Spherical Harmonic Transform with GPUs

Ioan Ovidiu Hupca[1,3], Joel Falcou[1,3], Laura Grigori[1,3], and Radek Stompor[2]

[1] LRI - INRIA Saclay-Ile de France
{ioanovidiu.hupca,laura.grigori}@inria.fr, joel.falcou@lri.fr
[2] Astroparticule et Cosmologie, CNRS, Université Paris Diderot, Paris, France
radek@apc.univ-paris7.fr
[3] Université Paris-Sud 11, Orsay, France

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 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 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 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.

Keywords: Spherical Harmonic Transform, NVIDIA CUDA, GPU, Cosmic Microwave Background.

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. Its 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. A recent stunning increase in CMB data sets sizes, driven by the quick improvement of the detector technologies, has posed a formidable challenge for the CMB data analysis, which can only be met if efficient numerical algorithms and the latest computer hardware are employed to provide a sufficient,

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 355–366, 2012.

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* 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 are particularly time and resources consuming. They are therefore the focus of this work, which discuss their implementation on the NVIDIA GPU architecture within the CUDA framework. We note that these 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 (`http://healpix.jpl.nasa.gov/`), CCSHT (`http://crd.lbl.gov/~cmc/ccSHTlib/doc/`), S^{2}HAT (`http://www.apc.univ-paris7.fr/~radek/s2hat.html`), GLESP (`http://www.glesp.nbi.dk/`), particularly popular in the CMB research. Here, we have used S^{2}HAT (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, only S^{2}HAT is not tied to any specific sky pixelization or discretization schemes. It is 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 performance tests presented in this paper focus specifically on the benefits due to GPUs and thus on single CPU/GPU. Cases with the multi-GPU/multi-CPU configurations are studied elsewhere [1].

2 Spherical Harmonic Transforms

2.1 Algebraic Background

Any real, band-limited, scalar field, $\boldsymbol{s}$, defined on the $\mathcal{S}^2$-sphere can be represented as,

$$\boldsymbol{s}\left(\theta_p, \phi_p\right) = \sum_{\ell=0}^{\ell_{max}} \sum_{m=-\ell}^{\ell} \boldsymbol{a}_{\ell m}\, Y_{\ell m}\left(\theta_p, \phi_p\right) \tag{1}$$

Here the coefficients $\boldsymbol{a}_{\ell m}$ define a harmonic representation of the field $\boldsymbol{s}$, $Y_{\ell m}$ stands for a spherical harmonic. We assume, as it is usually the case in practical applications that the field, $\boldsymbol{s}$ is to be computed only on a discrete set of points, hereafter typically identified with a centers of sky pixels described by standard spherical coordinates, (θ_p, ϕ_p). The upper limit, ℓ_{max} in Eq. 1 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 $\ell_{max} = \mathcal{O}\left(10^3 - 10^4\right)$. The transform's objective is to reconstruct, or synthesize, the field, $\boldsymbol{s}$, from its harmonic coefficients $\boldsymbol{a}_{\ell m}$ on a grid of points p, and we will refer to it as the `alm2map` transform.

The spherical harmonics are defined as (hereafter, we will drop the index p for shortness)

$$Y_{\ell m}\left(\theta, \phi\right) \equiv \mathcal{P}_{\ell m}\left(\cos\theta\right) e^{im\phi} \tag{2}$$

where *renormalized* associated Legendre functions, $\mathcal{P}_{\ell m}\left(\cos\theta\right)$ are solutions of the Hemholtz equations, e.g., [2], normalized to ensure that $Y_{\ell m}$ constitute an orthonormal basis on the sphere. They can be computed via a 2-point recurrence, e.g., [2], with respect to the multipole number, ℓ, i.e.,

$$\mathcal{P}_{\ell+2,m}\left(x\right) = \beta_{\ell+2,m}\left[x\,\mathcal{P}_{\ell+1,m}\left(x\right) + \frac{1}{\beta_{\ell+1,m}}\mathcal{P}_{\ell m}\left(x\right)\right] \tag{3}$$

where $\beta_{\ell m} = \sqrt{(4\,\ell^2 - 1)/(\ell^2 - m^2)}$. The recurrence is initialized by the starting values,

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

$$\mathcal{P}_{m+1,m}\left(x\right) = \beta_{\ell+1,m}\, x\, \mathcal{P}_{mm}\left(x\right), \tag{5}$$

and is numerically stable but requires double precision and care has to be taken to ensure it does not under- or overflows. We describe a relevant algorithm in the next Section. On introducing,

$$\boldsymbol{\Delta}_m\left(\theta\right) \equiv \begin{cases} \sum\limits_{\ell=m}^{\ell_{max}} \boldsymbol{a}_{\ell m}\, \mathcal{P}_{\ell m}\left(\cos\theta\right), & m \geq 0; \\ \sum\limits_{\ell=|m|}^{\ell_{max}} \boldsymbol{a}^{\dagger}_{\ell|m|}\, \mathcal{P}_{\ell|m|}\left(\cos\theta\right), & m < 0; \end{cases} \tag{6}$$

we can rewrite Eq. (1) as,

$$\boldsymbol{s}\left(\theta,\phi\right) = \sum_{m=-\ell_{max}}^{\ell_{max}} e^{im\phi}\,\boldsymbol{\Delta}_m\left(\theta\right). \tag{7}$$

Eqs. 6 and 7 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 [3,4]. Here we briefly outline the most important features, emphasizing the parallel aspects.

Numerical Complexity. From the sphere sampling considerations [5], we know that to properly sample a band-limited function with the band-limit set to ℓ_{max}, we need roughly $n_{pix} \sim \ell_{max}^2$ points on the sphere. Therefore to perform the operations required to calculate $\Delta\left(\theta\right)$, and as detailed in Eqs. 6, we need as many as $\mathcal{O}\left(n_{pix}^2\right)$ 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 $\mathcal{O}\left(\ell_{max}^2\right) \sim \mathcal{O}\left(n_{pix}\right)$ 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 [5]. 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 [5,6,3,7] in a number of applications. With these constraints imposed on the pixelization the scaling for Eq. 6 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. 7, 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 $\mathcal{O}[n_{pix}(\ln n_{pix})^2]$ [5] or $\mathcal{O}\left(n_{pix} \ln n_{pix}\right)$ [8]. However, they have a significant prefactor, involve complex algorithmic solutions, and have not been demonstrated to be numerically viable for $\ell_{max} \gg 100$.

Algorithm. The implementation of Eqs. 6 and 7 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 as the one in Eq. 3 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

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. 3;*
      – *update* $\boldsymbol{\Delta}_m(r)$*, given input* $\boldsymbol{a}_{\ell m}$ *and computed* $\mathcal{P}_{\ell m}$*, Eq. 6;*
    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)

Algorithm 2. 2-point associated Legendre recurrence

– *initialize the rescaling table;*
– *precompute* $\boldsymbol{\mu}$ *coefficients, Eq. 4;*
for *every ring* r ***do***
  for *every* $m = 0, ..., m_{max}$ ***do***
    – *initialize the recurrence:* $\mathcal{P}_{mm}$, $\mathcal{P}_{m+1,m}$*, Eqs. 4 & 5, using precomputed* $\boldsymbol{\mu}_m$*;*
    – *precompute recurrence coefficients,* $\beta_{\ell m}$ *(fixed* m*,* $\ell \in [m, \ell_{max}]$*), Eq. 3;*
    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. 3;*
      – *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* $\boldsymbol{\Delta}_m$*;*
    end for (ℓ)
  end for (m)
end for (r)

that accorded to a double precision number on a typical processor. To solve this problem, real-time rescaling is employed. The newly computed values are tested if they approach over- or underflow limits and are rescaled if needed. The rescaling coefficients (e.g., in form of their logarithms) are kept tracked of and used to scale back the computed values of $\mathcal{P}_{\ell m}$ at the end. The scaling vector, referred to hereafter as a rescale table, 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 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.

3 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. The loop computing the two-point recurrence is serial in nature and instead we consider two remaining choices for parallelization: the m-loop and the loop over the rings.

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

STEP 1 - $\boldsymbol{\Delta}_m$ CALCULATION
– STEP 1.1 - *assign rings for each thread*
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. 3;*
      – 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. 3;*
      – 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. 6;*
    end for (ℓ)
  end for (m)
end for (r)
GLOBAL COMMUNICATION
– *redistribute* $\{\boldsymbol{\Delta}_m(r),\ m \in \mathcal{M}_i,\ \text{all } r\} \overset{\texttt{MPI\_Alltoallv}}{\Longrightarrow} \{\boldsymbol{\Delta}_m(r),\ r \in \mathcal{R}_i,\ \text{all } m\}$
STEP 2
– *using FFT calculate* $\boldsymbol{s}(\theta, \phi)$ *for all samples for every,* $r \in \mathcal{R}_i$*, given* $\boldsymbol{\Delta}_m(\theta)$ *stored for* $r \in \mathcal{R}_i$ *&* ***all*** m*.*

The 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 due to shared memory limitations. The size of vector $\boldsymbol{\beta}_{\ell m}$, Eq. 3, 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. However, these expensive, repeating calculations would seriously limit performance.

Parallelizing the ring loop (step 1.1 in algorithm 3) 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. 4, 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 3 shows the outline of the GPU computing kernel. It can be observed that the three new steps (1.2, 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.

4 Optimizations for GPU

GPU code optimization follows different rules than regular, CPU based code optimization. In fact, in some cases [9] even the most direct algorithm can outperform the CPU optimized one. On GPUs the relationship between the cost of memory access and amount of computations per kernel is exacerbated and it can be far more beneficial to recompute large segments of constant values instead of fetching them from main memory [10]. Performance loss can also stem from thread divergence due to asymmetrical branching in control flow. Such divergence though detectable by profilers, can be hard to avoid. Based on guidelines for CUDA kernel optimization [11,12,13] and our previous experiences, we focused here on limiting the effect of the slow global memory by buffering, precalculating or reusing data, removing branching in performance-critical sections and canceling warp serialization.

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 that 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). The values $\boldsymbol{a}_{\ell m}$ are transferred in segments during the $P_{\ell m}$ computation in step 1.5. Optimal segment size for all

three vectors is input size and platform dependent and has been found here by manual testing, a process, which could be however automated.

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).

Branch Collapsing. Code branching can severely impair the performance of GPU code, as divergent code is executed sequentially, effectively canceling parallelism. This problem is solved by collapsing the branch into code that has the same outcome 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. On the GPU however binary operators are not applicable to floating point operands. An equivalent version, based on multiplications and subtraction (`v = tv*c + fv*(1-c)`) severely increases overhead and is 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 provides increased performance, they do offer some insight into the behavior 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, some values are accessed from the same bank. Bank access is not concurrent, so the threads are serialized. We tried splitting a 64 bit value into 2 32-bit ones stored in two different vectors, which are rejoined as needed [11, p. 156] but on the GT200 architecture, the computational cost outweighed that of warp serialization. The newer GT400 chips do not display such a problem.

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. Given its small size (21 64-bit values), the simplest approach for canceling serialization is to replicate the table for each thread. However, experiments showed that though no serialization occurs, the time gain is insignificant even for small inputs. Also, for a large

number of threads, the amount of shared memory used becomes a limiting factor (for just 64 threads, 10.5 KB are needed).

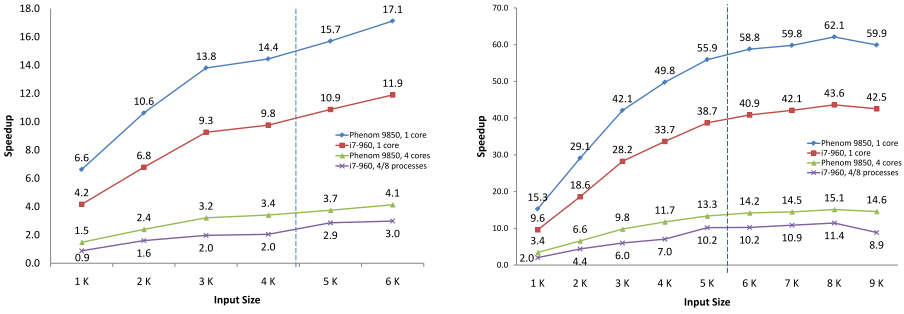
$\boldsymbol{\beta}_{\ell m}$ Precalculation. Based on the ability to execute a very large number of mathematical operations and the drawback of high device memory latency, a method for obtaining a good throughput is computing values on-the-fly instead of precalculating them. This trades computing cycles 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 $\boldsymbol{\beta}_{\ell m}$, making reuse essential. Computing the scaling factors on a need-basis showed a similar problem.

5 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 2.2 and 3.2 times larger for the two GPU platforms 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.

The S^{2}HAT Fortran algorithm was employed as reference for the CPU version. It was compiled with gfortran 4.3, using the default flags active at optimization level 3 (-O3). Manual tuning was applied to improve memory and cache performance. Accelerating spherical harmonics with SSE, could yield a 1.6-1.8x speedup, as suggested by [4] and our limited attempts on the code base, however more research is needed. Additional gains could also be obtained through proprietary compilers, like the Intel Fortran package, which is known to boost runtime by 5-20% over gfortran. Algorithm 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 used is 4096x4096 and 5120x5120, respectively. To assess performance for larger datasets, the output arrays were no longer allocated, leaving the entire card memory for the input. 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 extended up to 9216x9216.



(a) GTX 260 **(b)** GTX 480

Fig. 1. Improvement factor obtained for Δ_m calculation of `alm2map` with CUDA with respect to the MPI version ran on the AMD Phenom and Intel i7 CPUs

5.1 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.

The improvement factor of 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 and thus can run 8 threads on just 4 physical cores. However, we found that, in some cases, the 4 threads (MPI processes) version is faster. Therefore, one process and the best out of 4 or 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.

Figures 1a and 1b 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 could be expected to 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 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 appear to help by pushing scaling beyond the number of physical cores.

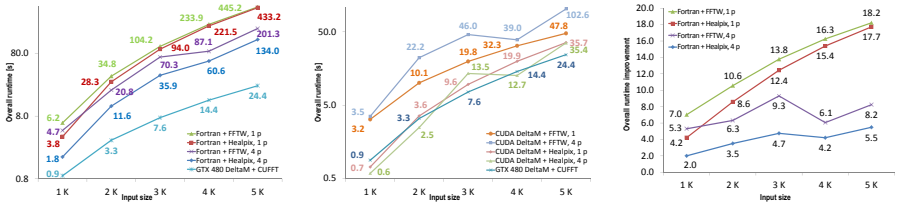


Fig. 2. `alm2map` overall runtime, Intel i7-960 (left) and NVIDIA GTX 480 (middle) and overall performance gain (right)

5.2 Overall Performance

The performance of the `alm2map` algorithm is greatly improved by offloading the $\boldsymbol{\Delta}_m$ computation onto a GPU. In the original CPU-only code, the FFTs, performed as the second step, constitute 5-10% of the total runtime. Accelerating $\boldsymbol{\Delta}_m$ calculation by a factor of 10 (Intel I7-960, 4 processes), results in the FFTs becoming dominant. We have tested two CPU FFT routines (FFT function implemented in HEALPIX [3] and FFTW[1] [14]) and one FFT routine for the NVIDIA GPUs (CUFFT [15]). We have not introduced any specific GPU optimizations in this part of the algorithm.

Left and middle panels of Fig. 2 show the overall ($\boldsymbol{\Delta}_m$ + FFT) runtimes for all combinations of $\boldsymbol{\Delta}_m$ 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 routine, the HEALPIX FFT performs better for both 1 and 4 processes. We also observe that the best runtimes belong to the code running on the GPU.

Right panel of Fig. 2 plots the overall runtime improvement over the CPU code versions with respect to the best performing GPU code (labeled "GTX480 $\boldsymbol{\Delta}_m$ + CUFFT" – middle panel). We observe that, in the best case, the improvement is just half of that obtained when considering only the $\boldsymbol{\Delta}_m$ computation (Fig. 1b), but also significant, reaching factors from 5 to 18.

6 Conclusions

We have described an algorithm for computing the inverse spherical harmonic transform on GPUs and compared it with the inverse spherical harmonic transform provided in the S^{2}HAT library, and based on Fortran and MPI. The GPU

[1] FFTW: `http://www.fftw.org/`

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.

Acknowledgment. This work has been supported in part by French National Research Agency (ANR) through COSINUS program (project MIDAS no. ANR-09-COSI-009) and used HPC resources from GENCI-CCRT/IDRIS (Grant 2011-066647).

References

1. Szydlarski, M., Esterie, P., Falcou, J., Grigori, L., Stompor, R.: Spherical harmonic transform on heterogeneous architectures using hybrid programming, INRIA, Rapport de recherche RR-7635 (April 2011), http://hal.inria.fr/inria-00597576/en/
2. Arfken, G.B., Weber, H.J.: Mathematical methods for physicists, 6th edn. Academic Press (2005)
3. Górski, K.M., et al.: HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. Astrophysical Journal 622, 759–771 (2005)
4. Reinecke, M.: Libpsht - algorithms for efficient spherical harmonic transforms. Astronomy and Astrophysics 526, A108+ (2011)
5. Driscoll, J.R., Healy, D.M.: Computing fourier transforms and convolutions on the 2-sphere. Advances in Applied Mathematics 15(2), 202–250 (1994)
6. Muciaccia, P.F., Natoli, P., Vittorio, N.: Fast Spherical Harmonic Analysis: A Quick Algorithm for Generating and/or Inverting Full-Sky, High-Resolution Cosmic Microwave Background Maps. Astrophysical Journal Letters 488, L63(1997)
7. Doroshkevich, A.G., et al.: First Release of Gauss-Legendre Sky Pixelization (GLESP) software package for CMB analysis. ArXiv Astrophysics e-prints (January 2005)
8. Tygert, M.: Fast algorithms for spherical harmonic expansions, ii. Journal of Computational Physics 227(8), 4260–4279 (2008)
9. Nukada, A., Matsuoka, S.: Auto-tuning 3-D FFT library for CUDA GPUs. In: SC 2009: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–10 (2009)
10. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: ACM/IEEE Conference on Supercomputing, SC 2008 (2008)
11. Nvidia, NVIDIA CUDA Programming Guide (2010)
12. Nvidia, NVIDIA CUDA Best Practices Guide (2010)
13. Nvidia, Tuning CUDA Applications for Fermi (2010)
14. Frigo, M., Johnson, S.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005)
15. Nvidia, CUDA CUFFT Library (2010)