Este programa toma una matriz tridimensional (en la practica una imagen 3D) y calcula una medida de interaccion entre los diferentes puntos. La medida que se ha escogido aqui es, $$ n_{i} = \sum_j{\frac{V_i V_j}{r^2}}$$ siendo r^2 la distancia euclidiana entre los dos puntos y V_i la actividad medida en el punto i-esimo.

En una imagen de tamaño standard MNI de 91x109x91 el procedimiento en serie es extremadamente largo por lo que se intento paralelizarlo. Esta es la forma mas simple de hacerlo pero tambien es poco eficiente.

Primeramente se define la estructura de la imagen, que se lee de un archivo, asi como los datos. Tras obtener los resultados, estos se guardan a disco.

```
#include <stdio.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;

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;
}

//......

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));

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

    get_img(matrix, x);

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


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

    //Free the mallocs!
    free(x);
    free(net);
    return 0;
}
```

Hasta aqui esto es solo un prototipo de lo que queremos hacer. Para empezar a trabajar, definimos una variable en la **Constant Memory** (esta memoria es muy rapida y compartida entre todos los hilos) y guardamos ahi la estructura de la matriz. Copiamos a la memoria de la tarjeta la matriz de trabajo y reservamos tambien espacio para la matriz de resultados.

Lo que se hara es que para cada punto de la imagen se lanza un proceso (*block*) dentro del cual se calcula la suma.

Despues de calcular copiaremos los reslutados de la tarjeta a la cpu y tras escribir los reslutados a disco liberaremos la memoria.

```
__constant__ image_info shape;

//.........

int main(int argc, char **argv){

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

    // 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));
//.............
    //Defino el tamaño de grid
    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,1>>> (x_dev, net_dev);
    // Copiar a host el resultado
    cudaMemcpy(net, net_dev, tmpd->vol*sizeof(float),
cudaMemcpyDeviceToHost);
//.............

    //Free the mallocs!
    free(x);
    free(net);
    cudaFree(x_dev);
    cudaFree(net_dev);
    cudaDeviceReset();

//.............
```

El trabajo real se realiza en el *kernel*, en este caso la funcion *network*.

```
__global__ void network (float *matrix, float *net){
    int myel = blockIdx.x + blockIdx.y*shape.imd[0]+ blockIdx.z*shape.plane;
    int i;
    float temp = 0;
    for(i=0; i<shape.vol; i++){
        if(myel!=i && matrix[myel]>0 && matrix[i]>0){
            temp += matrix[myel]*matrix[i]/distance(myel,i);
        }
    }
    net[myel] = temp;
    return;
}
```

Aqui se hace un ciclo por cada valor de la matriz y se guardan los valores de la suma. En cada punto

de este ciclo se llama a la funcion *distance*. Esta funcion debe ser definida en la tarjeta,

```
__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;
}
```

Notese que la variable *shape* que fue definida anteriormente puede llamarse desde cualquier funcion que se ejecute en la tarjeta.

El programa final es algo asi:

getnet_noshared.cu

```
#include <stdio.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){
    int myel = blockIdx.x + blockIdx.y*shape.imd[0]+
blockIdx.z*shape.plane;
    int i;
    float temp = 0;
    //printf("myel: %d, %d\n", myel, shape.vol);
    for(i=0; i<shape.vol; i++){
        if(myel!=i && matrix[myel]>0 && matrix[i]>0){
            temp += matrix[myel]*matrix[i]/distance(myel,i);
        }
    }
    net[myel] = temp;
    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);

    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,1>>> (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\n ", net[i]);
    }
    fclose(odf);

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