

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

DEEP LEARNING INSTITUTE

- Ventana 1, pestaña 1: Maestra.
- Ventana 1, pestaña 2: Jupyter.

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

Courses Manuel-student * Accelerating Applications with CUDA C/C++ □ Bookmark this page

Maestra (tu curso del DLI) I Jupyter (tu GPU en AWS) IUDVTer AC UM_NVPROF Last Checkpoint: 08/30/2018 (unsaved changer --- you can click on this and any of the source file links in this lab to open them for editing) is a naively erated vector addition program. Use the two code execution cells below (by CTRL+ clicking them). The first code tion cell will compile (and run) the vector addition program. The second code execution cell will profile the executable this After profiling the application, answer the following questions using information displayed in the profiling output · How long did it take this kernel to run? Record this time somewhere; you will be optimizing this application and will want to Success! All values calculated correctly

El editor de texto con tu programa CUDA

cudaDeviceSynchronize();

- Unlike much C/C++ code, launching kernels is asynchronous: the CPU code will continue to execute without waiting for the kernel launch to complete
- · A call to cudaDeviceSynchronize, a function provided by the CUDA runtime, will cause the host (CPU) code to wait until the device (GPU) code completes, and only then resume execution on the CPU

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.

> Previous Next >



Exercise 1: Write a Hello GPU Kernel

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

Exercise: Write a Hello GPU Kernel

The <u>01-hello-gpu.cu</u> ('<---- 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 "Helio 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 <u>cudaDeviceSynchronize</u>. Before compiling and running the code, take a guess at what will happen, recalling that kernels are launched asynchronously, and that <u>cudaDeviceSynchronize</u> is what makes host execution in walt for kernel execution to complete before proceeding. Replace the call to <u>cudaDeviceSynchronize</u> 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()
05
      printf("Hello from the CPU.\n");
06 }
07
08 /*
     * Refactor the `helloGPU` definition to be a kernel
     * that can be launched on the GPU. Update its message
     * to read "Hello from the GPU!"
12
     */
13
    void helloGPU()
15 {
      printf("Hello also from the CPU.\n");
17
18
    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
       * `helloGPU` kernel completion before continuing the CPU
32
33
       * thread.
34
       */
35 }
```

```
#include <stdio.h>
02
03 void helloCPU()
      printf("Hello from the CPU.\n");
05
06 }
07
98
09
     * Refactor the `helloGPU` definition to be a kernel
     * that can be launched on the GPU. Update its message
     * to read "Hello from the GPU!"
11
12
     */
13
    __global__ void helloGPU()
15
      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>>>();
       cudaDeviceSynchronize();
30
       * Add code below to synchronize on the completion of the
31
       * `helloGPU` kernel completion before continuing the CPU
32
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.

```
#include <stdio.h>
01
02
03
    void helloCPU()
04
      printf("Hello from the CPU.\n");
05
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
       * `helloGPU` kernel completion
22
       * before continuing the CPU thread.
23
24
25
```

```
#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
       * `helloGPU` kernel completion
22
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.

```
#include <stdio.h>
01
02
03
    void helloCPU()
04
      printf("Hello from the CPU.\n");
05
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
       * `helloGPU` kernel completion
22
       * before continuing the CPU thread.
23
24
25
```

```
#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
       * `helloGPU` kernel completion
22
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>
                                          01
                                              #include <stdio.h>
02
                                          02
03
    void helloCPU()
                                              void helloCPU()
                                          03
04
                                          04
05
      printf("Hello from the CPU.\n");
                                          05
                                                printf("Hello from the CPU.\n");
06
                                          06
07
                                          07
08
      _global__ void helloGPU()
                                          80
                                                _global___ void helloGPU()
09
                                          09
10
      printf("Hello from the GPU!\n");
                                          10
                                                printf("Hello from the GPU!\n");
11
                                          11
12
                                          12
13
    int main()
                                          13
                                              int main()
14
                                          14
15
                                          15
16
      helloCPU();
                                          16
                                                helloCPU();
17
                                          17
18
      helloGPU<<<1,1>>>();
                                                helloGPU<<<1,1>>>();
                                          18
      cudaDeviceSynchronize();
19
                                          19
                                                cudaDeviceSynchronize();
20
                                          20
                                                cudaDeviceSynchronize();
                                          21
```



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

```
#include <stdio.h>
01
                                              #include <stdio.h>
                                          01
02
                                          02
03
    void helloCPU()
                                              void helloCPU()
                                          03
04
                                          04
05
      printf("Hello from the CPU.\n");
                                          05
                                                printf("Hello from the CPU.\n");
06
                                          06
07
                                          07
08
      _global___ void helloGPU()
                                          80
                                                _global___ void helloGPU()
09
                                          09
10
      printf("Hello from the GPU!\n");
                                          10
                                                printf("Hello from the GPU!\n");
11
                                          11
12
                                          12
13
    int main()
                                          13
                                              int main()
14
                                          14
15
                                          15
16
      helloCPU();
                                          16
                                                helloCPU();
17
                                                helloGPU<<<1,1>>>();
                                          17
                                                 cudaDeviceSynchronize();
18
      helloGPU<<<1,1>>>();
                                          18
      cudaDeviceSynchronize();
19
                                          19
                                                helloCPU();
20
                                          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.

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



Exercise 2: Launch Parallel Kernels

```
#include <stdio.h>
                                                                      Exercise: Launch Parallel Kernels
02
                                                                      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
03
        * Refactor firstParallel so that
04

    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

05
        * it can run on the GPU.
                                                                         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

06
        */
                                                                         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
07
                                                                         message printed 25 times now after compiling and running.
       void firstParallel()
09
          printf("This should be running in parallel.\n");
10
11
12
       int main()
14
15
            * Refactor this call to firstParallel
16
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 }
```



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

```
#include <stdio.h>
02
03
     * Refactor firstParallel so that
04
05
     * it can run on the GPU.
06
07
   void firstParallel()
09
10
      printf("This should run in parallel.\n");
11
12
13
    int main()
14
15
       * Refactor this call to firstParallel
16
17
       * to execute in parallel on the GPU.
18
       */
19
      firstParallel();
20
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 }
```

```
#include <stdio.h>
02
03 /*
     * Refactor firstParallel so that
94
     * it can run on the GPU.
05
06
     */
07
80
    global void firstParallel()
09 {
      printf("This should run in parallel.\n");
10
11
12
   int main()
13
14
15
      /*
       * Refactor this call to firstParallel
16
       * to execute in parallel on the GPU.
17
18
       */
19
      firstParallel<<<1,1>>>();
20
21
      cudaDeviceSynchronize();
22
23
       * Some code is needed below so that
       * the CPU will wait for the GPU kernels
24
       * to complete before proceeding.
25
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.

```
#include <stdio.h>
02
03
     * Refactor firstParallel so that
04
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
       * Refactor this call to firstParallel
16
17
       * to execute in parallel on the GPU.
18
       */
19
      firstParallel<<<1,1>>>();
20
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 }
```

```
#include <stdio.h>
02
03
   /*
     * Refactor firstParallel so that
04
     * it can run on the GPU.
05
     */
06
07
80
      global void firstParallel()
09
      printf("This should run in parallel.\n");
10
11
12
    int main()
13
14
15
      /*
16
       * Refactor this call to firstParallel
       * to execute in parallel on the GPU.
17
18
       */
19
      firstParallel<<<1,15>>>();
20
21
      cudaDeviceSynchronize();
22
23
       * Some code is needed below so that
       * the CPU will wait for the GPU kernels
24
       * to complete before proceeding.
25
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.

```
#include <stdio.h>
                                                          #include <stdio.h>
02
                                                      02
03
                                                      03
                                                          /*
     * Refactor firstParallel so that
04
                                                           * Refactor firstParallel so that
                                                      04
05
     * it can run on the GPU.
                                                           * it can run on the GPU.
                                                      05
06
                                                           */
                                                      06
07
                                                      07
08
      global void firstParallel()
                                                      80
                                                            global void firstParallel()
09
                                                      09
10
      printf("This should run in parallel.\n");
                                                            printf("This should run in parallel.\n");
                                                      10
11
                                                      11
12
                                                      12
13
    int main()
                                                          int main()
                                                      13
14
                                                      14
15
                                                      15
                                                            /*
       * Refactor this call to firstParallel
16
                                                      16
                                                             * Refactor this call to firstParallel
17
       * to execute in parallel on the GPU.
                                                             * to execute in parallel on the GPU.
                                                      17
18
       */
                                                      18
                                                             */
19
                                                      19
      firstParallel<<<1,5>>>();
20
                                                            firstParallel<<<15,5>>();
                                                      20
21
                                                      21
                                                            cudaDeviceSynchronize();
22
                                                      22
23
       * Some code is needed below so that
                                                      23
                                                             * Some code is needed below so that
24
       * the CPU will wait for the GPU kernels
                                                             * the CPU will wait for the GPU kernels
                                                      24
25
       * to complete before proceeding.
                                                             * to complete before proceeding.
                                                      25
26
       */
                                                      26
                                                             */
27
                                                      27
28 }
                                                      28 }
```



Exercise 3: Use Specific Thread and Block Indices

```
Exercise: Use Specific Thread and Block Indices
      #include <stdio.h>
                                                             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
02
                                                             work. Refer to the solution if you get stuck.
03
        _global__ void printSuccess()
04
05
06
         if (threadIdx.x == 1023 && blockIdx.x == 255)
07
             printf("Success!\n");
08
         } else {
09
             printf("Failure. Update the execution configuration as necessary.\n");
10
11
12
13
14
      int main()
15
16
           * Update the execution configuration so that the kernel
17
           * will print `"Success!"`.
18
19
20
         printSuccess<<<1,1>>>();
21
22
```



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.

```
#include <stdio.h>
02
                                                  02
03
      global void printSuccess()
                                                  03
04
                                                  04
05
                                                  05
06
      if (threadIdx.x == 1023 &&
                                                  06
07
          blockIdx.x == 255)
                                                  07
98
                                                  80
        printf("Success!\n");
09
                                                  09
      } else {
10
                                                  10
        printf("Failure.\n");
11
                                                  11
12
                                                  12
13
                                                  13
14
                                                  14
15
    int main()
                                                  15
16
                                                  16
17
      /*
                                                  17
18
       * Update the execution
                                                  18
       * configuration so that
19
                                                  19
       * the kernel will print
20
                                                  20
       * `"Success!"`.
21
                                                  21
22
                                                         */
                                                  22
23
                                                  23
24
      printSuccess<<<1,1>>>();
                                                  24
25
                                                  25
                                                  26
```

```
#include <stdio.h>
 global void printSuccess()
  if (threadIdx.x == 1023 &&
      blockIdx.x == 255)
    printf("Success!\n");
  } else {
    printf("Failure.\n");
int main()
   * Update the execution
   * configuration so that
   * the kernel will print
   * `"Success!"`.
  printSuccess<<<1,1256,1024>>>();
  cudaDeviceSynchronize();
```



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

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

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

Currently, the loop function inside <a href="https://doi.org/10.25/10.25/20.25

```
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
    void loop(int N)
03
04
05
      for (int i = 0; i < N; ++i)
06
        printf("Iteration %d\n", i);
07
08
09
10
11
    int main()
12
13
      int N = 10;
      loop(N);
14
15
```

```
#include <stdio.h>
01
02
03
    __global__ void loop(int N)
04
      int i = blockIdx.x*blockDim.x
05
06
            + threadIdx.x:
07
      for (int i = 0; i < N; ++i)
80
        printf("Iteration %d\n", i);
09
10
11
12
    int main()
13
14
15
      int N = 10;
16
      loop<<<1,N>>>(N);
17
      cudaDeviceSynchronize();
18
```



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

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

```
#include <stdio.h>
01
02
                                         02
    void loop(int N)
03
                                         03
04
                                         04
05
      for (int i = 0; i < N; ++i)
                                         05
06
                                         06
        printf("Iteration %d\n", i);
07
                                         07
08
                                         80
09
                                         09
10
                                         10
11
    int main()
                                         11
12
                                         12
13
      int N = 10;
                                         13
      loop(N);
14
                                         14
15
                                         15
                                         16
                                         17
                                         18
```

```
#include <stdio.h>
__global__ void loop(int N)
  int i = blockIdx.x*blockDim.x
        + threadIdx.x:
  for (int i = 0; i < N; ++i)
    printf("Iteration %d\n", i);
int main()
  int N = 10;
  loop<<<2,N/2>>>(N);
  cudaDeviceSynchronize();
```



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)</pre>
   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)</pre>
    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) (0,0)
                                                                             (0,ni)
  for ( int j=1; j < nj-1; j++ ) {
                                                                i 0m1
                                                             im10 i00 ip10
    for ( int i=1; i < ni-1; i++ ) {
                                                                          (i-1,j)
                                                                      (i,j-1)(i,j)(i,j+1)
      // find indices into linear memory
                                                                         (i+1,j)
      // for central point and neighbours
                                                       (nj,0)
                                                                             (nj,ni)
      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
                                                        // i0p1 = (j+1)*ni + i
      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);
```



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;</pre>
    for ( int i=1; i < ni-1; i++ ) { i = blockIdx.y * blockDim.y + threadIdx.y;</pre>
      if (((j>0) && (i>0)) && ((j<nj-1) && (i<ni-1))) {
      int i00 = I2D(ni, i, j);
                                                 // i00 = j * ni + i
                                                   // im10 = j * ni + i-1
      int im10 = I2D(ni, i-1, j);
      int ip10 = I2D(ni, i+1, j);
                                                  // ip10 = j * ni + i+1
      int i0m1 = I2D(ni, i, j-1);
                                                   // i0m1 = (j-1)*ni + i
                                                      // i0p1 = (j+1)*ni + i
      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);
   }}
  }}
```



Producto de matrices en CPU

- \circ 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;
   }
}</pre>
```



Versión CUDA para el producto de matrices

```
B
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];
                                                            C
    sum += a*b;
 C[i*N + j] = sum;
```