CUDA Course : Exercises

Pedro Alonso

Contents

Exercise 1: What do I have on my computer? **3**

Exercise 2: Transference speed between CPU and GPU **3**

Exercise 3: Data transference CPU-GPU **4**

Exercise 4: Timing with CUDA **4**

Exercise 5: A Simple Kernel: Matrix sum **5**

Exercise 6: Matrix transposition example **7**

Exercise 7: Matrix multiplication with and without shared memory **9**

Exercise 8: Matrix-vector product **10**

Exercise 9: Using `CUDA` libraries: `CUBLAS` **12**

Exercise 10: Matrix multiplication with streams **13**

Exercise 11: MEX files for GPU (Opcional) **14**

Exercise 1: What do I have on my computer?

The first exercise consists of figuring out which GPU(s) there is(are) in my computer and the features it has(have). For this purpose, we'll use an NVIDIA sample provided with the package. This tool is a simple program called `deviceQuery` that makes use of a library routine called `cudaGetDeviceProperties` that access to all the NVIDIA GPUs available in the host and fills a structure of type `cudaDeviceProp`. Its scheme is basically the following:

Listing 1: CUDA structure `cudaDeviceProp`

```
int deviceCount = 0;
for (dev = 0; dev < cudaGetDeviceCount(&deviceCount); ++dev) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    /* Show properties accesing members of deviceProp */
}
```

Each member of this C-structure contains a given information regarding the GPU.
Follow these steps:

1. Enter `./exercises/deviceQuery`.
2. Compile the program typing `make`.
3. Run the program: `./deviceQuery`.
4. After few seconds you'll see the information acquired by the tool. Analyze each item and link them in your mind with data learnt in the previous session. Pay attention to special items like: Capability numbers of the GPU version, amount of global memory, Mutiprocesors (SM) and cores, warp size, and maximum numbers of cores per block, dimension of blocks and grid.

Puntuación: 0.00
Entrega: De este ejercicio no se entrega nada.

Exercise 2: Transference speed between CPU and GPU

Other important information we need to know about our CPU-GPU environment is the data transference performance between both CPU and GPU at both directions. For this purpose we'll use another application provided with the CUDA programming tools: `bandwidthTest`.

1. Enter `./exercises/bandwidthTest`.
2. Compile the program typing `make`.
3. Run the program: `./bandwidthTest`.

Without arguments, this application shows bandwidth form CPU to GPU, from GPU to CPU and between GPU to GPU (from/to global memory) with "PINNED" memory. *Pinned memory* is allocated in host memory by means of routine `cudaHostAlloc` and this type of memory is page-locked and accessible to the device. This memory can be read or written with much higher bandwidth than pageable memory. To obtain bandwidth with pageable memory use argument `--memory=pageable` and compare.
This tool makes us to be aware about limitations of GPUs for our purposes. Also, shows a desirable bandwidth we should reach inside a GPU by our kernel.

Puntuación: 0.00
Entrega: De este ejercicio no se entrega nada.

Exercise 3: Data transference CPU-GPU

We're going to transfer data from the host to the device and viceversa. Follow the next steps where they are marked into the sample source code (`TransferenceExample`):

1. Allocate matrices `A` and `B` in the host with the regular function `malloc` or `calloc`. Matrices should be in single precision (`float`). For example using `A = (float*) malloc(m*n*sizeof(float));`
2. Fill matrix `A` with random generated data (use function `rand`) where each element is in the range $[-1.0, 1.0]$. For instance, one element of matrix `A` can be assigned a random value as `A(i,j) = ( 2.0f * (float) rand() / RAND_MAX ) - 1.0f;`.
3. Allocate memory for the two m-by-n matrices called `d_A` and `d_B` in the device memory. Use CUDA function `cudaMalloc`. For example, using `CUDA_SAFE_CALL( cudaMalloc( (void **) &d_A, m*n*sizeof(float) ) );`.
4. Copy host memory, that is, matrix `A` to the device memory into the corresponding matrix `d_A` using function `cudaMemcpy`. For instance, using `CUDA_SAFE_CALL( cudaMemcpy( d_A, A, m*n*sizeof(float), cudaMemcpyHostToDevice ) );`. The first argument of `cudaMemcpy` is always the destination, and the second one is the source, whatever the direction of the transfer you use. The last argument denotes this direction, which should be `cudaMemcpyDeviceToHost` for a Device-To-Host transference.
5. Copy device matrix `d_A` into the device matrix `d_B` using function `cudaMemcpy`.
6. Copy back the (resulting) data from device matrix `d_B` into the host memory matrix `B`.
7. Deallocate matrices on the device memory (`d_A` and `d_B`) with routine `cudaFree`.
8. Deallocate the three host matrices `A` and `B` with routine `free`.

Run the program and check for syntax or execution errors. Increase the amount of memory that you use. Add to you program a calculation for the total amount of bytes allocated into the device to avoid running out of memory.

Puntuación: 0.15
Entrega: De este ejercicio se entrega el código (solo el fichero fuente).

Exercise 4: Timing with CUDA

There are many ways to time and application, either that using CUDA or not. For the case of a CUDA application we can use CUDA *events*. This method allows to time a CUDA application easily and accurately. Follow the next steps:

1. Use the former example of data transference and add the following instructions:

```
cp TransferenceExample/TransferenceExample.cu TimingExample/
cd TimingExample/
```

2. Edit `TransferenceExample.cu` and declare two variables of type `cudaEvent_t` to record events, and a `float` variable to store the elapsed time:

```
cudaEvent_t start, stop;
float elapsedTime;
```

3. Use CUDA function `cudaEventCreate` to create the `start` event (`cudaEventCreate(&start);`).
4. Record on the `start` event (`cudaEventRecord(start,0);`) at the beginning of the code to be timed.
5. Similarly, create and record on event `stop` at the end of the code to be timed.
6. Call function `cudaEventSynchronize` on event `stop` (`cudaEventSynchronize(stop)`). This function waits until the completion of all device work preceding the most recent event recorded. This function is not necessary here because data transference between the host and the device is synchronous, but it is in fact necessary when timing calls to kernel since these calls are not synchronous.
7. The elapsed time between two CUDA events can be obtained with the following function:

   ```
   cudaEventElapsedTime(&elapsedTime, start,stop);
   ```

   which returns the elapsed time measured in milliseconds between two events.
8. Now, only remains to show the elapsed time.

Puntuación: 0.10
Entrega: De este ejercicio se entrega el código (solo el fichero fuente).

Exercise 5: A Simple Kernel: Matrix sum

In this kernel we're going to perform the addition of two matrices:

$$C = A + B \ ,$$

where $A, B, C \in R^{m \times n}$.
Follow the following steps:

1. Enter `./exercises/SimpleMatrixSum`.
2. Type `make`.
3. Run the application, e.g.:

   ```
   ./SimpleMatrixSum 1000 1000 16 16
   ```

   You'll see a very large error, even Not-A-Number (`nan`). This is because the kernel is not yet implemented.
4. Open file `SimpleMatrixSum.cu`. You'll see the kernel at the top of the file (`compute_kernel`). Implement it!. See Figure 1. There are two matrices, A and B, that once added produce matrix C. In red there is represented an entry matrix that will be processed by a thread in a block. The grid of blocks in the example overlays the shape of the matrix so all entries have an associated thread which process them. To implement the kernel there are two actions to be done that are indicated through two comments in the kernel function and can be easily followed by looking at the figure. Follow the instructions. Clue: Check slice 10 and 14 on *CUDA-2-ProgramingModel*.

The kernel is called in function `cu_matrix_sum` at line 52. The threads are arranged in a two dimensional grid of `row_blocks`×`col_blocks` blocks, where each block is of size `block_rows`×`blocks_cols` threads. (The size of the block of threads has been provided as an argument by command line.) See Listings 2.

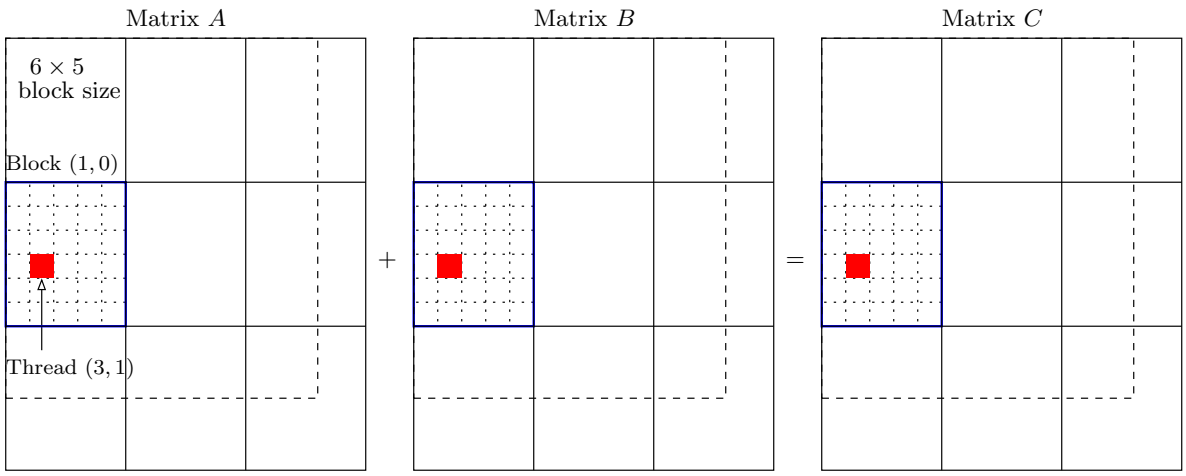


Figure 1: Example of matrix addition of two matrices of size 15×13 in a GPU. (Large dot rectangles represents matrices bounds.) The grid size is of 3×3 blocks of size 6×5. The figure represents the $(9, 1)$ element of the three matrices which is accessed by thread $(3, 1)$ of block $(1, 0)$.

Listing 2: Simple Matrix Sum kernel call

```
dim3 dimGrid( row_blocks, col_blocks );
dim3 dimBlock( block_rows, block_cols );
compute_kernel<<< dimGrid, dimBlock >>>( m, n, d_A, d_B, d_C );
```

Now, we're checking the efficiency of our kernel:

1. Call the CUDA profiler:

   ```
   prompt_$ computeprof &
   ```

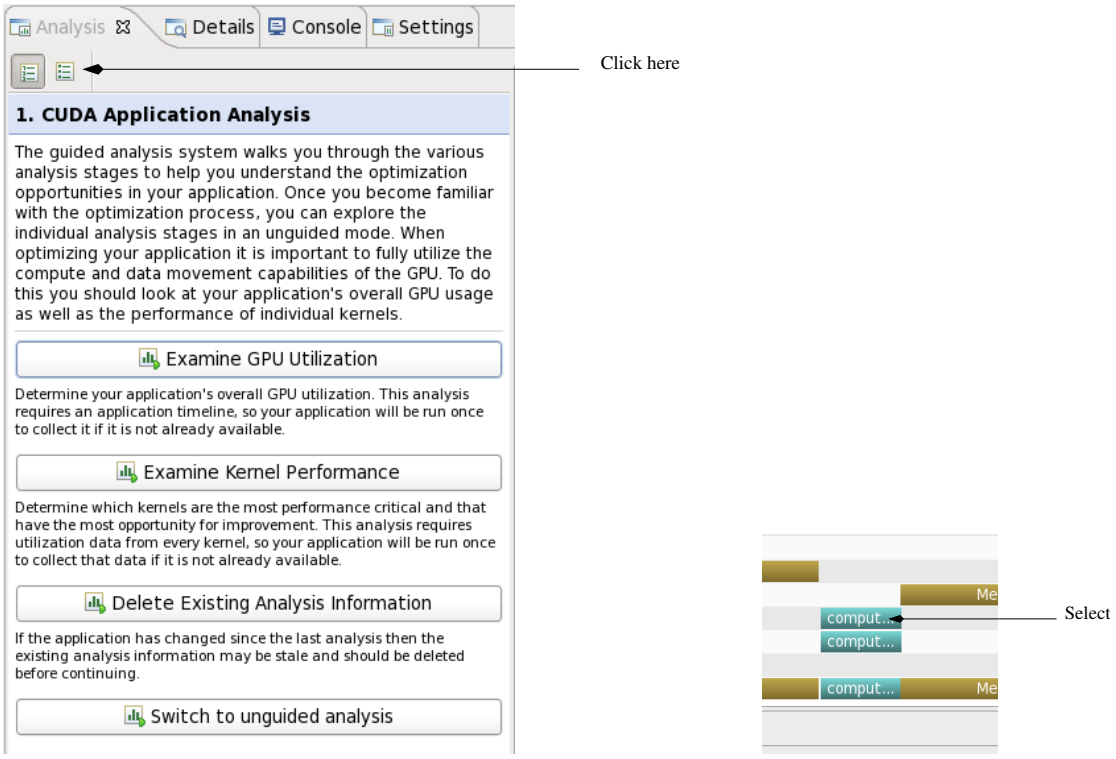
2. Create a new session: `File→New Session`. In the textbox `File` of the emerging dialog window select the executable file (`SimpleMatrixSum`) and write the arguments, e.g. `8192 8192 16 16`, in the arguments textbox. No more information is needed to create the session. Type `Next`, `Finish` and wait for the profiler to analyze the application.

3. On the left-down corner you'll see the image in Fig. 2 (a). Press the indicated button.

4. Now, select the kernel as shown in Fig. 2 (b). Press button `Analyze All`.

   The profiling tool shows many items of analysis. We'll focus our attention on the `Memory Access Pattern`. In our case, since the kernel is very simple and not very high computation is carried out with data and the productivity per date is very low, access pattern to memory is the main source for performance loss. You can select `Line / File` where Transaction Access count is provided to go the exact line in the code which performs this access to memory. A high ratio (e.g. 16) of L2 transactions per data access means that things can do better. Take into account that threads in a *warp* (those in the x dimension) should access consecutive memory locations and elements in the matrix are store by rows. Use Fig. 1 to understand the access pattern to use.

5. Once you have modified your kernel, recompile in command line with the `make` tool. Come back to the profiler, press the button `Reset All` and `Analyze All` again. Look the Memory Access Pattern now, any difference?



(a) Analyzer tool (b) Selection of the kernel to analyze

Figure 2: `computeprof`: CUDA profiler tool

Puntuación: 0.40
Entrega: De este ejercicio se entrega el código (solo el fichero fuente) con la versión final que contiene el alineamiento a memoria correcto.

Exercise 6: Matrix transposition example

In this exercise we're going to implement a kernel that performs a matrix transposition into the GPU. Given a matrix $A \in R^{m \times n}$ the kernel obtains a matrix $B \in R^{n \times m}$ that

$$B = A^T ,$$

that is, an out-of-place transposition.
The code provided (`MatrixTransposition_v1.cu`) reads values m and n and the block size, which will be square for simplicity. Then, it transposes the matrix in CPU and stores the transposed matrix in another one. Routine `cu_transpose` makes all needed to call the kernel. Before calling the kernel, a two-dimensional grid of square blocks of threads is formed (Listings 3).

Listing 3: Matrix Transposition kernel call

```
// Calculate blocks grid size
int blocks_row = (int) ceil( (float) m / (float) block_size );
int blocks_col = (int) ceil( (float) n / (float) block_size );
// Execute the kernel
dim3 dimGrid( blocks_col, blocks_row );
```

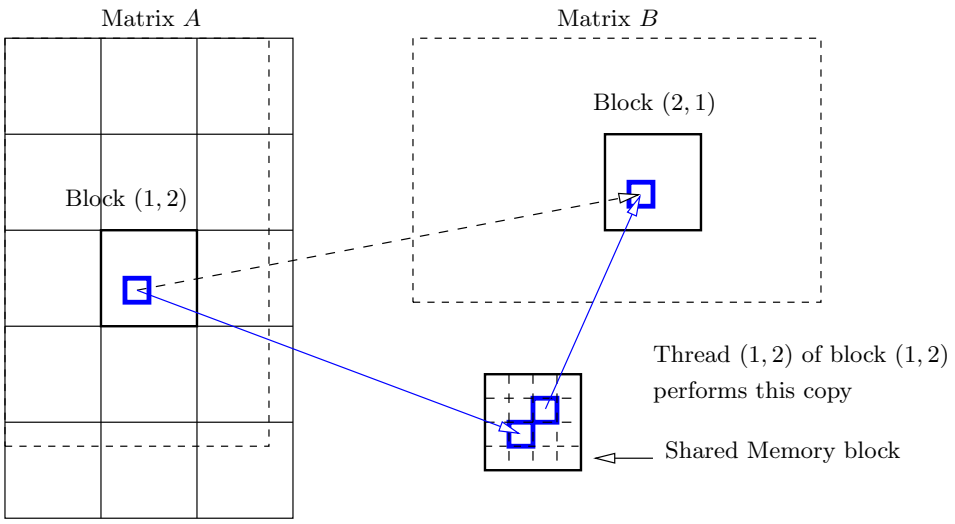


Figure 3: Example of transposition of a 17×11 matrix A into a 11×17 matrix B in a GPU using shared memory. Dot rectangle represents the matrix. A thread (Thread$(1,2)$) of a block (Block$(1,2)$) copies a matrix A element ($A(11,6)$) into an element of matrix B ($B(6,11)$) through shared memory.

```
dim3 dimBlock( block_size, block_size );
compute_kernel<<< dimGrid, dimBlock >>>( m, n, d_A, d_B );
```

To implement the kernel follow the steps:

1. Obtain the coordinates (x, y) of the thread in both dimensions inside the block.
2. Obtain the global index to a row of matrix A.
3. Obtain the global index to a column of matrix A.
4. Copy element $A(i, j)$ into $B(j, i)$ to form the transposed matrix. Take care of not accessing elements beyond matrix bounds.

The code you have implemented does not access memory positions properly for both matrices. Before proceeding, open the CUDA profiler:

1. Select the application and the arguments in the dialog window, e.g., `8192 8192 16`.
2. Select the box representing the kernel execution in the timeline (maybe in blue or green).
3. Select (down-left) click on the kernel in the timeline and press *Analyze All*.
4. Check on the *Memory Access Pattern*. You'll see on the right an issue. Changing the access pattern to the matrices does not solve the problem.

Now, we're using a different version that access consecutive elements of both matrices. To do this, we need to use *shared memory*.
The new version will be implemented in file `MatrixTransposition_v2.cu`. Follow the next steps:

1. Obtain the coordinates (x, y) of the thread in both dimensions inside the block.

2. Save block indexes into two variables, e.g., `BLOCK_X` and `BLOCK_Y`.

Look at Figure 3 to understand the next steps. In this version, we'll use a constant value for the block size `BLOCKSIZE`. This is because we'll use that value to declare shared memory and its allocation size must be known at compile time. For simplicity, we'll use a square block. On the left of the figure there's matrix A. Suppose a threads block is processing the matrix block $(1, 2)$. Since matrices are stored by rows, we'll use the first threads block dimension (`blockIdx.x`) to access row elements of A. In order to attain coalesced memory accesses, the same threads block will store block $(2, 1)$ of matrix B. Note that a block entry of A ($A(i, j)$) and the same coordinated block entry in B ($B(i, j)$), both must be managed by the same thread within the threads block in order to access both matrix elements by rows (as if it was a mere copy without transposition). To transpose elements of that block, we'll use an intermediate copy into share memory.
Continue with the following steps. Note that the column index should stream consecutive entries of the matrix and that the difference between matrix A and B falls in the threads block index.

3. Obtain the global index to a row of A (call it `r_A`).

4. Obtain the global index to a column of A (call it `c_A`).

5. Obtain the global index to a row of B (call it `r_B`).

6. Obtain the global index to a column of B (call it `c_B`).

7. Allocate a shared memory block of dimension `BLOCKSIZE`×`BLOCKSIZE`. This is a normal C-type two-dimensional array of type `float` preceded by word `__shared__`.

8. Within an `if` clause that prevents accessing A elements beyond the $m \times n$ limits, write the copy from the corresponding element of A to B through shared memory. Remember synchronize threads upon data has been saved into shared memory (`__synchthreads()`).

Use the CUDA profiler again to analyze memory access pattern and compare with the former version. Be aware of use always a matrix size that be multiple of `BLOCKSIZE`. It is left as an exercise the possibility of using any matrix size.

Puntuación: 0.50
Entrega: De este ejercicio se entrega el código (solo el fichero fuente) con la versión final que contiene el alineamiento a memoria correcto con la utilización de la memoria `shared`.

Exercise 7: Matrix multiplication with and without shared memory

(from NVIDIA samples)
The next two steps allow to see the performance of a straightforward matrix multiplication without the use of shared memory:

1. Enter `./exercises/matrixMul`.

2. type `make`, and run the code with `./matrixMul`.

To see the performance using shared memory follow:

1. type `make clean`.

2. type `make SHARED=1`, and run the code again with `./matrixMul`.

Puntuación: 0.00
Entrega: De este ejercicio no se entrega nada.

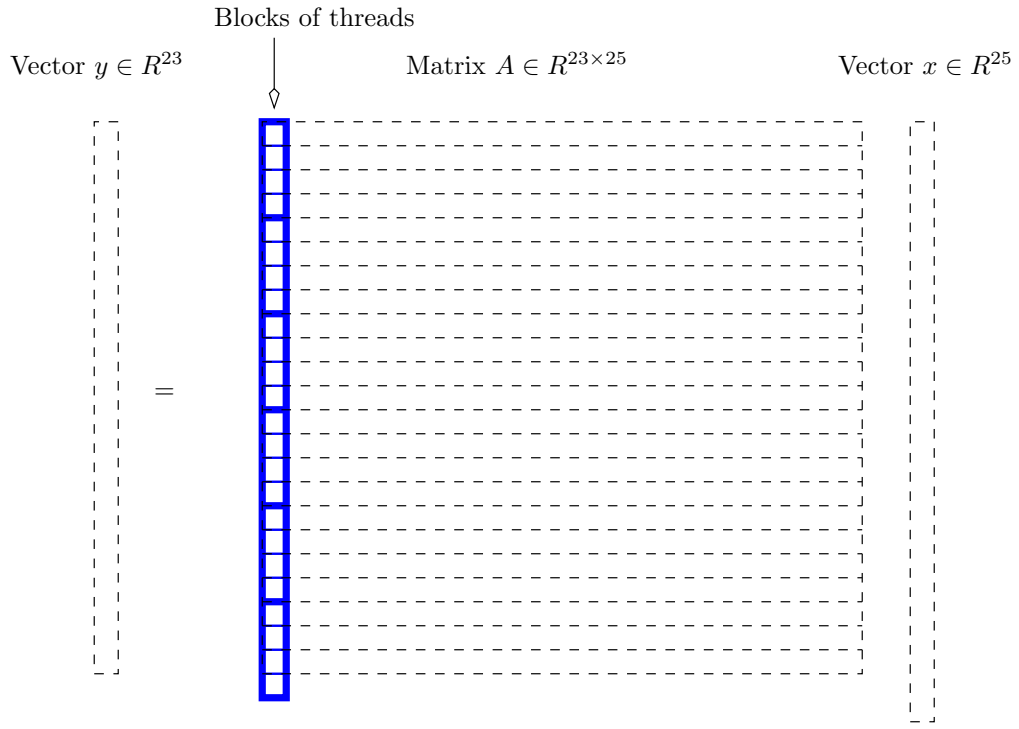


Figure 4: Arrangement of threads for the product of a matrix $A \in R^{23\times25}$ by a vector $v \in R^{25}$. There are 6 blocks of threads with four threads per block. Each thread computes one element of array $y \in R^{23}$.

Exercise 8: Matrix-vector product

In this kernel we're going to perform the matrix-vector product:

$$y = A \cdot x \ ,$$

where $A \in R^{m\times n}$, $x \in R^n$ and $y \in R^m$. We're going to perform two different versions of the same operation. In the first version, we'll use a simple arrangement of threads consisting of a one-dimensional grid of thread blocks, where each block is also a one-dimensional array of threads. Each thread will be in charge of computing one component of the resulting vector y, i.e., each thread computes:

$$y(i) = A(i,:) \cdot x \ ,$$

for $i = 0, \ldots, m-1$, where : denotes all the elements in that dimension. Figure 4 shows the arrangement desired for an example with a 23×25 size matrix. The grid used in the example has 6 blocks where each threads block contains 4 threads.

1. Enter `./exercises/MatrixVectorProduct`.
2. Type `make` and run the application, e.g.:

   ```
   make && ./MatrixVectorProduct_v1 8192 8192
   ```

   You'll see a large error. This is again because the kernel is not yet implemented. (When the kernel is implemented the error will be $\approx 10^{-4}$.)

3. Open file `MatrixVectorProduct_v1.cu`. The kernel is also at the top of the file with the same name as the former exercise (`compute_kernel`). Implement it according to the instructions above. Also, follow the comments in the source code.

4. When you finish, check out for memory access pattern issues with the profiling tool.

Some remarks about this code:

- The block size of threads block is defined with a C macro (`BLOCKSIZE`) as in the previous exercise, though here is not necessary since we don't use this value to declare a shared memory block.
- We use CUDA events to get the computing time of different parts of the code.
- We also compute the Gflops obtained by the application, a very used measure in HPC and GPU contexts to analyze the performance of our applications and the so good that our CUDA code is. In this case, the operation has a cost of $2n^2$ flops[1].

Now, we're designing a more efficient kernel. To this end we need to use Shared Memory.
First of all, look at Figure 5. We'll use a one-dimensional grid of thread blocks, where each block will be square. Following the instructions in the code (`MatrixVectorProduct_v2.cu`) obtain coordinate identifiers for the thread and the global row index (`i`). Also, declare shared memory space for an array of size `BLOCKSIZE` and a two-dimensional array of size `BLOCKSIZE`×`BLOCKSIZE`. Remember that an array in shared memory is declared like a regular C array preceded by modifier `__shared__`.
The kernel is partitioned into two parts: (Since the resulting data is value `d_y( i )`, all the kernel code will be within an `if` clause preventing not access components beyond `m-1`, as in previous exercises.)

1. A thread which has access to global index row `i`, multiplies $A(i, (\texttt{threadIdx.x:BLOCKSIZE:n-1}))$ by $x((\texttt{threadIdx.x:BLOCKSIZE:n-1}))$. This means that, in this example of Figure 5, thread $(3, 2)$ multiplies the red components of row 2 of A, one by one, by the red components of vector x. Pay attention that threads in the x dimension are consecutive (belong to the same warp) so they should traverse an array of consecutive elements.

   Note that all the threads in the same block column ($\text{Th}(X, :)$) reference the same vector x component. Thus, before doing the above-mentioned computation, threads in row `Y==0` should save a piece of x in the one-dimensional array previously allocated in shared memory and before reading it.

   The resulting data of this part can be stored in a local variable. Upon termination, each thread saves the content of this variable in its corresponding position in the shared memory square block.

   Remember use `__syncthreads()` each time you save data into shared memory.

2. This second part is carried out only by threads on the first column of the block (`X==0`). The computation consists of adding all the components of the columns along each row of the shared memory square block. The result is a vector (each thread of the first column has one component of this vector) that must be saved in array `d_y`.

Once the kernel is implemented, run it and compare results with the former version. Use the profiler to see the benefits of the new version. Now, declare a shared memory square block of size `BLOCKSIZE`×`BLOCKSIZE+1` and analyze results with the profiler.

Puntuación: 0.60
Entrega: De este ejercicio se entrega el código (solo el fichero fuente).

[1] **flops** = **fl**oating-point **o**perations **p**er **s**econd.

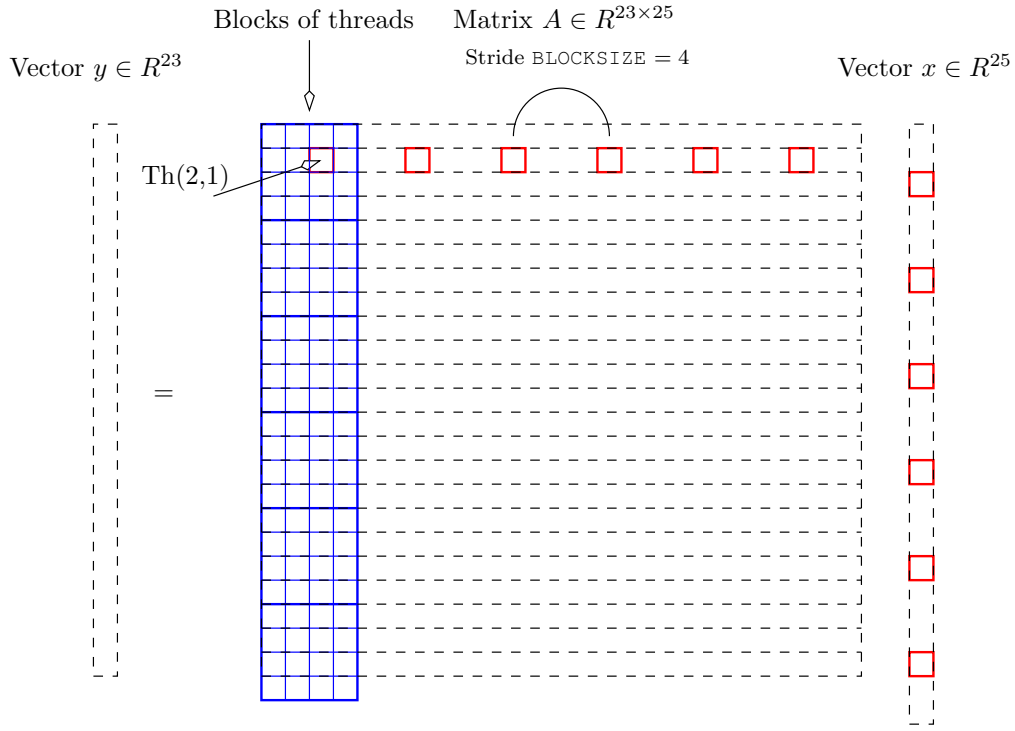


Figure 5: Arrangement of threads for the product of a matrix $A \in R^{23\times 25}$ by a vector $v \in R^{25}$. There are 6 blocks of threads of size 4×4. Components of matrix A colored in red are accessed by thread $(2, 1)$ to be multiplied by components of array x also colored in red.

Exercise 9: Using CUDA libraries: CUBLAS

In this exercise we are going to perform a matrix-matrix multiplication using the CUDA library CUBLAS. Follow the next steps:

1. Enter into the CUBLAS_Example_without_mkl folder and open the file MatrixMatrixMultiplication.cu.
2. The first you need to use CUBLAS is to include the suitable header file (line 8). Search for it in http://docs.nvidia.com/cuda/cublas/index.html
3. To use CUBLAS routines you need a CUBLAS handler (line 59). This is an object of type cublasHandle_t. Call it handle.
4. Initialize it using routine cublasCreate in the place signaled (line 60).
5. Fill the gaps inside macros CUDA_SAFE_CALL() to transfer matrices A and B to the device (lines 77 and 78).
6. Fill the gap inside macro CUBLAS_SAFE_CALL() by writing a call to the CUBLAS routine cublasDgemm (line 79). You'll find API information at http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemm. You can use the already declared constants ZERO and ONE.
7. Transfer matrix d_C to gpu_C (line 82).
8. Don't forget to destroy the CUBLAS handler at the end of the source code by using the routine.

 continued on next page...

Now compile and run your program.

Puntuación: 0.15
Entrega: De este ejercicio se entrega el código (solo el fichero fuente).

Exercise 10: Matrix multiplication with streams

With this exercise we'll learn the capability of CUDA streams to improve performance by overlapping data transference and GPU computation.
The exercise consists of implementing a matrix multiplication partitioning matrix A so that different streams can do different work at the same time. We partition the CPU matrix `A` (and `d_A` in GPU) of size $m \times p$ in blocks of size $k \times p$, as shown in Fig. 6. The idea is to have a number of streams equal to m/k, each one of them doing the same work on different data. The example (`./exercises/MatMulStreams`) asks for matrices size `m`, `p`, `n`, and the number of streams (`nstreams`), thus the block size is calculated as `k=m/nstreams`. The computation carried out by each stream consists of a sequence of three operations:

1. a data transference of a given block of `d_A` to the GPU,
2. a matrix multiplication of the block of `d_A` with matrix `d_B`, and
3. a data transference of the resulting block of `d_D` to the CPU.

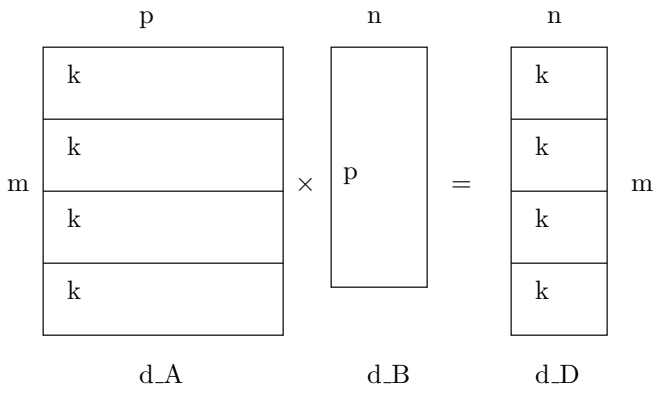


Figure 6: Matrix multiplication with streams.

The exercise consists of replacing the comments into the given gaps by the required actions in the source file. Remember to use pinned memory (`HostAlloc`) when it is required to allocate memory into the Host. For data transference it is convenient to use CUBLAS routines since it is more easy to select the appropriate matrix block to be sent/received (`cublasGetMatrixAsync` and `cublasSetMatrixAsync`). Use asynchronous versions since the operation must be asynchronous with regard to the Host to operate as expected. In order to set a CUBLAS operation into a given CUDA stream it is necessary to set this stream just before the calling function with `cudaSetStream`.
The program should show a small value as error to check for correctness. Varying the number of streams we can find a trade-off value which allows to reduce the execution time.

Puntuación: 0.25
Entrega: De este ejercicio se entrega el código (solo el fichero fuente) y una gráfica en la que se vea la ganancia obtenida con los *Streams*.

Exercise 11: MEX files for GPU (Opcional)

We're going to execute a matrix multiplication example in GPU using the Matlab environment. To this end, we'll make a mex file. Follow the next steps:

1. Enter folder `mex_gpu`. There's a source code (mex file `matprod.c`) for the product of two matrices from the Matlab command line. You can check that code by compilig it: `mex matprod.c`. The compilation can be carried out inside the Matlab application or in the Linux shell. Use the last option.

2. Copy file `matprod.c` in a new one called `cumatprod.cu`. This new source file will be the max file for the GPU version.

3. Function `mexFunction` in the file `cumatprod.cu` will remain unchanged. On the contrary, function `matprod` will be replaced by a new one. Change name of function `matprod` to `cumatprod`. The arguments are the same.

4. Include files `stdio.h` and the header file needed to use the CUBLAS package. This is because we're going to call the proper routine of this packege to perform the matrix product.

5. Include the definitions for the macros `CUDA_SAFE_CALL` and `CUBLAS_SAFE_CALL` use in the last exercise.

6. Now there's only left to implement routine `cumatprod`. Since we're using CUBLAS, follow the steps used in the last example to perform this product. In general, the steps are simple and clear, they can be summarized as:

   (a) Create a CUBLAS handle.

   (b) Allocate matrices in the GPU.

   (c) Copy data from the Host to de GPU.

   (d) Perform the product calling the multiplication routine of CUBLAS.

   (e) Copy back the resulting matrix from GPU to the CPU.

   (f) Destroy the CUBLAS context and free memory space in GPU.

7. Compilation. Compiling this mex file is not trivial. Two steps are required: compile the source with the NVIDIA compiler (`nvcc`), and link the application with the mex tool. In your case, use the provided Makefile and simply type `make`.

Puntuación: 0.25
Entrega: De este ejercicio se entrega el código (solo el fichero fuente) y un pantallazo de la ejecución en Matlab sin y con GPUs.