

# ACCELERATING SPHERICAL HARMONIC TRANSFORMS ON THE NVIDIA® GPU

Vikrant Soman

Department of Electrical Engineering  
University of Wisconsin, Madison, Wisconsin, USA

## Abstract

The Spherical Harmonic Transform is a critical computational kernel of the dynamics algorithms for numerical weather prediction and climate modeling. As atmospheric models push towards higher resolutions it has become necessary to accelerate this computationally intensive transform. Previous work has made attempts to parallelize and optimize the transform [1] [2] [3] [4], but none have exploited the advantages of the NVIDIA's General Purpose Graphics Processor Unit (GPGPU), a very recent SIMD type architecture. This paper describes a CPU-GPU type implementation for computation of Spherical Harmonic Transform. The implementation shows gain in terms of computation time and a low error rate, when compared to the implementation discussed in [1].

**Keywords:** Spherical Harmonic Transform, GPU, Parallel Computing

## 1 Introduction

The governing equations for global spectral weather model are derived from the conservation laws of mass, momentum, and energy. Vorticity, divergence, temperature, surface pressure and moisture equations are the main constituents of it. Expansion of the global field is done using spherical harmonics. Thus, spherical harmonic transform is a critical computational kernel of the dynamics algorithms for numerical weather prediction and climate modeling. The spherical harmonic transform is used to project grid point data on the sphere onto the spectral modes in an *analysis* step and an inverse transform reconstructs grid point data from the spectral information in a *synthesis* step. As atmospheric models push towards higher resolutions it has become necessary to accelerate this computationally intensive transform. This project aims to take a step in this direction by exploiting the recent technology of NVIDIA's Graphics Processors Unit (GPU) which is a SIMD type architecture.

The rest of the paper is organized as follows. Section 2 covers prior work in this area, Section 3 describes the Spherical Harmonic Transforms highlighting the data intensive parts in it and also describes the architecture of NVIDIA GPGPU. Section 4 gives an overview of the CPU-GPU implementation. Section 5 provides the results of the implementation Section 6 concludes the paper, Section 7 provides direction for future work and Section 8 provides references. Appendix A, B, C has codes for various kernels implemented as part of this work.

## 2 Prior Work and Motivation

The task of parallelizing the computation of spherical harmonic transforms is not a new art. In fact climate and weather modelers were one of the first users of parallel computers. The calculation of the spherical harmonic transform, is a two step process and methods mentioned in [2] and [4] take advantage of this and parallelize each of the steps. Other methods are on the lines of approximating the original transform equations in a Fourier basis since FFT implementations are computationally quicker. The algorithm explained in [1] gives a matrix implementation of the spherical harmonic transform which essentially makes the transform vectorizable. This is of interest in this context considering that the NVIDIA GPGPU provides some tremendous acceleration in terms of time for algorithms involving the BLAS operations. This provides the motivation for this work where a combined CPU-GPU implementation of the vectorizable algorithm in [1] is expected to give acceleration in terms of computation time and show gains over a conventional CPU implementation.

## 3 A. Spherical Harmonic Transforms

Spherical Harmonic Transforms (SHTs) are essentially Fourier transforms on the sphere. For flows in a global domain, the preferred basis set for approximation of functions on the sphere is the spherical harmonic basis. The spherical harmonic transform is used to project grid point data on the sphere onto the spectral modes in an *analysis* step and an inverse transform reconstructs grid point data from the spectral information in a *synthesis* step. The synthesis step is described in Equation (1). The analysis step is described by Equations (2) and (3) consisting of the computation of the Fourier coefficient  $\xi^m$  and the Legendre transform that incorporates the Gaussian weights corresponding to the Gaussian latitudes  $\mu_j = \sin(\theta_j)$ .

$$\xi(\lambda, \mu) = \sum_{m=-M}^M \sum_{n=|m|}^{N(m)} \xi_n^m P_n^m(\mu) e^{im\lambda} \dots\dots\dots (1)$$

$$\xi_n^m = \sum_{j=1}^J \xi^m(\mu_j) P_n^m(\mu_j) w_j \dots\dots\dots (2)$$

$$\xi^m(\mu_j) = \frac{1}{I} \sum_{i=1}^I \xi(\lambda_i, \mu_j) e^{-im\lambda_i} \dots\dots\dots (3)$$

Legendre functions can be shown as the angular solutions of the Laplacian equation in spherical coordinates.

$$\nabla^2 f = \frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial f}{\partial r} \right) + \frac{1}{r^2 \sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial f}{\partial \theta} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial^2 f}{\partial \varphi^2} = 0$$

In geodesic and other geo-potential applications the Legendre functions can be represented as

$$Y_\ell^m(\theta, \varphi) = \sqrt{(2\ell + 1) \frac{(\ell - m)!}{(\ell + m)!}} P_\ell^m(\cos \theta) e^{im\varphi}$$

For a Gaussian grid the triangular spectral truncation requires the number of longitudes  $I \geq 3M + 1$ , and number of latitudes  $J = I/2$ , where  $M$  refers to the modal truncation number. The code implemented in [1] which we accelerate using this truncation although it is easy to extend it to different truncations.

The Analysis step algorithm summarized is as below.

1. Compute Fourier coefficients using the direct Fourier transform.
2. Compute spectral coefficients by direct Legendre's transform.
3. Perform calculations in spectral domain.

The Synthesis step is summarized below.

1. Input spectral coefficients.
2. Compute Fourier coefficients using inverse Legendre's transform
3. Compute Gaussian grid point values using the Inverse Fourier Transforms.

The fig 1 shows both the steps in form of a flowchart.

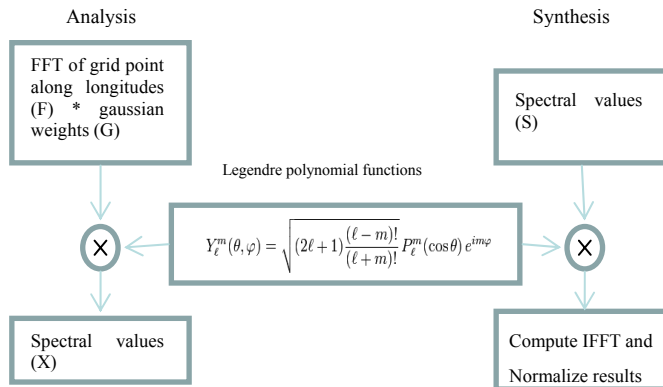


Figure 1. Flowchart for analysis and synthesis

## B. NVIDIA® GPGPU Architecture and CUDA

The sudden surge of popularity of the GPGPU for algorithm computation within the engineering fraternity has been due to the fact that GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 2.



Figure 2. CPU vs GPU.

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

As illustrated by Figure 3, each multiprocessor has on-chip memory of the four following types:

- One set of local 32-bit *registers* per processor,
- A parallel data cache or *shared memory* that is shared by all scalar processor cores and is where the shared memory space resides,
- A read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory
- A read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory

The local and global memory spaces are read-write regions of device memory and are not cached.

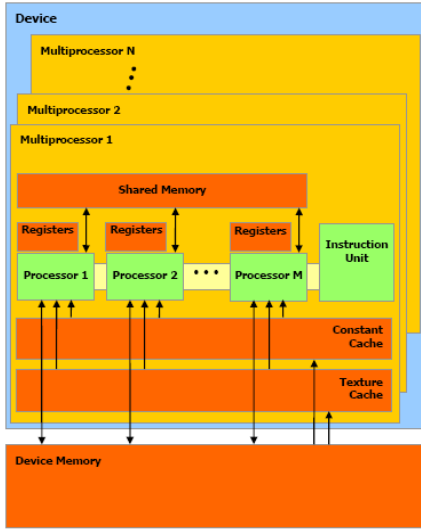


Figure 3. Memory structure of GPU

CUDA extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads for each call is specified using a new `<<<...>>>` syntax

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

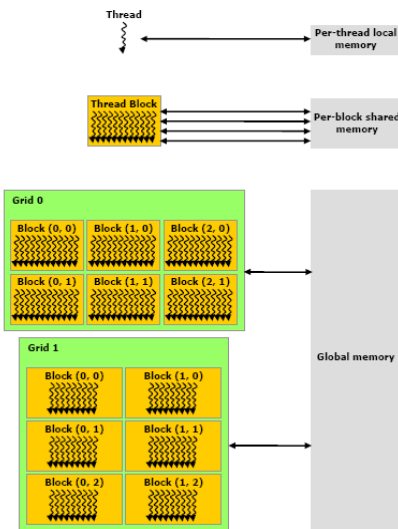


Figure 4. Memory hierarchy of GPU

An advantage that is offered by the GPU programming model is heterogeneity which allows a combined CPU-GPU implementation for an algorithm which is shown in this paper. Thus a small portion of a C program can be offloaded to GPU for computation and once results are retrieved back from the GPU the C code can continue its serial execution.

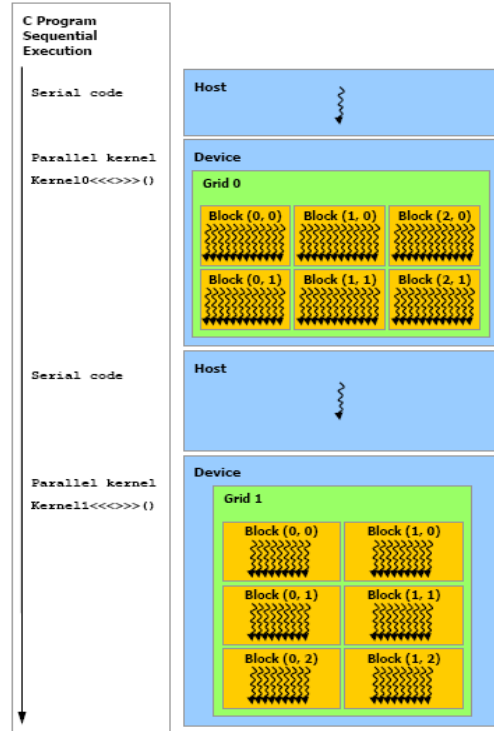


Figure 5. Heterogeneous programming model

## 4 Implementation Details

Algorithm exploits the heterogeneous programming model for GPU and extends the algorithm mentioned in [1] for SHT synthesis and analysis. The first step was profiling the code in [1] and identifying the data intensive and time consuming loops. For this purpose the analysis and synthesis code was executed on the CPU.

After that came the part of checking whether the areas in the code could be offloaded to the GPU for computation and achieve speed up in the code. It turned out that most of the loops could be modified to match the GPU architecture and transformed into a CUDA kernel that can be used on the GPU. The 3 main areas that were targeted were the Legendre Polynomial computation, the Analysis step projection loop and the Synthesis step reconstruction loop. We will consider each of the steps below.

A. Legendre polynomial calculation.

Figure 6 shows the combined CPU-GPU architecture for the Legendre polynomial calculations. As seen in the figure some operations are carried on the CPU while some are done on the GPU side.

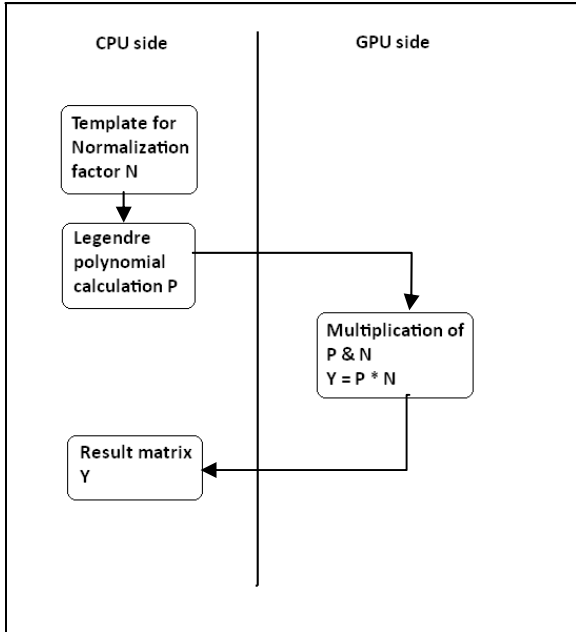


Figure 6. Flowchart Legendre Polynomial Computation

On the GPU side using the knowledge of the architecture the kernel is written in such a way that maximum elements are computed in parallel. Thus we see that as shown in the figure 7, the 3D matrix P and 2D matrix of FFT coefficients are the inputs while Y is the output returned back to the CPU. The arrow shows the direction of the kernel execution which is called multiple times along the depth of the matrix P.

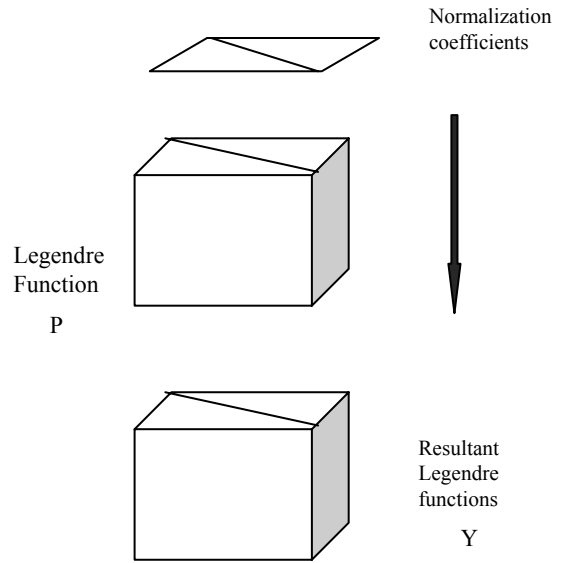


Figure 7. Kernel mapping for Legendre Polynomial Computation

### B. Analysis step

Figure 8 shows the implementation for the analysis step. The projection of points is offloaded to the GPU. Note that in this case the kernel in GPU is invoked for the real and the imaginary parts of the FFT coefficients.

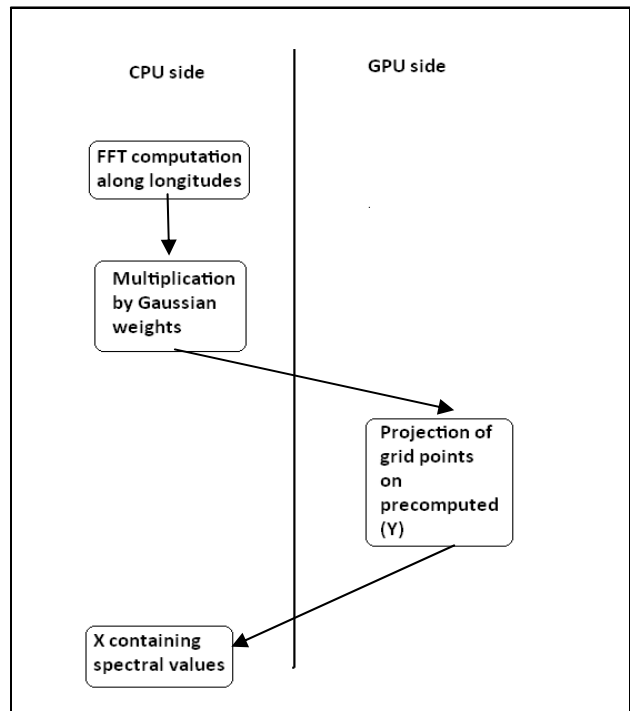


Figure 8. Flowchart Analysis Step

For the analysis step, the FFT is still computed on the CPU side since the GPU is shown to give good speed up for FFT only for a very high point FFT, while MATLAB® has been shown to have a highly optimized BLAS code for computing FFT on the CPU. Thus only the projection part is done on the GPU. The resultant projection is a 2D matrix as shown in the figure 9. The bold arrow shows the direction of kernel invocation as mentioned earlier.

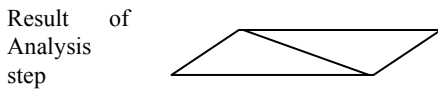
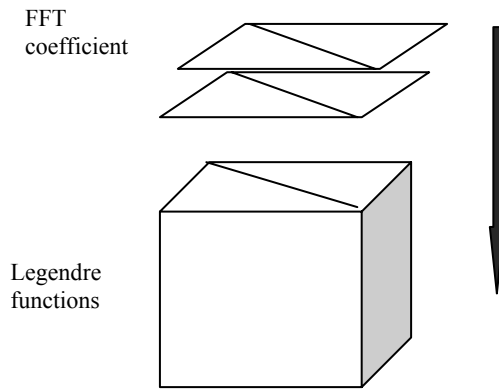


Figure 9. Kernel mapping for Analysis step

### C. Synthesis step

Figure 10 shows the implementation for the synthesis step. The reconstruction of grid points is offloaded to the GPU. Note that in this case as well the kernel in GPU is invoked for the real and the imaginary parts of the FFT coefficients.

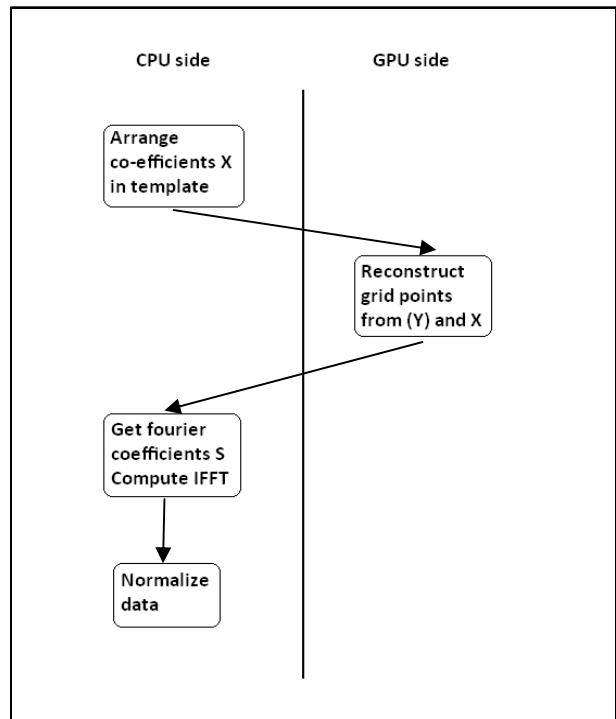


Figure 10. Flowchart for Synthesis step

Once again the IFFT is computed on CPU side for reasons mentioned before. Also in the synthesis step the difference is in the direction of the kernel invocation. One may notice that since we are reconstructing the grid points we move along the direction of the horizontal latitudes summing over the points to get back the fourier coefficients. The result is a 2D matrix as shown below.

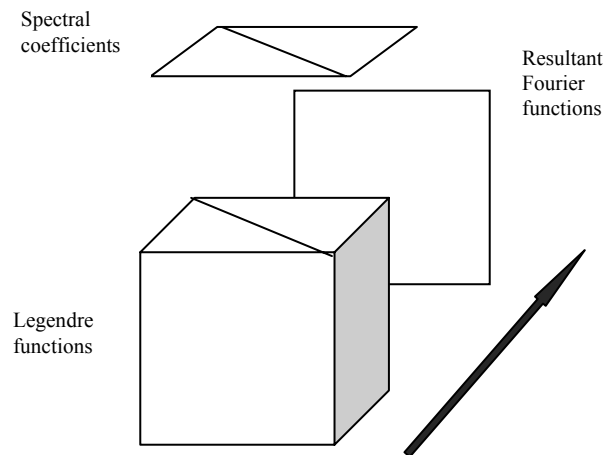


Figure 11. Kernel mapping for Synthesis step

The code mentioned in [1] has been written in MATLAB® and executed on the CPU. The GPU code was first written in CUDA and tested with sample inputs to check the indexing. For the actual interfacing between CPU and GPU the CUDA code was wrapped using MATLAB® extensions for CUDA provided by NVIDIA® called NVMEX. This allows a user to pass data to the GPU seamlessly via a function call within a MATLAB® code and returns the desired output data back to the calling MATLAB® code.

## 5 Results

The results were collected on a CPU with configuration as follows.

1. Intel® Quad Core
2. 2.5 Ghz
3. 2.5GB RAM

The GPU was a NVIDIA® 8800 GT. The results were collected for 3 mentioned areas. Error analysis was done for the Legendre polynomial calculation while the CPU versus CPU-GPU execution was compared.

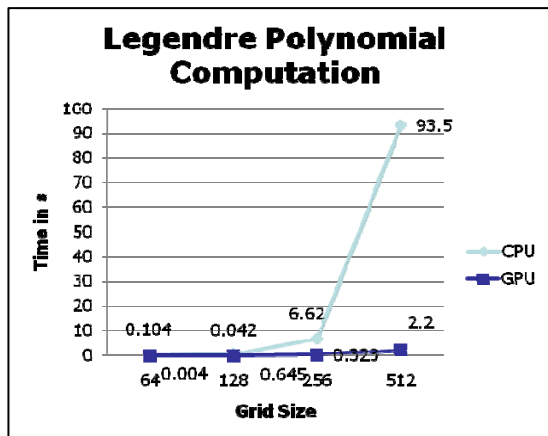


Figure 12. CPU vs CPU-GPU for Legendre polynomial computation

As we can see from the fig 12 the CPU-GPU implementation shows tremendous acceleration as grid sizes increase. For grid size 512 it gives a gain of 42x. The trend for higher grid sizes looks the same. The error analysis showed an error of E-10 between CPU values and CPU-GPU implementation.

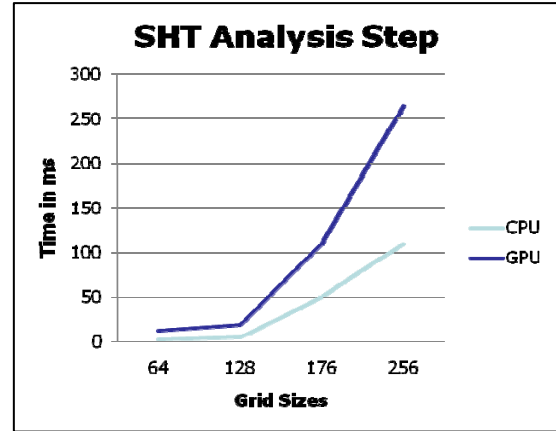


Figure 13. CPU vs CPU-GPU for Analysis step

The analysis step on the other hand didn't show any optimization over the traditional CPU implementation. This can be due to 2 reasons. The kernel had to be executed twice since in analysis step there were FFT values that were complex in nature. This increases the communication overhead slightly for each value resulting in a sub optimal performance. The only good part is that the CPU-GPU timing values do not deviate abnormally from the CPU timings and show similar nature

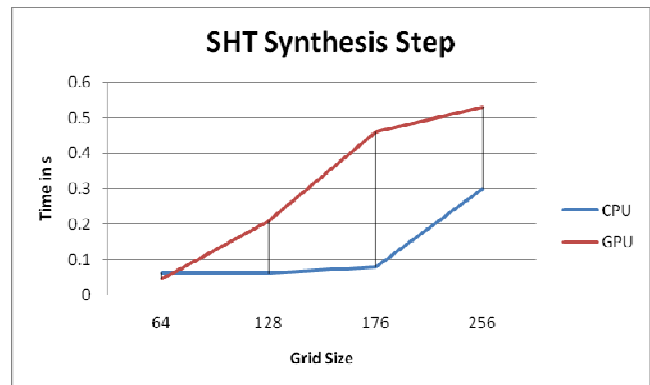


Figure 14. CPU vs CPU-GPU for Synthesis step

The synthesis step also dealt with complex values and had to face the penalty of communication overhead. Just like the analysis step the values are not exponentially deviant from the CPU timings but at the same time do not show any tremendous speed up like the Legendre polynomial computation. The 2 graphs do seem like they might meet each other for higher grid sizes but that could not be verified because of memory constraints on host machine.

## 6 Conclusions

The Legendre polynomial computation showed tremendous acceleration and for higher grid sizes the trend shows to be positive. The Analysis and Synthesis step did not so much

acceleration but the performance wasn't deviant from the pure CPU implementation. The errors were low less than E-10 on an average even though the GPU is a single precision device.

The GPU shows some tremendous acceleration for BLAS operations compared to the CPU. Assuming a zero communication overhead between CPU-GPU a heterogeneous programming model would be ideal. But in the real world there is some overhead and it conversely affects the speed up that might have been achieved by the GPU.

## 7 Future Work

It needs to be seen whether the synthesis and analysis computation for the Spherical Harmonic Transform shows any significant changes for higher grid sizes. Implementations for other truncation methods also need to be looked at.

## 8 References

1. **Drake, J. B., Worley, P., and D'Azevedo, E. 2008.** Algorithm 888: Spherical harmonic transform algorithms. *ACM Trans. Math. Softw.* 35, 3, Article 23 (October 2008)
2. **Akshara Kaginalkar, Sharad Purohit,** Benchmarking of Medium Range Weather Forecasting Model on PARAM -A parallel machine, Center for Development of Advanced Computing (C-DAC), Pune University Campus, Pune 411007 India
3. **Martin J. Mohlenkamp,** A Fast Transform for Spherical Harmonics, *The Journal of Fourier Analysis and Applications*, 1999
4. **Huadong Xiao, Yang Lu,** Parallel computation for spherical harmonic synthesis and analysis, *Computers & Geosciences*, Volume 33, Issue 3, March 2007 5.
5. **NVIDIA CUDA Programming Guide 2.0**

*Special thanks to Prof. Dan Negrut and Makarand Datar at the SBEL, University of Wisconsin, Madison for allowing access to their GPU machines and other facilities.*

## A. Legendre polynomial computation kernel

```
//headers

#include <stdlib.h>
#include "mex.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "cufft.h"
#include "driver_types.h"
#include <stdio.h>
#define BLOCK_SIZE 16
// Kernel for multiplication
#ifndef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_
#define CHECK_BANK_CONFLICTS 0
#if CHECK_BANK_CONFLICTS
#define AS(i, j) CUT_BANK_CHECKER(((float*)&As[0][0]), (BLOCK_SIZE * i + j))
#define BS(i, j) CUT_BANK_CHECKER(((float*)&Bs[0][0]), (BLOCK_SIZE * i + j))
#else
#define AS(i, j) As[i][j]
#define BS(i, j) Bs[i][j]
#endif
#define Csub(i,j) Csub[i][j]

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//! Matrix multiplication on the device:
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
__global__ void matrixMul( float* A, float* B,int wA,int wB,int k)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);

    // Index of the first sub-matrix of B processed by the block
    int bBegin_x = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);

    __shared__ float Csub[BLOCK_SIZE][BLOCK_SIZE];
```

```

// Declaration of the shared memory array As used to
// store the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Declaration of the shared memory array Bs used to
// store the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix

AS(ty, tx) = A[k*wA*wA + aBegin + wA * ty + tx];
BS(ty, tx) = B[bBegin_x + wB * ty + tx];

Csub (ty,tx) = 0;

// Synchronize to make sure the matrices are loaded
__syncthreads();
{
    Csub(ty,tx) = AS(ty,tx) * BS(ty,tx);
    __syncthreads();

    int c = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);
    A[k*wA*wA + c + tx + ty*wA] = Csub(ty,tx);

    __syncthreads();
}

#endif // #ifndef _MATRIXMUL_KERNEL_H_

```

## B. Analysis step kernel

```

//headers
#include <stdlib.h>
#include "mex.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "cufft.h"
#include "driver_types.h"
#include <stdio.h>
#define BLOCK_SIZE 16
// Kernel for multiplication
#ifndef _ANALYSIS_KERNEL_H_
#define _ANALYSIS_KERNEL_H_
#define CHECK_BANK_CONFLICTS 0
#if CHECK_BANK_CONFLICTS

```

```

#define AS(i, j) CUT_BANK_CHECKER(((float*)&As[0][0]), (BLOCK_SIZE * i + j))
#define BS(i) CUT_BANK_CHECKER(((float*)&Bs[0]), (BLOCK_SIZE * i))
#define CS(i) CUT_BANK_CHECKER(((float*)&Cs[0]), (BLOCK_SIZE * i))
#define ResSub(i,j) CUT_BANK_CHECKER(((float*)&ResSub[0][0]), (BLOCK_SIZE * i + j))
#else
#define AS(i, j) As[i][j]
#define BS(i) Bs[i]
#define CS(i) Cs[i]
#define ResSub(i,j) ResSub[i][j]
#endif
#define Csub(i,j) Csub[i][j]

////////////////////////////////////
//! Matrix multiplication on the device:
////////////////////////////////////
__global__ void
analysis( float* Res,float* A, float* B,float *C,int wA,int wB,int wC,int k)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);
    int resBegin = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);

    // Index of the first sub-matrix of B processed by the block
    int bBegin_x = BLOCK_SIZE * bx;
    int cBegin_x = BLOCK_SIZE * bx;

    __shared__ float Csub[BLOCK_SIZE][BLOCK_SIZE];

    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float ResSub[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE];
    __shared__ float Cs[BLOCK_SIZE];

    ResSub(ty,tx) = 0;
    AS(ty, tx) = A[k*wA*wA + aBegin + wA * ty + tx];
    ResSub(ty,tx) = Res[resBegin + wB * ty + tx];

```

```

    BS(tx) = B[bBegin_x + tx];
    CS(tx) = C[cBegin_x + tx];

    Csub (ty,tx) = 0;

    // Synchronize to make sure the matrices are loaded
    __syncthreads();

    Csub(ty,tx) = ResSub(ty,tx)+( AS(ty,tx) * BS(tx) + powf(-1,(ty-tx))*CS(tx)*AS(ty,tx));

    __syncthreads();

    int c = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);
    Res[c + wA * ty + tx] = Csub(ty,tx);
    __syncthreads();
}

#endif // #ifndef _ANALYSIS_KERNEL_H_

```

### C. Synthesis step kernel

```

//headers
#include <stdlib.h>
#include "mex.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "cufft.h"
#include "driver_types.h"
#include <stdio.h>
#define BLOCK_SIZE 16
// Kernel for multiplication
#ifndef _SYNTHESIS_KERNEL_H_
#define _SYNTHESIS_KERNEL_H_

#define CHECK_BANK_CONFLICTS 0
#if CHECK_BANK_CONFLICTS
#define AS(i, j) CUT_BANK_CHECKER(((float*)&As[0][0]), (BLOCK_SIZE * i + j))
#define BS(i) CUT_BANK_CHECKER(((float*)&Bs[0]), (BLOCK_SIZE * i))
#define ResSub(i,j) CUT_BANK_CHECKER(((float*)&ResSub[0][0]), (BLOCK_SIZE * i + j))
#else
#define AS(i, j) As[i][j]
#define BS(i) Bs[i]
#define ResSub(i,j) ResSub[i][j]
#endif
#define Csub(i,j) Csub[i][j]

```

```

////////////////////////////////////
//! Matrix multiplication on the device
////////////////////////////////////
__global__ void
synthesis( float* Res,float* A, float* B,int wA,int lA,int wB,int k)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);
    int resBegin = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);

    __shared__ float Csub[BLOCK_SIZE][BLOCK_SIZE];

    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Ressub[BLOCK_SIZE][BLOCK_SIZE];

    __shared__ float Bs[BLOCK_SIZE];//[BLOCK_SIZE];

    ResSub(ty,tx) = 0;
    AS(ty, tx) = A[k*wA*lA + aBegin + wA * ty + tx];
    ResSub(ty,tx) = Res[resBegin + wA * ty + tx];
    BS(tx) = B[k * wB + BLOCK_SIZE*bx+tx];
    Csub (ty,tx) = 0;

    // Synchronize to make sure the matrices are loaded
    __syncthreads();

    Csub(ty,tx) = ResSub(ty,tx)+( AS(ty,tx) * BS(tx) );

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    // __syncthreads();

    int c = bx*BLOCK_SIZE + by*BLOCK_SIZE*BLOCK_SIZE*(wA/BLOCK_SIZE);
    Res[c + wA * ty + tx] = Csub(ty,tx);
    __syncthreads();
}

#endif // #ifndef _SYNTHESIS_KERNEL_H_

```