GRID

Grupo de Investigación en Redes, Información y Distribución

Programación para Unidades de Procesamiento Gráfico de Propósito General

Una oportunidad para el cálculo científico

(Parte III)

JULIAN ESTEBAN GUTIERREZ POSADA
CHRISTIAN ANDRÉS CANDELA URIBE
LUIS EDUARDO SEPÚLVEDA RODRÍGUEZ

Cali, agosto 2016

UNIVERSIDAD DEL QUINDÍO

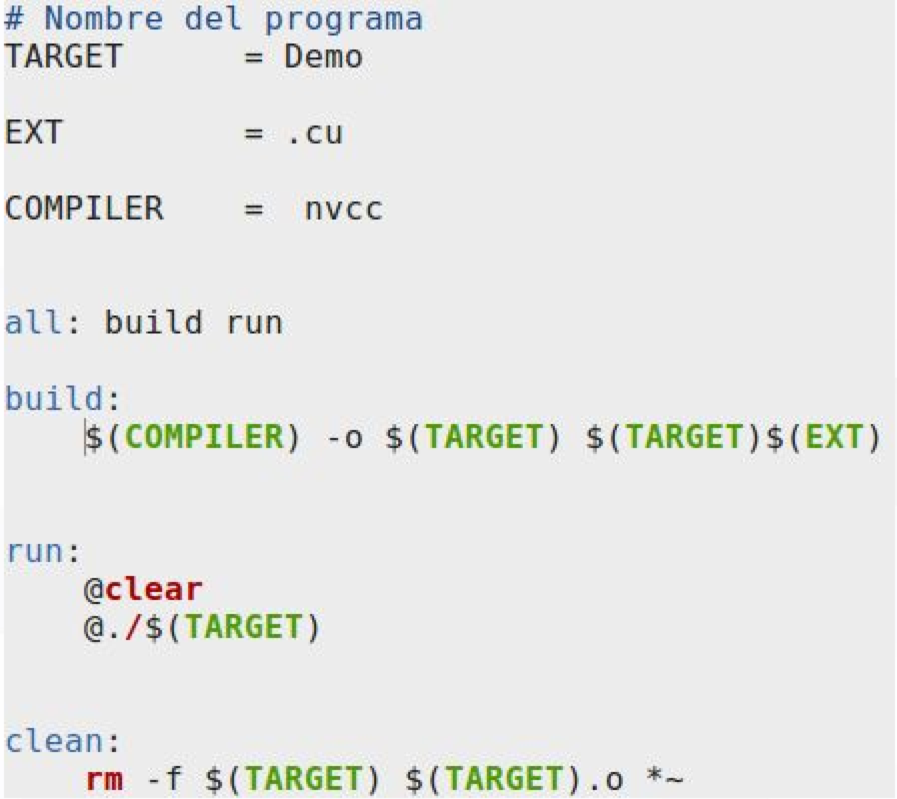
Universidad del Valle

Contenido

- Makefile
- Transpuesta de una matriz
- Producto de dos matrices (dos versiones)

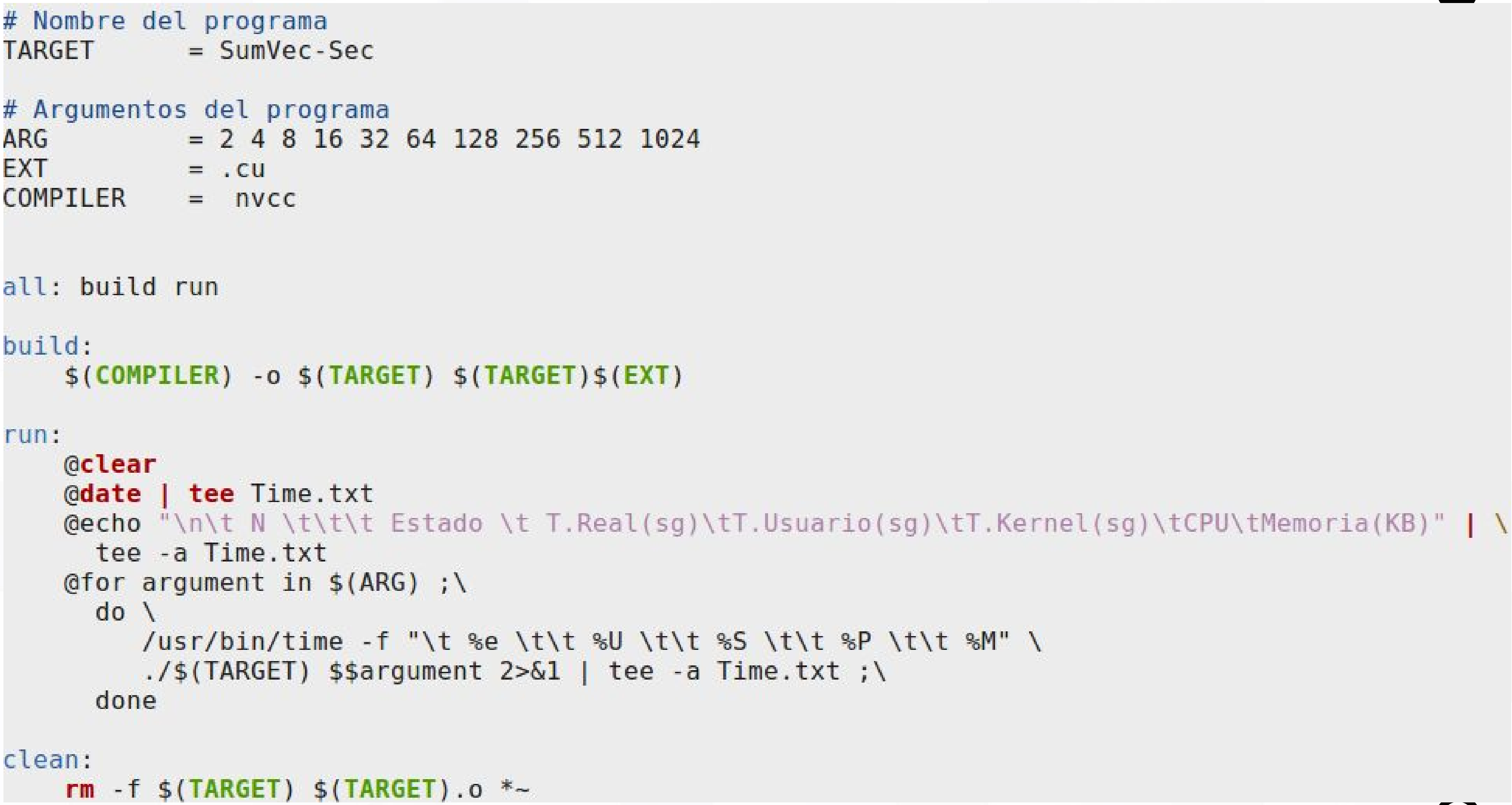
Makefile

```
make

make all

make build

make run

make clean
```



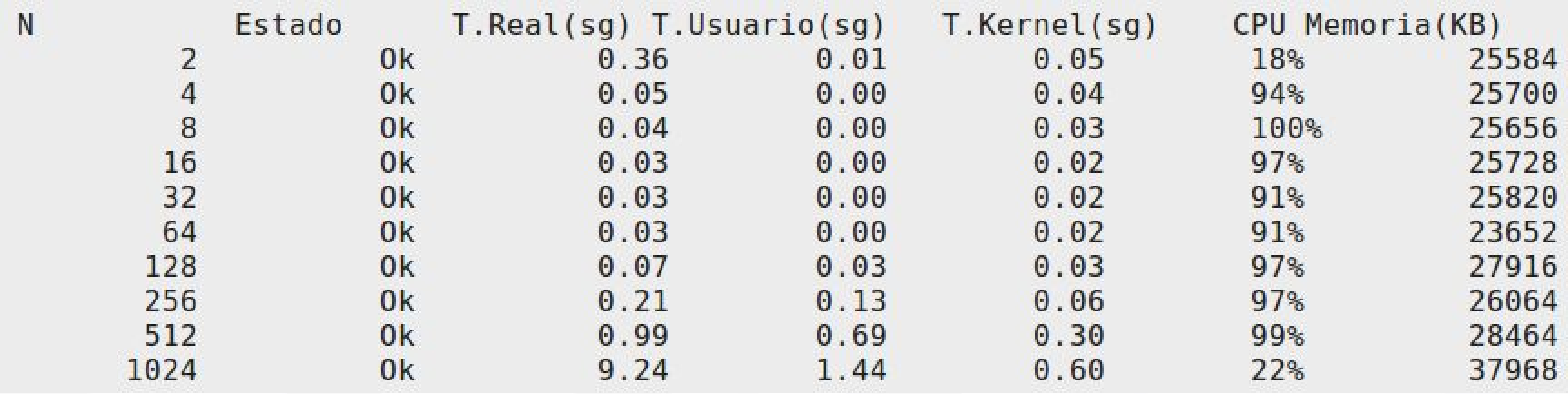
```
# Nombre del programa
TARGET      = Demo

EXT         = .cu

COMPILER    =  nvcc


all: build run

build:
    $(COMPILER) -o $(TARGET) $(TARGET)$(EXT)


run:
    @clear
    @./$(TARGET)


clean:
    rm -f $(TARGET) $(TARGET).o *~
```

Makefile



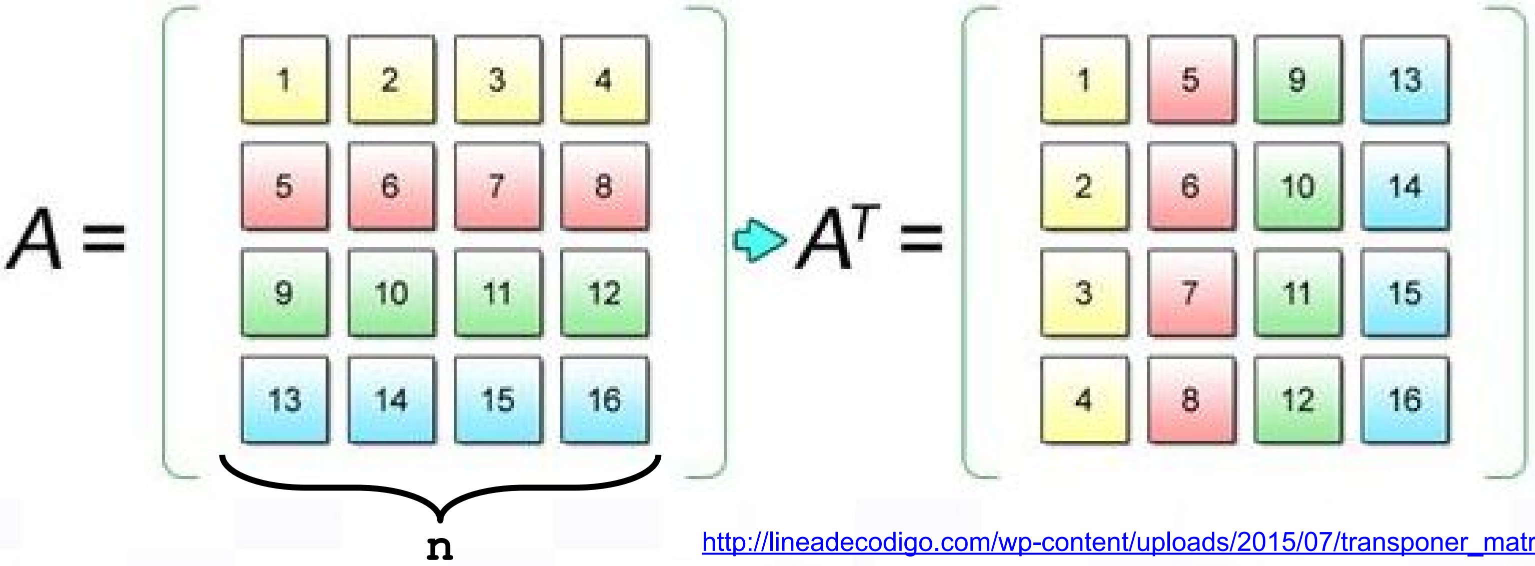
```
# Nombre del programa
TARGET      = SumVec-Sec

# Argumentos del programa
ARG         = 2 4 8 16 32 64 128 256 512 1024
EXT         = .cu
COMPILER    =  nvcc


all: build run

build:
    $(COMPILER) -o $(TARGET) $(TARGET)$(EXT)

run:
    @clear
    @date | tee Time.txt
    @echo "\n\t N \t\t\t Estado \t T.Real(sg)\tT.Usuario(sg)\tT.Kernel(sg)\tCPU\tMemoria(KB)" | \
      tee -a Time.txt
    @for argument in $(ARG) ;\
      do \
         /usr/bin/time -f "\t %e \t\t %U \t\t %S \t\t %P \t\t %M" \
         ./$(TARGET) $$argument 2>&1 | tee -a Time.txt ;\
      done

clean:
    rm -f $(TARGET) $(TARGET).o *~
```

Makefile - Salida



```
N           Estado      T.Real(sg) T.Usuario(sg)   T.Kernel(sg)     CPU Memoria(KB)
        2           Ok        0.36         0.01         0.05         18%        25584
        4           Ok        0.05         0.00         0.04         94%        25700
        8           Ok        0.04         0.00         0.03         100%       25656
       16           Ok        0.03         0.00         0.02         97%        25728
       32           Ok        0.03         0.00         0.02         91%        25820
       64           Ok        0.03         0.00         0.02         91%        23652
      128           Ok        0.07         0.03         0.03         97%        27916
      256           Ok        0.21         0.13         0.06         97%        26064
      512           Ok        0.99         0.69         0.30         99%        28464
     1024           Ok        9.24         1.44         0.60         22%        37968
```

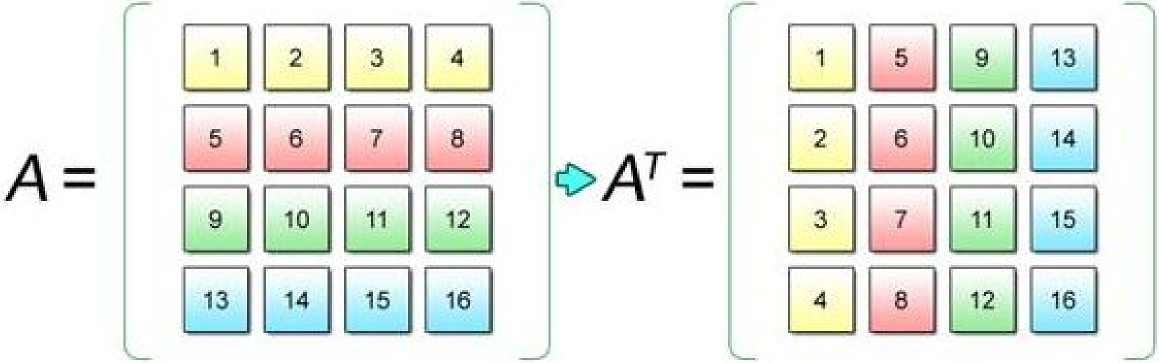
Transpuesta de matriz



$$A = \underbrace{\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}}_{\mathbf{n}} \Rightarrow A^{T} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

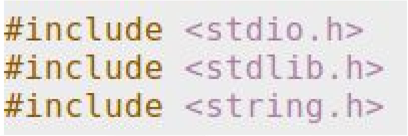
http://lineadecodigo.com/wp-content/uploads/2015/07/transponer_matrices.jpg

`n` -> Es dado como argumento del programa.

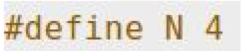


$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

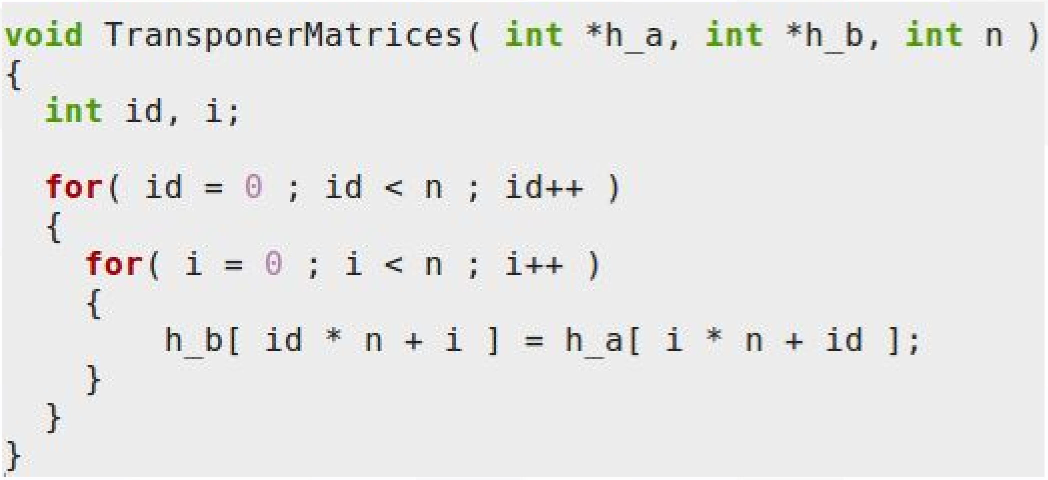
CPU



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



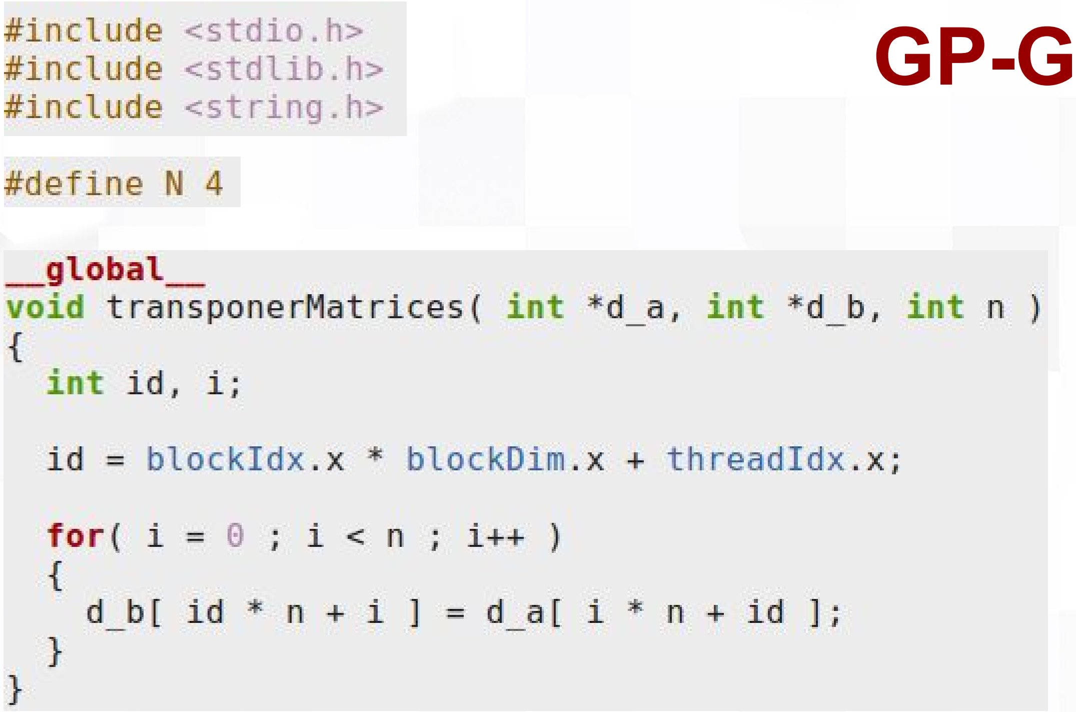
```
#define N 4
```



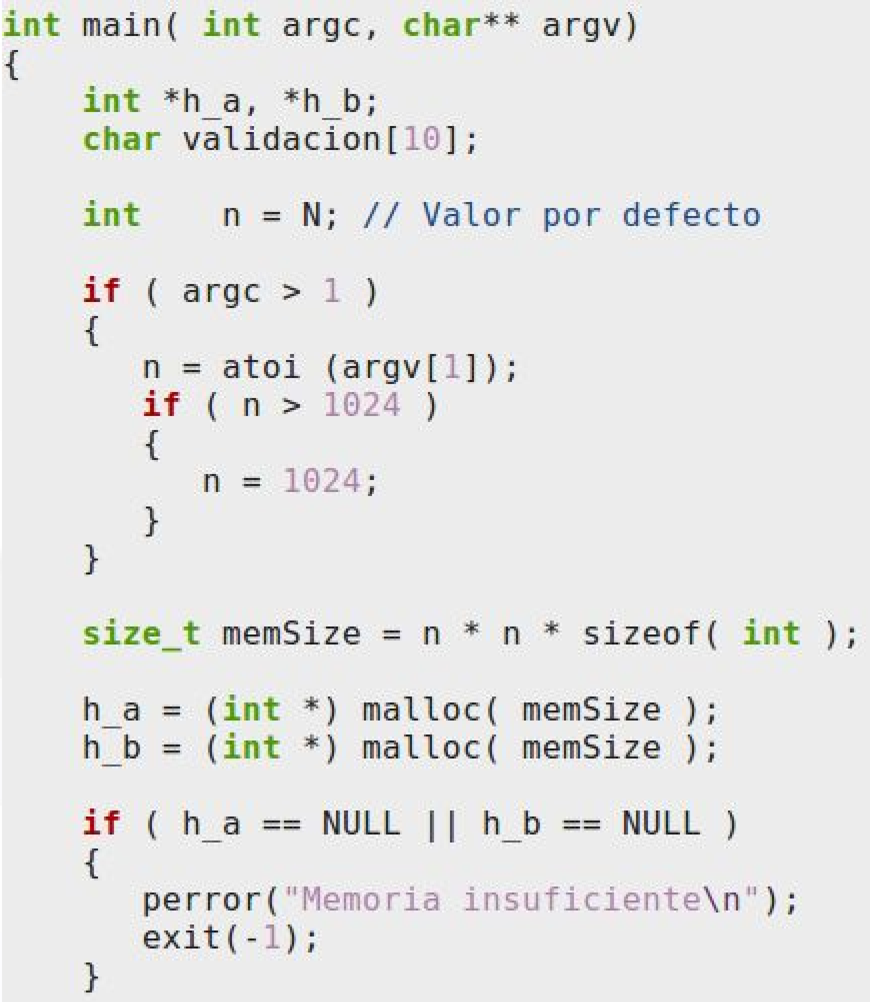
```
void TransponerMatrices( int *h_a, int *h_b, int n )
{
  int id, i;

  for( id = 0 ; id < n ; id++ )
  {
    for( i = 0 ; i < n ; i++ )
    {
        h_b[ id * n + i ] = h_a[ i * n + id ];
    }
  }
}
```

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

GP-GPU



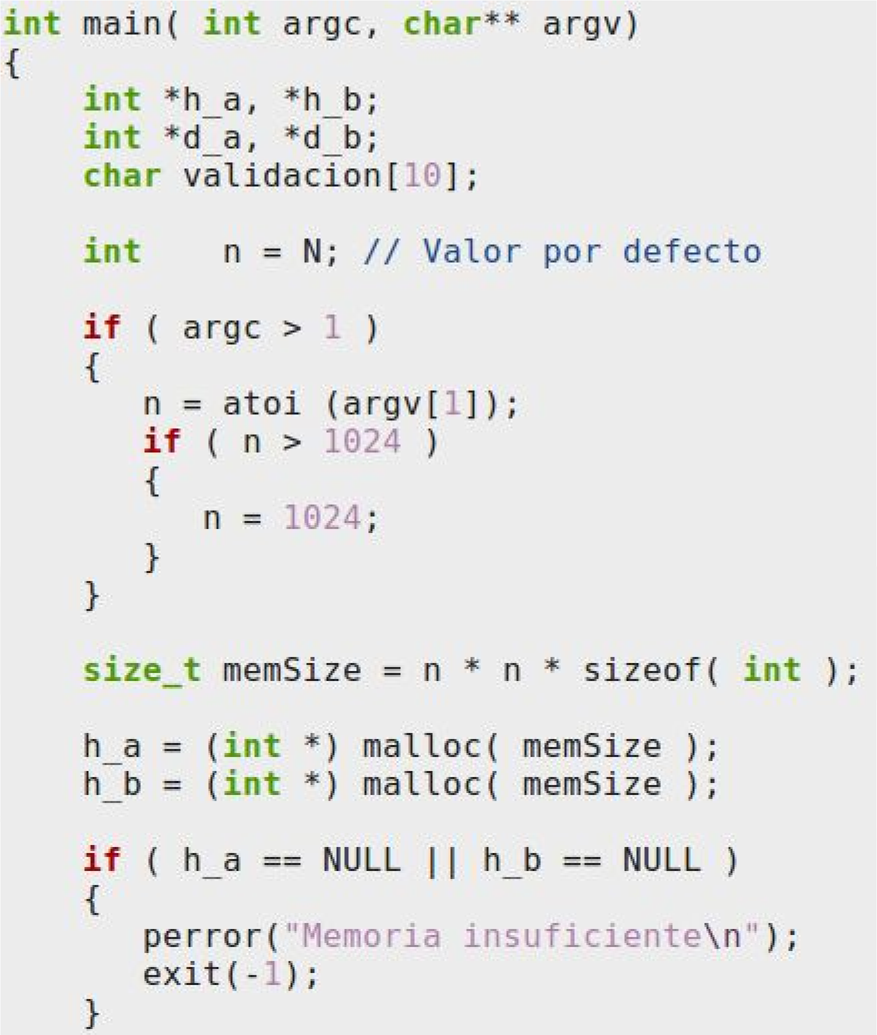
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 4

__global__
void transponerMatrices( int *d_a, int *d_b, int n )
{
  int id, i;

  id = blockIdx.x * blockDim.x + threadIdx.x;

  for( i = 0 ; i < n ; i++ )
  {
    d_b[ id * n + i ] = d_a[ i * n + id ];
  }
}
```



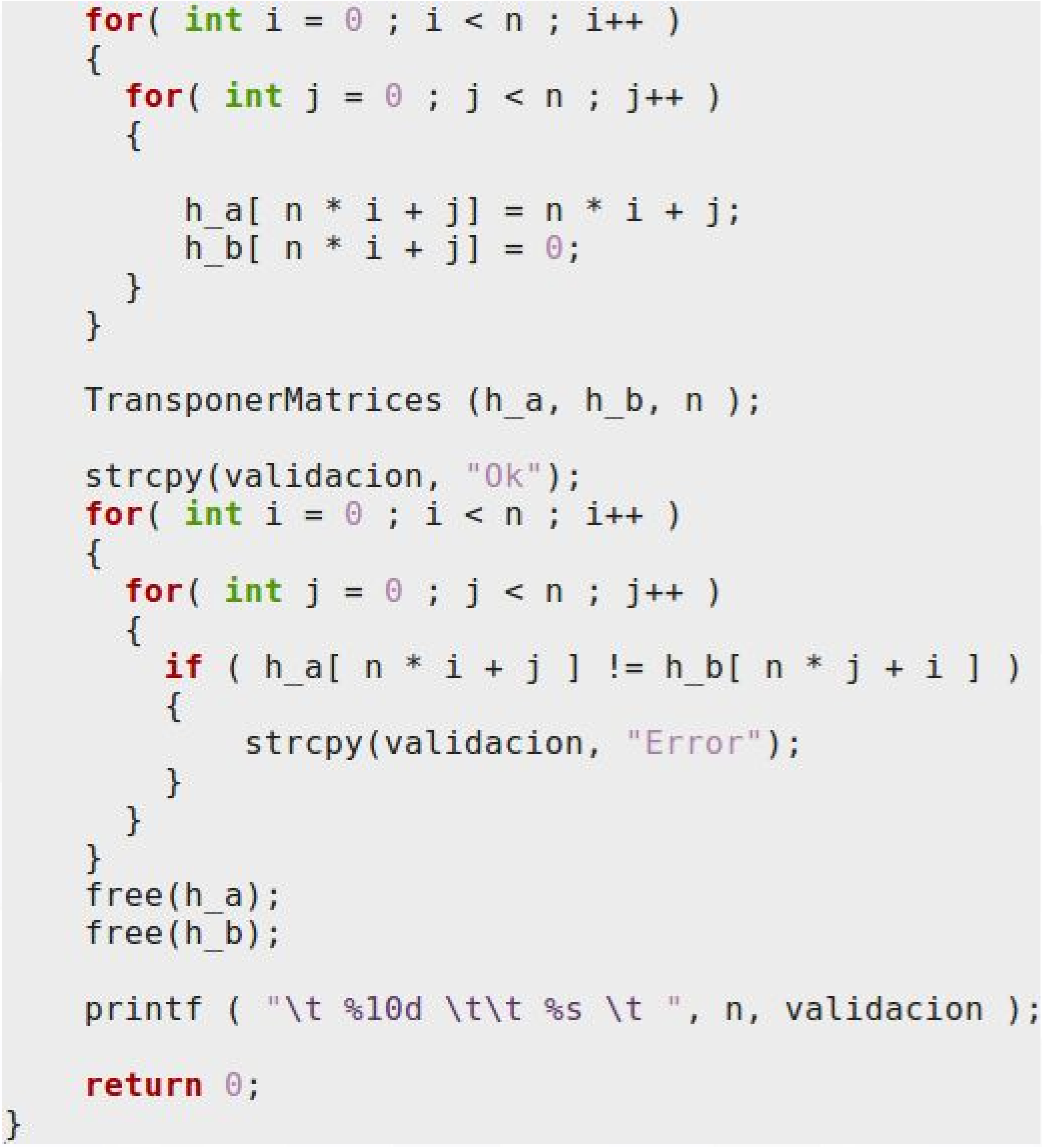
```
int main( int argc, char** argv)
{
    int *h_a, *h_b;
    char validacion[10];

    int     n = N; // Valor por defecto

    if ( argc > 1 )
    {
       n = atoi (argv[1]);
       if ( n > 1024 )
       {
          n = 1024;
       }
    }

    size_t memSize = n * n * sizeof( int );

    h_a = (int *) malloc( memSize );
    h_b = (int *) malloc( memSize );

    if ( h_a == NULL || h_b == NULL )
    {
       perror("Memoria insuficiente\n");
       exit(-1);
    }
```

CPU

GP-GPU



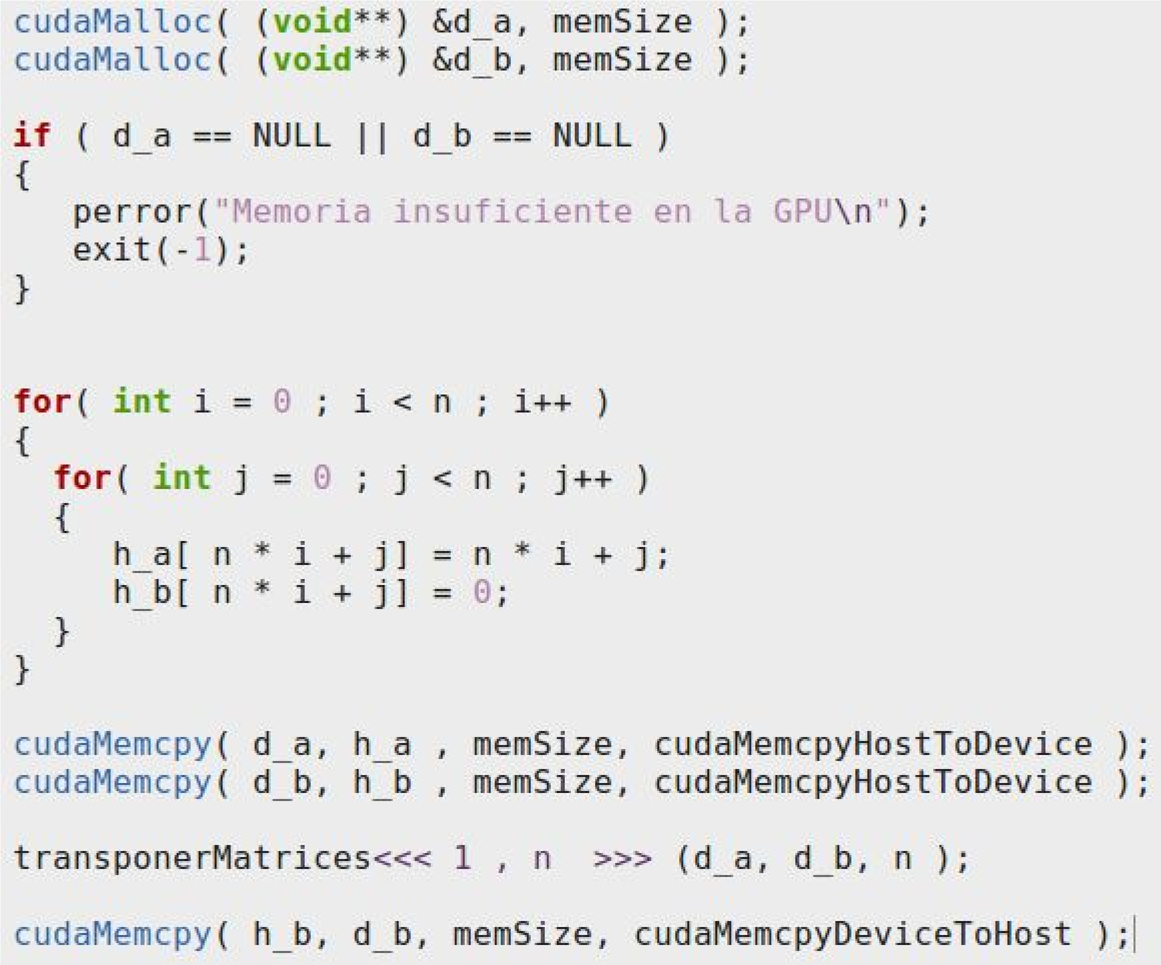
```
int main( int argc, char** argv)
{
    int *h_a, *h_b;
    int *d_a, *d_b;
    char validacion[10];

    int    n = N; // Valor por defecto

    if ( argc > 1 )
    {
       n = atoi (argv[1]);
       if ( n > 1024 )
       {
          n = 1024;
       }
    }

    size_t memSize = n * n * sizeof( int );

    h_a = (int *) malloc( memSize );
    h_b = (int *) malloc( memSize );

    if ( h_a == NULL || h_b == NULL )
    {
       perror("Memoria insuficiente\n");
       exit(-1);
    }
```



```
    for( int i = 0 ; i < n ; i++ )
    {
      for( int j = 0 ; j < n ; j++ )
      {

         h_a[ n * i + j] = n * i + j;
         h_b[ n * i + j] = 0;
      }
    }

    TransponerMatrices (h_a, h_b, n );

    strcpy(validacion, "Ok");
    for( int i = 0 ; i < n ; i++ )
    {
      for( int j = 0 ; j < n ; j++ )
      {
        if ( h_a[ n * i + j ] != h_b[ n * j + i ] )
        {
            strcpy(validacion, "Error");
        }
      }
    }
    free(h_a);
    free(h_b);

    printf ( "\t %10d \t\t %s \t ", n, validacion );

    return 0;
}
```

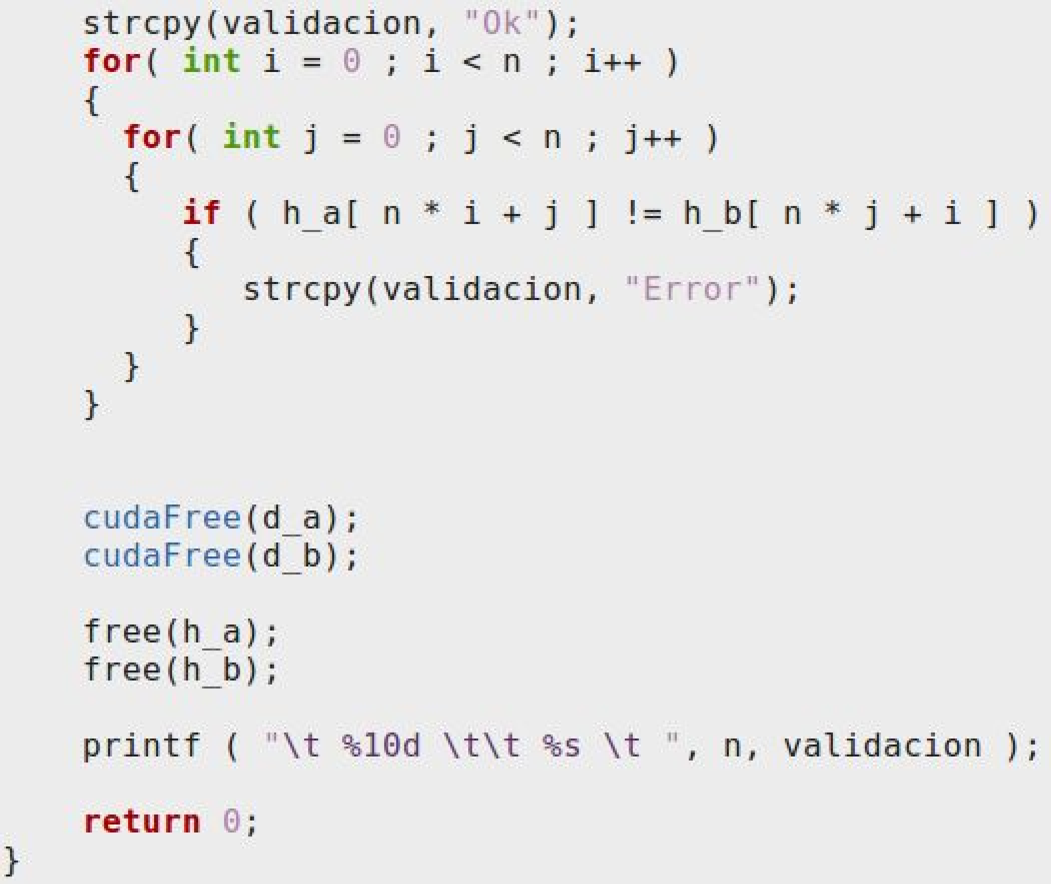
CPU

GP-GPU



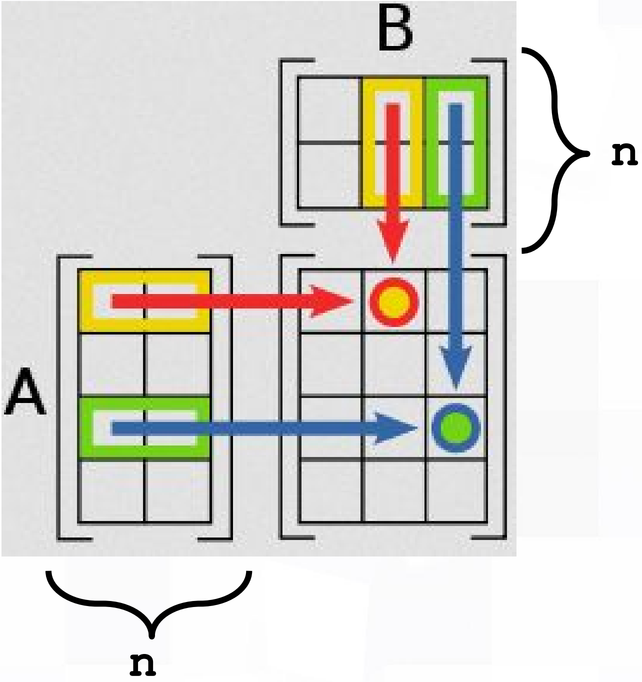
```
cudaMalloc( (void**) &d_a, memSize );
cudaMalloc( (void**) &d_b, memSize );

if ( d_a == NULL || d_b == NULL )
{
   perror("Memoria insuficiente en la GPU\n");
   exit(-1);
}


for( int i = 0 ; i < n ; i++ )
{
  for( int j = 0 ; j < n ; j++ )
  {
     h_a[ n * i + j] = n * i + j;
     h_b[ n * i + j] = 0;
  }
}

cudaMemcpy( d_a, h_a , memSize, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, h_b , memSize, cudaMemcpyHostToDevice );

transponerMatrices<<< 1 , n  >>> (d_a, d_b, n );

cudaMemcpy( h_b, d_b, memSize, cudaMemcpyDeviceToHost );
```

GP-GPU



```
    strcpy(validacion, "Ok");
    for( int i = 0 ; i < n ; i++ )
    {
      for( int j = 0 ; j < n ; j++ )
      {
         if ( h_a[ n * i + j ] != h_b[ n * j + i ] )
         {
            strcpy(validacion, "Error");
         }
      }
    }


    cudaFree(d_a);
    cudaFree(d_b);

    free(h_a);
    free(h_b);

    printf ( "\t %10d \t\t %s \t ", n, validacion );

    return 0;
}
```

Producto de matrices - v1

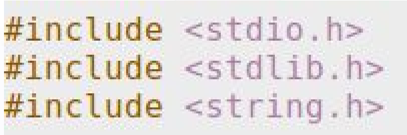


https://upload.wikimedia.org/wikipedia/commons/thumb/1/11/Matrix_multiplication_diagram.svg/250px-Matrix_multiplication_diagram.svg.png

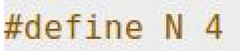
`n` -> Es dado como argumento del programa.

Producto de matrices - v1

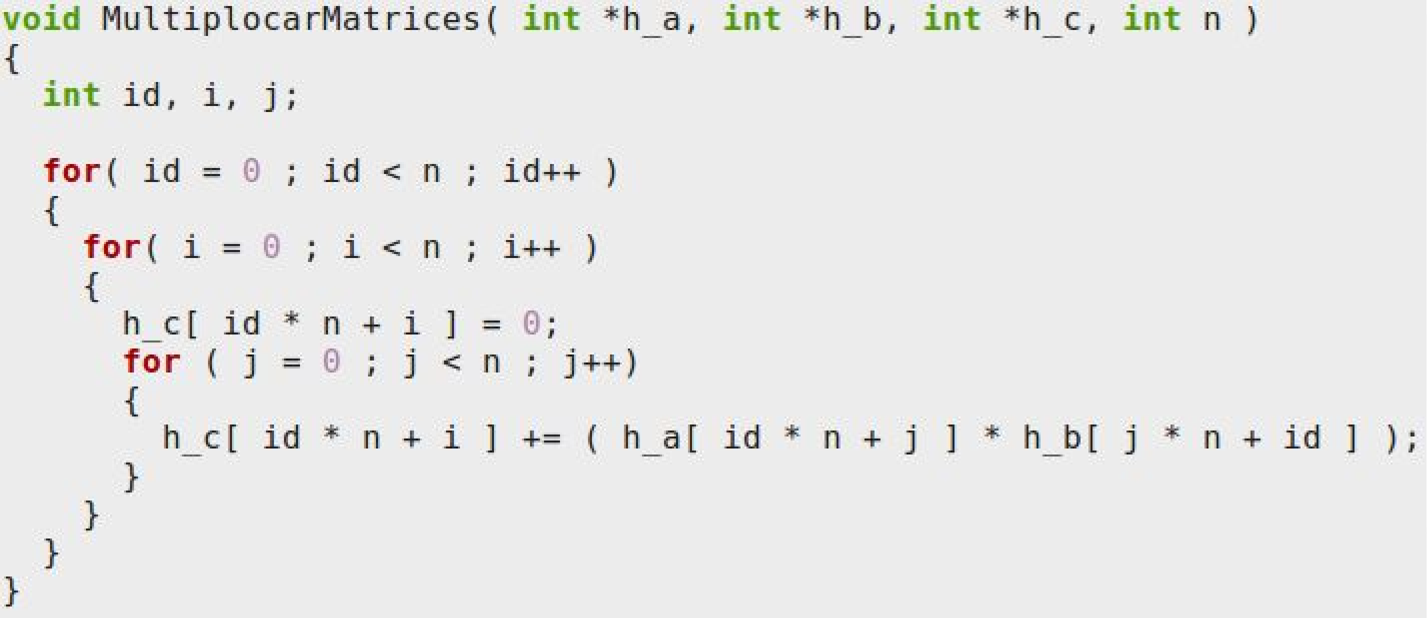
CPU



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



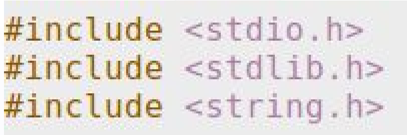
```c
#define N 4
```



```c
void MultiplocarMatrices( int *h_a, int *h_b, int *h_c, int n )
{
  int id, i, j;

  for( id = 0 ; id < n ; id++ )
  {
    for( i = 0 ; i < n ; i++ )
    {
      h_c[ id * n + i ] = 0;
      for ( j = 0 ; j < n ; j++)
      {
        h_c[ id * n + i ] += ( h_a[ id * n + j ] * h_b[ j * n + id ] );
      }
    }
  }
}
```

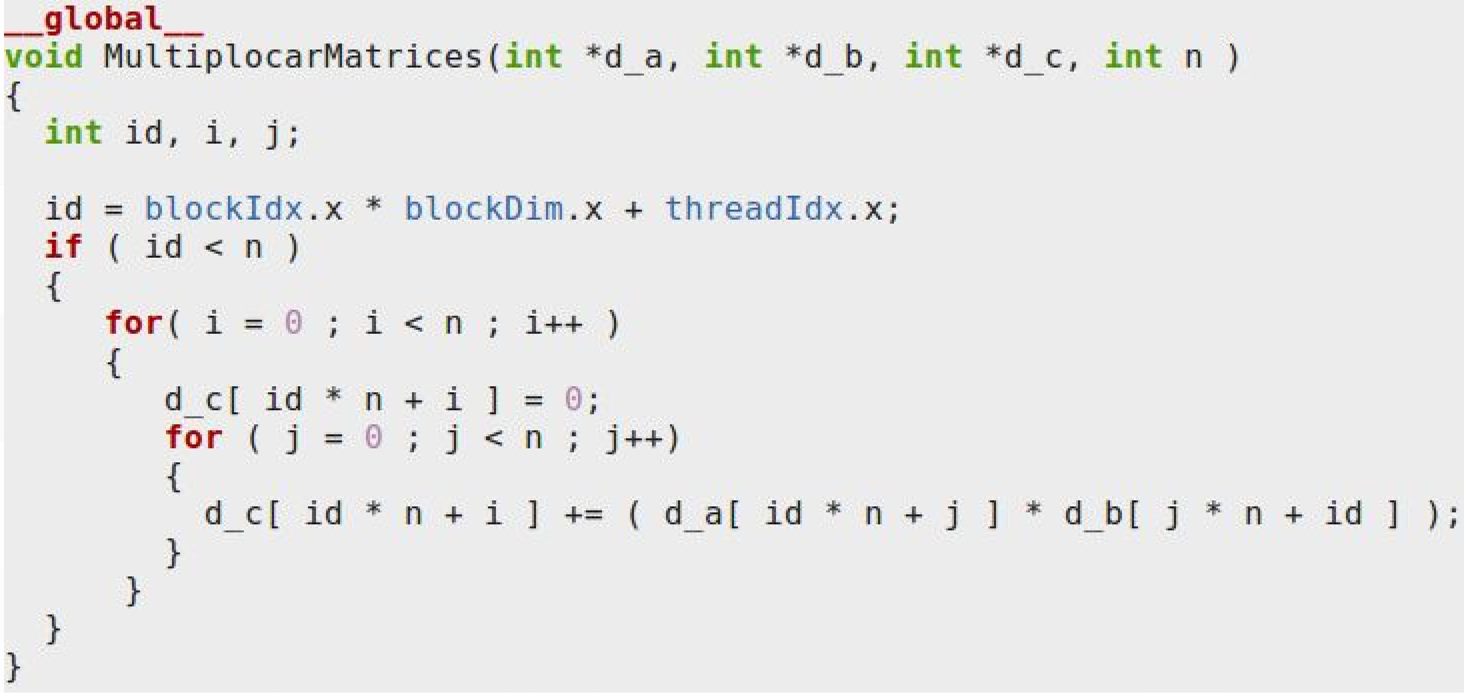
Producto de matrices - v1

GP-GPU



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

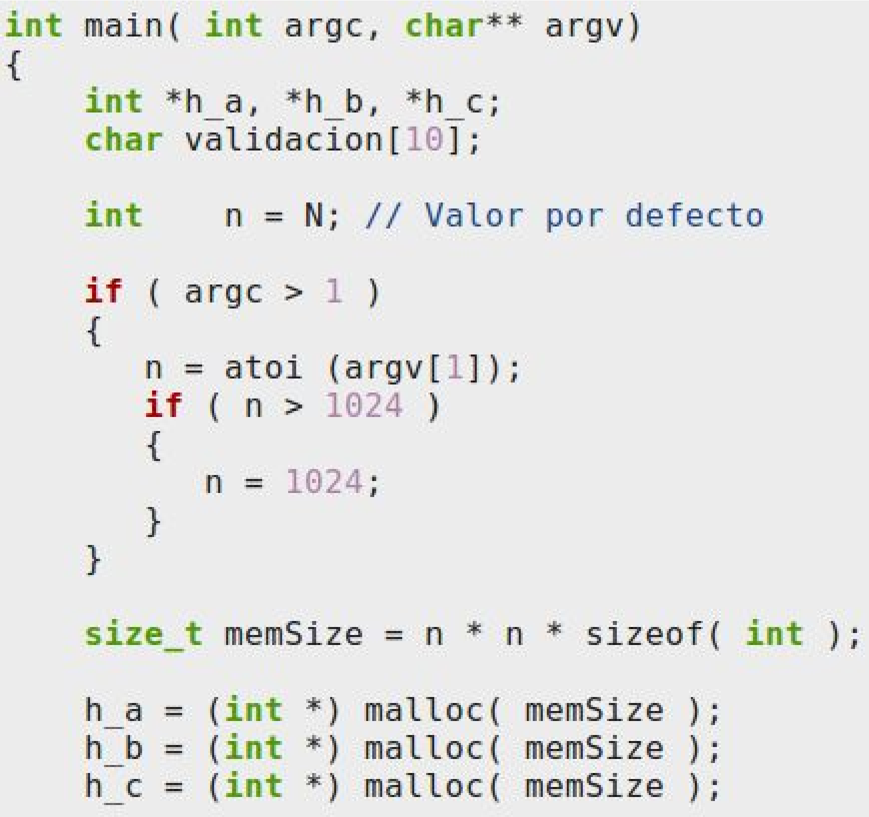
```
#define N 4
```



```
__global__
void MultiplocarMatrices(int *d_a, int *d_b, int *d_c, int n )
{
  int id, i, j;

  id = blockIdx.x * blockDim.x + threadIdx.x;
  if ( id < n )
  {
     for( i = 0 ; i < n ; i++ )
     {
        d_c[ id * n + i ] = 0;
        for ( j = 0 ; j < n ; j++)
        {
          d_c[ id * n + i ] += ( d_a[ id * n + j ] * d_b[ j * n + id ] );
        }
      }
  }
}
```

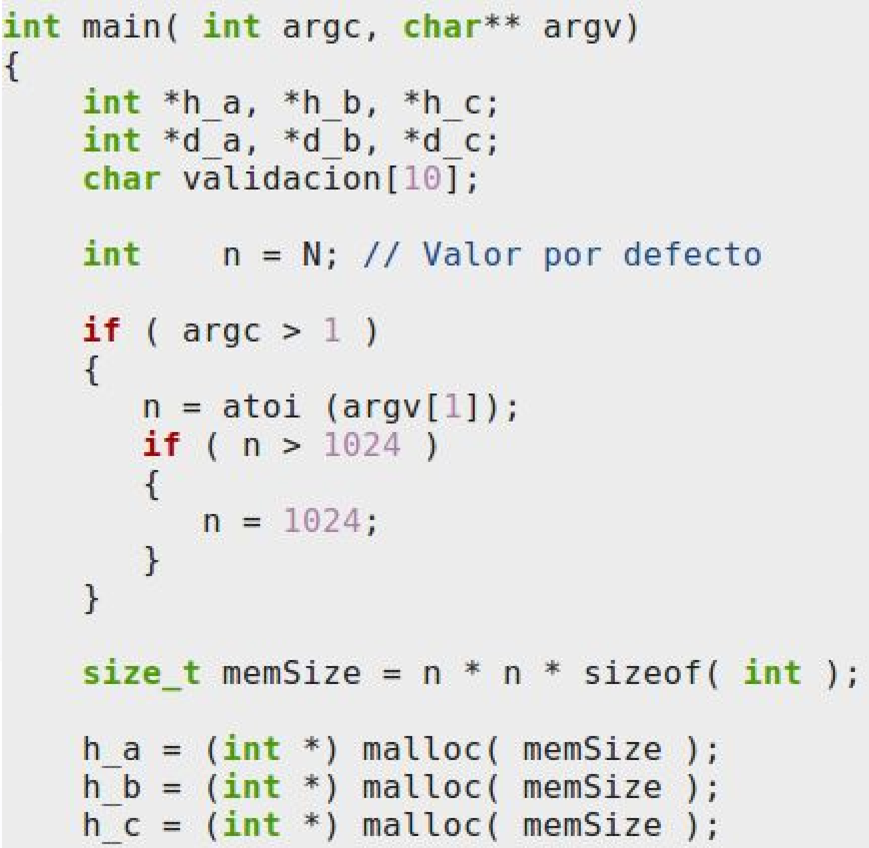
Producto de matrices - v1

CPU



```c
int main( int argc, char** argv)
{
    int *h_a, *h_b, *h_c;
    char validacion[10];

    int     n = N; // Valor por defecto

    if ( argc > 1 )
    {
       n = atoi (argv[1]);
       if ( n > 1024 )
       {
          n = 1024;
       }
    }

    size_t memSize = n * n * sizeof( int );

    h_a = (int *) malloc( memSize );
    h_b = (int *) malloc( memSize );
    h_c = (int *) malloc( memSize );
```

Producto de matrices - v1

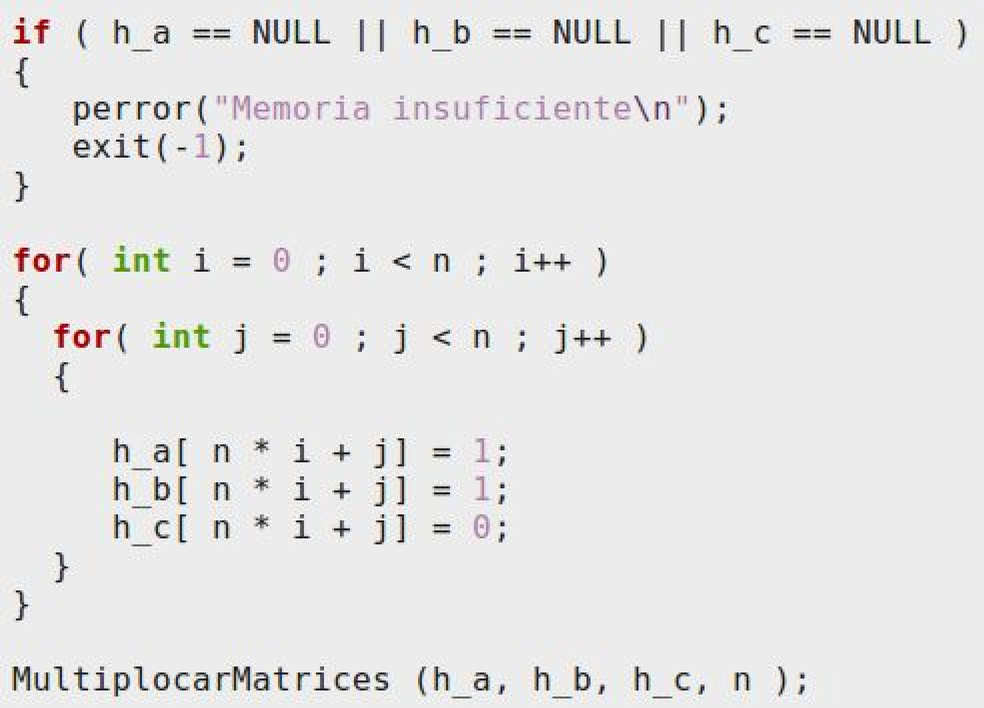


```
int main( int argc, char** argv)
{
    int *h_a, *h_b, *h_c;
    int *d_a, *d_b, *d_c;
    char validacion[10];

    int    n = N; // Valor por defecto

    if ( argc > 1 )
    {
       n = atoi (argv[1]);
       if ( n > 1024 )
       {
          n = 1024;
       }
    }

    size_t memSize = n * n * sizeof( int );

    h_a = (int *) malloc( memSize );
    h_b = (int *) malloc( memSize );
    h_c = (int *) malloc( memSize );
```

GP-GPU

Producto de matrices - v1

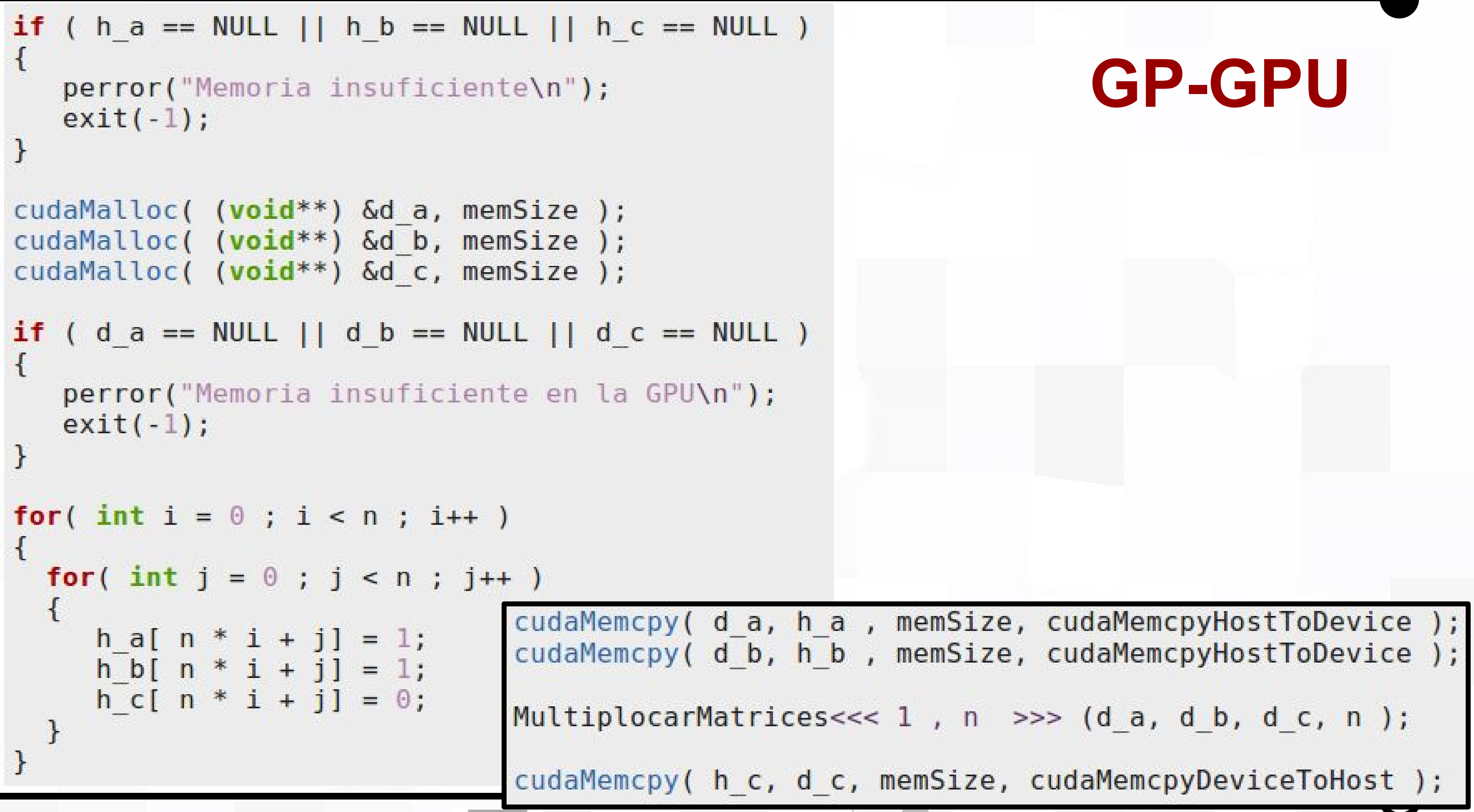
CPU



```
if ( h_a == NULL || h_b == NULL || h_c == NULL )
{
   perror("Memoria insuficiente\n");
   exit(-1);
}

for( int i = 0 ; i < n ; i++ )
{
  for( int j = 0 ; j < n ; j++ )
  {

     h_a[ n * i + j] = 1;
     h_b[ n * i + j] = 1;
     h_c[ n * i + j] = 0;
  }
}

MultiplocarMatrices (h_a, h_b, h_c, n );
```

Producto de matrices - v1

GP-GPU

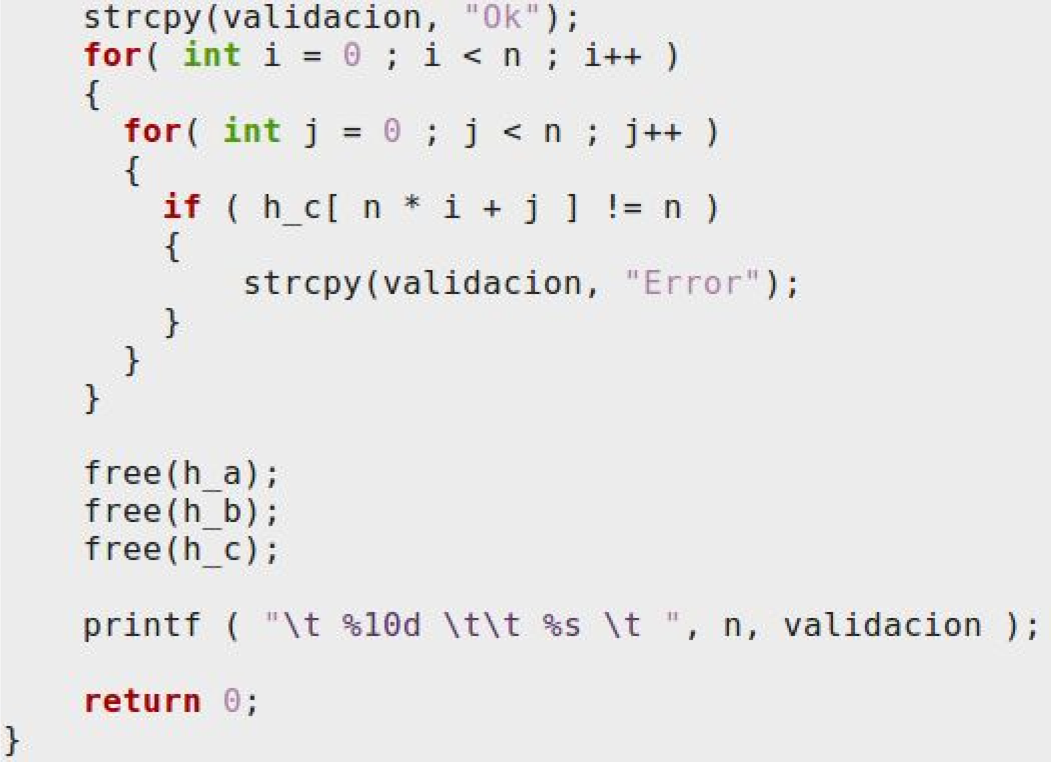


```
if ( h_a == NULL || h_b == NULL || h_c == NULL )
{
   perror("Memoria insuficiente\n");
   exit(-1);
}

cudaMalloc( (void**) &d_a, memSize );
cudaMalloc( (void**) &d_b, memSize );
cudaMalloc( (void**) &d_c, memSize );

if ( d_a == NULL || d_b == NULL || d_c == NULL )
{
   perror("Memoria insuficiente en la GPU\n");
   exit(-1);
}

for( int i = 0 ; i < n ; i++ )
{
  for( int j = 0 ; j < n ; j++ )
  {
     h_a[ n * i + j] = 1;
     h_b[ n * i + j] = 1;
     h_c[ n * i + j] = 0;
  }
}
```

```
cudaMemcpy( d_a, h_a , memSize, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, h_b , memSize, cudaMemcpyHostToDevice );

MultiplocarMatrices<<< 1 , n  >>> (d_a, d_b, d_c, n );

cudaMemcpy( h_c, d_c, memSize, cudaMemcpyDeviceToHost );
```

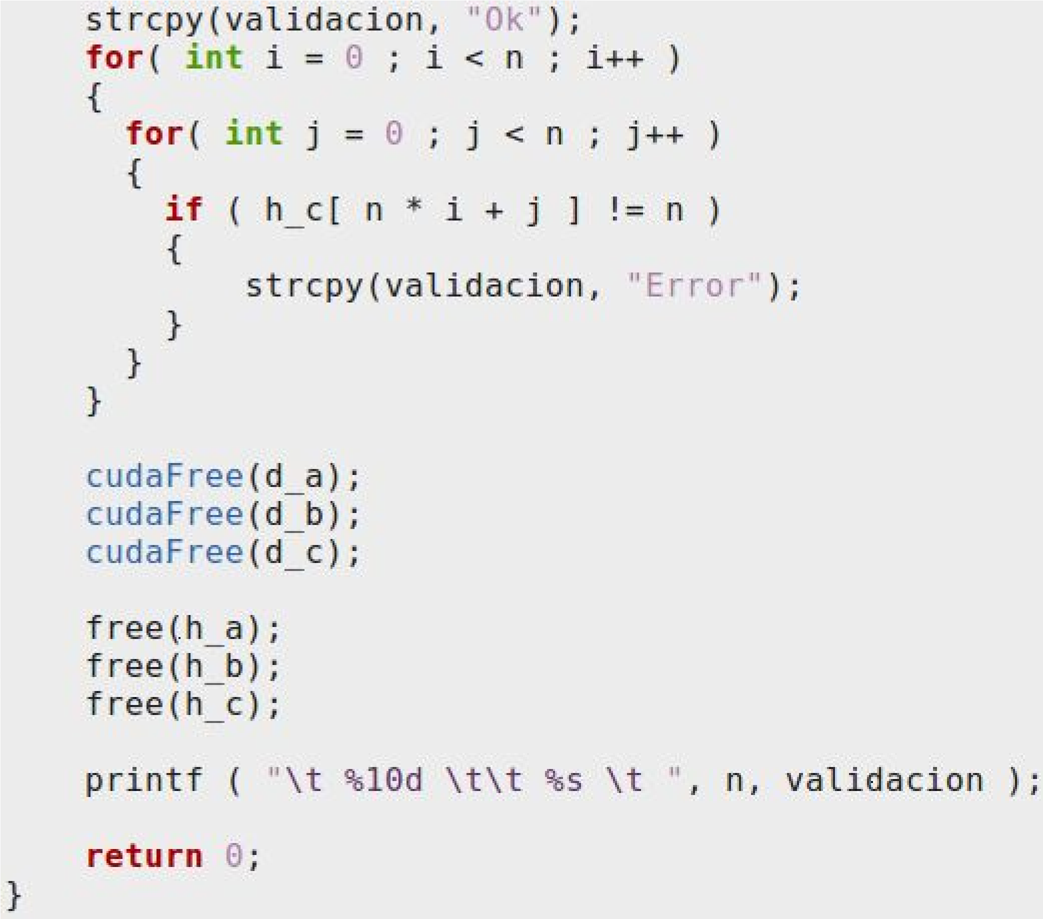
Producto de matrices - v1

CPU



```
    strcpy(validacion, "Ok");
    for( int i = 0 ; i < n ; i++ )
    {
      for( int j = 0 ; j < n ; j++ )
      {
        if ( h_c[ n * i + j ] != n )
        {
            strcpy(validacion, "Error");
        }
      }
    }

    free(h_a);
    free(h_b);
    free(h_c);

    printf ( "\t %10d \t\t %s \t ", n, validacion );

    return 0;
}
```

Producto de matrices - v1

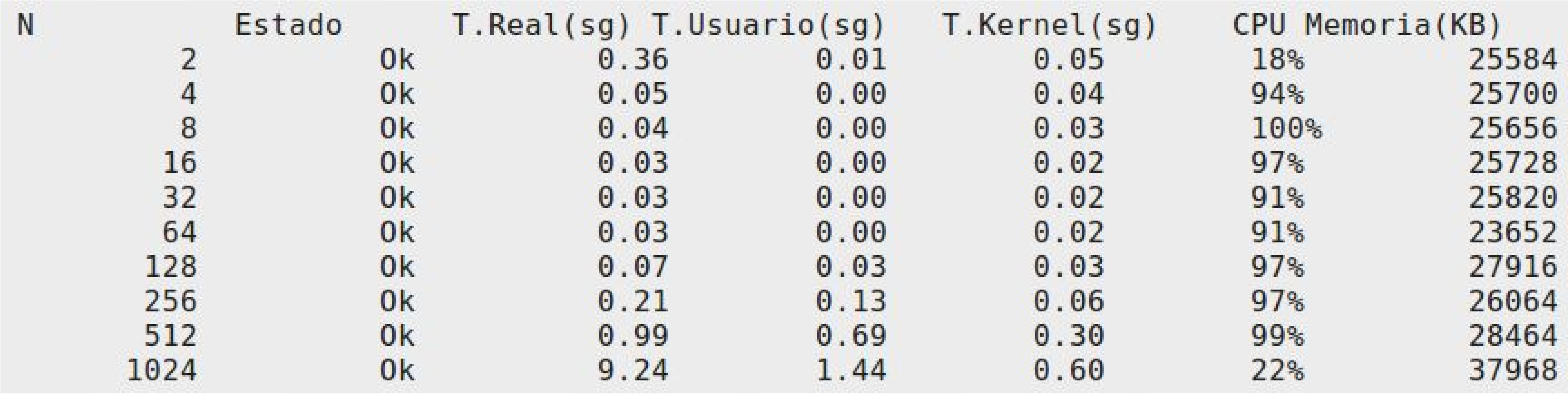


```
    strcpy(validacion, "Ok");
    for( int i = 0 ; i < n ; i++ )
    {
      for( int j = 0 ; j < n ; j++ )
      {
        if ( h_c[ n * i + j ] != n )
        {
            strcpy(validacion, "Error");
        }
      }
    }

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    free(h_a);
    free(h_b);
    free(h_c);

    printf ( "\t %10d \t\t %s \t ", n, validacion );

    return 0;
}
```

GP-GPU

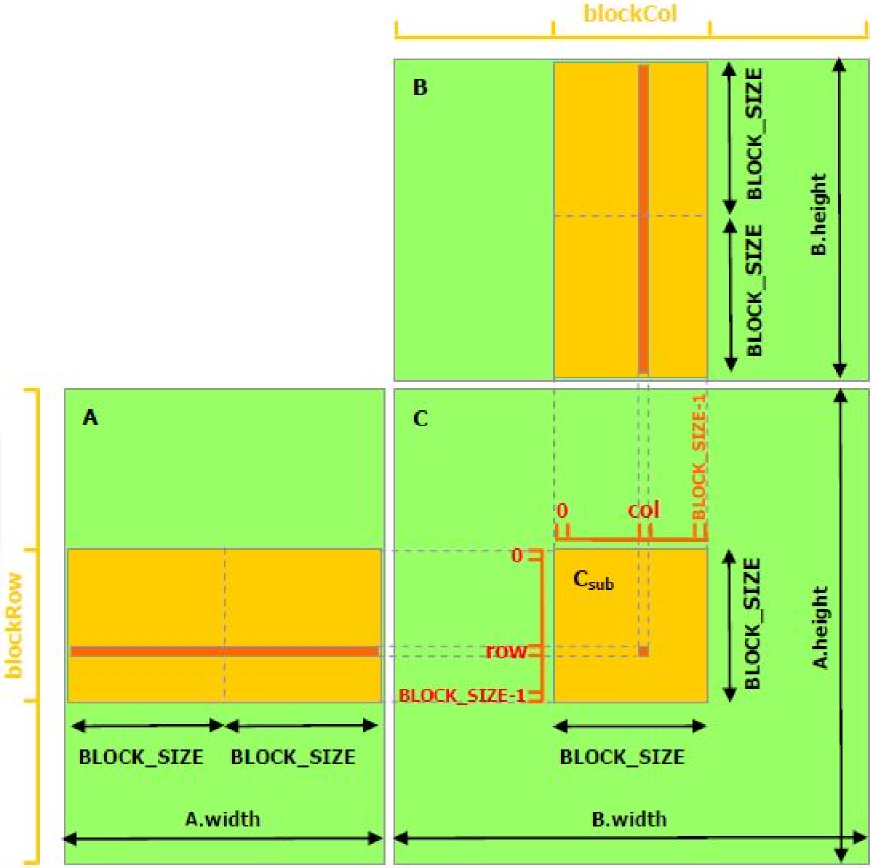
Producto de matrices - v1

GP-GPU



```
N           Estado      T.Real(sg) T.Usuario(sg)   T.Kernel(sg)     CPU Memoria(KB)
         2          Ok        0.36          0.01          0.05      18%          25584
         4          Ok        0.05          0.00          0.04      94%          25700
         8          Ok        0.04          0.00          0.03      100%         25656
        16          Ok        0.03          0.00          0.02      97%          25728
        32          Ok        0.03          0.00          0.02      91%          25820
        64          Ok        0.03          0.00          0.02      91%          23652
       128          Ok        0.07          0.03          0.03      97%          27916
       256          Ok        0.21          0.13          0.06      97%          26064
       512          Ok        0.99          0.69          0.30      99%          28464
      1024          Ok        9.24          1.44          0.60      22%          37968
```

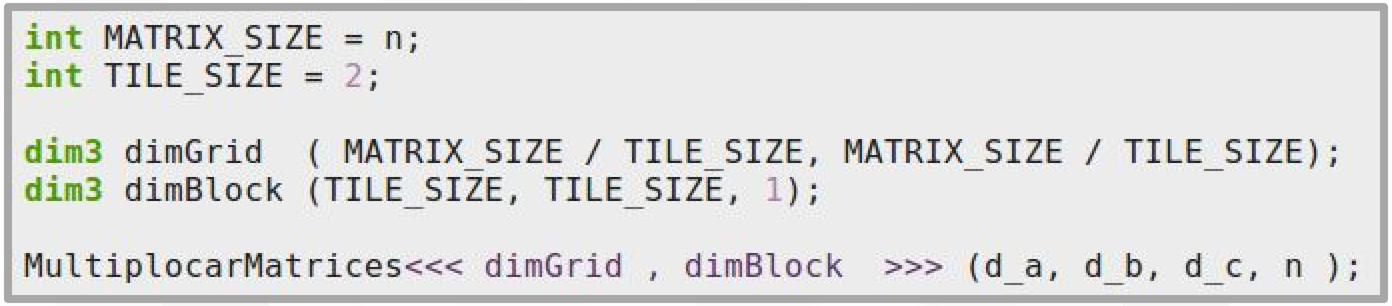
Producto de matrices - v2



https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/matrix-multiplication-with-shared-memory.png

Producto de matrices - v2

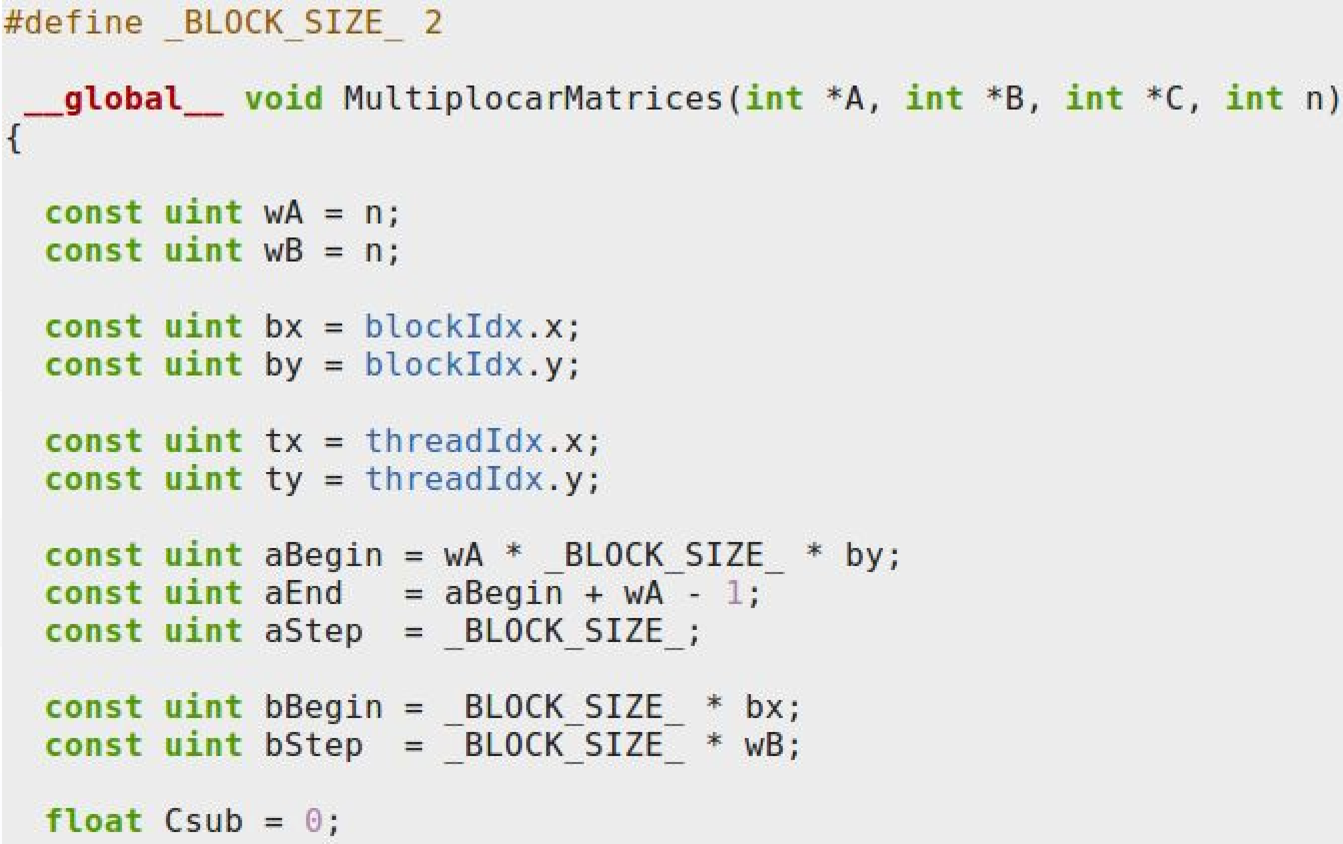
GP-GPU



```
int MATRIX_SIZE = n;
int TILE_SIZE = 2;

dim3 dimGrid  ( MATRIX_SIZE / TILE_SIZE, MATRIX_SIZE / TILE_SIZE);
dim3 dimBlock (TILE_SIZE, TILE_SIZE, 1);

MultiplocarMatrices<<< dimGrid , dimBlock  >>> (d_a, d_b, d_c, n );
```

Producto de matrices - v2

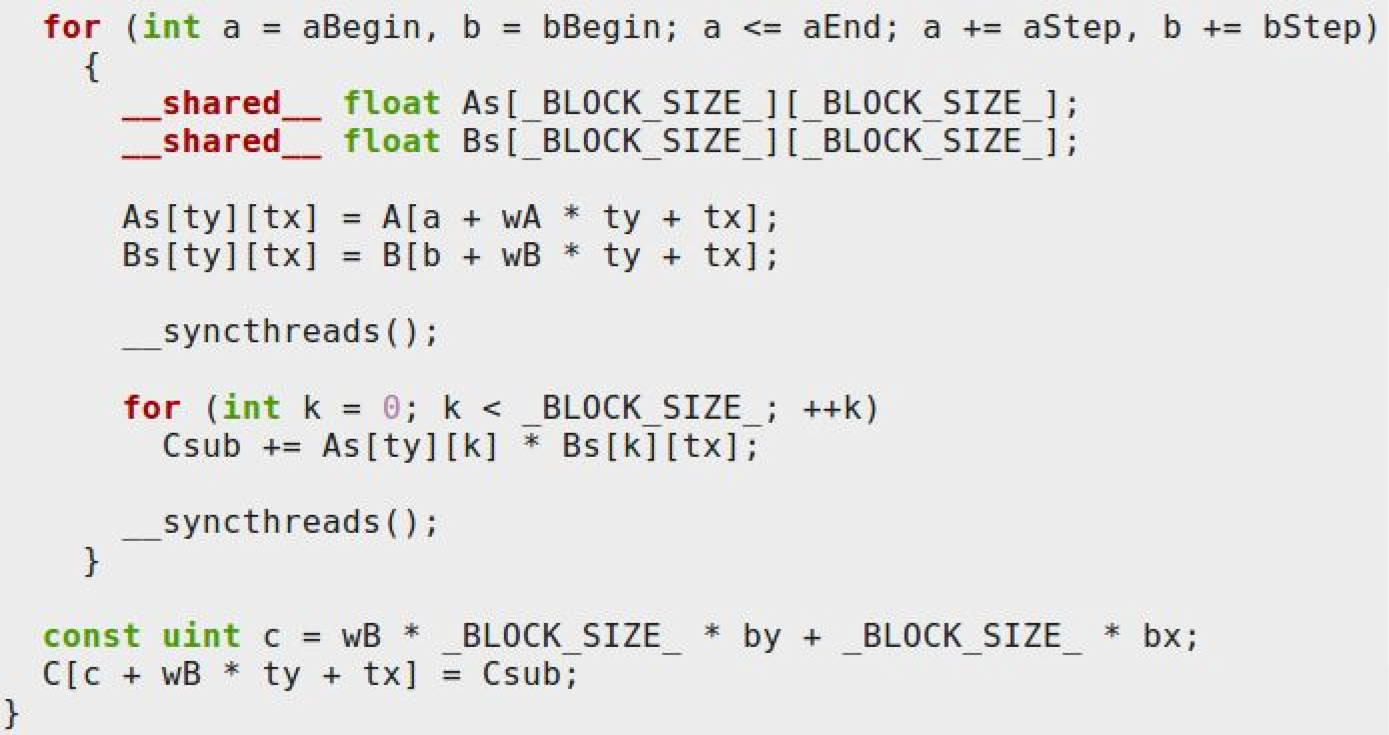
GP-GPU



```
#define _BLOCK_SIZE_ 2

 __global__ void MultiplocarMatrices(int *A, int *B, int *C, int n)
{

  const uint wA = n;
  const uint wB = n;

  const uint bx = blockIdx.x;
  const uint by = blockIdx.y;

  const uint tx = threadIdx.x;
  const uint ty = threadIdx.y;

  const uint aBegin = wA * _BLOCK_SIZE_ * by;
  const uint aEnd   = aBegin + wA - 1;
  const uint aStep  = _BLOCK_SIZE_;

  const uint bBegin = _BLOCK_SIZE_ * bx;
  const uint bStep  = _BLOCK_SIZE_ * wB;

  float Csub = 0;
```

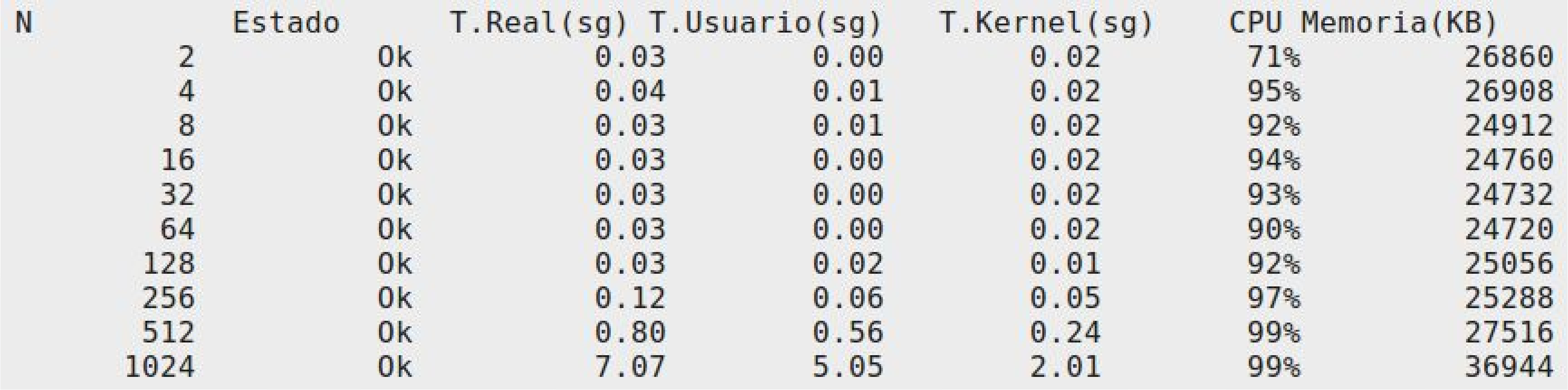
Producto de matrices - v2

GP-GPU



```
 for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
  {
   __shared__ float As[_BLOCK_SIZE_][_BLOCK_SIZE_];
   __shared__ float Bs[_BLOCK_SIZE_][_BLOCK_SIZE_];

   As[ty][tx] = A[a + wA * ty + tx];
   Bs[ty][tx] = B[b + wB * ty + tx];

   __syncthreads();

   for (int k = 0; k < _BLOCK_SIZE_; ++k)
    Csub += As[ty][k] * Bs[k][tx];

   __syncthreads();
  }

 const uint c = wB * _BLOCK_SIZE_ * by + _BLOCK_SIZE_ * bx;
 C[c + wB * ty + tx] = Csub;
}
```

Producto de matrices - v2



```
N           Estado       T.Real(sg) T.Usuario(sg)    T.Kernel(sg)      CPU Memoria(KB)
         2          Ok         0.03          0.00          0.02         71%         26860
         4          Ok         0.04          0.01          0.02         95%         26908
         8          Ok         0.03          0.01          0.02         92%         24912
        16          Ok         0.03          0.00          0.02         94%         24760
        32          Ok         0.03          0.00          0.02         93%         24732
        64          Ok         0.03          0.00          0.02         90%         24720
       128          Ok         0.03          0.02          0.01         92%         25056
       256          Ok         0.12          0.06          0.05         97%         25288
       512          Ok         0.80          0.56          0.24         99%         27516
      1024          Ok         7.07          5.05          2.01         99%         36944
```

Gracias