Este programa es idéntico en proposito al descrito en getnet (no shared memory) pero para acelerar los calculos utiliza la memoria compartida de la tarjeta. **Los resultados en aumento de velocidad son asombrosos**.

La magia está en no lanzar un solo *thread* por proceso sino varios y que cada uno haga una parte del calculo. Esto se hace tomando de las propiedades de la tarjeta cuantos hilos pueden lanzarse,

```
    cudaGetDeviceProperties( &prop, 0 );
    dim3 numthreads(prop.maxThreadsPerBlock,1,1);

//.......................

    network<<<numgrids,numthreads>>> (x_dev, net_dev);
```

Así, dentro del kernel no hay que realizar un ciclo por cada punto de la mtriz sino solo un numero mucho menor (1/prop.maxThreadsPerBlock).

```
__global__ void network (float *matrix, float *net){
    __shared__ float tnet[NPROC];
    int myel = blockIdx.x + blockIdx.y*shape.imd[0]+ blockIdx.z*shape.plane;
    int extel = threadIdx.x;
    int sindex = threadIdx.x;
    float temp = 0;

    while (extel<shape.vol) {
        if(myel!=extel && matrix[myel]>0 && matrix[extel]>0){
            temp += matrix[myel]*matrix[extel]/distance(myel,extel);
        }
        extel += blockDim.x;
    }
    tnet[sindex] = temp;
//.............................................
}
```

los resultados se van almacenando en un array definido dentro de cada hilo

```
    __shared__ float tnet[NPROC];
//.............................................
    int sindex = threadIdx.x;
//.............................................
    tnet[sindex] = temp;
```

pero despues estos resultados deben sumarse dentro de cada bloque. Para ello se usa la tecnica de reduction por lo que debe sincronizarse y efecturar el *folding*,

```
    __syncthreads();
    size_t i = blockDim.x/2;
    while (i!=0) {
        if (sindex < i) {
            tnet[sindex] += tnet[sindex + i];
        }
```

```
        __syncthreads();
        i/=2;
    }
    if (sindex == 0){
        net[myel] = tnet[0];
    }
    return;
}
```

Tras lo cual se devuelve el resultado.

Este es el codigo completo,

getnet.cu

```
/*
 * getnet.c
 *
 * Copyright 2013 O. Sotolongo <osotolongo@fundacioace.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301, USA.
 *
 *
 */


#include <stdio.h>
#include <string.h>
#include <cuda.h>

#define IMDIM 3
#define NPROC 1024
#define imin(a,b) (a<b?a:b)

typedef struct {
    int imd[IMDIM];
    int plane, vol;
    float pixd[IMDIM];
```

```
    } image_info;

__constant__ image_info shape;

void get_img(char *myfile, float *img){
    int i = 0;
    FILE *f = fopen(myfile,"r");
    while (fscanf(f, "%f", &img[i])!=EOF){i++;}
    fclose(f);
    return;
}

void get_img_info(char *myfile, image_info *img){
    FILE *f = fopen(myfile,"r");
    fscanf(f, "%d", &img->imd[0]);
    fscanf(f, "%d", &img->imd[1]);
    fscanf(f, "%d", &img->imd[2]);
    fscanf(f, "%f", &img->pixd[0]);
    fscanf(f, "%f", &img->pixd[1]);
    fscanf(f, "%f", &img->pixd[2]);
    fclose(f);
    return;
}

__device__ float distance (int i, int j){
    int k = abs(i-j);
    int x,y,z;
    int xy;
    z = (int) k/shape.plane;
    xy = k%shape.plane;
    y = (int) xy/shape.imd[0];
    x = xy%shape.imd[0];
    return shape.pixd[0]*x*shape.pixd[0]*x +
shape.pixd[1]*y*shape.pixd[1]*y + shape.pixd[2]*z*shape.pixd[2]*z;
}

__global__ void network (float *matrix, float *net){
    __shared__ float tnet[NPROC];
    int myel = blockIdx.x + blockIdx.y*shape.imd[0]+
blockIdx.z*shape.plane;
    int extel = threadIdx.x;
    int sindex = threadIdx.x;
    float temp = 0;

    while (extel<shape.vol) {
        if(myel!=extel && matrix[myel]>0 && matrix[extel]>0){
            temp += matrix[myel]*matrix[extel]/distance(myel,extel);
        }
        extel += blockDim.x;
    }
    tnet[sindex] = temp;
```

```
    __syncthreads();
    size_t i = blockDim.x/2;
    while (i!=0) {
        if (sindex < i) {
            tnet[sindex] += tnet[sindex + i];
        }
        __syncthreads();
        i/=2;
    }
    if (sindex == 0){
        net[myel] = tnet[0];
    }
    return;
}

int main(int argc, char **argv){
    float *x, *x_dev, *net, *net_dev;
    image_info *tmpd = (image_info*) malloc (sizeof(image_info));
    char img_name[20] = "", results[20] = "", info[20] = "", matrix[20]
= "";
    FILE *odf;
    cudaDeviceProp  prop;

    strcpy(img_name, argv[1]);
    strcpy(results, img_name);
    strcpy(info, img_name);
    strcpy(matrix, img_name);
    strcat(results, ".net");
    strcat(info, ".info");
    strcat(matrix, ".asc");

    get_img_info(info, tmpd);
    tmpd->plane = tmpd->imd[0]*tmpd->imd[1];
    tmpd->vol = tmpd->plane*tmpd->imd[2];

    // Definir los arrays en host
    x = (float *) malloc(tmpd->vol*sizeof(float));
    net = (float *) malloc(tmpd->vol*sizeof(float));
    // Definir los arrays en device
    cudaMalloc((void **)&x_dev, tmpd->vol*sizeof(float));
    cudaMalloc((void **)&net_dev, tmpd->vol*sizeof(float));
    cudaMalloc((void **)&shape, sizeof(image_info));

    get_img(matrix, x);

    // Cojo la primera (por ahora) GPU y guardo las propiedades
    cudaGetDeviceProperties( &prop, 0 );
    dim3 numthreads(prop.maxThreadsPerBlock,1,1);
    dim3 numgrids(tmpd->imd[0],tmpd->imd[1],tmpd->imd[2]);
    // Copiar a memoria constante las caracteristicas de la imagen
```

```
    cudaMemcpyToSymbol(shape, tmpd, sizeof(image_info));
    // Copiar a device la matriz
    cudaMemcpy(x_dev, x, tmpd->vol*sizeof(float),
cudaMemcpyHostToDevice);

    // the real deal
    network<<<numgrids,numthreads>>> (x_dev, net_dev);

    // Copiar a host el resultado
    cudaMemcpy(net, net_dev, tmpd->vol*sizeof(float),
cudaMemcpyDeviceToHost);

    //write network to disk
    odf = fopen(results, "w");
    int i;
    for (i = 0; i<tmpd->vol; i++){
        fprintf(odf, "%f ", net[i]);
    }
    fclose(odf);

    //Free the mallocs!
    free(x);
    free(net);
    cudaFree(x_dev);
    cudaFree(net_dev);
    cudaDeviceReset();
    return 0;
}
```