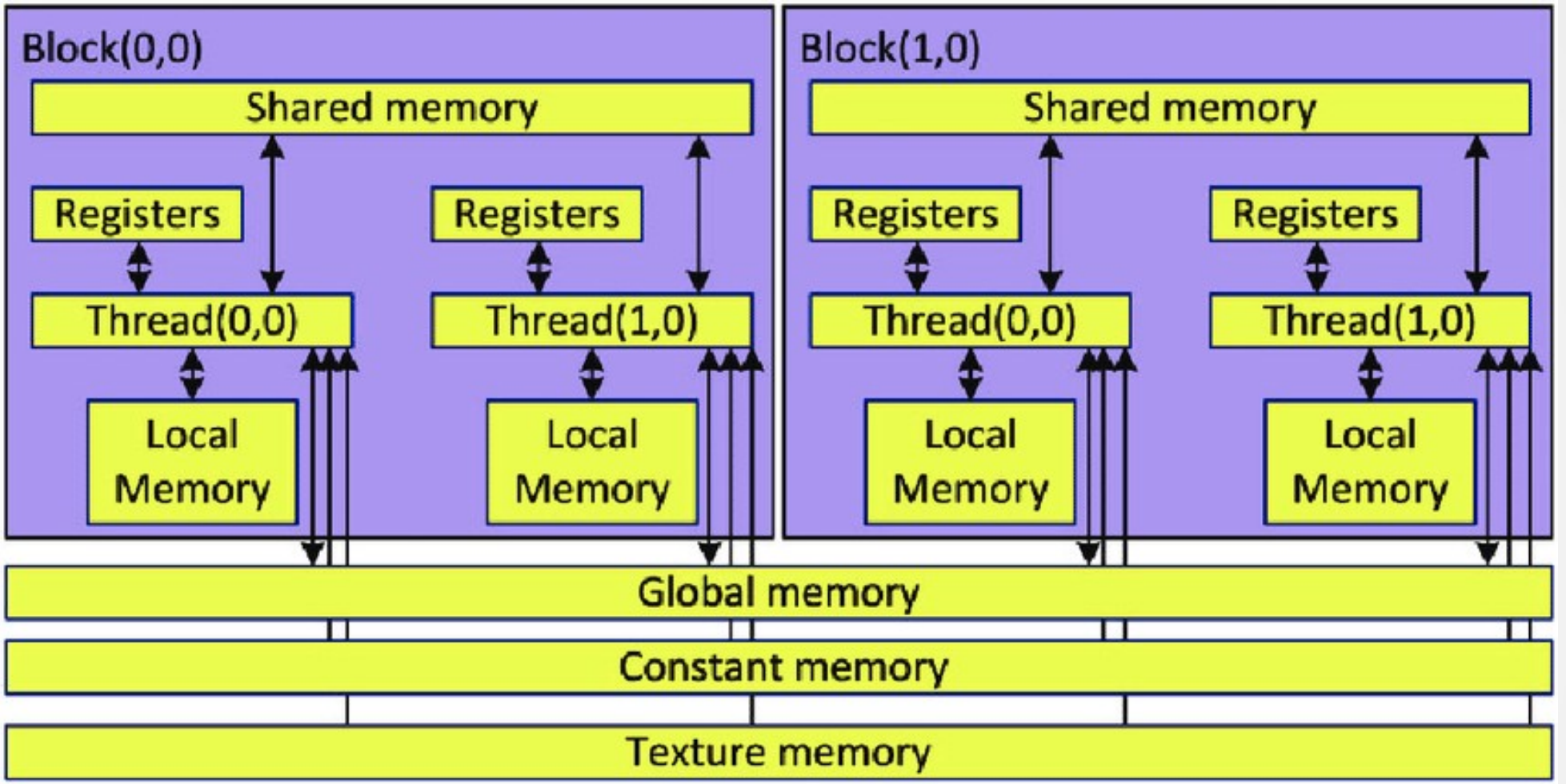
Reduccion en CUDA

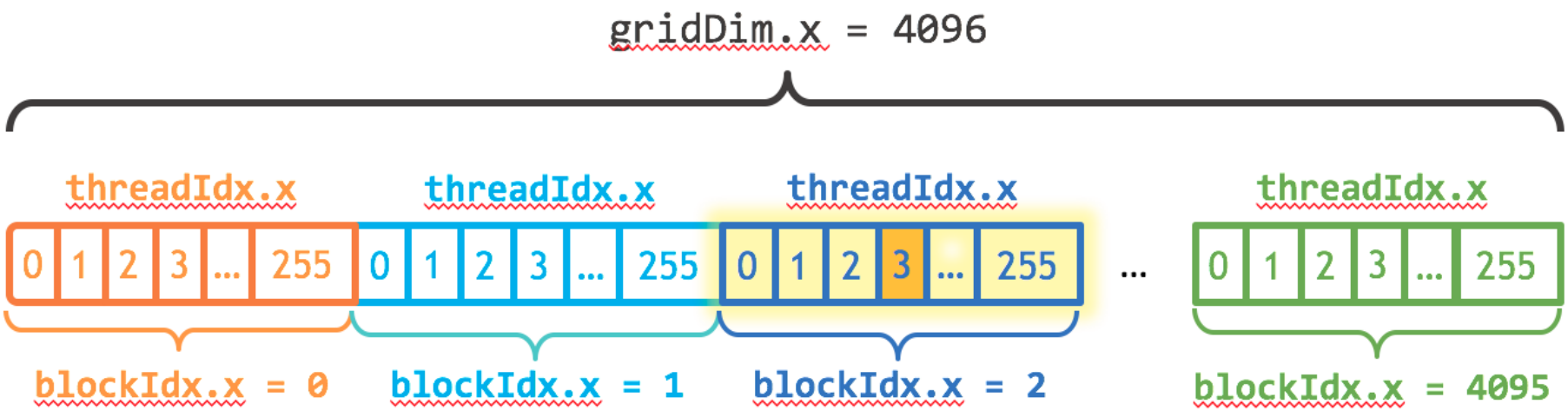
Aquitectura CUDA



- Source: https://www.researchgate.net/figure/Hardware-structure-and-memory-hierarchy-of-CUDA_fig2_319217827
- Autors:

  Fatma Ezahra Sayadi,Haythem Bahri,Chouchene Marwa,Mohamed Atri

Kernel CUDA




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

- Imagen obtenida de: https://devblogs.nvidia.com/even-easier-introduction-cuda/
- Autor: Mark Harris

¿Porque y para que usamos la reduccion en CUDA?

- ¿Porque?

  → No hay sincronizacion global entre hebras debido a que su implementacion el el hardware seria cara

  → Podria reducir la eficiencia global

- ¿Para que?

  → Resolucion de problemas tipo:

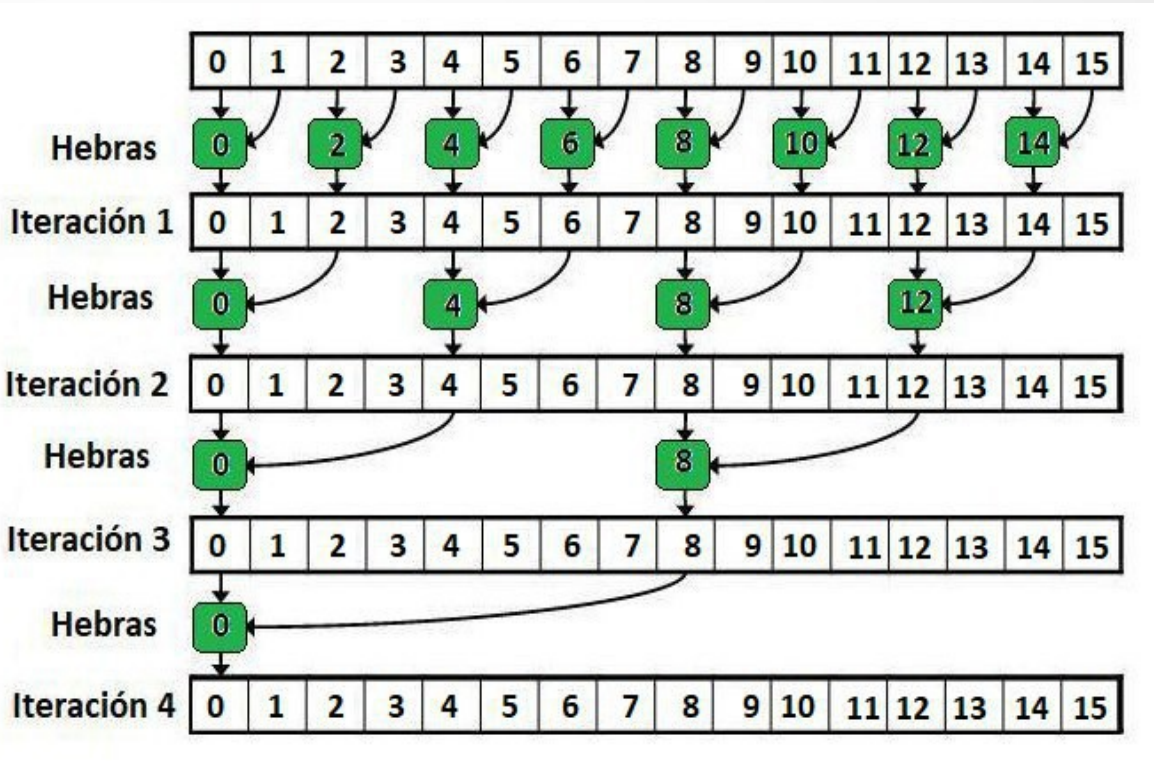
  - Min/max de un array
  - Sumar todos los valores de un vector
  - Juntar resultados de un kernel en un vector de salida

Metodos de reduccion en CUDA

- Reduccion por intervalos
  - (0,1) (2,3) (3,4) (5,6)…..
- Reduccion secuencial
  - (0,N/2) (1,N/2+1)(2,N/2+2)….
- Desenrollado de bucle parcial
- Desenrollado de bucle total

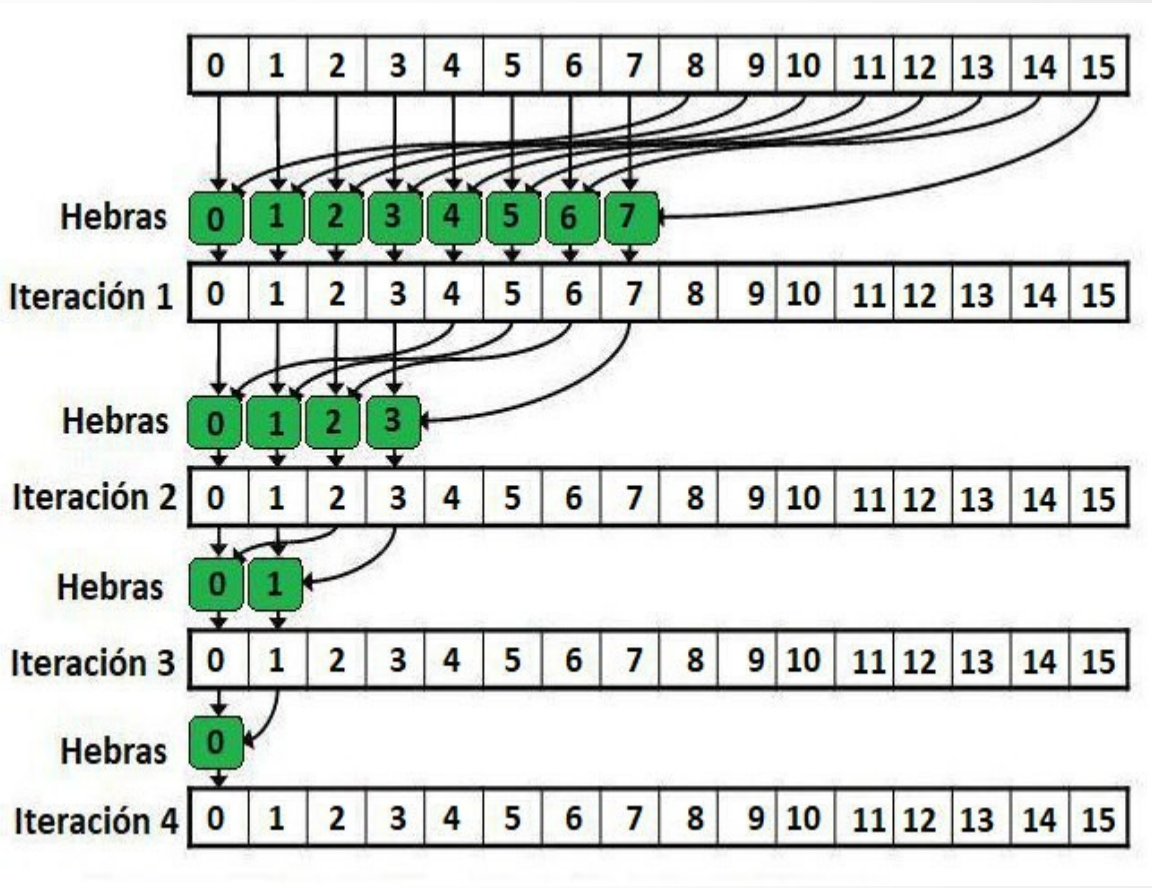
Reduccion por intervalos

```
extern __shared__ int datos[];

int tid = threadIdx.x; //numero de hebra

int posicion = blockIdx.x * blockDim.x + threadIdx.x;
int index = 0;

datos[tid] = device_vector[posicion];

for(int i = 1; i > blockDim.x; i *= 2){
  index = 2 * i * tid;

  if(index < blockDim.x){

    if(datos[tid] < datos[tid+i]){

      datos[tid] = datos[tid+i];

    }

  }
    __syncthreads();
}


//Guardo los resultados en el vector D
  if(device_salida[blockIdx.x] == 0){

    device_salida[blockIdx.x] = datos[0];
```



Reduccion secuencial

```
extern __shared__ int datos[];

int tid = threadIdx.x; //numero de hebra

int posicion = blockIdx.x * blockDim.x + threadIdx.x;

datos[tid] = device_vector[posicion];


__syncthreads();

for(int i = blockDim.x/2; i > 0; i >>= 1){

  if(tid < i){

    if(datos[tid] < datos[tid+1]){

      datos[tid] = datos[tid+1];

    }

  }
    __syncthreads();
}


//Guardo los resultados en el vector D
  if(device_salida[blockIdx.x] == 0){

    device_salida[blockIdx.x] = datos[0];
  }
```



Desenrrollado parcial

```
extern __shared__ int datos[];

int tid = threadIdx.x; //numero de hebra

int posicion = blockIdx.x * blockDim.x + threadIdx.x;

datos[tid] = device_vector[posicion];


for(int i = blockDim.x/2; i > 32; i >>= 1){

  if(tid < i){

    if(datos[tid] < datos[tid+1]){

      datos[tid] = datos[tid+1];

    }

  }
    __syncthreads();
}

if(tid < 32) desenrrollado_reduce_32(datos,tid);

//Guardo los resultados en el vector D
  if(device_salida[blockIdx.x] == 0){

    device_salida[blockIdx.x] = datos[0];
  }
```

```
__device__ void desenrrollado_reduce_32(volatile int* datos, int tid){

    if(datos[tid] < datos[tid+32]) datos[tid] = datos[tid+32];
    if(datos[tid] < datos[tid+16]) datos[tid] = datos[tid+16];
    if(datos[tid] < datos[tid+8]) datos[tid] = datos[tid+8];
    if(datos[tid] < datos[tid+4]) datos[tid] = datos[tid+4];
    if(datos[tid] < datos[tid+2]) datos[tid] = datos[tid+2];
    if(datos[tid] < datos[tid+1]) datos[tid] = datos[tid+1];


}
```

Desenrrollado total

```
extern __shared__ int datos[];

int tid = threadIdx.x; //numero de hebra

int posicion = blockIdx.x * blockDim.x + threadIdx.x;

datos[tid] = device_vector[posicion];

if(blockDim.x >= 2048){
    if(tid < 1024){
        if(datos[tid] < datos[tid + 1024]){
            datos[tid] = datos[tid + 1024];
        }
    }
}

if(blockDim.x >= 1024){
    if(tid < 512){
        if(datos[tid] < datos[tid + 512]){
            datos[tid] = datos[tid + 512];
        }
    }
    __syncthreads();
}

if(blockDim.x >= 512){
    if(tid < 256){
        if(datos[tid] < datos[tid + 256]){
            datos[tid] = datos[tid + 256];
        }
    }
    __syncthreads();
}
```

```
if(blockDim.x >= 256){
    if(tid < 128){
        if(datos[tid] < datos[tid + 128]){
            datos[tid] = datos[tid + 128];
        }
    }
    __syncthreads();
}
if(blockDim.x >= 128){
    if(tid < 64){
        if(datos[tid] < datos[tid + 64]){
            datos[tid] = datos[tid + 64];
        }
    }
    __syncthreads();
}

if(tid < 32) desenrrollado_reduce_32(datos,tid);

__syncthreads();
//Guardo los resultados en el vector D
  if(device_salida[blockIdx.x] == 0){

    device_salida[blockIdx.x] = datos[0];
  }
```

Pruebas

- Hardware

  Nvidia GTX 1060 6GB

  Memoria RAM DDR4 2400 MHz

  I5-7600K 3.80GHz

- Sotware

  Ubuntu 18.04 LTS

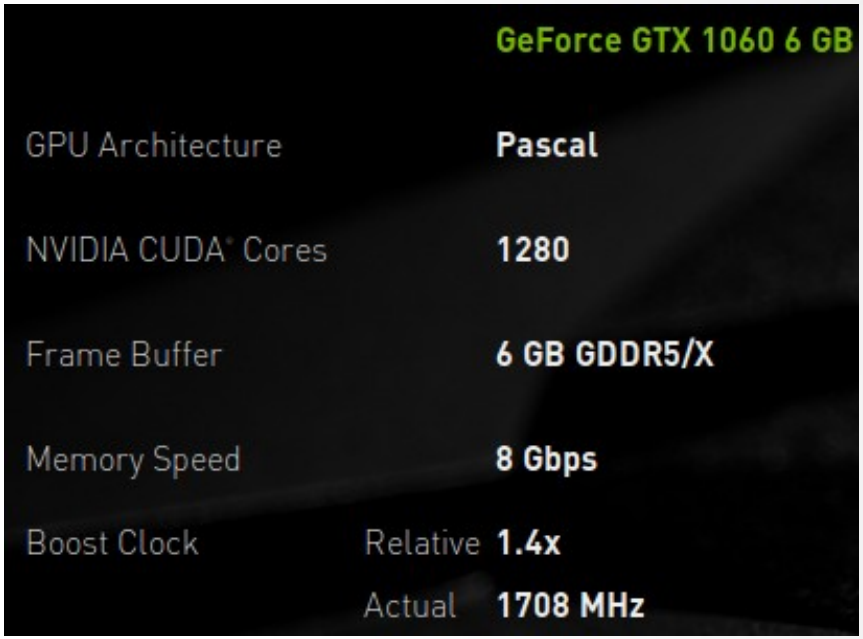
- Codigo disponible en:

  https://github.com/Espectro123/kernel_reduction_CUDA_tests

- Presentacion disponible en:

  https://github.com/Espectro123/Charlas

- Nvidia GTX 1060 6GB



Resultados experimentales

- Tamaño de bloque

  → 2048
- Tamaño del problema

  → 2^21 = 2097152
- Numero de hebras

  → Ceil(2^21/2048) = 1024 hebras por bloque
- 1000 mediciones por cada kernel

Resultados

- Tiempo medio CPU

  → 0.00426755741000002
- Tiempo medio secuencial

  → 1.375173750000039e-05
- Tiempo medio intervalos

  → 1.312898730000032e-05
- Tiempo medio desenrrollado parcial

  → 1.421021790000035e-05
- Tiempo media desenrrollado total

  → 1.331662420000076e-05

Speed up de kernels vs CPU

- Speed up
  - → CPU vs Desenrrollado completo
    - 320.46
  - → CPU vs Desenrrollado parcial
    - 300.316
  - → CPU vs Reduccion secuencial
    - 310.32
  - → CPU vs Reduccion por intervalos
    - 325.048

Speed up entre kernels

- Desenrrollado completo vs parcial
  - 1.0671
- Reduccion por intervalos vs secuencial
  - 1.0474
- ¿Porque hay tan poca ganancia?
  - Kernel de prueba muy ligero
  - Capacidad computacional del device muy alta

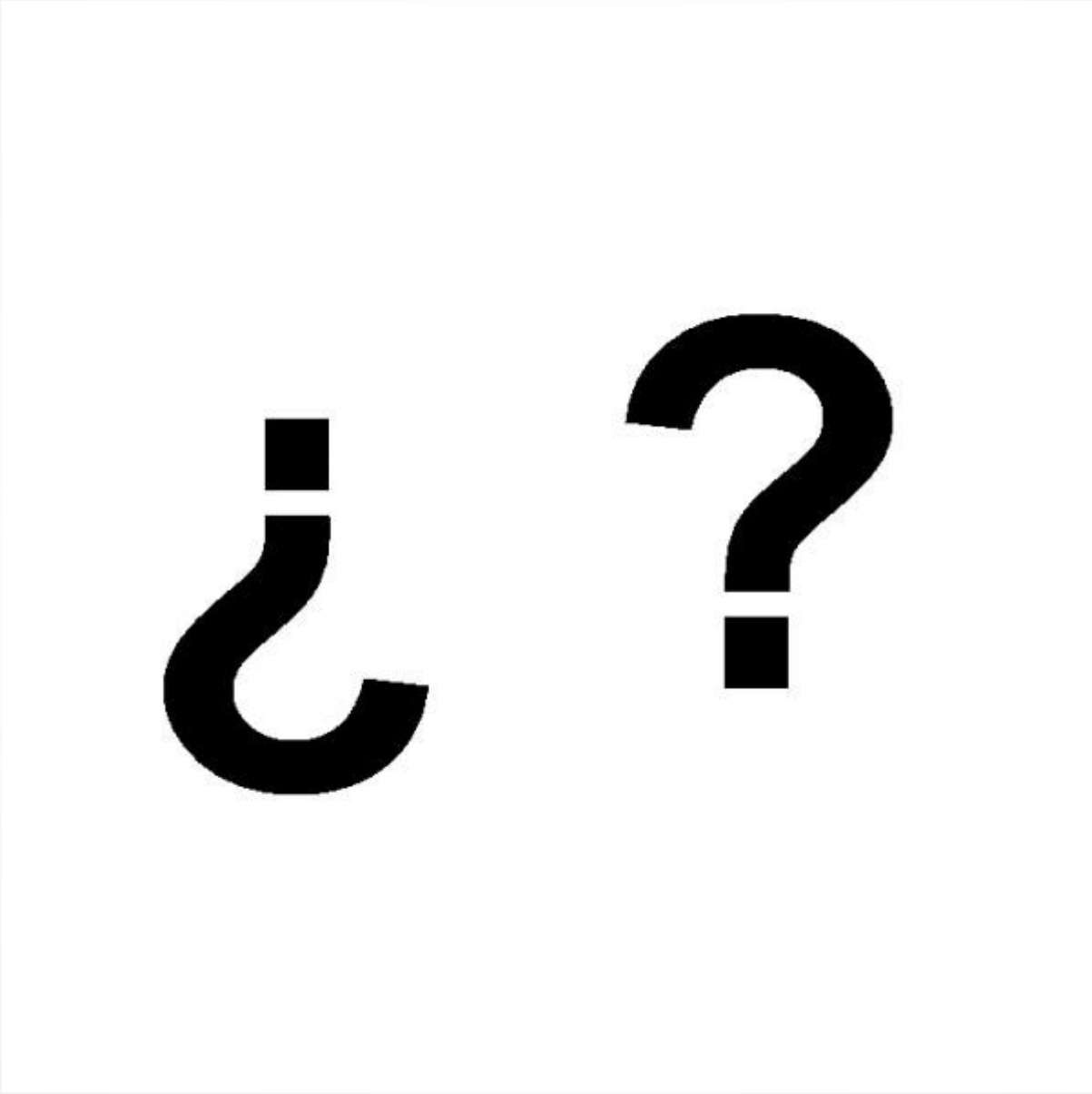
Experimento: Suma global

- Nvidia GPU G80(384 bit memory, 900 Mhz DDR)

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

¿Preguntas?



Bibliografia

- Professional CUDA C Programming
- The CUDA Handbook: A Comprehensive Guide to GPU Programming
- https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf