\dagger}_{\ell |m|}\, \mathcal{P}_{\ell |m|}\left(\cos\theta\right), & m < 0, \end{cases} \quad (7)$$

and rewrite Eq. (4) as,

$$\boldsymbol{s}\left(\theta, \phi\right) = \sum_{m=-\ell_{max}}^{\ell_{max}} e^{im\phi}\, \boldsymbol{\Delta}_m\left(\theta\right). \quad (8)$$

The associated Legendre functions can be computed via a 2-point recurrence, e.g., [1], with respect to the multipole number, ℓ. It reads,

$$\mathcal{P}_{\ell+2,m}\left(x\right) = \beta_{\ell+2,m}\left[x\,\mathcal{P}_{\ell+1,m}\left(x\right) + \frac{1}{\beta_{\ell+1,m}}\mathcal{P}_{\ell m}\left(x\right)\right] \quad (9)$$

where

$$\beta_{\ell m} = \sqrt{\frac{4\,\ell^2 - 1}{\ell^2 - m^2}}. \quad (10)$$

The recurrence is initialized by the starting values,

$$\mathcal{P}_{mm}\left(x\right) = \frac{1}{2^m\, m!}\sqrt{\frac{(2m+1)!}{4\pi}}\,\left(1 - x^2\right)^m$$

$$\equiv \mu_m\, \left(1 - x^2\right)^m, \quad (11)$$

$$\mathcal{P}_{m+1,m}\left(x\right) = \beta_{\ell+1,m}\, x\, \mathcal{P}_{mm}\left(x\right). \quad (12)$$

The recurrence is numerically stable but requires double precision and a care has to be taken to ensure it does not under- or overflows. We describe a relevant algorithm in the next Section. Eqs. 7 and 8 provide a basis for the numerical implementation of the spherical harmonic transforms.

2.2 Current Approach

A detailed description of the efficient serial implementation of the transforms can be found elsewhere [6]. Here we briefly outline the most important features, highlighting in particular the parallel aspects.

2.2.1 Numerical complexity

From the sphere sampling considerations [4], we know that to properly sample a band-limited function with the band-limit set to ℓ_{max} we need roughly $n_{pix} \sim \ell^2_{max}$ points on the sphere. Therefore to perform the operations required to calculate $\Delta\left(\theta\right)$, and as detailed in Eqs. 7, we need as many as $\mathcal{O}\left(n^2_{pix}\right)$ floating point operations (FLOPs). This is because for each of n_{pix} pixels we have to do the $\mathcal{P}_{\ell m}$ recurrence for all ℓ and m numbers, and there are $\mathcal{O}\left(\ell^2_{max}\right) \sim \mathcal{O}\left(n_{pix}\right)$ of (ℓ, m) pairs for a properly sample field. This is clearly a prohibitive scaling. It can however become more favorable if the problem is restricted to some specific sky pixelization/discretization schemes [4]. In particular, in the following we will always assume that all pixels/sky samples are arranged in a number of so-called iso-latitudinal rings, each of which have the same value of the polar angle, θ. Typically there will be $n_{rings} \sim \ell_{max}$ rings with each ring uniformly sampled $n_\phi \sim \ell_{max}$ times. Moreover, we will assume that the sky is pixelized symmetrically with respect to the

equator. Such schemes indeed have been proposed and demonstrated to work well in practice [4, 7, 6, 3] in a number of applications. With these constraints imposed on the pixelization the scaling for Eq. 7 is now $\mathcal{O}(n_{pix}^{3/2})$, given that the full $\mathcal{P}_{\ell m}$ recurrence needs to be now done only ones for each of the rings. The numerical cost of the final summation, Eq. 8, is then sub-dominant as it can be implemented using Fast Fourier transform (FFT) techniques, at the total cost of $\mathcal{O}\left(n_{pix} \ln n_{\phi}\right)$ FLOPs.

We note here in passing that for this class of pixelizations even faster algorithms have been proposed with the complexity either on order of $\mathcal{O}[n_{pix}(\ln n_{pix})^2]$ [4] or $\mathcal{O}\left(n_{pix} \ln n_{pix}\right)$ [14]. However, they have a significant prefactor, involve complex algorithmic solutions, and have not been demonstrated to be numerically viable for $\ell_{max} \gg 100$.

2.2.2 Algorithm

The implementation of Eqs. 7 and 8 is rather straightforward. The pseudo code is outlined as Algorithm 1. Two steps which require somewhat more attention are the recurrence and the FFT. The two point recurrence

Algorithm 1 BASIC `alm2map` ALGORITHM

STEP 1 - $\boldsymbol{\Delta}_m$ CALCULATION
COMMENT: *Algorithm 2 has to be embedded below.*
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 the 2-point recurrence, Eq. 9;
      – update $\boldsymbol{\Delta}_m(r)$, given input $\boldsymbol{a}_{\ell m}$ and computed $\mathcal{P}_{\ell m}$, Eq. 7;
    end for *(ℓ)*
  end for *(m)*
  STEP 2 - $\boldsymbol{s}$ CALCULATION
  – calculate $\boldsymbol{s}$ via FFT, given $\boldsymbol{\Delta}_m(r)$ pre-calculated for all m;
end for *(r)*

as the one in Eq. 9 spans a huge dynamic range of values. This range depends on the values of ℓ, which need to be considered, but already for values as low as $\mathcal{O}\left(10^2\right)$ it exceeds that accorded to a double precision number on a typical processor. As we have freedom to rescale all the values of $\mathcal{P}_{\ell m}$, we can try to make use of it to rescale the recurrence starting values, e.g., $\mathcal{P}_{mm}$ and $\mathcal{P}_{m+1,m}$, appropriately to avoid the overflows later. However, this simple fix works only as long as the rescaled values of $\mathcal{P}_{mm}$ and $\mathcal{P}_{m+1,m}$ do not cause underflows. Though this on its own is not an issue, as these two will be typically set to zero what is clearly a good approximation to their values, our 2-point recurrence as a result will never produce any other outcome than zero. This will apply also to those Legendre functions, which initially produced the overflow and were supposed to be brought to within the representable range of values via the rescaling. A more robust solution to the problem is that of real-time rescaling. In the scheme usually used for this the newly computed values are tested if they approach over- or underflow limits and whenever this is the case they are rescaled appropriately. The rescaling coefficients (e.g., in form of their logarithms) are kept tracked of and used to rescale the computed values of $\mathcal{P}_{\ell m}$ at the end as required. This scheme is based on two facts. First, that the values of the Legendre functions calculated via the recurrence change gradually and rather slowly on each step. Second, that their final values are well-within the range of the double precision values.

The specific implementation of these ideas used in the S^{2}HAT software, and derived from the solution coded in the HEALPIX package, uses 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. The scaling vector, referred to hereafter as a rescale table is used to compare the values of $\mathcal{P}_{lm}$ computed on each step of the recurrence, and then used to rescale them if needed.

The respective pseudo-code for the Legendre function recurrence is presented as Algorithm 2. The associated Legendre function recurrence is normally performed on-the-fly and Algorithm 2 is thus merged with the algorithm for the `alm2map` transform, Algorithm 1.

The application of the FFTs in the last step of Algorithm 1 also requires some care. This is because the number of samples per ring may be either larger or smaller than the number of available m-modes. The former case can be dealt with by assuming the missing mode amplitudes to be zero. In the latter case the extra m modes have to be wrapped up and co-added to the modes present in the box. We refer the reader to paper [6] for more details of the involved calculations.

Algorithm 2 2-POINT ASSOCIATED LEGENDRE RECURRENCE

– *initialize the rescaling table;*
– *precompute $\boldsymbol{\mu}$ coefficients, Eq. 11;*
for *every ring r* **do**
  for *every $m = 0, ..., m_{max}$* **do**
    – *initialize the recurrence: $\mathcal{P}_{mm}$, $\mathcal{P}_{m+1,m}$, Eqs. 11 & 12, using precomputed $\boldsymbol{\mu}_m$;*
    – *precompute recurrence coefficients, $\beta_{\ell m}$ (fixed m, $\ell \in [m, \ell_{max}]$), Eq. 10;*
    for *every $\ell = m + 2, ..., \ell_{max}$* **do**
      – *compute $\mathcal{P}_{\ell,m}$ given $\mathcal{P}_{\ell-1,m}$ and $\mathcal{P}_{\ell-2m}$, given precomputed $\boldsymbol{\beta}_{\ell m}$, Eq. 9;*
      – *test the value of $\mathcal{P}_{\ell+2m}$ against the rescaling table;*
      – *rescale $\mathcal{P}_{\ell+2,m}$ and $\mathcal{P}_{\ell+1,m}$ if needed, keep the info about the rescaling coefficient;*
      COMMENT: *$\mathcal{P}_{\ell m}$ needs to be scaled back before being used in the calculations of the functions, $\boldsymbol{\Delta}_m$;*
    end for *(ℓ)*
  end for *(m)*
end for *(r)*

2.2.3 MPI parallelism

The structure of the parallel implementation of Algorithm 1 is determined by the data layout. For a properly sampled full sphere the input set of the harmonic coefficients, $\boldsymbol{a}_{\ell m}$, and the output sky map, $\boldsymbol{s}$, are roughly of the same size as $\sim n_{pix} \sim \ell_{max}$. These two objects are typically large and preferably have to be distributed.

S^{2}HAT divides the 2-dimensional $\boldsymbol{a}_{\ell m}$ array by assigning to a processor i a subset of all coefficients with predefined m values, $\mathcal{M}_i$. The 2-dimensional sky map $\boldsymbol{s}$ is treated as a collection of rings, r, so a subset, $\mathcal{R}_i$ of complete rings is distributed to a processor i. Recall that r corresponds to a subset of pixels, identified by a unique θ. This ensures the memory scalability of the algorithm if only the sizes of all the sets, $\mathcal{M}_i$ and $\mathcal{R}_i$, decrease as roughly $\sim 1/n_{procs}$.

The algorithm, see Algorithm 3, proceeds then in two steps. First, given the input subset of all $\{\boldsymbol{a}_{\ell m}, m \in \mathcal{M}_i\}$ coefficients, each processor calculates, using Eq. 7, the $\boldsymbol{\Delta}_m$ functions for *every* ring, r, of the sphere and $m \in \mathcal{M}_i$. Once this is done, the global communication is performed so that at the end each processor has in its memory $\boldsymbol{\Delta}_m$ functions calculated for the subset of rings as assigned to this processor, $\mathcal{R}_i$, and *all* m values. Given these data each processor performs the second step of the sky calculations, Eq. 8, using FFTs. We note that once the data distribution is performed then steps 1 and 2 of the algorithm are embarrassingly parallel. The memory required to store the intermediate products, $\boldsymbol{\Delta}_m$, is on order of $\mathcal{O}(n_{pix}/n_{procs})$ and therefore are comparable to that used to store the input and output objects in their distributed form. Also like the latter they decrease as a number of employed processors increases, preserving the overall memory-scalability of the algorithm. In its current form the S^{2}HAT implementation is

Algorithm 3 S^{2}HAT `alm2map` ALGORITHM - MPI IMPLEMENTATION

COMMENT: *Code executed by each MPI process.*
STEP 1 - $\boldsymbol{\Delta}_m$ CALCULATION
– STEP 1.1 - *initialize the rescaling table;*
STEP 1.2 - *precompute $\boldsymbol{\mu}$ coefficients, Eq. 11;*
for *every ring r* **do**
  for *every $m \in \mathcal{M}_i$:* **do**
    – STEP 1.3 - *initialize the recurrence: $\mathcal{P}_{mm}$, $\mathcal{P}_{m+1,m}$, Eqs. 11 & 12, using precomputed $\boldsymbol{\mu}_m$;*
    – STEP 1.4 - *precompute recurrence coefficients, $\beta_{\ell m}$ (fixed m, $\ell \in [m, \ell_{max}]$), Eq. 10;*
    for *every $\ell = m + 2, ..., \ell_{max}$* **do**
      – STEP 1.5 – *compute $\mathcal{P}_{\ell m}$ via the 2-point recurrence, given precomputed $\boldsymbol{\beta}_{\ell m}$, Eq. 9;*
      – STEP 1.6 - *test the value of $\mathcal{P}_{\ell m}$ against the rescaling table;*
      – STEP 1.7 - *update $\boldsymbol{\Delta}_m(r)$, given input $\boldsymbol{a}_{\ell m}$ and computed $\mathcal{P}_{\ell m}$, Eq. 7;*
    end for *(ℓ)*
  end for *(m)*
end for *(r)*

GLOBAL COMMUNICATION
– *redistribute* $\{\boldsymbol{\Delta}_m(r),\ m \in \mathcal{M}_i,\ \text{all}\ r\} \xrightarrow{\texttt{MPI_Alltoallv}} \{\boldsymbol{\Delta}_m(r),\ r \in \mathcal{R}_i,\ \text{all}\ m\}$

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

memory-distributed and implemented using MPI. Adding the openMP layer could be certainly useful as it could help to alleviate the communication bottleneck and thus facilitate running the library at even higher concurrencies. Nevertheless openMP is expected to have no major impact on the computational efficiency of steps 1 and 2 of the algorithm due to their embarrassingly parallel character. Considering the accelerators, such as GPUs, is therefore a potentially attractive avenue to explore.

3 NVIDIA CUDA Programming Model

NVIDIA CUDA is a general purpose parallel computing architecture - with a new parallel programming model and instruction set architecture - that takes advantage of the parallel compute engine in NVIDIA GPUs. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Other languages and application programming interfaces are also supported.

A CUDA-enabled GPU is basically a manycore chip consisting of hundreds of simple cores, called Streaming Processors (SP), together with control logic for the different levels of encapsulation. Each SP executes instructions sequentially, has a pipeline, 2 arithmetical-logical units and one floating point unit. The older generations do not have a cache. Several SPs are encapsulated in a Streaming Multiprocessor (SM). Multiple SMs form a Texture/Processor Cluster (TPC). All the TPCs form the Streaming Processor Array.

Inside one SM, the SPs execute in a SIMD fashion, while different SMs may execute different parts of the instruction stream in a SPMD fashion. Each SM also manages hundreds of active threads in a cyclic pipeline in order to hide memory latency. In practice, 32 threads are grouped together and scheduled as a Warp, executing the same instructions.

The latest two architecture generations available from NVIDIA are GT200 and GF100 (Fermi). The best model of GT200 has a total of 240 SPs (8 SP/SM x 3 SM/TPC x 10 TPC), with 16 KB of shared memory per SM and 16K available registers per block. It has a better implementation for memory fetching, reducing the earlier generation (G80) bottleneck produced by uncoalesced read/writes. Its theoretical peak performance is 1062.72 GFLOPS, in single precision. For double precision, FLOP count is 8 times lower. GF100 increases the number of SPs to 512 (32 SP/SM x 4 SM/GPC x 4 GPC), shared memory to 64 KB and adds a Level 1 caching mechanism. Theoretical peak performance for single precision is 1344.96 GFLOPS and double precision is half this value. However, for the commercial GTX 480 graphics card, the double precision performance is intentionally limited to one eighth of single precision. The only products using the entire DP capacity of the chip are those in the NVIDIA Tesla line.

The CUDA threads are grouped together into a series of thread blocks. All threads in a thread block execute on the same SM, and can synchronize and exchange data using shared memory. Synchronization between thread blocks is not possible. One more level of encapsulation is possible as the thread blocks can be grouped in independent grids. The Fermi chip is equipped with a resizable Level 1 cache (using shared memory for storage) which has the purpose of removing manual data copy between the slow device global memory and the fast shared memory. It is enabled by default, but its size can be switched between 16 KB to 48 KB.

Various limitations and roadblocks have to be taken into account while developing algorithms on such architectures. First, the performance for the total chain of computation has to take into account the transfer time between the CPU and GPU main memory. For smaller algorithms, this transfer can represent more than ten times the computation step itself. One is then tempted to fit large segments of data in the GPU memory to limit the number of times the data has to be sent and received. However, doing so will put a heavy constraint over the memory management code inside the kernel. To fasten access to the global memory, data is usually fetched into some local memory segments. However, using large segments of local memory reduces the size of the registers bank, thus leading to a large amount of slower code spill. Balancing between transfer time, size and actual memory management strategy is a very important problem to solve for any high performance algorithm executing on GPUs. We address these issues in the following section in the context of our application.

We also considered OpenCL as alternative for the development, but opted for CUDA, which we expect is best suited to exploit the capabilities of the studied hardware.

4 `alm2map` with CUDA

Programming philosophy for CUDA dictates using fine grained parallelism and launching a very large number of threads in order to use all the available cores and hide memory latency. Since the loop computing the two-point recurrence is serial in nature and therefore cannot be broken into parallel segments, there are two remaining choices for parallelization: the m-loop and the ring loop.

The initial CPU approach involved parallelizing only the m-loop, by having each process compute all the ring values for a subset of m values. This method of parallelization makes it easy to write code for MPI, as each process works on a subset of m values.

This approach is not appropriate for the GPU however, because of shared memory limitations. The size of vector $\boldsymbol{\beta}_{\ell m}$, Eq. 9, depends on ℓ_{max} and therefore cannot be stored in shared memory. Its values need to be recomputed for each m and are 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. However, these expensive, repeating calculations would seriously limit performance.

Parallelizing the ring loop (step 1.1 in algorithm 4) avoids this problem and has additional advantages. Each thread is assigned a number of rings for which it computes the 2-point recurrence for all m-values. The consequence is that each thread processes $\boldsymbol{a}_{\ell m}$ values at the same m and ℓ coordinates, in parallel. This makes it easy to plan the computation of $\boldsymbol{\beta}_{\ell m}$ and $\boldsymbol{\mu}_m$, Eq. 11, in segments, as well as caching the $\boldsymbol{a}_{\ell m}$ values. An important added benefit is reusing these two vectors, by sharing them inside a thread block. Algorithm 4 shows the outline of the GPU computing kernel. It is observable that the three new steps (1.2,

Algorithm 4 S^{2}HAT `alm2map` ALGORITHM - CUDA IMPLEMENTATION

STEP 1 - $\boldsymbol{\Delta}_m$ CALCULATION

– STEP 1.1 - *assign rings for each* <u>*thread*</u>
for *every* $r \in \mathcal{R}_j$ ***do***
  for *every* $m \in \mathcal{M}_i$ ***do***
    – STEP 1.2 - *thread 0 in block computes a segment of* $\boldsymbol{\mu}_m$*;*
    for *every* $\ell = m+2, ..., \ell_{max}$ ***do***
      – STEP 1.3 - *use precomputed or, if needed, precompute in parallel a segment of* $\boldsymbol{\beta}_{\ell m}$*, Eq. 10;*
      – STEP 1.4 - *use fetched or, if needed, fetch in parallel a segment of* $\boldsymbol{a}_{\ell m}$ *map data;*
      – STEP 1.5 - *compute* $\mathcal{P}_{\ell m}$ *via the 2-point recurrence, Eq. 9;*
      – STEP 1.6 - *handle overflow/underflow using rescaling table;*
      – STEP 1.7 - *update* $\boldsymbol{\Delta}_m(r)$*, given prefetched* $\boldsymbol{a}_{\ell m}$ *and computed* $\mathcal{P}_{\ell m}$*, Eq. 7;*
    end for *(*ℓ*)*
  end for *(*m*)*
end for *(*r*)*

GLOBAL COMMUNICATION
– *redistribute* $\{\boldsymbol{\Delta}_m(r),\ m \in \mathcal{M}_i,\ \text{all } r\} \stackrel{\texttt{MPI_Alltoallv}}{\longrightarrow} \{\boldsymbol{\Delta}_m(r),\ r \in \mathcal{R}_i,\ \text{all } m\}$

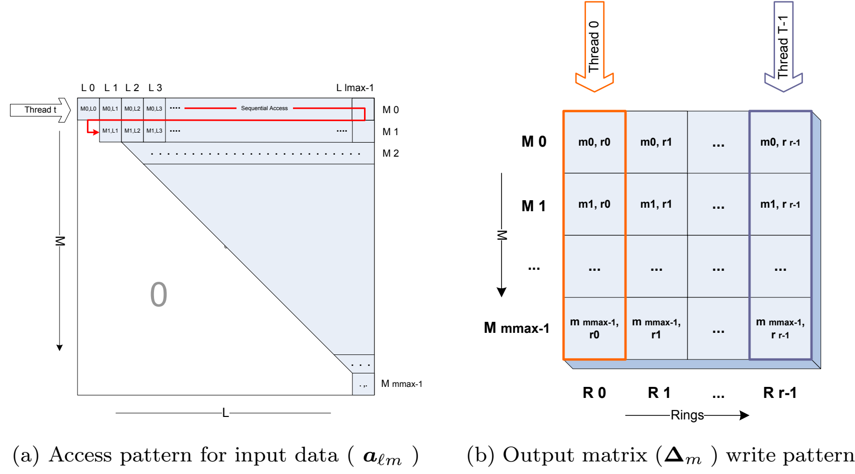
STEP 2
– *using FFT calculate* $\boldsymbol{s}(\theta,\phi)$ *for all samples for every,* $r \in \mathcal{R}_i$*, given pre-computed* $\boldsymbol{\Delta}_m(\theta)$ *for* $r \in \mathcal{R}_i$ *and* <u>*all*</u> m*.*

1.3 and 1.4) are designed to work around the high latency device memory and take advantage of the fast, but small, shared memory. Steps 1.2 and 1.3 calculate the values of the $\boldsymbol{\mu}_m$ and $\boldsymbol{\beta}_{\ell m}$ vectors in segments, as they do not fit in shared memory and it would be slow and wasteful to store them in global memory. Step 1.4 tries to keep a supply of $\boldsymbol{a}_{\ell m}$ values for the 2-point recurrence, therefore allowing a more continuous operation of the floating point units by decreasing memory wait time. Step 1.1 is where the threads select the rings on which to work upon. Since the m-loop and ring-loops are interchangeable, unlike the CPU version, the ring loop is first, allowing the sharing of the $\boldsymbol{\mu}_m$ and $\boldsymbol{\beta}_{\ell m}$ vectors. Figure 2 displays the acces by each thread of the 2-dimensional arrays $\boldsymbol{a}_{\ell m}$ and $\boldsymbol{\beta}_{\ell m}$.

5 Optimizations for GPU

GPU code optimization uses different rules than regular, CPU based code optimization. Due to the massively parallel structure of the target architecture, GPU code needs to fulfill a different set of constraints. Among these, the relationship between the cost of memory access and amount of computations per kernel is exacerbated. As shown in [15] for example, it can be far more beneficial to recompute large segments of constant values instead of fetching them from main memory. Others [8] show that, in some cases, the most direct algorithm can outperform the CPU optimized one. Another source of performance loss is thread divergence due to asymmetrical branching in control flow. Such divergence is usually detected by various profilers, but can prove hard to remove.

Based on guidelines for CUDA kernel optimization [11, 10, 12] and previous experiences, we mainly looked at limiting the effect of the slow global memory by buffering, precalculating or reusing data, removing branching in performance-critical sections and canceling warp serialization.



(a) Access pattern for input data ($\boldsymbol{a}_{\ell m}$) (b) Output matrix ($\boldsymbol{\Delta}_m$) write pattern

Figure 2: Input and output data access patterns

Array segmentation. Due to shared memory small size, it is required to compute the $\boldsymbol{\beta}_{\ell m}$ vector in segments, on the fly (step 1.3). Pre-calculating it entirely in device memory (akin to the original CPU implementation) would be very slow, as completing one $P_{\ell m}$ value requires reading the entire vector. $\boldsymbol{\beta}_{\ell m}$ segments are computed inside the ℓ-loop. Since $\boldsymbol{\beta}_{\ell m}$ is accessed sequentially, a portion of the vector is computed then used in the following steps of the recurrence. When existing values are exhausted, the next portion is computed. The size of the segment influences code performance, as it can be seen in the performance section. The same philosophy is applied for the $\boldsymbol{\mu}_m$ vector. Only difference is the segments are computed inside the m-loop (step 1.2). The advantage of having the code process the same $\boldsymbol{a}_{\ell m}$ data is that the two vectors are computed only once (in a parallel and serial fashion, respectively) and then reused by all threads in a thread block. As expected, the runtime decreases with increase in the number of threads.

A similar approach to segmentation is employed for offsetting memory latency for reading the $\boldsymbol{a}_{\ell m}$ coefficients and transferring them only once before being used by all threads in a block (step 1.4). $\boldsymbol{a}_{\ell m}$ values are transferred in segments during the Plm computation in step 1.5. Optimal segment size for all three vectors is found through testing. It is input size and platform dependent. Because of this, an autotuner is the best solution for obtaining the best possible performance. By running it for an input of the size targeted for computation, with all representative segment size and thread count variations, the best set of segment sizes can be selected. Currently, combinations that provide acceptable performance in all cases have been found by manual inspection and are being used as defaults in the code.

The nature of $\boldsymbol{\beta}_{\ell m}$ and $\boldsymbol{a}_{\ell m}$ allows their values to be obtained in parallel, by computing or fetching (steps 1.3 and 1.4, respectively). The number of threads which perform this operation is directly linked to segment size. In particular, the segment size must be a multiple of the number of threads. This avoids additional code for handling outlier indexes in performance-critical sections. Keeping in line with the CUDA guidelines on shared memory access for avoiding bank conflicts, the threads in a block calculate values sequentially, with a stride of block size. Due to its serial nature, $\boldsymbol{\mu}_m$ is computed by a single thread, while others wait for its completion (step 1.2).

There is one more type of global information, called pixelization data. It also comes in the form of two arrays, but they are not cached. This is because they are rarely accessed and caching would complicate the code with no speed benefits.

Branch collapsing. Code branching can severely impair the performance of GPU code, as divergent code is executed sequentially, effectively canceling parallelism. For example, when an "if" statement is encountered, some threads execute the true branch, while the other wait, then execute the false branch with the first group waiting. This problem is solved by collapsing the branch into code that has the same outcome as before, but can be executed in parallel by all threads. The computational overhead is smaller than that incurred by process-and-wait execution. Conditional assignments like `if (c) v=tv else v=fv` are converted to `v=c | tv & !c | fv`. The use of binary operators makes this expression very fast to compute. It is a common technique when converting code to SIMD operations. On the GPU however, this can be applied only on operations with integer numbers, as binary operators are not applicable to floating point operands. An equivalent version is based on multiplications and subtraction: `v = tv*c + fv*(1-c)`. This severely increases the overhead and makes it applicable only in some cases. For S^{2}HAT code, this version was

employed in both full and short form (if-then) resulting in decreased branching but with limited influence on execution time.

Other approaches have been tried for using the resources of the GPU as much as possible. While none of them provide increased performance, they do offer some insight into the operation of this new platform and serve as lessons for the future. We describe them briefly in the following.

Warp serialization. Warp serialization for arrays of double precision floating point stored in shared memory is a problem for GT200 chips. Since a memory bank holds only 32 bit values, a 64 bit value is stored in two different banks. When the number of threads grows beyond half the number of banks, values are accessed concurrently from the same bank. However, a bank can only service one request at a time and threads making additional requests are serialized, waiting for the previous request to complete. NVIDIA Programming guide suggests one possible solution as splitting the 64 bit value into 2 32-bit ones, storing them in two different vectors and then rejoining at use [11, p. 156]. However, after testing this technique on each vector of double precision data stored in shared memory, it proved useless. On the GT200 architecture, the computational cost of splitting and joining the values outweighed that of warp serialization. Moreover, the newer GT400 chips addressed this problem and 64 bit floats no longer cause warp serialization.

Dedicated scaling table. The scaling table is subject to a different kind of warp serialization. When threads in a block enter the rescaling phase, they access the data inside the array in a random fashion (some elements accessed by a single thread, others by multiple). Being small in size (21 64-bit values), the simplest approach for canceling serialization is having a copy of each table for each thread. However, experiments showed that while serialization does not occur, the time gain is insignificant even for small inputs. Also, as the number of threads increases, the amount of shared memory used becomes a limiting factor (for just 64 threads, 10.5 KB are needed).

$\boldsymbol{\beta}_{\ell m}$ precalculation. Based on the ability to execute a very large number of mathematical operations and the drawback of high device memory latency, a method for obtaining a good throughput is computing values on-the-fly instead of precalculating them. This trades computing cycles for memory cycles and some algorithms gained significant performance in this manner. $\boldsymbol{\beta}_{\ell m}$ calculation inside the ℓ-loop turned out to greatly increase computation time over both precalculation-based version and segment-based version. This is due to the high number of expensive operations involved in computing a single value of $\boldsymbol{\beta}_{\ell m}$ (multiplications, divisions and square roots), making reuse essential. Computing the scaling factors at usage-time had the same problem of expensive operations (powers, in particular).

Branch collapsing for scaling. Each iteration of the inner, ℓ-loop involves checking the value to be inside a validity interval and apply a scaling factor if this is not the case. This requires two "if" checks and can result in branching, impairing performance, as threads can go on 3 different execution paths. By collapsing the code using the technique described above, branching was reduced, but had the adverse effect of increased execution time. This is caused by the high number of multiplications forced on each thread by both the scaling code (which does not always execute) and operations introduced by the collapse method itself, since bitwise operators are not available for floating point.

6 Experimental results

Two platforms have been used for testing the code: GTX 260 for NVIDIA GT200 architecture and GTX 480 for the new NVIDIA GF100 (Fermi). Their host systems are: AMD Phenom 9850 (4 cores) with 8 GB of PC3200 DDR2 memory running on a MSI MS-7376 motherboard and Intel Core i7-960 CPU (4 cores, 8 processes with Hyperthreading) with 8 GB of PC3200 DDR2 memory running on a Gigabyte EX58-UD5, respectively.

The number of theoretical double precision FLOPS is significantly in favor of the GPUs, with a ratio of 1:2.2 and 1:3.2, respectively, when compared to the 51.2 GFLOPS double precision performance of Intel i7-960. The GPU FLOPS counts a FMADD operation as two separate ones, for an easier comparison with the CPU. It is also taken into account the fact that the Fermi chip can process a FMADD and ADD operation in parallel.

Also, the GPU architecture's massive parallelism of 260 and 480 SPs indicates a large advantage over the 4 physical cores of the referece CPUs. Even though the algorithm is known for near-linear scaling, due to the memory bound nature of the code (obtaining 10-15% of the theoretical peak performance), the algorithm was expected to get a significant, but limited, runtime improvement. It was anticipated that the high latency of the GPU global memory would further stall execution.

On the CPU, the Fortran algorithm was used as reference. Its efficiency was computed using the FLOP count returned by the PAPI package.

For the GPU, the execution time is calculated using the `gettimeofday()` library call between kernel launch and result retrieval. Because the consumer-grade cards used have limited memory, the largest dataset

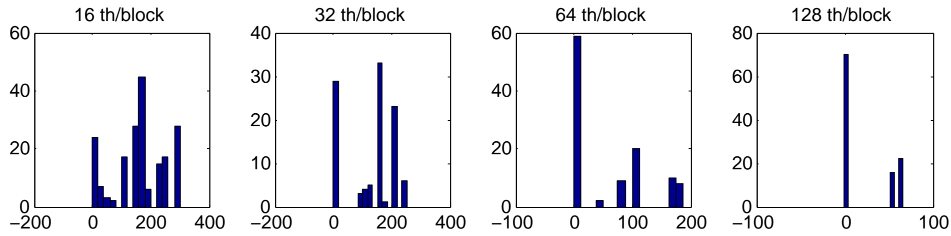


Figure 3: Histograms showing performance decrease (in percentages) due to L1 cache enabling on GTX 480

used is 4096x4096 and 5120x5120, respectively. In order to evaluate the possible runtime improvement for larger datasets, the output arrays were no longer allocated, allowing the input to fill the entire card memory. Results were written in a very small buffer (one value per thread), in order to maintain memory access and not distort the results. In this manner, the dataset limit was pushed to 9216x9216.

6.1 Parameter setup for $\boldsymbol{\Delta}_m$

Algorithm performance is dependent on the following parameters: number of threads in a block, number of rings per thread, size of the buffers used for computing the $\boldsymbol{\mu}_m$ and $\boldsymbol{\beta}_{\ell m}$ values, size of the input buffer, usage of 64-bit floating point split into 2 32 bit integers for storage in shared memory, usage of L1 cache (for Fermi) and the usage of a shared or dedicated rescaling table for each block. We tested each of these parameters for their influence on runtime and selected the best results.

Since computing the values for each ring can be done independently, each thread can calculate the values for any given number of rings. This would allow running a fixed number of threads, for a theoretical performance improvement (for example, running a number of threads equal to that of the SPs, resulting in no overhead from thread context switching – [15]). However, testing has shown that the optimum amount of rings for a thread is 1. Higher numbers result in significant performance degradation for any combination of parameters. Since other values would always return sub-par (and therefore useless) runtimes, all subsequent tests are performed with just one ring per thread.

One of the difficulties raised by the GT200 architecture is the method used for storing 64 bit values in the shared memory banks. When the number of threads grows beyond half the number of banks, values are accessed concurrently from the same bank, triggering serialization which results in latency for data read/write. The solution for avoiding it employs a workaround method, suggested in the NVIDIA Programming Guide, by splitting the data into two 32 bit values, storing them in shared memory then merging back to the original form at retrieval. This is no longer necessary on the Fermi, as 64 bit array access no longer generates warp serialization. This technique was tested on the $\boldsymbol{\mu}_m$, $\boldsymbol{\beta}_{\ell m}$ and $\boldsymbol{a}_{\ell m}$ arrays, in all their possible combinations (with or without using the L1 cache) as well as with all thread counts. It resulted in overall performance degradation for all cases. The reason for this is that the overhead for splitting and merging the values is greater than the time gained by avoiding serialization. The following tests were performed with 32-bit splitting disabled in all cases.

Another aspect tested was the cache system of the Fermi chip. This is designed to handle automatic fetching of values from device memory and store them into the fast shared memory, a task usually performed by hand by the CUDA programmer. Since array data is accessed sequentially (see figure 2a), caching should be straightforward and benefit from a hardware implementation. However, after running tests with all combinations of segment sizes, thread count per block and L1 cache on or off, it was determined that, for this particular algorithm, the caching implemented by the GPU never improves and often degrades performance. This is visible in figure 3, which contains the histograms for performance degradation values when enabling L1 preference, for the different thread counts used. This decrease is most apparent for low thread counts (average speed degradation of 200%). For 64 threads per block, activating L1-preference has no impact in half of the tests and ranges between 50% and 200% runtime increase for the rest. At 128 threads per block, half of the tests are not influenced, whilst the other suffer a 50-60% speed decrease. The input size used is large, 4096x4096, in order to obtain a relevant result. We can conclude that, for this particular case, manual buffering outperforms the L1 cache system of the Fermi.

The rescaling table is the read only array most accessed by all threads. Each time an overflow or underflow event takes place inside the L-loop, a value in the table is read (algorithm 4 - step 1.6). Therefore, optimizing its access can have a significant impact on overall algorithm performance. Due to its small size (21 64 bit values), giving a dedicated copy of the table to each thread is a good way of avoiding warp serialization triggered by concurrent random access. However, as the number of threads increases, shared

memory becomes a limiting factor. This is especially true on the GTX 280, which has just 16 KB available. Running the algorithm with the rescaling table shared by the thread block has shown that, contrary to expectation, the algorithm is faster by 30-50%.

Subsequent experiments used a shared rescaling table, had L1 cache preference (when ran on the Fermi) and 32-bit splitting disabled as well as treated just one ring per thread. The last three parameters to test are the lengths of the $\boldsymbol{\mu}_m$, $\boldsymbol{\beta}_{\ell m}$ and $\boldsymbol{a}_{\ell m}$ buffers. As expected, they have a major impact on the overall algorithm performance. Final performance values have been obtained by running all segment length combinations (16, 64, 128, 256 and 512 elements) with thread counts per block (16, 32, 64, 128, 256 and 512) for all input sizes. Since, for most cases, m_{max} equals ℓ_{max} (the $\boldsymbol{a}_{\ell m}$ matrix is square), the data sets used as input use the same restriction.

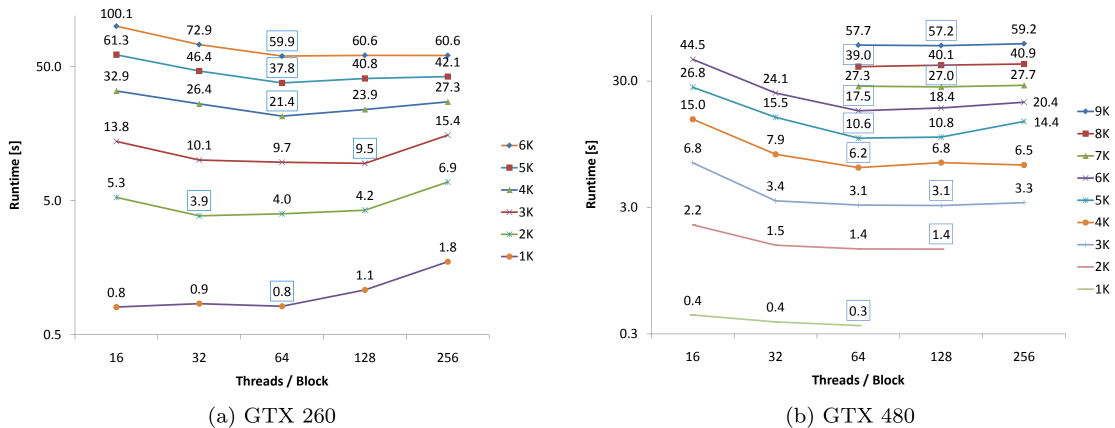
Having such a large set of experimental data, analysis on the influence of each parameter was attempted, since it was not freely observable. It was revealed that segment length and thread count have a direct influence on the runtime, but a correlation between their association and runtime was not found. In addition, combinations that give the best results for a certain input size do not keep that property for other datasets. Also, it was discovered that the runtimes can be split into clusters of values, as defined by different parameter combinations. However, this grouping differs with the input size.

In order to obtain the best performance for any input size, an autotuner is the best solution. Combinations providing near-best runtimes have been found by manually taking the parameters that provide the best time for a certain dataset and analyzing its performance when applied for the other datasets. Out of these parameter sets, the best overall was selected and used as default. As sufficient experimental data is available, this process can be further refined.

By executing the algorithm on the massively parallel chip that is the GPU, the number of running threads and their configuration relative to the processing units influences runtime considerably. In order to produce the final performance results, the best time is selected when varying both the thread count in a block as well as the segment length.

Figures 4a and 4b show the runtimes (in seconds) for different numbers of threads in a block. The entire range of inputs is considered, including those that fit into memory only with output allocation disabled. The best (lowest) times are marked by a box. We observe that, for both cards, the best runtime is obtained generally with 64 threads per block. In the cases where this is not the case (usually for 128 threads), the difference is almost negligible.

The two chips powering the GTX 260 and GTX 480, have a related, but significantly different architecture. However, as observed from the figures, the configurations that best use their capabilities are similar, using 64 or 128 threads per block. Unlike usual CPU logic, running more threads than execution units is beneficial. This is due to the high latency in accessing the device global memory. By using many threads, the Block Scheduler can replace blocks waiting for memory fetching with those ready for execution. The result is a higher throughput due to high reuse of idle threads.



(a) GTX 260 (b) GTX 480

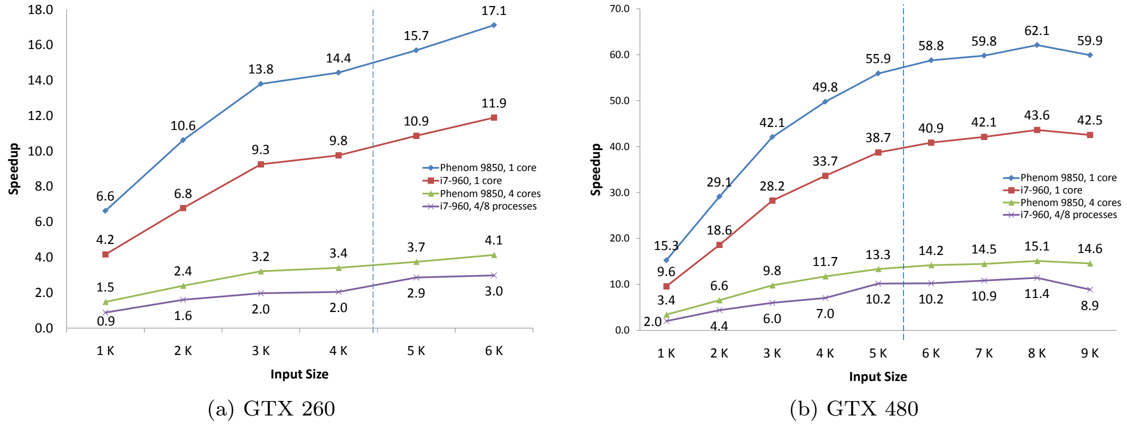
Figure 4: Runtimes for different thread counts per block. The input data size varies from 1K to 6K. The best runtime for each input size is displayed in a box.

6.2 Performance of Δ_m computation

In this section we discuss the performance of the code on the two GPU platforms (from the latest two generations), with respect to the CPU implementation running on two different processors. The entire

range of input sizes is tested with all variations of segment lengths. The best times are then selected and used for calculating the runtime improvement relative to the CPU implementation. Efficiency and GFLOPS for each graphics card are then computed.

The improvement factor from the GPU version is calculated against the reference Fortran MPI code running on the CPU. For AMD, the time duration obtained by running the program with 1 and 4 processes is used. The Intel i7-960 is equipped with Hyperthreading, meaning it can run 8 threads on just 4 physical cores. However, it was found that, in some cases, the 4 threads (MPI processes) version is faster. Therefore, one process and the best out of 4 and 8 processes is used as reference. The final runtime improvement factor for each input size is obtained by dividing the best time for each CPU by the best time of the GPU. When single-core is used as reference, the time measured while running the algorithm with just one process is divided by the best time of the GPU.



(a) GTX 260 (b) GTX 480

Figure 5: Improvement factor obtained by $\boldsymbol{\Delta}_m$ calculation of `alm2map` with CUDA with respect to the MPI version ran on the AMD Phenom and Intel i7 CPUs

Figures 5a and 5b show the runtime improvement for the two platforms used for testing (the latest generation GTX 480 and the older GTX 260) while using the entire range of inputs. We observe how larger inputs result in a higher improvement factor. Values rise sharply before starting to level at 4K (GTX 260) or 5K (GTX 480). The graphs plot the values for input sizes that normally fit the cards used for testing as well as those that require output disabling. They are separated by a vertical line (normal inputs on the left).

The AMD Phenom is slower than the Intel i7, therefore the improvement factor obtained will naturally be higher. When comparing the GTX 480 runtimes to those of single core CPU code, the performance ratio levels out at 60x for the Phenom and at 42x for the i7. For the older generation GTX 260, the factor is 3-3.5 times lower, at 17x and 12x, respectively. However, the relevant values are those obtained when using the CPUs to their full potential, with all their cores. The algorithm scales almost perfectly with the number of physical cores, the improvement values being generally one fourth of single core, with 14x and 10x for GTX 480 and 4x and 3x for GTX 260. Intel Hyperthreading does not seem able to provide advantages by pushing scaling beyond the number of physical cores.

Figure 6 displays the variation of efficiency with respect to the size of the input. The aspect of the curve is similar to that of the speedup graph, rising sharply before levelling at a 3K or 5K input. Being a memory-starved algorithm on the CPU, an adaptation to a faster chip, with a high-latency memory, was bound to suffer of the same problem. Running S^{2}HAT on the Intel i7-960, it reaches an efficiency of just 10%. As it can be seen in the graph, for the GTX 260, it mantains roughly the same value of 10%. On the GTX 480, however, it improves by a factor of 3, peaking at 30%. Since on the commercial GTX 480 double precision is limited to a quarter of its performance, running the algorithm on a Tesla card (which does not have it) will probably decrease efficiency by exposing the memory latency issue, hidden by this limitation, but should significantly improve performance.

In figures 7a and 7b we display the GFLOPS values obtained for each input size, for different counts of threads per block. Floating point operations methodology is as follows: additions and multiplications are computed as one operation each; due to lack of good references, divisions, square roots and logarithms are each counted as 20 operations. Rudimentary testing shows this value (20) to be an adequate estimate and, in either case, these operations combined are just 0.55% of the total number of floating point operations counted. Since GPUs do not yet have hardware counters for floating point operations, the operations were counted manually (adding values to variables in each thread followed by summation for the final results).

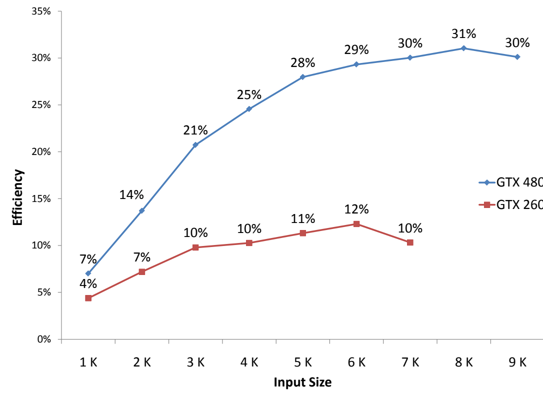
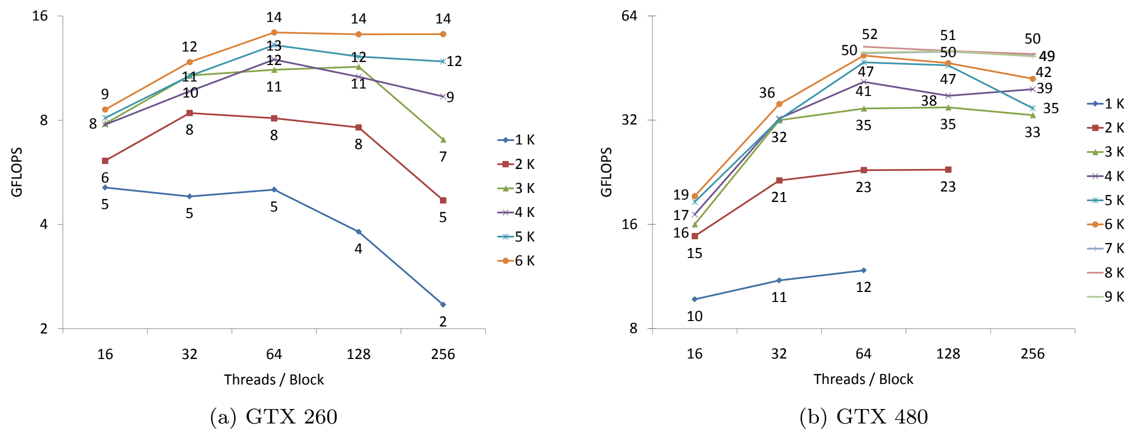


Figure 6: `alm2map` efficiency on GPUs



(a) GTX 260

(b) GTX 480

Figure 7: GFLOPS per thread count

As resulting from figures 7a and 7b, the number of resulting GFLOPS increases with the size of the input, with values leveling at 50-52 GFLOPS for GTX 480 (figure 7b) and 14 for GTX 260 (figure 7a)

6.3 Overall performance

The performance of the `alm2map` algorithm is greatly improved by offloading the $\mathbf{\Delta}_m$ computation onto a GPU. However, the second step, FFT calculation, requires attention also. In the original CPU-only code, the FFTs occupy 5-10% of the total runtime. Improving $\mathbf{\Delta}_m$ timing by a factor of 10 (Intel I7-960, 4 processes), results in the FFTs becoming dominant.

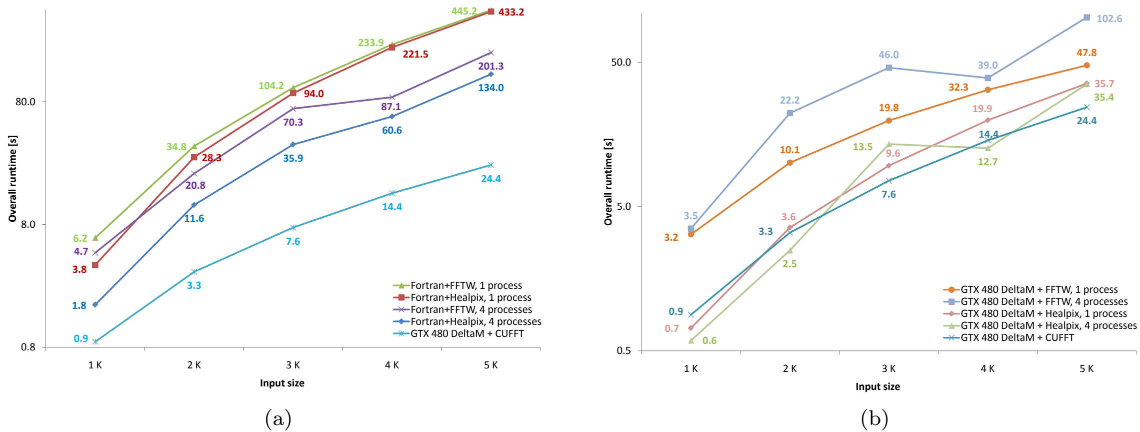
Since different FFT packages have different runtime characteristics, two CPU-only FFT libraries have been tested: one as implemented in Healpix [6] and the other – FFTW[6] [5]. Also, an FFT library for the NVIDIA GPUs, CUFFT [9], is employed. The current `alm2map` FFT implementation is a direct port of the original CPU version, with no specific GPU optimizations.

Figure 8 shows the overall ($\mathbf{\Delta}_m$ + FFT) runtimes for all combinations of $\mathbf{\Delta}_m$ computing code (Fortran on Intel i7-960 or CUDA on NVIDIA GTX 480), CPU FFT packages (Healpix or FFTW) and process count (1 or 4). Also, the runtime for a full GPU computation is plotted. Only the Intel i7-960 with NVIDIA GTX 480 results are shown.

We notice that, relative to the FFTW version, the Healpix package performs better for both 1 and 4 processes. We also observe that the best runtimes belong to the code running on the GPU.

Figure 9 plots the overall runtime improvement over the CPU code versions with respect to the best performing GPU code (labeled "GTX480 $\mathbf{\Delta}_m$ + CUFFT" in figure 8). We observe that, in the best case, the improvement is just half that obtained when considering just the $\mathbf{\Delta}_m$ computation (figure 5b), but also significant, reaching factors from 5 to 18 (when comparing to the best and worst, respectively, performing code).

[6]FFTW: http://www.fftw.org/



(a) (b)

Figure 8: `alm2map` overall runtime, Intel i7-960 and NVIDIA GTX 480

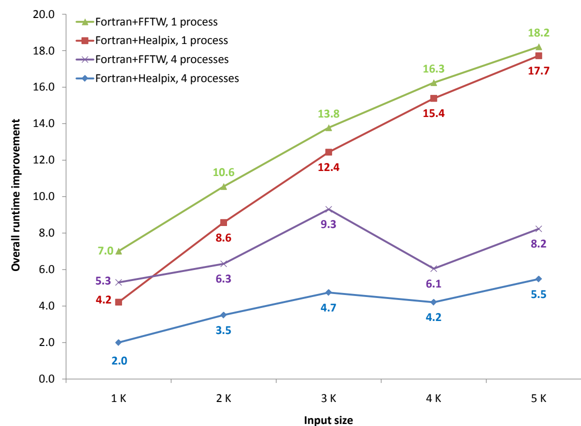


Figure 9: `alm2map` overall performance improvement

7 Conclusions and Future Work

This paper describes an algorithm for computing the inverse spherical harmonic transform on GPUs. The algorithm is compared with the implementation of the inverse spherical harmonic transform provided in S^{2}HAT library, implemented using Fortran and MPI. The GPU algorithm leads to an improvement of up to a factor of 18 with respect to S^{2}HAT on a single core and up to a factor of 5.5 with respect to S^{2}HAT on 4 cores of an Intel i7-960 machine. The improvement is limited by the performance of Fast Fourier transforms.

Even though a single GPU offers high computing power, employing several is the easiest way of scaling performance as well as handling larger inputs. The algorithm has been designed with multi-GPU use in mind and it can directly fit into, and benefit from, the S^{2}HAT MPI structure enabling straightforwardly distributed GPU computing. However special care has to be taken to ensure a good load balance among processors. This is the object of our current work.

Acknowledgments

This work has been supported in part by French National Research Agency (ANR) through COSINUS program (project MIDAS no. ANR-09-COSI-009).

References

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

[2] C. L. Bennett, M. Bay, M. Halpern, G. Hinshaw, C. Jackson, N. Jarosik, A. Kogut, M. Limon, S. S. Meyer, L. Page, D. N. Spergel, G. S. Tucker, D. T. Wilkinson, E. Wollack, and E. L. Wright. The Microwave Anisotropy Probe Mission. *ApJ*, 583:1–23, Jan. 2003.

[3] A. G. Doroshkevich, P. D. Naselsky, O. V. Verkhodanov, D. I. Novikov, V. I. Turchaninov, I. D. Novikov, P. R. Christensen, and L. Chiang. First Release of Gauss-Legendre Sky Pixelization (GLESP) software package for CMB analysis. *ArXiv Astrophysics e-prints*, Jan. 2005.

[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] 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.

[7] 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.

[8] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, New York, NY, USA, 2009. ACM.

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

[10] Nvidia. *NVIDIA CUDA Best Practices Guide*, 2010.

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

[12] Nvidia. *Tuning CUDA Applications for Fermi*, 2010.

[13] G. F. Smoot, C. L. Bennett, A. Kogut, E. L. Wright, J. Aymon, N. W. Boggess, E. S. Cheng, G. de Amici, S. Gulkis, M. G. Hauser, G. Hinshaw, P. D. Jackson, M. Janssen, E. Kaita, T. Kelsall, P. Keegstra, C. Lineweaver, K. Loewenstein, P. Lubin, J. Mather, S. S. Meyer, S. H. Moseley, T. Murdock, L. Rokke, R. F. Silverberg, L. Tenorio, R. Weiss, and D. T. Wilkinson. Structure in the COBE differential microwave radiometer first-year maps. *ApJ*, 396:L1–L5, Sept. 1992.

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

[15] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[16] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. *ACM/IEEE Conference on Supercomputing (SC08)*, 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