An Introduction to GPU Computing for Numerical Simulation

José Miguel Mantas, Marc De la Asunción, and Manuel J. Castro

Abstract Graphics Processing Units (GPUs) have proven to be a powerful accelerator for intensive numerical computations. The massive parallelism of these platforms makes it possible to achieve dramatic runtime reductions over a standard CPU in many numerical applications at a very affordable price. Moreover, several programming environments, such as NVIDIA's Compute Unified Device Architecture (CUDA) have shown a high effectiveness in the mapping of numerical algorithms to GPUs. These notes provide an introduction to the development of CUDA programs for numerical simulation using CUDA C/C++, the most popular GPU programming toolkit. An overview of CUDA programming will be illustrated through the CUDA implementation of simple numerical examples for PDEs. These CUDA implementations will be studied and run on modern GPU-based platforms.

Keywords Graphics processing units • CUDA • Numerical solution of PDEs • CUDA C programming

1 Introduction to GPU Computing

A *graphics processing unit (GPU)* is a programmable single-chip processor which is used primarily for things such as: rendering of 3D graphics scenes, 3D object processing and 3D motion. All these tasks are computing-intensive and highly parallel. A GPU performs arithmetic operations in parallel on multiple data to render images. Generally, GPUs are embedded in stand-alone cards which include their own memory. In order to use them, these cards must be connected to the motherboard of a CPU-based computer system, by using the suitable high speed connection (Peripheral Component Interconnect Express bus, PCI-Express [11]) (Fig. 1).

J.M. Mantas (✉)
Departamento de Lenguajes y Sistemas informáticos, Universidad de Granada, Granada, Spain
e-mail: jmmantas@ugr.es

M. De la Asunción • M.J. Castro
Dpto. Análisis Matemático, Universidad de Málaga, Málaga, Spain
e-mail: marcah@uma.es; castro@anamat.cie.uma.es

Fig. 1 GPU cards

Modern GPUs are highly programmable and can be used for non-graphics applications. They offer hundreds or thousands of processing units optimized for massively performing floating-point operations in parallel. In fact, they can be a cost-effective way to obtain a substantially higher performance in computationally intensive tasks [5, 25, 26, 28].

In many compute-intensive applications, there is a large performance gap between GPUs and multicore CPUs [28]. Nowadays the peak performance of a modern and powerful GPU is approximately seven times the peak performance of a corresponding CPU in single precision and close to five times in double precision [5, 21]. This is mainly due to two reasons:

- While CPUs are designed to minimize the execution latency of a single task (latency-oriented platforms), the design goal of GPUs is to maximize the total execution throughput of a large number of parallel tasks (they are massively parallel throughput-oriented computing platforms).
- The graphics chip memories are usually much faster than the corresponding CPU chip memories.

Moreover, since massively parallel GPUs are easily accessible in comparison with huge cluster-based parallel machines, GPU-based platforms have made it possible to increase the use and spread of the high performance computing in scientific and engineering environments.

GPU computing consists of using GPUs together with CPUs to accelerate the solution of compute-intensive science, engineering and enterprise problems. Since the numerical simulation based on Partial Differential Equations (PDEs) exhibits a lot of exploitable parallelism, there has been an increasing interest in the acceleration of these simulations by using GPU-based computer systems.

Initially, graphics-specific programming languages and interfaces [8, 9, 14, 16, 26, 27] were used to program GPUs. However, the use of these languages complicates the programming of GPUs in scientific applications, and the code obtained is difficult to understand and maintain. Later, NVIDIA developed the CUDA programming toolkit [21] which includes an extension of the C language which facilitates the programming of GPUs for general purpose applications [10] by preventing the programmer to deal with the graphics details of the GPU. Additionally, several languages and interfaces such as OpenCL [7, 12], OpenACC [24] or Thrust [4], have been developed and spread to make the GPU computing easier.

There is a widespread use of CUDA-based platforms to accelerate numerical solvers for PDEs [1–3, 6]. For this reason, these notes intend for making it easier the exploitation of CUDA-enabled platforms to accelerate PDE-based numerical simulations, by providing the suitable CUDA C programming foundations.

2 CUDA Introduction

The *CUDA (Compute Unified Device Architecture)* framework [21] is a hardware and software platform that makes it possible to exploit efficiently the potential of NVIDIA GPUs to accelerate the solution of many costly computational problems. This framework includes a unified architectural view of the GPU and a multithreaded programming model which uses an extension to the C programming language (called CUDA C) to implement general-purpose applications on NVIDIA GPUs. Other languages are also supported [21] for CUDA programming (C++, Fortran, Java, etc.).

According to the CUDA framework, a GPU is viewed as a computing device which works as a coprocessor for the main CPU (host). Both the CPU and the GPU maintain their own Dynamic Random Access Memory (DRAM) (see Fig. 2) and it is possible to copy data from CPU memory to GPU memory and vice versa.

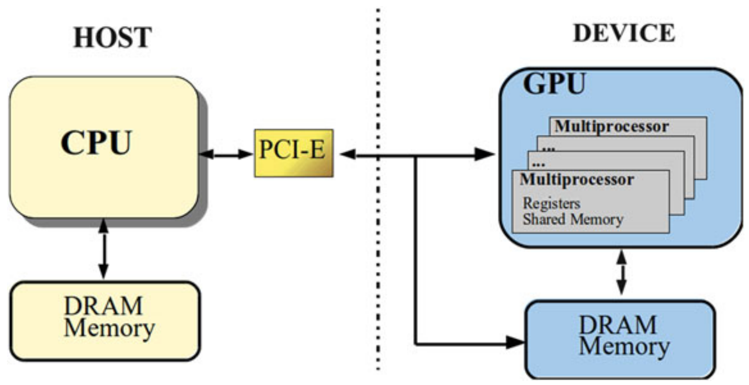

Fig. 2 GPU system (device) as a coprocessor of the CPU system (host)

3 Brief Introduction to the CUDA Hardware Model

In the CUDA framework, we consider that a CUDA enabled GPU is formed by N multiprocessors (N depends on the particular GPU), each one having M processors or cores (see Fig. 4) where the value of M also depends on the specific GPU architecture (it can be 8, 32, 192, …). At any clock cycle, each core of the multiprocessor executes the same instruction, but operates on different data. The multiprocessors of the GPU are specialized in the parallel execution of multiple CUDA threads. A *CUDA thread* represents a sequential computational task which executes an instance of a function. Each CUDA thread is executed logically in parallel with respect to other CUDA threads (associated to the same function but operating on different data) on the cores of a GPU multiprocessor (see Fig. 3).

3.1 CUDA Memory Model

A CUDA thread that runs on a multiprocessor of the GPU has access to the following memory spaces (see Fig. 4):

- **On-chip memories**:
  - *Registers*: Each thread has its own readable and writeable registers. Each GPU model has a particular number of registers per multiprocessor, which are split and assigned to the threads that run concurrently on that multiprocessor.
  - *Shared memory*: Each multiprocessor includes a small memory (between 16 and 48 KB) called *shared memory*. This memory is readable and writeable

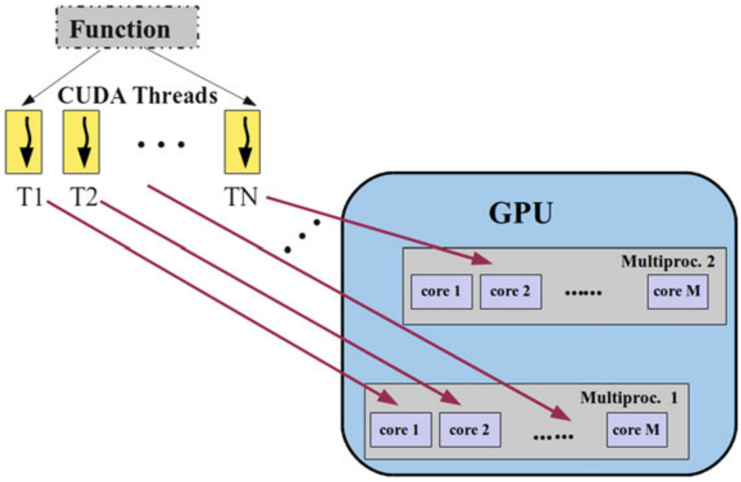

Fig. 3 Execution of multiple CUDA Threads (associated with the same function)

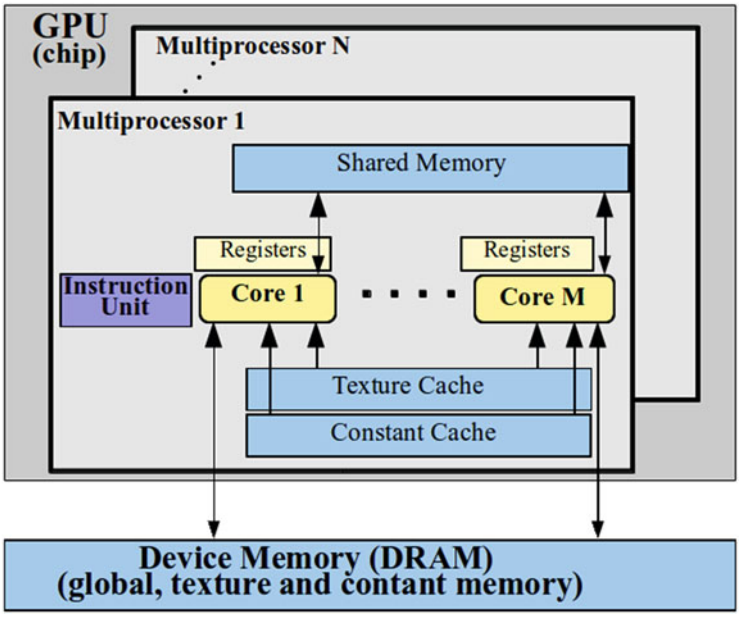

Fig. 4 CUDA hardware model

only from the CUDA threads running on the particular multiprocessor of the GPU and is much faster than global memory.

- **Off-chip memories** (in DRAM memory): These spaces are not resident in the GPU chip and they are shared by all CUDA threads which are being executed on the GPU (on several multiprocessors).
  - *Global memory*: This memory is readable and writeable from CPU and GPU. It is slow in comparison with the shared memory due to its high latency (500 times slower approximately).
  - *Constant memory*: It is readable from GPU and writeable from CPU. It is cached, making it faster than global memory if the data is in cache.
  - *Texture memory*: It is readable from GPU and writeable from CPU. Under several circumstances, the use of the texture memory can provide performance benefits because it is cached (there is a small texture cache memory for each multiprocessor) and optimized for 2D spatial locality. Texture cache size varies between 6 and 8 KB per multiprocessor.

4 CUDA Programming Model

4.1 CUDA Kernels and Threads

In order to specify the function to be executed by each thread on the GPU, CUDA C allows the programmer to define special C functions, called CUDA *kernels*. A CUDA kernel function is called from the CPU and is executed *N* times on the GPU by an array of *N* CUDA threads (see Fig. 5).

CUDA Threads are extremely light because they exhibit a very fast creation and context switching. As a consequence, it is recommendable to use fine-grain parallel decompositions (running very many thousands of CUDA threads) to obtain high efficiency.

Every thread executes the same code (the kernel function) but the specific action (and the data subdomain which is processed) to be performed depends on the thread identifier. The thread identifier can be obtained from the built-in variable `threadIdx` and is used to compute memory addresses and take control decisions (see Fig. 5).

Listing 1 shows a naive CUDA program which includes the declaration of a straightforward kernel and the corresponding kernel launch statement. The kernel (`VecAdd`) is used to add two vectors (with *N* floating point elements), `A` and `B`, obtaining the result in a vector `C`. As can be seen, a kernel function is declared using the modifier `__global__` and it must return `void` type. We can see that each thread of the kernel computes one element of the output vector `C`. In order to launch a kernel, the programmer must specify several parameters which determine, in particular, the number of CUDA threads which are to be created to execute it. In this example, the kernel invocation code indicates that an array of *N* threads (one for each element of the vectors) is launched on the GPU. The number and organization of the threads used to execute a kernel is given by several parameters between <<< and >>> as we will see later.

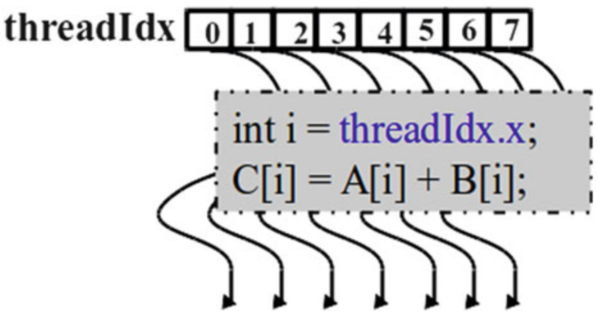

Fig. 5 Thread array for a naive CUDA kernel

```
...
__global__ void VecAdd(float* A, float* B, float* C)
{ int i = threadIdx.x;
   C[i] = A[i] + B[i]; }
...
int main()
 {
   ...
   // Kernel call with N Threads
   VecAdd <<<1, N>>> (A, B, C);
 }
```

Listing 1 Outline of a naive one-block CUDA program to add two vectors

4.2 *CUDA Thread Organization: Grids and Blocks*

The set of CUDA threads which are created in a kernel launch is organized forming a grid of thread blocks that run logically in parallel [21].

4.2.1 CUDA Thread Blocks

Each thread is identified with a (1D, 2D or 3D) thread Index (stored in the predefined variable `threadIdx`) in a *thread block*. A CUDA block is the unit used to map threads to multiprocessors. Each thread block runs on a single multiprocessor of the GPU.

Each multiprocessor can process a maximum number of blocks at a particular moment in time, and each block can include a maximum number of threads. It depends on the so-called *Compute Capability* of the GPU. The predefined variable `blockDim`, with type `dim3`, stores the block dimensions for a kernel (see Fig. 6). The `dim3` type is a `struct` with 3 unsigned integer fields (`x`, `y` and `z`).

The threads of a block can communicate among themselves using the shared memory of the multiprocessors assigned to that block.

As can be seen in the naive code of Listing 1, that kernel is designed to launch only one thread block. Consequently the size of the vectors (N) can not be greater than the maximum number of threads per block (which depends on the particular GPU architecture).

4.2.2 Grid of Blocks

As we have seen, several CUDA threads which will be executed on the same multiprocessor, are organized forming a (1D, 2D or 3D) matrix of threads called block. In a similar way, all the blocks which are created in a kernel launch are

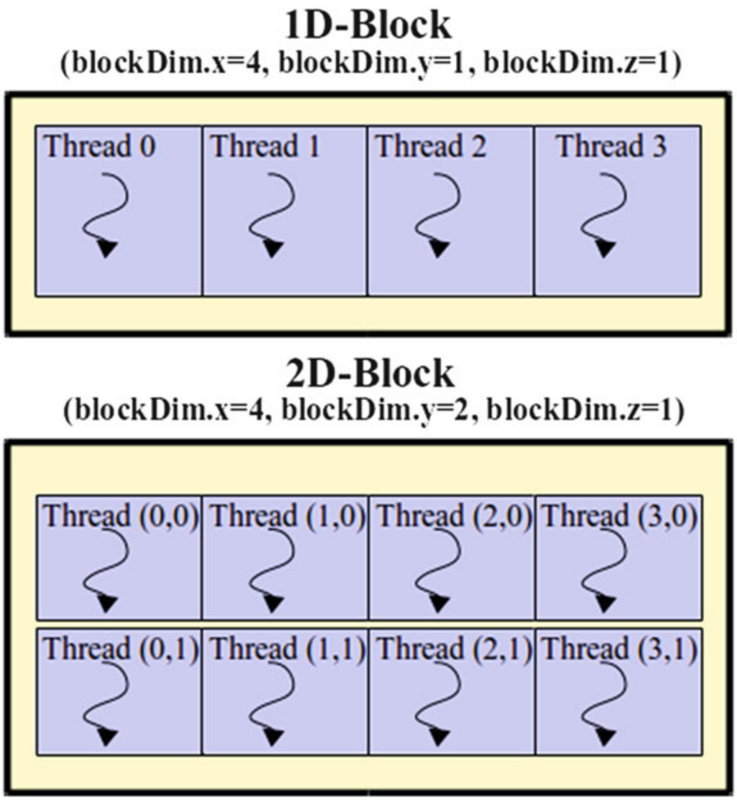

Fig. 6 Structure of 1D and 2D blocks of CUDA threads

organized as a (1D, 2D or 3D) matrix called *grid* (see Fig. 7). In general, the grid is a 3D array of blocks but the programmer can choose to use fewer dimensions

The number of threads per block and the number of blocks per grid used to launch a kernel are specified with the following syntax:

```
<<<   blocks_per_grid, threads_per_block >>>
```

The first parameter gives the number of thread blocks in the grid. The second gives the number of threads in each thread block. In the program of Listing 1, we run one grid with only one block of N threads. This program would only exploit one multiprocessor of the GPU. A more practical CUDA program to add vectors is given in Listing 2, where a one-dimensional grid with $\lceil \frac{N}{256} \rceil$ one-dimensional blocks with 256 threads is used (see Fig. 8).

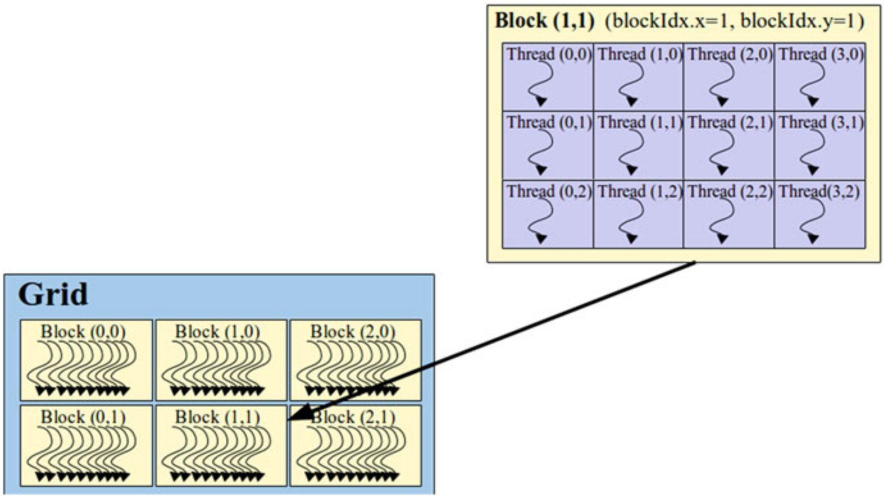

Fig. 7 Grid of thread blocks

```
#define BSIZE 256
__global__ void VecAdd(float* A, float* B, float* C, int N)
   { int i = blockIdx.x * blockDim.x + threadIdx.x;
     if (i<N)   C[i] = A[i] + B[i];
   }
...
int main()
 {
  ...
  // Kernel call with N Threads using 256 threads 1D-blocks
  VecAdd <<<ceil((float)N/BSIZE), BSIZE>>> (A,B,C,N);
  ...
 }
```

Listing 2 A more general CUDA program to add vectors

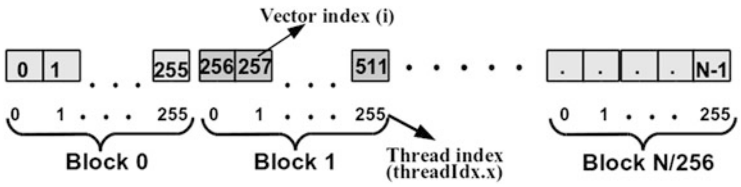

Fig. 8 Mapping of a vector to one-dimensional blocks

In order to identify several parameters of a particular thread (to distinguish it among the others), programmers can use the following predefined variables within the kernel function:

- `uint3 blockIdx`: identifies the coordinates of a particular block in the grid. The `uint3` type has the same structure as `dim3` (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`).
- `dim3 blockDim`: identifies the dimensions of the block.
- `dim3 gridDim`: maintains the grid dimensions.

Using these spatial indices (including `threadIdx`), the programmer can specify what particular data subdomain will be operated by each CUDA thread. In Listing 2, these indices are used to identify what elements of the input and output vectors are processed by a particular thread. Since not all threads will compute elements of the output vector, a conditional `if` statement is needed to test if the global index values of a thread are within the valid range (if $i \geq N$ nothing is computed).

The grid and block dimensions for a kernel must be chosen by the programmer depending on the nature of the data in order to maximize efficiency. For example, to add two 2D matrices of `float` elements, it can be convenient to use a 2D grid which consists of 2D blocks of threads, where each thread computes one element of the output matrix. Figure 9 shows this type of 2D mapping to compute the addition of 45×45 matrices. In this figure, it is assumed that we are using 16×16 2D-blocks (`blockDim.x` = `blockDim.y` = 16, `blockDim.z` = 1). As can be seen, we need a 3×3 grid of blocks (`gridDim.x` = `gridDim.y` = 3, `gridDim.z` = 1). Note that we have three extra threads in the x and y directions.

Listing 3 shows an outline of a CUDA program to add $N \times N$ matrices. In the `MatAdd` kernel, we initially compute the position of the current thread in the horizontal and vertical directions, using the built-in variables `blockIdx` and `threadIdx` (see Fig. 9). These position values (`i` and `j`) are used to derive the global 1D index for the input and output matrices. This 1D index is used to compute the corresponding element of matrix `C` if the previously computed position values (`i` and `j`) of the thread are within the valid range.

4.3 Transparent Scalability

A CUDA thread block can be executed independently with respect the rest of CUDA blocks of a kernel. Therefore the execution of several blocks can follow any relative execution order. Taking into account that the thread blocks are automatically mapped onto multiprocessors by the CUDA runtime system, this flexibility in the execution order enables the automatic adaptation of a kernel execution to the number of multiprocessors of the available GPU (see Fig. 10). This is called *transparent scalability*, because the blocks are mapped to the specific number of multiprocessors of a particular GPU in an efficient way without any intervention from the user.

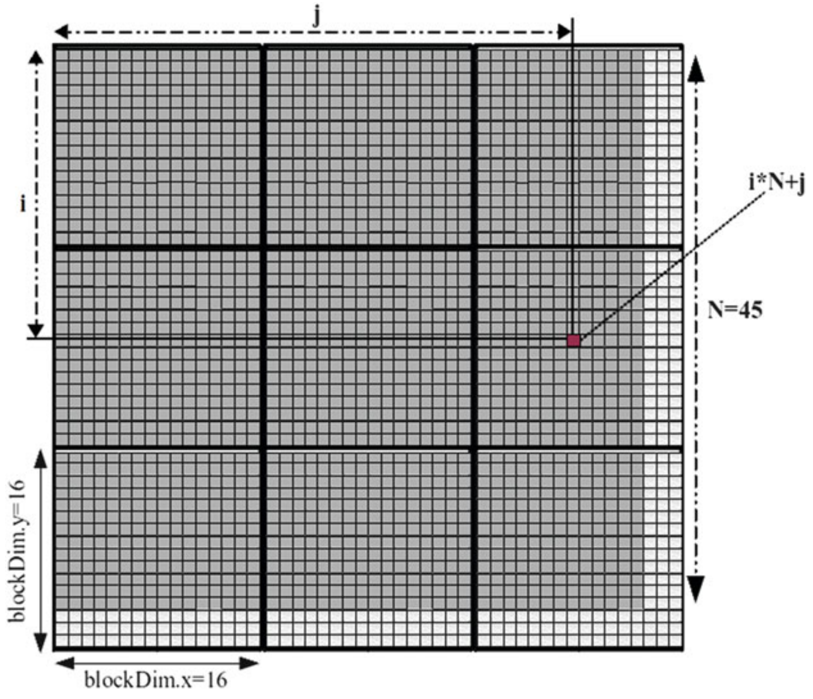

Fig. 9 Mapping of a 2D grid of 16 × 16 blocks to a 45 × 45 matrix

4.4 CUDA Program Structure

A CUDA C program must include the code for the CPU which includes the calls to several CUDA kernels and the specification of these kernel functions using the suitable C extensions (see Fig. 11). A kernel can only start on a GPU when the previous CUDA function call has returned.

The qualifiers for functions which run on GPU are the following:

- The qualifier `__global__` introduces a function (a kernel) which runs on GPU (device) but is called from the host:

```
__global__ void Kernel_name (...) {.....}
```

```
...
__global__ void MatAdd (float *A, float *B, float * C, int N)
     {
      // Compute row index in A, B, and C
      int j = blockIdx.x * blockDim.x + threadIdx.x;
      // Compute column index in A, B, and C
      int i = blockIdx.y * blockDim.y + threadIdx.y;
      // Compute global 1D index in A, B, and C
      int index=i*N+j;
      // Thread computes C element if within valid range
      if (i < N && j < N)
         C[index] = A[index] + B[index];
     }

  int main()
     {  .......
      // Kernel Launch code
      dim3 threadsPerBlock (16, 16);
      dim3 numBlocks( ceil ((float)(N)/threadsPerBlock.x),
                  ceil ((float)(N)/threadsPerBlock.y) );
      MatAdd <<<numBlocks, threadsPerBlock>>> (A, B, C, N);
      }
```

Listing 3 CUDA program to add matrices

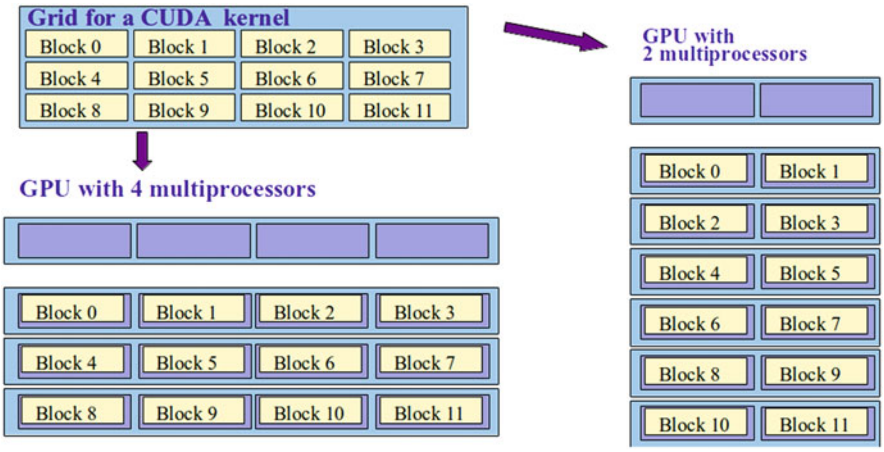

Fig. 10 Transparent scalability for CUDA programs

- The qualifier `__device__` introduces a function which runs on GPU and is called from the GPU (device):

```
__device__ int Function_name (...) {.....}
```

These device functions are invoked in the kernel function.

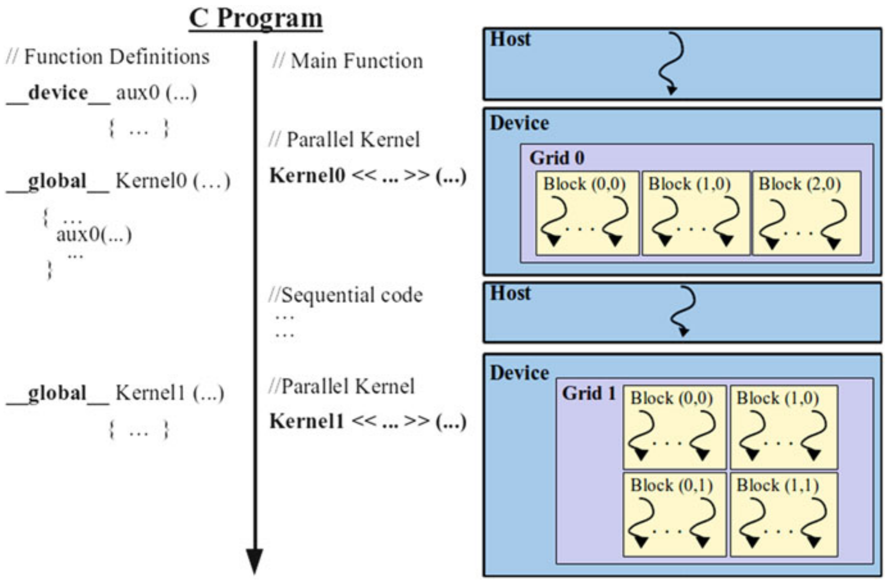

Fig. 11 Structure of a CUDA program

4.5 Memory Organization and Variable Declarations

As we introduced in Sect. 3.1, CUDA supports several types of memory. Each memory type has usage constraints and can be used by programmers declaring suitably the variables in the CUDA program. The adequate use of the available memory types is very important to obtain a good performance in a kernel execution. Each type of memory has different characteristics that we can denote by:

- **Functionality**: it is the intended use of the memory space.
- **Size**: it is measured in number of Kbytes.
- **Average access speed**: it indicates the global efficiency in accessing data which are allocated in the particular memory space.
- **Variable declaration**: it specifies how we must declare the variables which are resident in that memory space.
- **Scope**: it identifies the group of threads which can access a variable which is resident in that memory space.
- **Lifetime**: it denotes the phase of the program execution where the variable (resident in that memory space) is available for use.

Now we present the properties of the multiple memory spaces where CUDA threads can access data during the execution of a CUDA program (see Fig. 12):

- **Registers**: This memory space is used to hold variables which are frequently accessed by each thread in a kernel function. Each CUDA thread has its own

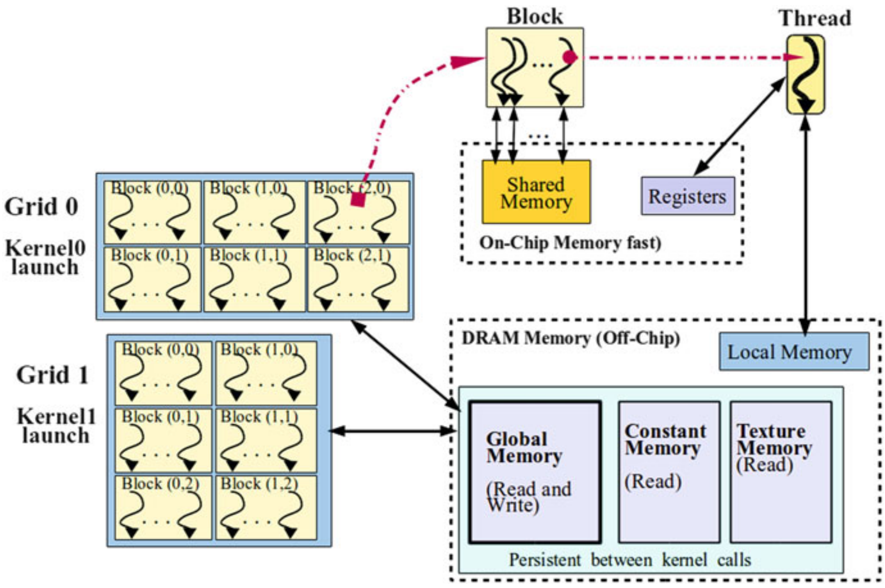

Fig. 12 Memory organization in a CUDA program

private registers. The speed access to registers is extremely high. While the capacity of the register storage is not exceeded, all scalar variables (not arrays) declared (without qualifiers) in kernel and device functions are placed into registers. The scope of these variables is the single owner thread (when the thread finishes its execution, these variables are also terminated). The number of 32-bit registers per thread and multiprocessors for several modern GPU architectures is shown in Table 1.

- **Local memory**: Array variables declared (without qualifiers) in a kernel and several scalar variables (also declared without qualifiers) which exceed the limited capacity of the register storage are not stored in registers. These variables have the same scope and lifetime as register variables but they are not stored into the registers (they are usually stored in DRAM device memory). Therefore, the speed access to local memory could be very low and this can generate long access delays.
- **Shared memory**: Shared memory is part of the memory space which resides on the processor chip. This implies that the access speed is also very high (in comparison with the access to variables resident in global memory) but lower than accessing directly to registers. The variables resident in shared memory (shared memory variables) are accessible, to read and write, for all the threads within the same block (the threads of the block share that space memory) and only last for the thread block activity period. They provide an excellent means by which threads within a block can communicate and collaborate efficiently on computations.

Table 1 CUDA Compute Capability of Fermi, Kepler and Maxwell GPUs

	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110	Maxwell GM107
Compute capability	2.0	2.1	3.0	3.5	5.0
Threads/warp	32	32	32	32	32
Max warps/multiprocessor	48	48	64	64	64
Max threads/multiprocessor	1536	1536	2048	2048	2048
Max blocks/multiprocessor	8	8	16	16	16
32-bit registers/multiprocessor	32,768	32,768	65,536	65,536	65,536
Max numb. registers/thread	63	63	63	255	255
Max numb. threads/block	1024	1024	1024	1024	1024
Shared memory config.	16K	16K	16K	16K	16K
	48K	48K	48K	48K	48K
			32K	32K	32K

In order to declare a shared memory variable in a kernel or device function, we use the qualifier `__shared__`. Hence, for example, the following code can be used to declare a shared memory vector of 32 `float` values in a kernel code:

```
__shared__ float vector[32];
```

- **Global memory**: Variables in global memory can be written and read by the host by calling API functions as we will see in Sect. 5.4. These variables are located off the processor chip, into the so-called device DRAM memory (as the local memory variables). Since the DRAM technology is used to implement this memory, the speed access is much lower than in on-chip memories (registers and shared memory).

  The lifetime of a variable resident in global memory is the duration of the entire CUDA application (it is shared and available for all kernels in the application).

  The `__device__` qualifier is used to declare device variables which are resident in global memory. These variables are accessible from all threads through the entire CUDA application.

```
__device__ float Global_A[32];
```

Global memory variables can be used to implement the thread block collaboration in a kernel call, but the main goal of these variables is providing a way to communicate data between different CUDA grids associated to different kernel calls (see Fig. 12).

We can define pointers to global memory data in CUDA. Thus, the programmer can declare a pointer variable in a host function (for example, in the `main` function) and then to use the function `cudaMalloc` to allocate global device memory for that pointer. For example, to declare and allocate a pointer to a vector of 1024 integers in global memory, we can write:

```
int numbytes = 1024*sizeof(int);
int *a_d;
cudaMalloc( (void**)&a_d, numbytes );
```

This pointer can be passed to a kernel function as a parameter. For example, the parameters `A`, `B` and `C` of the kernels shown in Listings 1, 2 and 3 are samples of this class of pointers. Listing 4 shows the declaration, allocation, usage in a kernel and freeing of several pointers to global memory data.

```
__global__ void VecAdd(float* A, float* B, float* C,int N){ ...
     }
...
int main()
{ int N = ...; int size = N * sizeof(float);
float* h_A = (float*) malloc(size); float* h_B = (float*)
    malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);
float* d_A; float* d_B;float* d_C; cudaMalloc((void**)&d_C,
    size);
cudaMalloc((void**)&d_A, size); cudaMalloc((void**)&d_B, size);

// Step 1
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

// Step 2
VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B,
    d_C, N);

// Step 3
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```

Listing 4 CUDA program with host-device data transfers

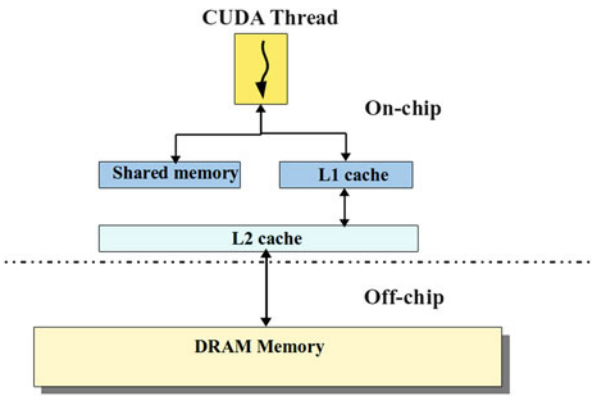

Fig. 13 Memory hierarchy in modern CUDA-enabled GPUs

- **Constant memory**: The constant variables are global variable declarations which should be outside any function in the source file. The `__constant__` qualifier is used to declare device variables in constant memory. The following code declares a constant vector of 64 `float` values:

```
__constant__ float A[64];
```

  These variables are accessible from all threads in a CUDA application but only to be read. They last for the entire CUDA application. These variables are also stored in the device DRAM memory but they are cached (using a small cache memory) and this makes it possible to obtain high speed access if the patterns to read the data are suitable.

  These variables are used to provide input values (they will not change during the execution) to a kernel function.

It is important to note that in modern CUDA-enabled GPUs, there exists an on-chip space memory that can be partially dedicated to cache memory for each kernel call (see Fig. 13). This cache memory might reduce the global memory access cost and avoids the explicit control of the programmer [17]. Moreover, the programmer can specify the amount of memory which is allocated to L1 cache and shared memory.

4.6 CUDA Thread Scheduling

Each active block resulting from a kernel launch is divided into warps to be executed on the assigned multiprocessor. A *warp* is a group of 32 threads (see Table 1) with consecutive indices (the thread ordering depends on `ThreadIdx`) that run physically in parallel on a multiprocessor. All threads in a warp execute the same

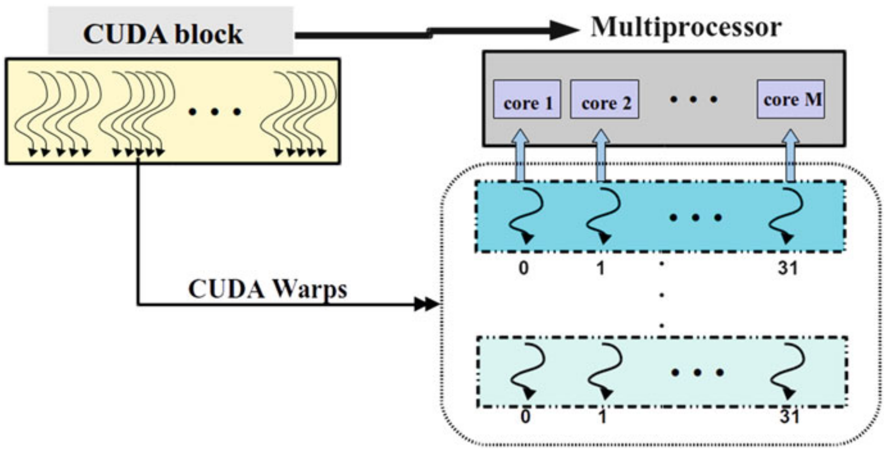

Fig. 14 Execution of a thread block by generating several warps

instruction of the kernel simultaneously, following the *Single Instruction Multiple Data* (SIMD) execution model (Fig. 14).

When a block is assigned to a multiprocessor of the GPU, the block is split into warps and periodically a scheduler of the multiprocessor switches from one warp to another (without penalties). This allows to hide the high latency when accessing the GPU device memory, since some threads can continue their execution while other threads are waiting. Therefore, the definition of a high number of threads when launching a kernel improves the efficiency in the kernel execution because makes it possible to hide the latency.

The decomposition of a thread block into warps is based on thread indices (the `threadIdx` field values). When only `threadIdx.x` is used (1D blocks), the threads of a warp and its ordering are imposed by the values of `threadIdx.x`. So, for example, for the CUDA program of Listing 2 (the block size is 256 threads), the arrangement of the threads would be (T_i denotes the thread with `threadIdx.x` $= i$):

First warp:	$T_0, \ldots\ldots.., T_{31}.$
Second warp:	$T_{32}, \ldots\ldots., T_{63}.$
....	....
....	....
Last warp (8th):	$T_{224}, \ldots\ldots, T_{255}.$

When the block size is not a multiple of 32, the last warp will be completed with additional threads. For that reason it is usual to define a block size that is a multiple of 32.

For a 2D or 3D block, the dimensions are mapped to a linear 1D arrangement in order to decompose the block into warps. This linear order is obtained by advancing forward firstly the x-dimension, secondly the y-dimension and finally the z-dimension. For instance, in Listing 3, where 2D 16×16 blocks are used, we would have the following arrangement ($T_{i,j}$ denotes the thread with `threadIdx.x` $= i$ and `threadIdx.y` $= j$):

First warp:	$T_{0,0},\ T_{1,0}, \ldots\ldots.,\ T_{15,0},\ T_{0,1},\ T_{1,1}, \ldots\ldots.,\ T_{15,1}.$
Second warp:	$T_{0,2},\ T_{1,2}, \ldots\ldots.,\ T_{15,2},\ T_{0,3},\ T_{1,3}, \ldots\ldots.,\ T_{15,3}.$
....	....
....	....
Last warp (8th):	$T_{0,14},\ T_{1,14}, \ldots\ldots.,\ T_{15,14},\ T_{0,15},\ T_{1,15}, \ldots\ldots.,\ T_{15,15}.$

We can check that the mapping of threads to elements of the input and output matrices in Listing 3 implies that the threads of a warp access to matrix elements with are located contiguously in the global memory. This way of mapping threads to data is very important to obtain a good performance because makes it possible to take advantage of the on-chip cache for the global memory space which is available in modern GPUs.

4.7 Block Thread Synchronization

Although there are no synchronization constraints between the CUDA blocks of a kernel, CUDA provides a mechanism to coordinate the execution of the threads in the same block, provoking a barrier-type synchronization. The following function

```
void __syncthreads();
```

can be used to synchronize all the threads in the same block of a kernel launch. It establishes a synchronization barrier for all the threads in a block. When this function is called in a particular point of a kernel function, all threads in each generated block will wait until the rest of threads also reaches the same execution point in the kernel.

It is used to prevent the threads from accessing variables before these have been written by other threads, as will be seen later. This function should only be called in a conditional code when the condition has the same evaluation for all threads in the block.

5 CUDA C/C++ Programming Interface

CUDA C/C++ is a minimal extension of C to define kernels as C++ functions. In order to program with a high abstraction level in CUDA C, one must use the CUDA runtime API [19], which provides functions to perform the following tasks:

- To query the properties of multiple devices.
- To allocate and free device memory.
- To transfer data between host memory and device memory.

The CUDA Runtime API requires the usage of the compiler driver **nvcc**.

It is also possible to develop CUDA programs using the CUDA driver API [18] but it is harder to program with this interface.

5.1 Querying GPU Properties

The CUDA Runtime API provides functions to find out the available resources and capabilities of the underlying GPU hardware. The CUDA runtime system numbers all the available GPUs in the system from 0 to `num_GPUs` − 1.

The `cudaGetDeviceCount` function makes it possible to know the number of available CUDA GPUs in a particular computer system. Thus, for instance, to obtain the number of GPUs into an integer variable, we write:

```
int num_GPUs;
cudaGetDeviceCount ( &num_GPUs );
```

In order to know the characteristics of a particular GPU, CUDA Runtime API provides the built-in type `cudaDeviceProp` which is a C structure with fields representing different properties of a CUDA GPU. Using this type, the function `cudaGetDeviceProperties` can be used to query the characteristics of a GPU in the system. For example, this code piece:

```
cudaDeviceProp GPU_prop; int dev_id=...;
cudaGetDeviceProperties ( &GPU_prop, dev_id);
```

returns the properties of the GPU which has the number `dev_id` in the `GPU_prop` variable. Some of the properties we can query in `GPU_prop` are:

- `GPU_prop.totalGlobalMem`: Global memory available on device in bytes.
- `GPU_prop.sharedMemPerMultiprocessor`: Shared memory available per multiprocessor in bytes.
- `GPU_prop.maxThreadsPerBlock`: maximal number of threads allowed in a block (512, 1024,…).

- `GPU_prop.multiProcessorCount`: number of multiprocessors.
- `GPU_prop.maxThreadsDim[dim_id]`: maximal number of threads allowed along dimension `dim_id` of a block.
- `dev_prop.maxGridSize [dim_id]`: maximal number of blocks allowed along dimension `dim_id` of a grid.
- `dev_prop.warpSize`: Number of threads in a warp (unit of thread scheduling in multiprocessors).

The *CUDA Compute Capability* (CCC) indicates the architecture and features of the particular GPU. It is defined by a major revision number and a minor revision number `X.X`. Devices with the same major revision number denote the same core architecture. Table 1 shows some features of modern GPU architectures (Fermi, Kepler and first generation of Maxwell architecture) with different CCC.

5.2 Reporting Errors

All CUDA API calls return an error code (with built-in type `cudaError_t`). This error can be caused by:

- Error in the API call itself. For example:

```
cudaError_t err; err=cudaGetDevice(&devID);
if (err!=cudaSuccess) {cout<<"ERROR!!"<<endl;}
```

- Error in an earlier asynchronous operation (for example in a kernel call):

```
kernel<<<blocksPerGrid,threadsPerBlock >>>(a,b,c);
err = cudaGetLastError();
if (err != cudaSuccess) {...}
```

We can get a string which describes the particular error by using the following code piece:

```
char *cudaGetErrorString(cudaError_t)
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

5.3 Compiling CUDA Code

The CUDA compilation includes two stages [22]:

- *Independent on GPU* (Virtual): It generates a code written in a lower level intermediate language called PTX code (Parallel Thread eXecution) [23].
- *Physical*: It generates object code for a particular GPU.

The CUDA Toolkit includes a compilation tool called `nvcc`, which is the compiler driver. It performs the calls to the required compilers and tools: cudac, g++, etc. This tool decouples CPU and GPU code. The CPU C code must be compiled by another tool. PTX object code must be adapted to a particular GPU architecture and CUDA Capability.

In order to execute code on devices of an specific CUDA Compute Capability, it is necessary to load code that is compatible with this CCC. Which PTX and binary code gets embedded in a CUDA C application is controlled by the `-arch` and `-code` compiler options or, more concisely, by using the -gencode compiler option.

- `-arch=<compute_xy>`: it generates PTX code for capability `x.y`.
- `-code=<sm\_xy>`: it generate binary code for capability `x.y`, by default same as `-arch`.

Example

```
nvcc filename.cu -gencode arch=compute_35,
                     code=sm_35 -o filename
```

This command generates binary and PTX code compatible with CCC 3.5.

Host code is generated to automatically select at runtime the most appropriate code.

In order to ensure 64-bit or 32-bit compatibility, we can use the following switches:

- `m64`: compile device code in 64-bit mode.
- `m32`: compile device code in 32-bit mode.

5.4 Device Memory Management

CPU and GPU have separated memory spaces. To enable the data management and the communication, there are runtime functions to (Fig. 15):

- Allocate and free DRAM device global memory.
- Set a value in several positions of device global memory.
- Transfer data between global (device) memory and host memory.

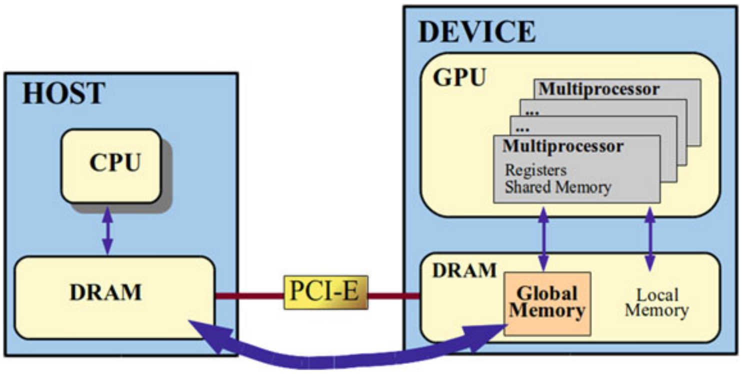

Fig. 15 Host-device memory transfer

We can use the function `cudaMalloc` to allocate memory device with the following syntax:

```
cudaMalloc(void ** pointer, size_t numbytes)
```

The `size_t` type is an unsigned integer type used to represent the sizes of data objects as a number of bytes.

We use `cudaFree` to free memory device:

```
cudaFree(void* pointer)
```

The function `cudaMemset` makes it possible to set a particular `value` in linear device memory:

```
cudaMemset(void * pointer, int value, size_t nbytes)
```

Example Reset all elements of a vector with 1024 integers

```
int numbytes = 1024*sizeof(int);
int *a_d;
cudaMalloc( (void**)&a_d, numbytes );
cudaMemset( a_d, 0, nbytes);
    ...
    ...
cudaFree(a_d); // Free a_d when it is not needed
```

The function `CudaMemcpy` is used to transfer data between global memory and host memory:

```
CudaMemcpy (void *dest, void *source, size_t nbytes,
                    enum cudaMemcpyKind direction);
```

This function copies `nbytes` bytes from the memory area pointed to by `source` to the memory area pointed to by `dest`, where the location (host or device) for `dest` and `source` is given by `direction`:

- `cudaMemcpyHostToDevice`: From host to device.
- `cudaMemcpyDeviceToHost`: From device to host.
- `cudaMemcpyDeviceToDevice`: Between two global memory positions in device.

The invocation of this function blocks the CPU thread and only returns when the data copy has been completed. The data transfer does not begin if any previous CUDA call have not returned.

The program in Listing 4 shows the CUDA calls which are necessary in order to execute the vector addition kernel. After initializing vectors *A* and *B* (stored in a memory area pointed to by `h_A` and `h_B`), the area for the input and output vectors in device memory is allocated using `cudaMalloc` (pointers `d_A`, `d_B` and `d_C`). Then input vectors are transferred to the memory area pointed to by `d_A` and `d_B` before the kernel launch. After the kernel execution, the resultant value of *C*, obtained in device global memory (pointed to by `d_C`), must be copied to the host memory. Finally, the device memory area pointed to by `d_A`, `d_B` and `d_C` is freed. Figure 16 illustrates the main steps of this program.

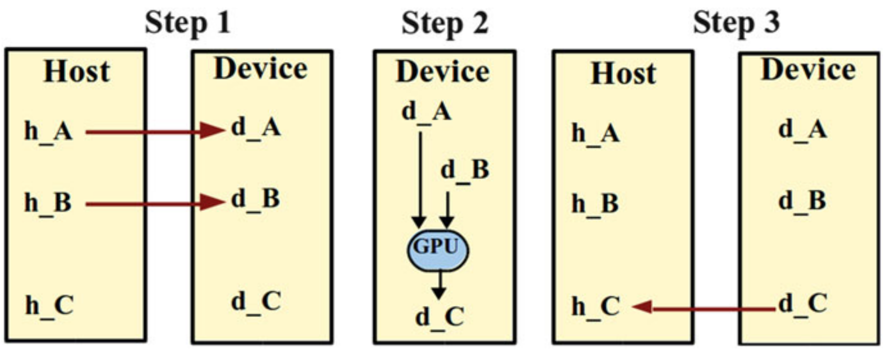

Fig. 16 Description of data transfers and kernel launch in a CUDA program

5.5 CUDA Samples

To assist in the development of CUDA software, the CUDA Samples [20] provide several examples with source code which can be useful to build a wide range of interesting applications.

6 Solving the 1D Linear Advection Equation in CUDA

6.1 1D Linear Advection

The 1D advection equation [15] is the simplest hyperbolic equation and can be written as:

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0, \tag{1}$$

where $\phi = \phi(x,t)$ is a real function with $x \in [a,b]$ and $t > 0$. The scalar $u \in \mathbb{R}$ represents the speed at which information propagates along the x-direction and is assumed to be constant. The initial state for this equation is specified as

$$\phi(x,0) = \phi_0(x),$$

and we assume periodic boundary conditions.

We can verify that the exact solution for this equation is given by

$$\phi(x,t) = \phi_0(x - ut).$$

6.2 The Lax-Friedrichs Method

We discretize the spatial and time domain using uniform grids which consist of:

- $n+1$ spatial grid points:

$$x_i, \quad i = 0,\dots,n, \text{ where } x_i = a + i\Delta x, \qquad \text{with } \Delta x = \frac{b-a}{n}.$$

- M time steps:

$$t^1, t^2, \dots, t^M \text{ with a constant } \Delta t, \quad (t^{k+1} = t^k + \Delta t).$$

Now, we can define the *Lax-Friedrichs* finite difference scheme to approximate the solution of equation (1) as follows (see [15]):

$$\phi_i^{k+1} = \frac{1}{2}\left(\phi_{i-1}^k + \phi_{i+1}^k - C(\phi_{i-1}^k - \phi_{i+1}^k)\right), \tag{2}$$

$$C = \frac{u\,\Delta t}{\Delta x}, \quad i = 0,\ldots,n, \quad k = 1,\ldots,M.$$

In the previous scheme $\phi(x_i, t^k)$ denotes ϕ_i^k.

The boundary conditions are imposed by defining:

$$\phi_{-1}^k = \phi_n^k, \qquad \phi_{n+1}^k = \phi_0^k.$$

This method is linearly stable and convergent under the usual CFL condition provided $|C| \le 1$ (see [15]).

6.3 A Sequential C Program to Implement the Method

Listing 5 shows a sequential C program to approximate the 1D linear advection equation using the Lax-Friedrichs scheme on a uniform grid. As we can see, at each time step, each data element of the output vector `phi_new` is computed from two neighbouring input elements of vector `phi`. Figure 17 shows graphically the data dependencies between the input and output vectors at each time step. Vector `phi` includes elements at the edges to store the ghost values needed to perform the stencil computation.

After each state vector update the pointers, `phi` and `phi_new`, must be swapped (using the `swap_pointers` function), because the state vector computed in a time step will be the previous state vector in the next time step.

6.4 A CUDA Linear Advection Solver

Since the calculation of all output elements of `phi_new` for a time step can be done in parallel (see Listing 5), we can assign a different thread to the computation of each element of `phi_new` at each time step. We will need to use at least $n+1$ threads and the body of each thread is given by a kernel function with four arguments:

- A pointer to array, `d_phi`, denoting the initial state vector (ϕ^k in Eq. (2)).
- A pointer to array, `d_phi_new`, denoting the new state vector ϕ^{k+1}.
- An integer `cu` representing the value of the Courant number ($\texttt{cu} = \frac{u\,\Delta t}{\Delta x}$).
- The integer value `n`, which denotes the value of n in Eq. (2).

The definition of this kernel function is given in Listing 6.

```
...
void swap_pointers(float * *a, float * *b)
  {float * tmp=*a;*a=*b;*b=tmp;}
...
int main(int argc, char* argv[]){
   // Define Delta x, Delta t and velocity (u)
  int n= ... ;float dx = ...;float dt=..; float u=...
  // Compute the Courant number
  float cu = u*dt/dx;
  //Declare the initial and final state vectors
  float * phi =new float[n+3]; float * phi_new=new float[n+3];
  ..
  Initialize(phi); // Set phi values at t=0
  // Time steps
  for(int k=1; k <= M; k++)
   {// Impose Homogeneous Neumann Boundary Conditions
    phi[index(-1)] =phi[index(n)];
    phi[index(n+1)]=phi[index(0)];

    //Lax-Friedrichs Update
    for(int i=0;i<=n;i++)
          phi_new[index(i)]=0.5*((phi[index(i+1)]+phi[index(i-1)])
                         -cu*(phi[index(i+1)]-phi[index(i-1)]));

    swap_pointers (&phi,&phi_new);
   }
...}
```

Listing 5 Sequential C program which implements the Lax-Friedrichs scheme

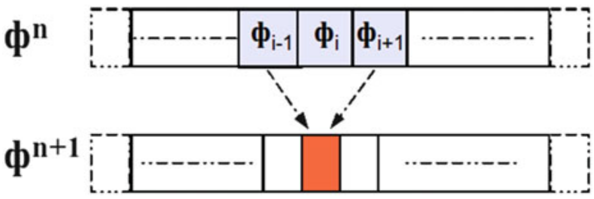

Fig. 17 Two-point stencil of the Lax-Friedrichs method

As can be seen in Listing 6, we follow a similar approach as the vector addition kernel (see Listing 2). All threads read two elements of the same input vector except for two threads which read an additional element to impose the boundary conditions. The work of threads which are not assigned to elements of `d_phi_new` is avoided (threads verifying that $i \geq n+2$). The same threads which have computed the values close to the edges of `d_phi_new` (positions `1` and `n+1`) impose the boundary conditions by assigning values on the edges of `d_phi_new`.

```
__global__ void FD_kernel1(float * d_phi,
                          float * d_phi_new,
                               float cu, int n)

 {int i=threadIdx.x+blockDim.x*blockIdx.x+1;

 // Lax-Friedichs Stencil
 if (i<n+2)
   d_phi_new[i]=0.5*((d_phi[i+1]+d_phi[i-1])
                     -cu*(d_phi[i+1]-d_phi[i-1]));

 // Impose Boundary Conditions
 if (i==1) d_phi_new[0]=d_phi_new[n+1];
 if (i==n+1) d_phi_new[n+2]=d_phi_new[1];}
```

Listing 6 CUDA kernel to apply the Lax-Friedrichs method stencil

Listing 7 shows a CUDA C program to implement the Lax-Friedrichs scheme using the previously introduced kernel (`FD_kernel1`). Before launching the kernel, the memory of the input and output state vectors (`d_phi` and `d_phi_new`) must be allocated on the device using calls to `cudaMalloc()`. Using `cudaMemcpy()`, the initial values of the state vector (computed by calling a initialization function) are copied to the device memory. Then, at each time step, we launch the `FD_kernel` from the host code by using blocks of 256 threads. After each kernel launch the memory device pointers, `d_phi` and `d_phi_new`, must be swapped. Finally, when the time integration loop has finished, the final state vector is copied from device memory to the host memory.

6.5 A Tiled CUDA Advection Solver

In the kernel of Listing 6, the ratio of floating-point calculations regarding with the global memory accesses is low. This prevents us from achieving high performance. One approach that could improve the performance consists of managing the shared memory which is much faster than global memory. This involves to replace global memory accesses with shared memory accesses (into each block). This approach frequently involves a reorganization of the code to enable the reuse of data which are placed in shared memory.

The typical approach to use shared memory in order to reduce the traffic to global memory consists of two steps:

1. The input data elements are organized in small pieces, called *tiles*.
2. Threads of a thread block collaborate to load a tile into shared memory before the threads use these elements.

```
#define BLOCKSIZE 256

... //FD_kernel function definition

int main(int argc, char* argv[]){
  int size=(n+3)*sizeof(float);
  float * d_phi=NULL; float * d_phi_new=NULL;
  cudaMalloc((void **) &d_phi, size);
  cudaMalloc((void **) &d_phi_new, size);

  Initialize(phi); // Set phi values at t=0

  cudaMemcpy(d_phi,phi,size,cudaMemcpyHostToDevice);
  // Time Steps
  for(int k=1; k <= M ;k++)
   {

    int blocksPerGrid =(int) ceil((float)(n+1)/BLOCKSIZE);

    // ********* CUDA Kernel Launch
        ************************************
    FD_kernel1<<<blocksPerGrid, BLOCKSIZE >>> (d_phi, d_phi_new,
         cu, n);

    swap_pointers (&d_phi,&d_phi_new);
   }

  cudaMemcpy(phi_GPU, d_phi, size, cudaMemcpyDeviceToHost);
  ...
}
```
Listing 7 CUDA program to implement the Lax-Friedrichs method

3. Once the threads of the same block have loaded its corresponding tile, they re-use the elements by accessing the shared memory.

In these so-called *tiled algorithms* [13], the size of the tiles must be chosen to fit into the shared memory (which is quite small).

In order to apply this approach to the Lax-Friedrichs kernel of Listing 6, we can load all input elements of `d_phi`, which are needed to compute all output elements of `d_phi_new` for a thread block, into the shared memory. The number of elements to be loaded must be equal to `BLOCKSIZE` + 2 because we need two additional elements at the edges to compute `BLOCKSIZE` elements of `d_phi_new`. Listing 8 shows the tiled kernel which implements this computation.

```
__global__ void FD_kernel2 (float * d_phi, float * d_phi_new,
                                    float cu, const int n)
  {
   int li=threadIdx.x+1; //local index in shared memory vector
   int gi= blockDim.x*blockIdx.x+threadIdx.x+1; // global
       memory index
   int lstart=0; // start index in the block (left value)
   int lend=BLOCKSIZE+1; // end index in the block (right value
       )

   __shared__ float s_phi[BLOCKSIZE + 2]; //shared mem. vector

   float result;

   // Load internal points of the tile in shared memory
   if (gi<n+2) s_phi[li] = d_phi[gi];

   // Load the halo points of the tile in shared memory
   if (threadIdx.x == 0) // First Thread (in the current block)
      s_phi[lstart]=d_phi[gi-1];

   if (threadIdx.x == BLOCKSIZE-1) // Last Thread
          if (gi>=n+1) // Last Block
             s_phi[(n+2)%BLOCKSIZE]=d_phi[n+2];
          else
             s_phi[lend]=d_phi[gi+1];


   __syncthreads();

  if (gi<n+2)
   {
    // Lax-Friedrichs Update
     result=0.5*((s_phi[li+1]+s_phi[li-1])
         -cu*(s_phi[li+1]-s_phi[li-1]));
     d_phi_new[gi]=result;
   }


  // Impose Boundary Conditions
  if (gi==1) d_phi_new[0]=d_phi_new[n+1];
  if (gi==n+1) d_phi_new[n+2]=d_phi_new[1];

  }
```

Listing 8 Lax-Friedrichs CUDA kernel using shared memory

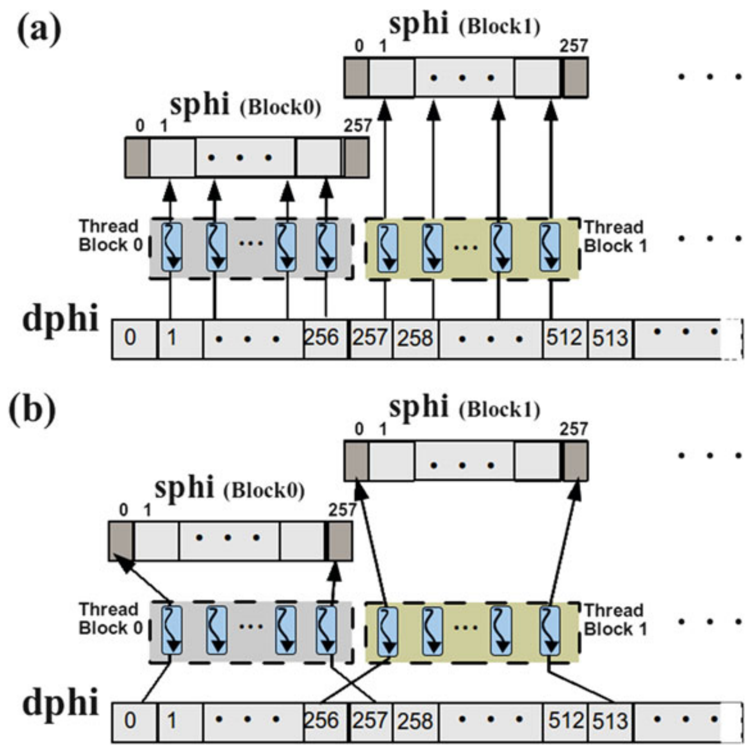

Fig. 18 Loading the tile in shared memory: (**a**) Loading internal points; (**b**) Loading the halo points

Initially, a shared memory array with `BLOCKSIZE` + 2 elements, `s_phi`, is declared to hold all elements to be loaded in order to compute the corresponding `BLOCKSIZE` elements of `d_phi_new`. Then the tile is loaded in shared memory in two phases (see Fig. 18):

(a) All threads in the block load in parallel the internal elements of the input tile (without taking into account the elements on the edges).
(b) The first and last threads in the block load the elements at the edges of the tile (the halo elements).

After loading the tile in shared memory, it is necessary to make sure that all threads in a block have finalized the load before using the tile to compute the output data elements. This is done by invoking the function `__syncthreads()`.

When the tile is completely loaded for the particular thread block, we can compute the output `BLOCKSIZE` elements of `d_phi_new` reading only elements of `s_phi`.

Finally, the boundary conditions are imposed in a similar way as in Listing 6.

Acknowledgements This work has been partially supported by FEDER and the Spanish and Andalusian research projects MTM2014-52056-P, MTM2012-38383-C02-01, P11-FQM8179 and P11-RNM7069.

References

1. de la Asunción, M., Castro, M.J., Fernández-Nieto, E.D., Mantas, J.M., Ortega, S., González, J.M.: Efficient GPU implementation of a two waves TVD-WAF method for the two-dimensional one layer shallow water system on structured meshes. Comput. Fluids **80**, 441–452 (2012)
2. de la Asunción, M., Mantas, J.M., Castro, M.J.: Programming CUDA-based GPUs to simulate two-layer shallow water flows. In: D'ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Ischia. Lecture Notes in Computer Science, vol. 6272, pp. 353–364. Springer (2010)
3. de la Asunción, M., Mantas, J.M., Castro, M.J.: Simulation of one-layer shallow water systems on multicore and CUDA architectures. J. Supercomput. **58**(2), 206–214 (2011)
4. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems Jade Edition, pp. 359–371. Morgan Kaufmann, Waltham (2011)
5. Brodtkorb, A.R., Hagen, T.R., SæTra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. J. Parallel Distrib. Comput. **73**(1), 4–13 (2013)
6. Castro, M.J., Ortega, S., de la Asunción, M., Mantas, J.M., Gallardo, J.M.: GPU computing for shallow water flow simulation based on finite volume schemes. Comptes Rendus Mécanique **339**(2–3), 165–184 (2011)
7. Fang, J., Varbanescu, A., Sips, H.: A comprehensive performance comparison of cuda and opencl. In: 2011 International Conference on Parallel Processing (ICPP 2011), Taipei, pp. 216–225 (2011)
8. Fernando, R.: GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison-Wesley, Boston (2004)
9. Fernando, R., Kilgard, M.J.: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley, Boston (2003)
10. Hubert, N.: GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, Boston (2007)
11. Jackson, M., Budruk, R., Winkles, J., Anderson, D.: PCI Express Technology 3.0. MindShare Press, Monument (2012)
12. Khronos OpenCL Working Group: The OpenCL Specification. https://www.khronos.org/opencl/ (2015)
13. Kirk, D., Wen-mei, H.: Programming Massively Parallel Processors. A Hands-on Approach, 2nd edn. Morgan Kaufmann, Waltham (2012)
14. Lastra, M., Mantas, J.M., Ureña, C., Castro, M.J., García, J.A.: Simulation of shallow-water systems using graphics processing units. Math. Comput. Simul. **80**(3), 598–618 (2009)
15. Leveque, R.: Finite Difference Methods for Ordinary and Partial Differential Equations. SIAM, Philadelphia (2007)
16. Matt, P., Randima, F.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, Upper Saddle River (2005)
17. NVIDIA: CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (2014)
18. NVIDIA: Cuda driver api. http://docs.nvidia.com/cuda/cuda-driver-api/index.html (2014)
19. NVIDIA: CUDA Runtime API. http://docs.nvidia.com/cuda/cuda-runtime-api/index.html (2014)
20. NVIDIA: CUDA Samples. http://docs.nvidia.com/cuda/cuda-samples (2014)
21. NVIDIA: CUDA Zone. http://www.nvidia.com/object/cuda_home.html (2014)

22. NVIDIA: NVIDIA CUDA Compiler Driver NVCC. http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (2014)
23. NVIDIA: Parallel tread execution isa 3.2. http://docs.nvidia.com/cuda/pdf/ptx_isa_3.2.pdf (2014)
24. OpenACC-Standard.org: The OpenACC Application Programming Interface, Version 1.0. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf (2011)
25. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proc. IEEE **96**(5), 879–899 (2008)
26. Rumpf, M., Strzodka, R.: Graphics processor units: new prospects for parallel computing. In: Bruaset, A.M., Tveito, A. (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers. Lecture Notes in Computational Science and Engineering, vol. 51, pp. 89–134. Springer, Berlin (2006)
27. Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1. Addison-Wesley Professional, Upper Saddle River (2007)
28. Ujaldon, M.: High performance computing and simulations on the GPU using CUDA. In: 2012 International Conference on High Performance Computing & Simulation (HPCS 2012), Madrid, pp. 1–7. Curran Associates (2012)