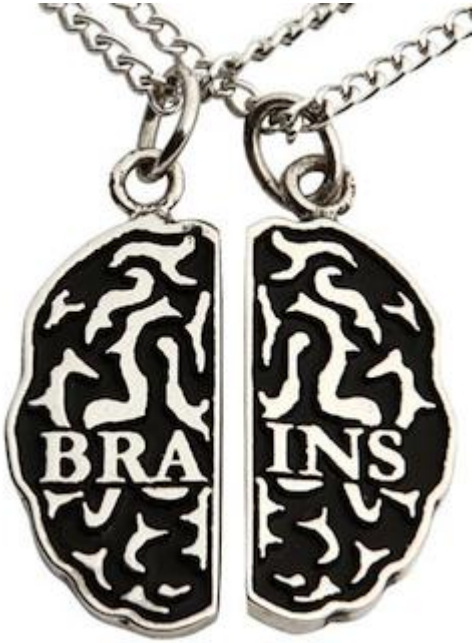


MANEJO DE MEMORIA EN CUDA

Netz Romero

Declarando funciones

`__global__` - función llamada desde el host, y es ejecutada en el device.

`__device__` - función llamada desde el device y es ejecutada en el device.

`__host__` - función llamada desde el host y es ejecutada en el host.

Calificadores de variables en el device

`__device__` - es almacenada en la memoria global.

- accesible para todos los threads.
- tiempo de vida durante la aplicación.

`__constant__` - es almacenada en la memoria constante

- solo de lectura para los threads.
- tiempo de vida durante la aplicación.

Calificadores de variables en el device

`__shared__` - es almacenada en la memoria compartida (muy baja latencia)

- accesible para todos los threads dentro del bloque
- tiempo de vida durante el bloque

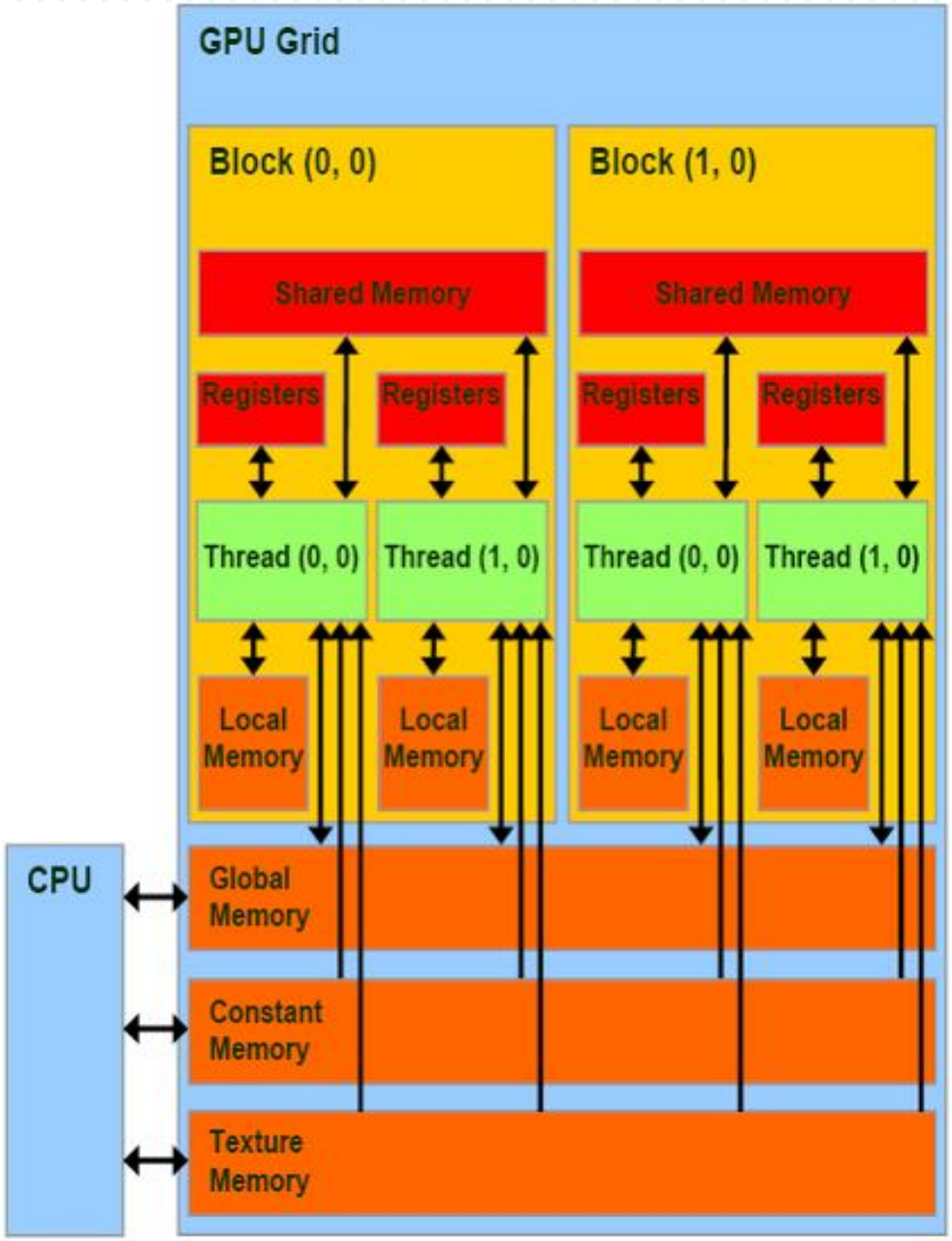
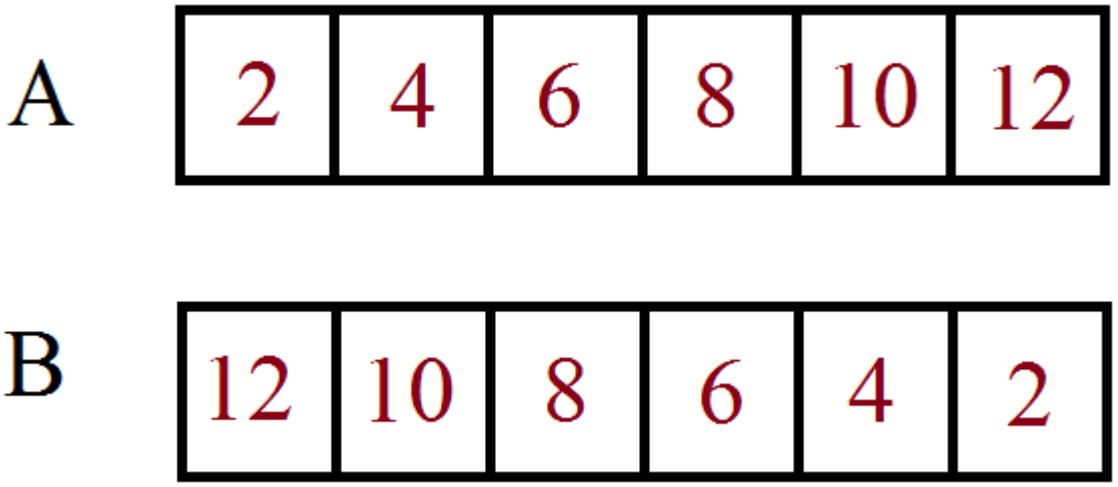

GPU Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
CPU
Global Memory
Constant Memory
Texture Memory

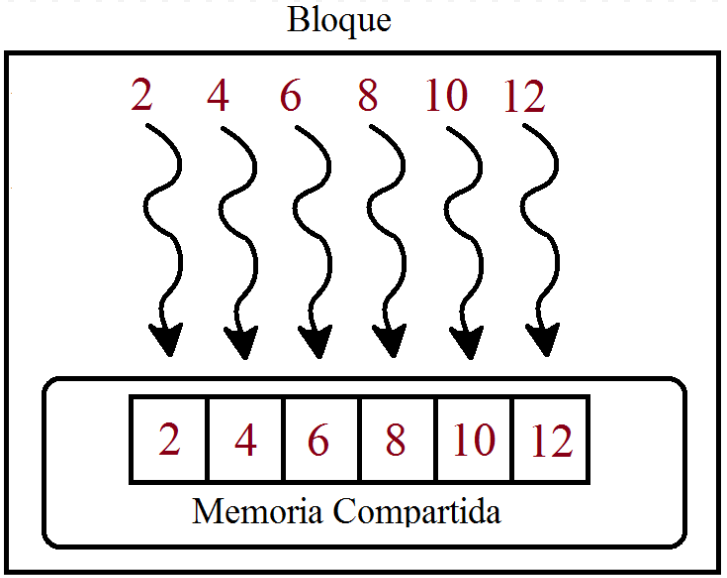
Tabla para el tipo de calificador

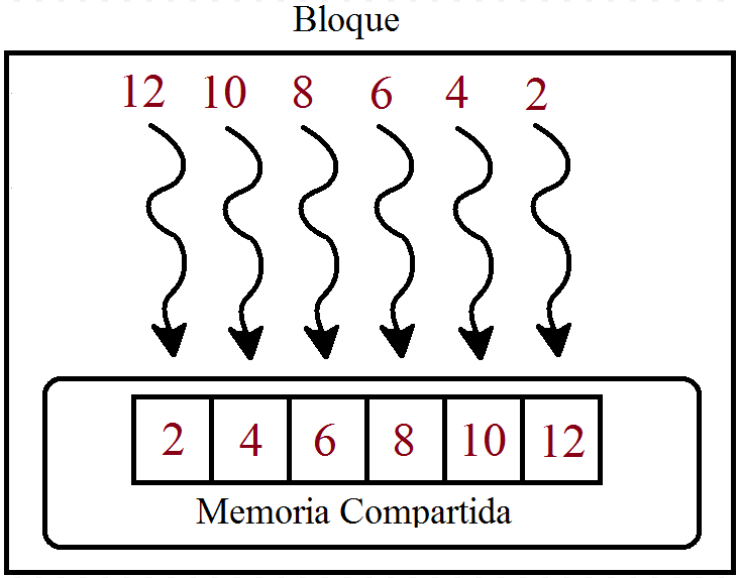
Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`int LocalArray[10];`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

Ejemplo de Memoria Compartida

- Se van a copiar los elementos de un arreglo a otro pero invirtiendo los valores.




Bloque
2 4 6 8 10 12
2 4 6 8 10 12
Memoria Compartida


Bloque
12 10 8 6 4 2
2 4 6 8 10 12
Memoria Compartida

```
1   #include <stdio.h>
2   #define n 6
3
4   __global__ void staticReverse(int *d, int size)
5   {
6     __shared__ int s[6];
7     int t = threadIdx.x;
8     int tr = size-t-1;
9     s[t] = d[t];
10    __syncthreads();
11    d[t] = s[tr];
12  }
13  int main(void)
14  {
15    int a[n], d[n];
16    int *d_d;
```

```
17  for (int i = 0; i < n; i++) {
18    a[i] = (i + 1)*2;
19    printf("a[%d]=%d\t", i, a[i]);
20  }
21  printf("\n");
22  cudaMalloc(&d_d, n * sizeof(int));
23  cudaMemcpy(d_d, a, n*sizeof(int),cudaMemcpyHostToDevice);
24  staticReverse<<<1, n>>>(d_d, n);
25  cudaMemcpy(d, d_d, n*sizeof(int),cudaMemcpyDeviceToHost);
26  for(int i = 0; i < n; i++)
27     printf("d[%d]=%d\t",i,d[i]);
28  printf("\n");
29  cudaFree(d_d);
30  return 0;
31 }
```

kernel

```
correKernel<<<nBlocks,
             nThreads,
             sharedMem >>>(...);
```

donde:

- `nBlocks` es el número de bloques en un grid
- `nThreads` es el número de hilos en un bloque
- `sharedMem` es el espacio a reservar en la memoria compartida

Reservando dinámicamente

Código del kernel

```
__global__ void dynamicReverse(int *d, int n) {
    extern __shared__ int s[];
    ...
}
```

Código del host

```
...
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d, n);
...
```

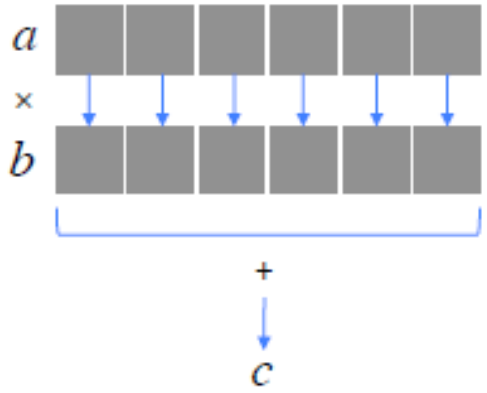
Ejercicio: Producto Punto

- El producto punto se define:

$$\begin{aligned}\mathbf{A} \cdot \mathbf{B} &= (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) \\ &= a_1 b_1 + a_2 b_2 + ... a_n b_n \\ &= \sum a_i \cdot b_i\end{aligned}$$

- Realizar el programa en forma secuencial.

$$c = \sum_i a_i b_i$$



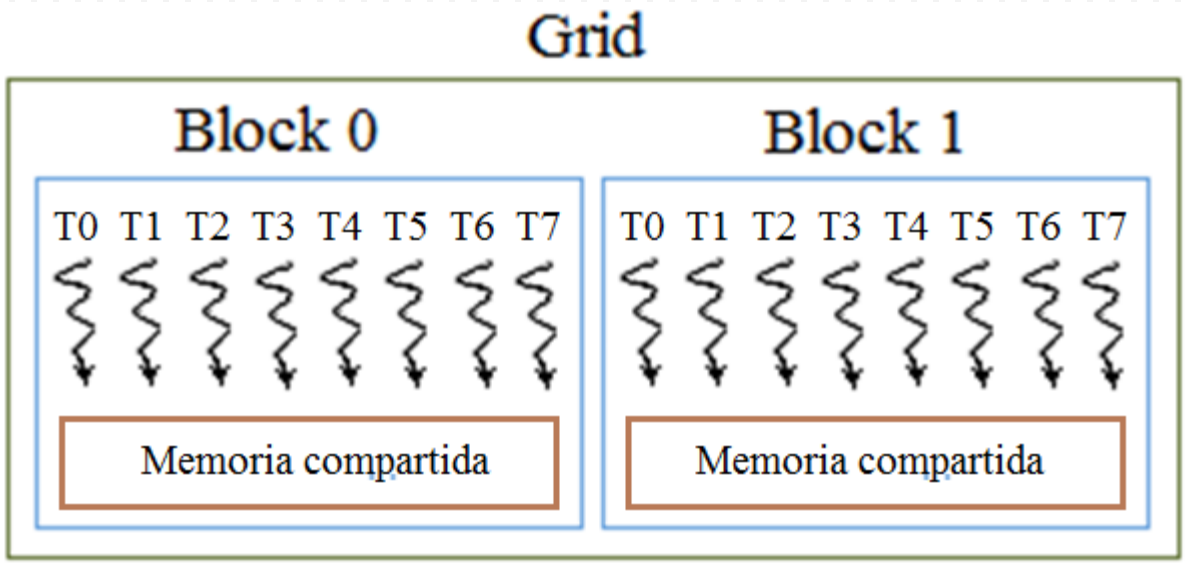
La multiplicación se realiza en paralelo

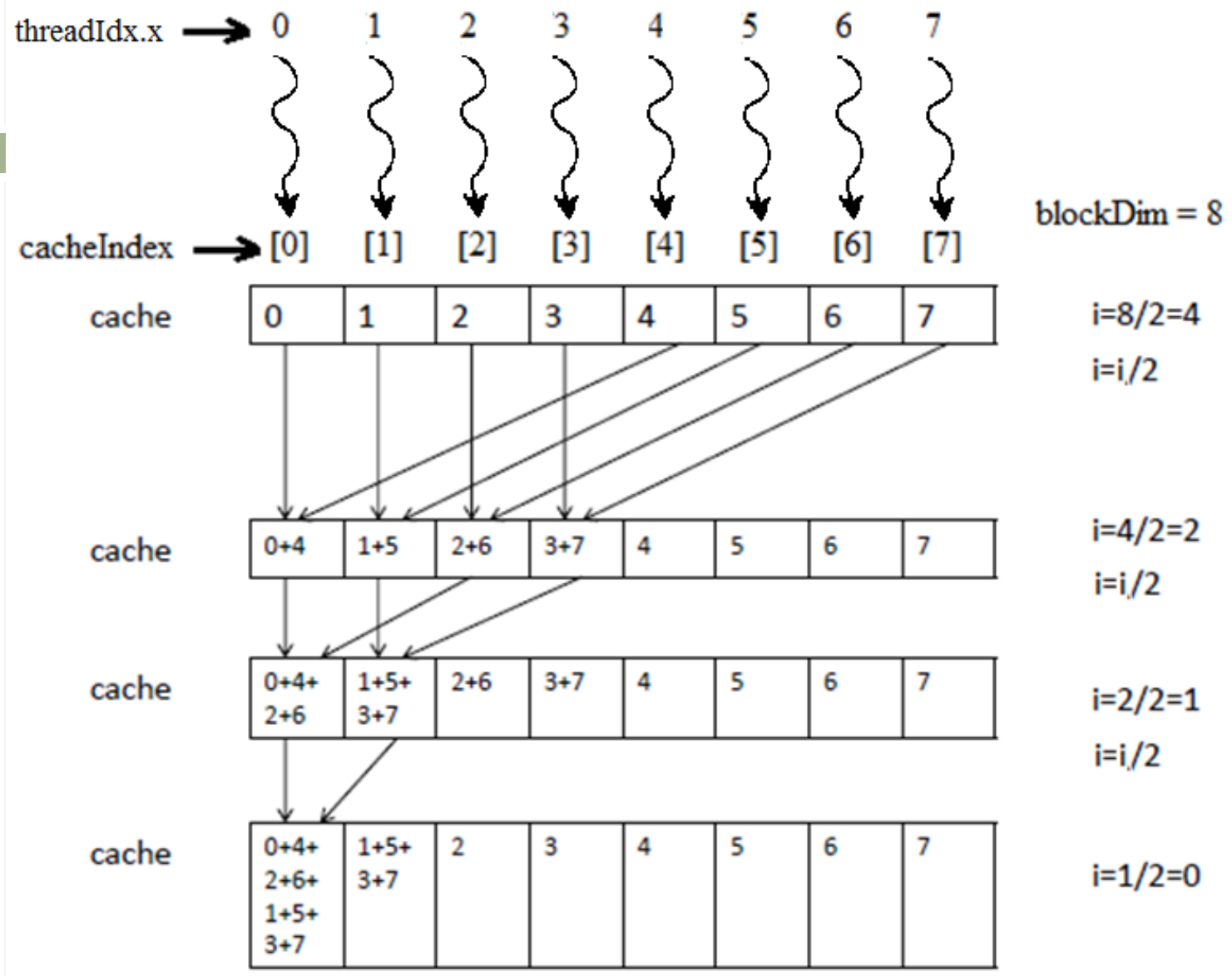
La suma se realiza en forma secuencial

```
__global__ void prodPunto(int a[], int b[], int *c)
{
  __shared__ int prod[N];
  int i = threadIdx.x;
  if (i < N)
    prod[i] = a[i] * b[i];
  __syncthreads();
  if (threadIdx.x == 0)
  {
    int sum = 0;
    for (int k = 0; k < N; k++)
    sum += prod[k];
    *c = sum;
  }
}
```

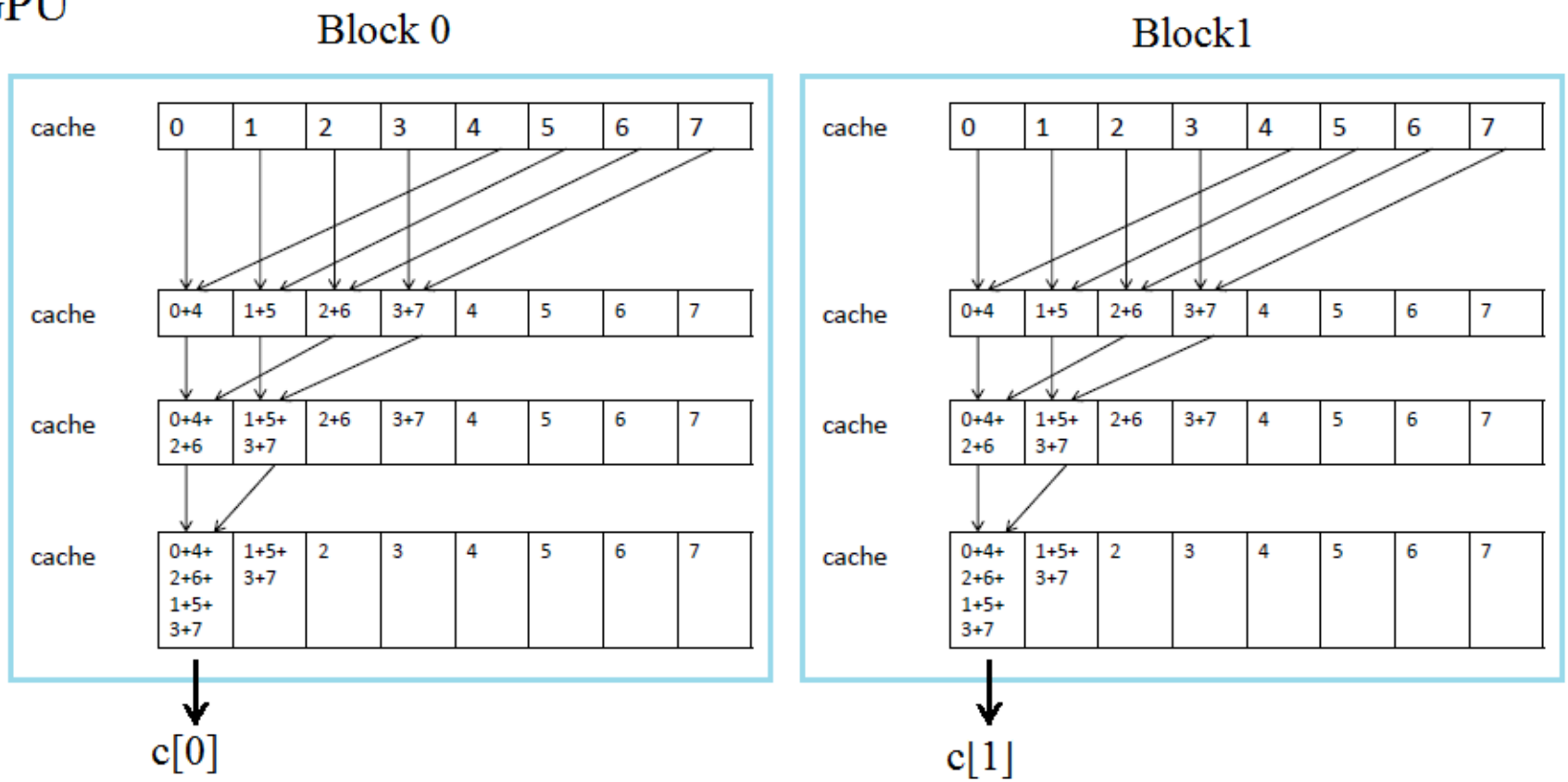
Ejercicio: Producto Punto, versión 2

- Para resolver el problema se van a utilizar dos bloques en el grid y ocho threads en el bloque.




threadIdx.x
0 1 2 3 4 5 6 7
blockDim = 8
cacheIndex
[0] [1] [2] [3] [4] [5] [6] [7]
cache
0 1 2 3 4 5 6 7
i=8/2=4
i=i/2
cache
0+4 1+5 2+6 3+7 4 5 6 7
i=4/2=2
i=i/2
cache
0+4+ 2+6 1+5+ 3+7 2+6 3+7 4 5 6 7
i=2/2=1
i=i/2
cache
0+4+ 2+6+ 1+5+ 3+7 1+5+ 3+7 2 3 4 5 6 7
i=1/2=0

GPU



CPU c = c[0] + c[1]

```
__global__ void dot( float *a, float *b, float *c ) {
  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;
  while (tid < N) {
     temp += a[tid] * b[tid];
     tid += blockDim.x * gridDim.x;
  }
  cache[cacheIndex] = temp;
  __syncthreads();
  int i = blockDim.x/2;
  while (i != 0) {
     if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
     __syncthreads();
     i /= 2;
  }
  if (cacheIndex == 0)
     c[blockIdx.x] = cache[0];
}
```

Constantes

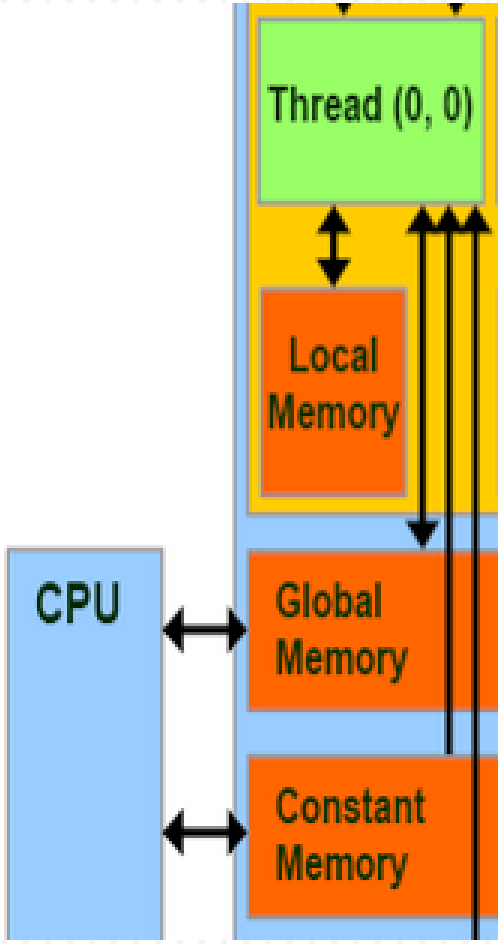
```
...
#define N 8
#define threadsPerBlock 8
#define blocksPerGrid 2
...
```

Código del host

```
...
float c = 0;
float suma_parcial[blocksPerGrid];
...
cudaMalloc((void**)&dev_c, blocksPerGrid*sizeof(float));
...
for (int i=0; i<blocksPerGrid; i++) {
   c += suma_parcial[i];
}
...
```

Memoria Constante

- CUDA dispone de un tipo de memoria conocido como memoria constante y los datos no cambiarán en el transcurso de la ejecución del kernel.



- Para declarar un arreglo en la memoria constante

```
__constant__ float cst_ptr[size];
```

- La instrucción para acceder desde el host a la memoria constante es:

```
cudaMemcpytoSymbol
```

```
1   #include<stdio.h>
2   #define n 10
3   __constant__ float a[n];
4   __global__ void kernel(float *out) {
5     if(threadIdx.x < n)
6         out[threadIdx.x] = a[threadIdx.x];
7   }
8   int main(void) {
9     float *ad;
10    size_t sz = size_t(n) * sizeof(float);
11    float avals[n]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
12    float ah[n];
13    cudaMemcpyToSymbol(a, avals, sz);
14    cudaMalloc((void **)&ad, sz);
15    kernel<<<dim3(1), dim3(16)>>>(ad);
16    cudaMemcpy(ah, ad, sz, cudaMemcpyDeviceToHost);
17    for(int i=0; i<n; i++) {
18        printf("%d %f\n", i, ah[i]);
19    }
20    return 0;
21  }
```

- La salida del programa es:

```
[netzrod@tonatiuh programas]$ ./constMem_00
0 1.000000
1 2.000000
2 3.000000
3 4.000000
4 5.000000
5 6.000000
6 7.000000
7 8.000000
8 9.000000
9 10.000000
```

```
cudaError_t cudaMemcpyToSymbol(const char*   symbol,
          const void*             src,
          size_t                  count,
          size_t                  offset = 0,
          enum cudaMemcpyKind     kind = cudaMemcpyHostToDevice)
```

Copia `count` bytes de la memoria apuntada por src a la memoria apuntada por `offset` bytes al inicio de `symbol`. `symbol` puede ser una variable que reside en la memoria global o constante.

Parametros:

- `symbol` – destino en el device
- `src` – dirección del destino
- `count` – Tamaño en bytes a copiar
- `offset` – desplazamiento de inicio para symbol en bytes
- `kind` – Tipo de transferencia

Regresos:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevice, cudaErrorInvalidMemcpyDirection

	Código del kernel
2	`__constant__ float a[n];`
3	`__global__ void kernel(float *out)`
4	`{`
5	`  if(threadIdx.x < n){`
6	`      a[threadIdx.x] = threadIdx.x + 10;`
7	`      out[threadIdx.x] = a[threadIdx.x];`
8	`  }`
9	`}`

```
$ nvcc constMem.cu -o constMem
./constMem.cu(6): Error: Store to read-only data
```

	Código del kernel
2	`__device__ float a[n];`
3	`__global__ void kernel(float *out)`
4	`{`
5	`  if(threadIdx.x < n){`
6	`      a[threadIdx.x] = threadIdx.x + 10;`
7	`      out[threadIdx.x] = a[threadIdx.x];`
8	`  }`
9	`}`

- La salida del programa es:

```
[netzrod@tonatiuh programas]$ ./constMem_01
0 10.000000
1 11.000000
2 12.000000
3 13.000000
4 14.000000
5 15.000000
6 16.000000
7 17.000000
8 18.000000
9 19.000000
```

Referencias

- Sito de NVIDIA, https://developer.nvidia.com/
- CUDA by Examples, NVIDIA
- Memoria compartida, http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/