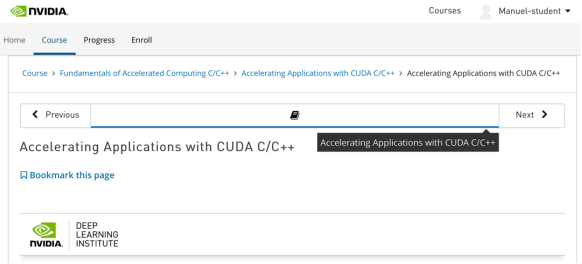
Ejercicios del primer módulo del DLI

Pertenecientes al workshop
“Fundamentals of Accelerated Computing with CUDA C/C++”

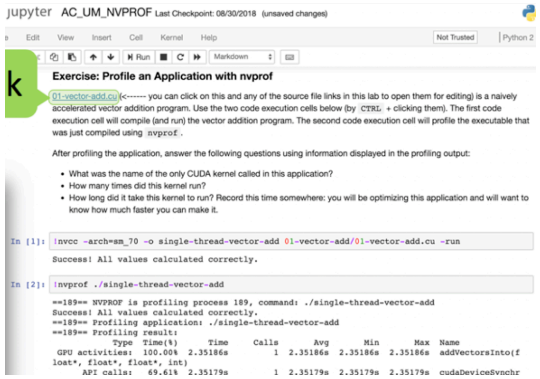
Entorno de trabajo recomendable en tu navegador Web para trabajar en CUDA

- Ventana 1, pestaña 1: Maestra.
- Ventana 1, pestaña 2: Jupyter.
- Ventana 2, pestaña 1: Tu editor de texto con el programa CUDA.

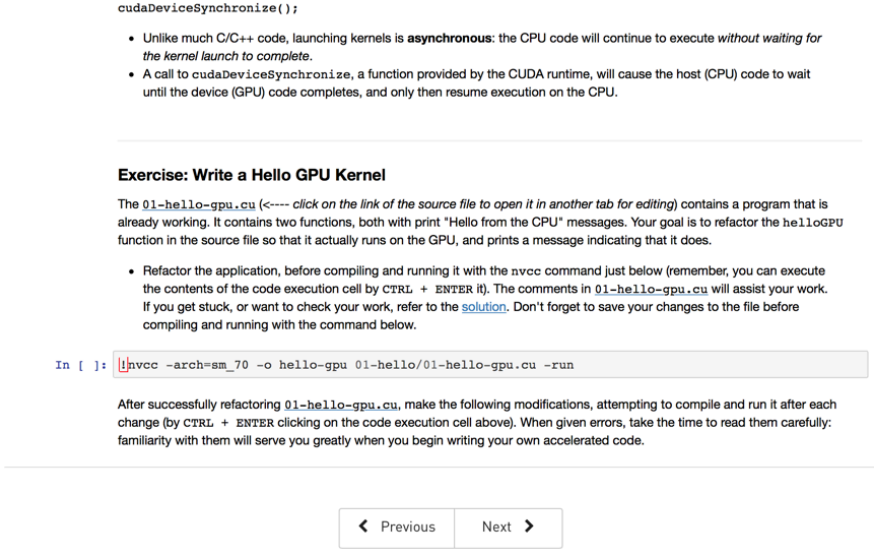
Maestra (tu curso del DLI)



Jupyter (tu GPU en AWS)



El editor de texto con tu programa CUDA



Exercise 1: Write a Hello GPU Kernel

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  /*
09   * Refactor the `helloGPU` definition to be a kernel
10   * that can be launched on the GPU. Update its message
11   * to read "Hello from the GPU!"
12   */
13
14  void helloGPU()
15  {
16    printf("Hello also from the CPU.\n”);
17  }
18
19  int main()
20  {
21
22    helloCPU();
23
24    /*
25     * Refactor this call to `helloGPU` so that it launches
26     * as a kernel on the GPU.
27     */
28
29    helloGPU();
30    /*
31     * Add code below to synchronize on the completion of the
32     * `helloGPU` kernel completion before continuing the CPU
33     * thread.
34     */
35  }
```

Exercise: Write a Hello GPU Kernel

The `01-hello-gpu.cu` ("<---- click on the link of the source file to open it in another tab for editing") contains a program that is already working. It contains two functions, both with print "Hello from the CPU" messages. Your goal is to refactor the `helloGPU` function in the source file so that it actually runs on the GPU, and prints a message indicating that it does.

- Refactor the application, before compiling and running it with the `nvcc` command just below (remember, you can execute the contents of the code execution cell by `CTRL + ENTER` it). The comments in `01-hello-gpu.cu` will assist your work. If you get stuck, or want to check your work, refer to the solution. Don't forget to save your changes to the file before compiling and running with the command below.

```
In [ ] !nvcc -arch=sm_70 -o hello-gpu 01-hello/01-hello-gpu.cu -run
```

After successfully refactoring `01-hello-gpu.cu`, make the following modifications, attempting to compile and run it after each change (by `CTRL + ENTER` clicking on the code execution cell above). When given errors, take the time to read them carefully: familiarity with them will serve you greatly when you begin writing your own accelerated code.

- Remove the keyword `__global__` from your kernel definition. Take care to note the line number in the error: what do you think is meant in the error by "configured"? Replace `__global__` when finished.
- Remove the execution configuration: does your understanding of "configured" still make sense? Replace the execution configuration when finished.
- Remove the call to `cudaDeviceSynchronize`. Before compiling and running the code, take a guess at what will happen, recalling that kernels are launched asynchronously, and that `cudaDeviceSynchronize` is what makes host execution in wait for kernel execution to complete before proceeding. Replace the call to `cudaDeviceSynchronize` when finished.
- Refactor `01-hello-gpu.cu` so that `Hello from the GPU` prints **before** `Hello from the CPU`.
- Refactor `01-hello-gpu.cu` so that `Hello from the GPU` prints **twice**, once **before** `Hello from the CPU`, and once **after**.

1.0 Refactor the helloGPU function in the source file so that it actually runs on the GPU, and prints a message indicating that it does.

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  /*
09   * Refactor the `helloGPU` definition to be a kernel
10   * that can be launched on the GPU. Update its message
11   * to read "Hello from the GPU!"
12   */
13
14  void helloGPU()
15  {
16    printf("Hello also from the CPU.\n”);
17  }
18
19  int main()
20  {
21
22    helloCPU();
23
24    /*
25     * Refactor this call to `helloGPU` so that it launches
26     * as a kernel on the GPU.
27     */
28
29    helloGPU();
30    /*
31     * Add code below to synchronize on the completion of the
32     * `helloGPU` kernel completion before continuing the CPU
33     * thread.
34     */
35  }
```

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  /*
09   * Refactor the `helloGPU` definition to be a kernel
10   * that can be launched on the GPU. Update its message
11   * to read "Hello from the GPU!"
12   */
13
14  __global__ void helloGPU()
15  {
16    printf("Hello also from the CPU.GPU!\n”);
17  }
18
19  int main()
20  {
21
22    helloCPU();
23
24    /*
25     * Refactor this call to `helloGPU` so that it launches
26     * as a kernel on the GPU.
27     */
28    helloGPU<<<1,1>>>();
29    cudaDeviceSynchronize();
30   /*
31    * Add code below to synchronize on the completion of the
32    * `helloGPU` kernel completion before continuing the CPU
33    * thread.
34    */
35  }
```

1.1 Remove the keyword __global__ from your kernel definition. Take care to note the line number in the error: what do you think is meant in the error by “configured”? Replace __global__ when finished.

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17
18    helloGPU<<<1,1>>>();
19    /*
20     * Add code below to synchronize
21     * on the completion of the
22     * `helloGPU` kernel completion
23     * before continuing the CPU thread.
24     */
25  }
```

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global____global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17    helloGPU<<<1,1>>>();
18    cudaDeviceSynchronize();
19    /*
20     * Add code below to synchronize
21     * on the completion of the
22     * `helloGPU` kernel completion
23     * before continuing the CPU thread.
24     */
25  }
```

1.2 Remove the execution configuration: does your understanding of “configured” still make sense? Replace the execution configuration when finished.

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17
18    helloGPU<<<1,1>>>();
19    /*
20     * Add code below to synchronize
21     * on the completion of the
22     * `helloGPU` kernel completion
23     * before continuing the CPU thread.
24     */
25  }
```

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17    helloGPU<<<1,1>>><<<1,1>>>();
18    cudaDeviceSynchronize();
19    /*
20     * Add code below to synchronize
21     * on the completion of the
22     * `helloGPU` kernel completion
23     * before continuing the CPU thread.
24     */
25  }
```

1.3 Remove the call to `cudaDeviceSynchronize`. Before compiling and running the code, take a guess at what will happen, recalling that kernels are launched asynchronously, and that `cudaDeviceSynchronize` is what makes host execution in wait for kernel execution to complete before proceeding. Replace the call to `cudaDeviceSynchronize` when finished.

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17
18    helloGPU<<<1,1>>>();
19    cudaDeviceSynchronize();
20  }
```

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17
18    helloGPU<<<1,1>>>();
19    cudaDeviceSynchronize();
20    cudaDeviceSynchronize();
21  }
```

1.4 Refactor 01-hello-gpu.cu so that *Hello from the GPU* prints before *Hello from the CPU.*

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17
18    helloGPU<<<1,1>>>();
19    cudaDeviceSynchronize();
20  }
```

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloCPU();
17    helloGPU<<<1,1>>>();
18    cudaDeviceSynchronize();
19    helloCPU();
20  }
```

1.5 Refactor `01-hello-gpu.cu` so that *Hello from the GPU* prints **twice**, once **before** *Hello from the CPU* and once **after**.

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloGPU<<<1,1>>>();
17    cudaDeviceSynchronize();
18    helloCPU();
19
20  }
```

```
01  #include <stdio.h>
02
03  void helloCPU()
04  {
05    printf("Hello from the CPU.\n");
06  }
07
08  __global__ void helloGPU()
09  {
10    printf("Hello from the GPU!\n”);
11  }
12
13  int main()
14  {
15
16    helloGPU<<<1,1>>>();
17    cudaDeviceSynchronize();
18    helloCPU();
19    helloGPU<<<1,1>>>();
20    cudaDeviceSynchronize();
21
22  }
```

Exercise 2: Launch Parallel Kernels

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  void firstParallel()
09  {
10    printf("This should be running in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel();
21
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

Exercise: Launch Parallel Kernels

`01-first-parallel.cu` currently makes a very basic function call that prints the message `This should be running in parallel.` Follow the steps below to refactor it first to run on the GPU, and then, in parallel, both in a single, and then, in multiple thread blocks. Refer to the solution if you get stuck.

- Refactor the `firstParallel` function to launch as a CUDA kernel on the GPU. You should still be able to see the output of the function after compiling and running `01-first-parallel.cu` with the `nvcc` command just below.
- Refactor the `firstParallel` kernel to execute in parallel on 5 threads, all executing in a single thread block. You should see the output message printed 5 times after compiling and running the code.
- Refactor the `firstParallel` kernel again, this time to execute in parallel inside 5 thread blocks, each containing 5 threads. You should see the output message printed 25 times now after compiling and running.

2.1 Refactor the `firstParallel` function to launch as a CUDA kernel on the GPU. You should still be able to see the output of the function after compiling and running `01-basic-parallel.cu`

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  void firstParallel()
09  {
10    printf("This should run in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel();
21
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  __global__ void firstParallel()
09  {
10    printf("This should run in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel<<<1,1>>>();
21    cudaDeviceSynchronize();
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

2.2 Refactor the `firstParallel` kernel to execute in parallel on 5 threads, all executing in a single thread block. You should see the output message printed 5 times after compiling and running the code.

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  __global__ void firstParallel()
09  {
10    printf("This should run in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel<<<1,1>>>();
21
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  __global__ void firstParallel()
09  {
10    printf("This should run in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel<<<1,15>>>();
21    cudaDeviceSynchronize();
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

2.3 Refactor the `firstParallel` kernel again, this time to execute in parallel inside 5 thread blocks, each containing 5 threads. You should see the output message printed 25 times now after compiling and running.

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  __global__ void firstParallel()
09  {
10    printf("This should run in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel<<<1,5>>>();
21
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

```
01  #include <stdio.h>
02
03  /*
04   * Refactor firstParallel so that
05   * it can run on the GPU.
06   */
07
08  __global__ void firstParallel()
09  {
10    printf("This should run in parallel.\n");
11  }
12
13  int main()
14  {
15    /*
16     * Refactor this call to firstParallel
17     * to execute in parallel on the GPU.
18     */
19
20    firstParallel<<<15,5>>>();
21    cudaDeviceSynchronize();
22    /*
23     * Some code is needed below so that
24     * the CPU will wait for the GPU kernels
25     * to complete before proceeding.
26     */
27
28  }
```

Exercise 3: Use Specific Thread and Block Indices

```
01  #include <stdio.h>
02
03  __global__ void printSuccess()
04  {
05
06    if (threadIdx.x == 1023 && blockIdx.x == 255)
07    {
08      printf("Success!\n");
09    } else {
10      printf("Failure. Update the execution configuration as necessary.\n");
11    }
12  }
13
14  int main()
15  {
16    /*
17     * Update the execution configuration so that the kernel
18     * will print `"Success!"`.
19     */
20
21    printSuccess<<<1,1>>>();
22  }
```

Exercise: Use Specific Thread and Block Indices

Currently the `01-thread-and-block-idx.cu` file contains a working kernel that is printing a failure message. Open the file to learn how to update the execution configuration so that the success message will print. After refactoring, compile and run the code with the code execution cell below to confirm your work. Refer to the solution if you get stuck.

```
In [ ] !nvcc -arch=sm_70 -o thread-and-block-idx 03-indices/01-thread-and-block-idx.cu -run
```

3.1 Update the execution configuration so that the success message will print. After refactoring, compile and run the code with the code execution cell below to confirm your work.

```
01  #include <stdio.h>
02
03  __global__ void printSuccess()
04  {
05
06    if (threadIdx.x == 1023 &&
07        blockIdx.x == 255)
08    {
09      printf("Success!\n");
10    } else {
11      printf("Failure.\n");
12    }
13  }
14
15  int main()
16  {
17    /*
18     * Update the execution
19     * configuration so that
20     * the kernel will print
21     * `"Success!"`.
22     */
23
24    printSuccess<<<1,1>>>();
25  }
```

```
01  #include <stdio.h>
02
03  __global__ void printSuccess()
04  {
05
06    if (threadIdx.x == 1023 &&
07        blockIdx.x == 255)
08    {
09      printf("Success!\n");
10    } else {
11      printf("Failure.\n");
12    }
13  }
14
15  int main()
16  {
17    /*
18     * Update the execution
19     * configuration so that
20     * the kernel will print
21     * `"Success!"`.
22     */
23
24    printSuccess<<<1,1256,1024>>>();
25    cudaDeviceSynchronize();
26  }
```

Exercise 4: Accelerating a For Loop with a Single Block of Threads

```
01  #include <stdio.h>
02
03  void loop(int N)
04  {
05    for (int i = 0; i < N; ++i)
06    {
07      printf("Iteration %d\n", i);
08    }
09  }
10
11  int main()
12  {
13    int N = 10;
14    loop(N);
15  }
```

Exercise: Accelerating a For Loop with a Single Block of Threads

Currently, the `loop` function inside `01-single-block-loop.cu`, runs a for loop that will serially print the numbers `0` through `—`. Refactor the `loop` function to be a CUDA kernel which will launch to execute `N` iterations in parallel. After successfully refactoring, the numbers `0` through `—` should still be printed. Refer to the solution if you get stuck.

In [ ]
```
!nvcc -arch=sm_70 -o single-block-loop 04-loops/01-single-block-loop.cu -run
```

4.1 Refactor the loop function to be a CUDA kernel which will launch to execute N iterations in parallel. After successfully refactoring, the numbers 0 through 9 should still be printed.

```
01  #include <stdio.h>
02
03  void loop(int N)
04  {
05    for (int i = 0; i < N; ++i)
06    {
07      printf("Iteration %d\n", i);
08    }
09  }
10
11  int main()
12  {
13    int N = 10;
14    loop(N);
15  }
```

```
01  #include <stdio.h>
02
03  __global__ void loop(int N)
04  {
05    int i = blockIdx.x*blockDim.x
06          + threadIdx.x;
07    for (int i = 0; i < N; ++i)
08    {
09      printf("Iteration %d\n", i);
10    }
11  }
12
13  int main()
14  {
15    int N = 10;
16    loop<<<1,N>>>(N);
17    cudaDeviceSynchronize();
18  }
```

Exercise 5: Accelerating a For Loop with Multiple Blocks of Threads

```
01  #include <stdio.h>
02
03  void loop(int N)
04  {
05    for (int i = 0; i < N; ++i)
06    {
07      printf("Iteration %d\n", i);
08    }
09  }
10
11  int main()
12  {
13    int N = 10;
14    loop(N);
15  }
```

Exercise: Accelerating a For Loop with Multiple Blocks of Threads

Currently, the `loop` function inside `02-multi-block-loop.cu` runs a for loop that will serially print the numbers `0` through `—`. Refactor the `loop` function to be a CUDA kernel which will launch to execute `N` iterations in parallel. After successfully refactoring, the numbers `0` through `—` should still be printed. For this exercise, as an additional constraint, use an execution configuration that launches *at least 2 blocks of threads*. Refer to the solution if you ge stuck.

In [ ]
```
!nvcc -arch=sm_70 -o multi-block-loop 04-loops/02-multi-block-loop.cu -run
```

5.1 Refactor the loop function to be a CUDA kernel which will launch to execute N iterations in parallel. After refactoring, the numbers 0 through 9 should still be printed. Use at least 2 blocks.

```
01  #include <stdio.h>
02
03  void loop(int N)
04  {
05    for (int i = 0; i < N; ++i)
06    {
07      printf("Iteration %d\n", i);
08    }
09  }
10
11  int main()
12  {
13    int N = 10;
14    loop(N);
15  }
```

```
01  #include <stdio.h>
02
03  __global__ void loop(int N)
04  {
05    int i = blockIdx.x*blockDim.x
06          + threadIdx.x;
07    for (int i = 0; i < N; ++i)
08    {
09      printf("Iteration %d\n", i);
10    }
11  }
12
13  int main()
14  {
15    int N = 10;
16    loop<<<2,N/2>>>(N);
17    cudaDeviceSynchronize();
18  }
```

Final exercise: Accelerate Vector Addition Application

```
void AddVectorsInto (float *result, float *a, float *b, int N)
{  // Versión secuencial en CPU
  for (int i=0; i<N; i++)
    result[i] = a[i] + b[i];
}
```

```
__global__ void AddVectorsInto (float *result, float *a, float *b, int N)
{  // Solución paralela en GPU con CUDA propuesta por el DLI
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i=index; i<N; i+=stride)
    if (i<N)
      result[i] = a[i] + b[i];
}
```

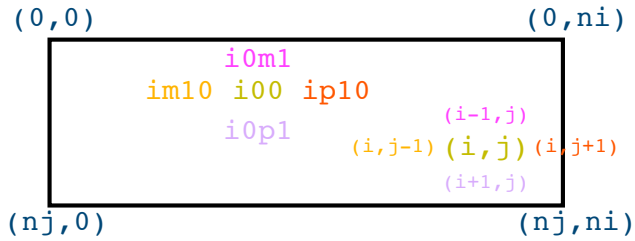
```
__global__ void AddVectorsInto (float *result, float *a, float *b, int N)
{  // Solución paralela en GPU menos elegante pero más eficiente
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index<N)
    result[index] = a[index] + b[index];
}
```

Código CPU para manejar la memoria y recoger los resultados de GPU

```
unsigned int numBytes = N * sizeof(float);
// Aloja memoria en la CPU
float* h_A = (float*) malloc(numBytes);
float* h_B = (float*) malloc(numBytes);
... inicializa h_A y h_B ...
// Aloja memoria en la GPU
float* d_A = 0;  cudaMalloc((void**)&d_A, numBytes);
float* d_B = 0;  cudaMalloc((void**)&d_B, numBytes);
float* d_C = 0;  cudaMalloc((void**)&d_C, numBytes);
// Copiar los datos de entrada desde la CPU a la GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
AddVectorsInto<<<N/256,256>>>(d_A,d_B,d_C);
// Copiar los resultados desde la GPU a la CPU
float* h_C = (float*) malloc(numBytes);
cudaMemcpy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);
// Liberar la memoria de vídeo
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

Heat Conduction en CPU

```
#define I2D(num, c, r) ((r)*(num)+(c))
void kernel(int ni, int nj, float fact, float* temp_in, float*temp_out)
  // loop over all points in domain (except boundary)
  for ( int j=1; j < nj-1; j++ ) {
    for ( int i=1; i < ni-1; i++ ) {
      // find indices into linear memory
      // for central point and neighbours
      int i00  = I2D(ni, i,   j);
      int im10 = I2D(ni, i-1, j);
      int ip10 = I2D(ni, i+1, j);
      int i0m1 = I2D(ni, i,   j-1);
      int i0p1 = I2D(ni, i,   j+1);

      // evaluate derivatives
      float d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
      float d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

      // update temperatures
      temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
    }
  }
```




```
// i00 =  j * ni   + i
// im10 = j * ni   + i-1
// ip10 = j * ni   + i+1
// i0m1 = (j-1)*ni + i
// i0p1 = (j+1)*ni + i
```

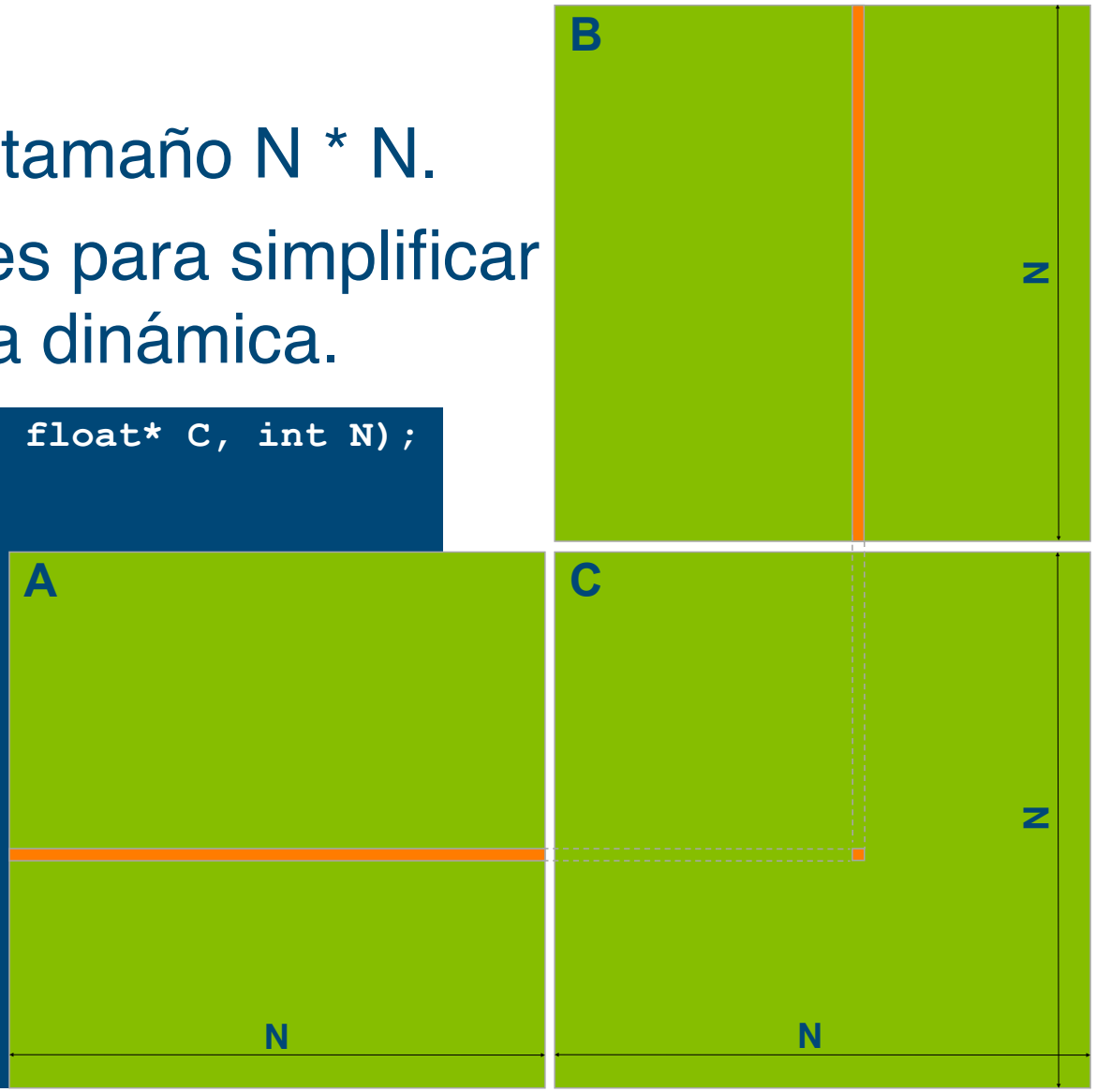
Heat Conduction en GPU

```
#define I2D(num, c, r) ((r)*(num)+(c))
__global__ void kernel(int ni, int nj,
                       float fact, float* temp_in, float *temp_out)
{ // loop over all points in domain (except boundary)
  for ( int j=1; j < nj-1; j++ ) {   j = blockIdx.x * blockDim.x + threadIdx.x;
    for ( int i=1; i < ni-1; i++ ) { i = blockIdx.y * blockDim.y + threadIdx.y;
      if (((j>0) && (i>0)) && ((j<nj-1) && (i<ni-1))) {
      int i00  = I2D(ni, i, j);                    // i00  = j * ni   + i
      int im10 = I2D(ni, i-1, j);                  // im10 = j * ni   + i-1
      int ip10 = I2D(ni, i+1, j);                  // ip10 = j * ni   + i+1
      int i0m1 = I2D(ni, i, j-1);                  // i0m1 = (j-1)*ni + i
      int i0p1 = I2D(ni, i, j+1);                  // i0p1 = (j+1)*ni + i

      // evaluate derivatives
      float d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
      float d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

      // update temperatures
      temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
    }}
  }}
```

Producto de matrices en CPU

- C = A * B
- Matrices cuadradas de tamaño N * N.
- Linearizadas en vectores para simplificar el alojamiento de memoria dinámica.

```
void MxMonCPU(float* A, float* B, float* C, int N);
{
  forall (int i=0; i<N; i++)
    forall (int j=0; j<N; j++)
    {
      float sum=0;
      for (int k=0; k<N; k++)
      {
 A[i][k] float a = A[i*N + k];
 B[k][j] float b = B[k*N + j];
        sum += a*b;
      }
      C[i*N + j] = sum;
    }
}
```



Versión CUDA para el producto de matrices

```
void MxMonGPU(float* A, float* B, float* C, int N);
{
  float sum=0;
  int i, j;

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k=0; k<N; k++)
  {
    float a = A[i*N + k];
    float b = B[k*N + j];
    sum += a*b;
  }
  C[i*N + j] = sum;
}
```