Solving the heat transfer equation by a finite difference method using multi-dimensional arrays in CUDA as in standard C

Josefina Sanchez-Noguez[1], Carlos Couder-Castañeda[2], J.J. Hernández-Gómez[2], Itzel Navarro-Reyes[3]
[1] Facultad de Estudios Superiores Acatlán, UNAM, Programa de Matemáticas Aplicadas y Computación
[2] Centro de Desarrollo Aeroespacial, Instituto Politécnico Nacional
[3] Escuela Superior de Física y Matemáticas, Instituto Politécnico Nacional

Abstract

In recent years the increasing necessity to speed up the execution of numerical algorithms has leaded researchers to the use of co-processors and graphic cards such as the NVIDIA GPU's. Despite CUDA C meta-language was introduced to facilitate the development of general purpose-applications, the solution to the common question: How to allocate (cudaMalloc) two-dimensional array?, is not simple. In this paper, we present a memory structure that allows the use of multidimensional arrays inside a CUDA kernel, to demonstrate its functionality, this structure is applied to the explicit finite difference solution of the non-steady heat transport equation.

Motivation

The motivation of this paper arises during the development of an application based on finite difference method to solve the heat transfer equation, to facilitate the implementation of the algebraic expressions. The main difficulty when implementing a finite-difference code on a GPU comes from the computational stencil. For example, a fourth-order spatial operator, the thread that handles the calculation of point (i, j, k) needs to access the arrays points $(i+1, j, k)$, $(i+2, j, k)$,$(i-1, j, k)$,$(i-2, j, k)$,$(i, j+1, k)$ and so on. This implies that 13 accesses to the memory are needed on the GPU to approximate fourth-order finite difference. Due to the number of discrete points that have to be handled, the algebraic expression implementation could be error-prone. For this reason in this work is proposed a data structure that allows access to the *multidimensional indices inside a CUDA kernel* improving the readability and programmability.

Study Case: non-steady heat transport equation

The bidimensional non-steady heat transport is used,

$$\frac{\partial T}{\partial t} = C\left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2}\right), \quad 0 \leq x, y \leq 1, t \geq 0.$$

in the explicit finite difference form, expressed as:

$$T_{i,j}^{l+1} = T_{i,j}^{l} + C\Delta t\left(\frac{T_{i+1,j}^{l} - 2T_{i,j}^{l} + T_{i-1,j}^{l}}{\Delta x^2} + \frac{T_{i,j+1}^{l} - 2T_{i,j}^{l} + T_{i,j-1}^{l}}{\Delta y^2}\right),$$

to probe the structure proposed.
Also the tridimensional non-steady heat transport equation, is solved

$$\frac{\partial T}{\partial t} = C\left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2} + \frac{\partial T^2}{\partial z^2}\right) \quad 0 \leq x, y, z \leq 1, t \geq 0,$$

using its explicit finite difference form:

$$T_{i,j,k}^{l+1} = T_{i,j,k}^{l} + C\Delta t\left(\frac{T_{i+1,j,k}^{l} - 2T_{i,j,k}^{l} + T_{i-1,j,k}^{l}}{\Delta x^2} + \frac{T_{i,j+1,k}^{l} - 2T_{i,j,k}^{l} + T_{i,j-1,k}^{l}}{\Delta y^2} + \frac{T_{i,j,k+1}^{l} - 2T_{i,j,k}^{l} + T_{i,j,k-1}^{l}}{\Delta z^2}\right).$$

Traditional kernel implementation for the 2D case

```
__global__ void kernel(float **T, float **Tn, int ny, int\
nx, float dt, float C, float dx2, float dy2)
{ int i,j,l;
j = blockIdx.x*blockDim.x+threadIdx.x;
i = blockIdx.y*blockDim.y+threadIdx.y;
float Txx; float Tyy
if( (i > 0) && (i<(ny-1)) && (j > 0) && (j < (nx-1))) {
l=i*nx+j
Txx = (T[l+1] -2.0f*T[l] + T[l-1])/dx2;
Tyy = (T[l+nx] -2.0f*T[l] + T[l-nx])/dy2;
Tn[l] = T[l]+dt*C*(Txx+Tyy);}
}
```

Notice the classical use of **only one index** (l) to operate with complex computational molecules.

Contiguous memory allocation for 2D array in CUDA C

To can handle **two indices inside a CUDA Kernel** is necessary to create two pointers, the first one to transfer data between CPU and GPU, and the second one to allocate a 2D array similar to standard C.

```
void Array_2D_GPU(float **P,float ***M,int n,int m) \
int i; float ** P_M, ** dev_M;
P_M = (float **) malloc(n*sizeof(float *));
if (P_M == NULL){printf("Memory error"); exit(-1); }
cudaMalloc((void**)&P_M[0],n*m*sizeof(float));
if (P_M[0]==NULL){ printf("Memory error"); exit(-1); }
for (i=1; i<n; i++) P_M[i] = P_M[0] + i * m;
cudaMalloc((void***)&dev_M,n*sizeof(float*));
if (dev_M==NULL){printf("Memory error"); exit(-1); }
cudaMemcpy(dev_M,P_M,n*sizeof(float*), cudaMemcpyHostToDevice);
*(P) = P_M[0]; *(M) = dev_M;
```

dev_M points to the 2D array that could be used as a 2D array inside a kernel, and P_M is used to transfer data between CPU and GPU.

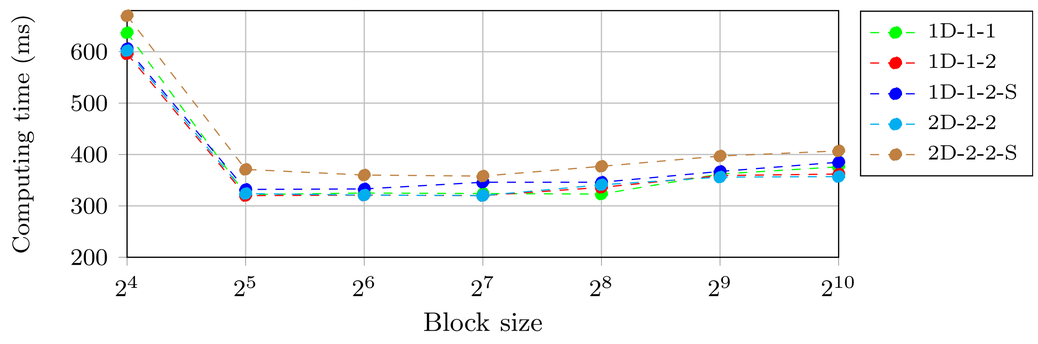
Proposed kernel implementation for the 2D case

```
__global__ void kernel(float **T, float **Tn, int ny, int\
nx, float dt, float C, float dx2, float dy2)
{ int i,j;
j = blockIdx.x*blockDim.x+threadIdx.x;
i = blockIdx.y*blockDim.y+threadIdx.y;
float Txx; float Tyy
if( (i > 0) && (i<(ny-1)) && (j > 0) && (j < (nx-1))) {
Txx = (T[i][j+1] -2.0*T[i][j] +T[i][j-1])/dx2;
Tyy = (T[i+1][j] -2.0*T[i][j] +T[i-1][j])/dy2;
Tn[i][j] = T[i][j]+dt*C*(Txx+Tyy);}
}
```

In this case, two indices are operating inside the kernel, giving more expressiveness, and **provides more clarity to code implementation**.

Performance

Several numerical experiments were conducted using double and single precision over a mid-range card RTX 2060, with 8GB of global memory with CUDA 11.4. For the 2D Case, the Block size was configured in different sizes: 4×4, 8×4, 8×8, 16×8, 16×16, 32×16 and 32×32. The notation used is XD-I, where X is the dimension of the array used inside the kernel, and I the number of indices used and the S indicates the use of shared memory.



The obtained results regarding performance shown that there is no overload due to the introduction of the 2D structure.

Final Remark

This work is introduced a data structure that allows creating multi-arrays in CUDA as it is done in standard C language. It consists of defining an auxiliary pointer to transfer data between CPU and GPU. This structure improves the readability of the source code because it can handle [][] or [][][] indices for 2D and 3D arrays respectively, easing the maintenance and modification of the application, especially in the management of the boundary conditions. The solution of the heat transport equation by the finite difference method is selected as a study case.