Ejemplo de aplicación Reducción en CUDA

Universidad Nacional de La Plata

Facultad de Informática

Prof. Dr. Adrian Pousa

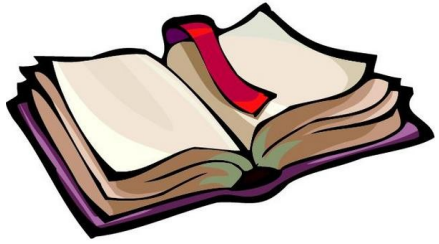
Agenda

I. Reducción

II. Suma por reducción

   i. Estrategia y solución en GPU

   ii. Optimización usando memoria compartida



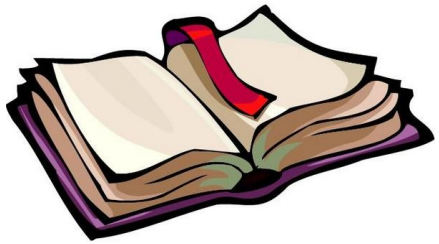
Agenda

I. Reducción

II. Suma por reducción

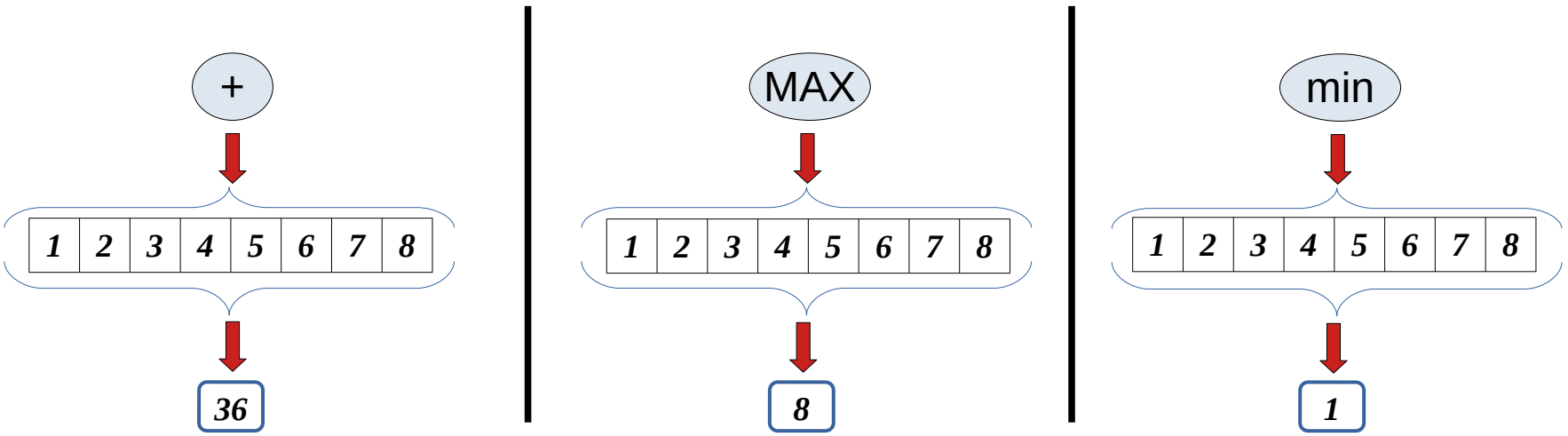
   i. Estrategia y solución en GPU

   ii. Optimización usando memoria compartida



Reducción

- **Reducción:** reducir los valores de un vector a un valor simple.
- Es posible hacerlo aplicando un operador asociativo:
  - +, *, MAX, min, etc.



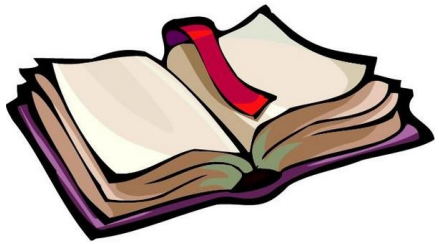
Agenda

I. Reducción

II. Suma por reducción

   i. Estrategia y solución en GPU

   ii. Optimización usando memoria compartida



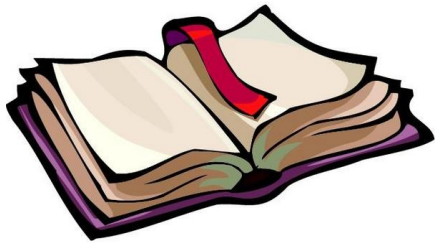
Agenda

I. Reducción

II. Suma por reducción

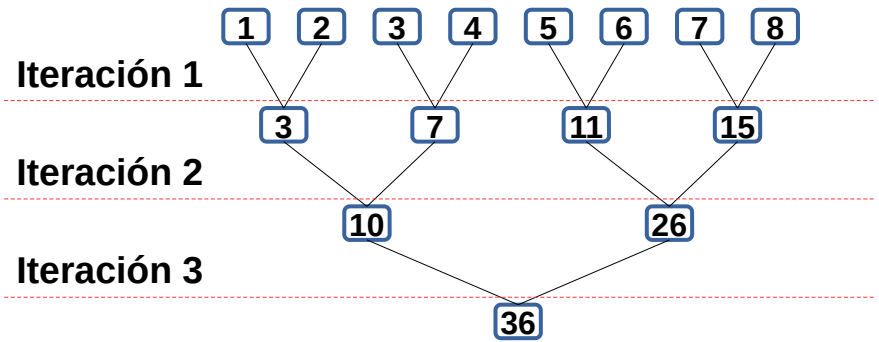
   i. Estrategia y solución en GPU

   ii. Optimización usando memoria compartida



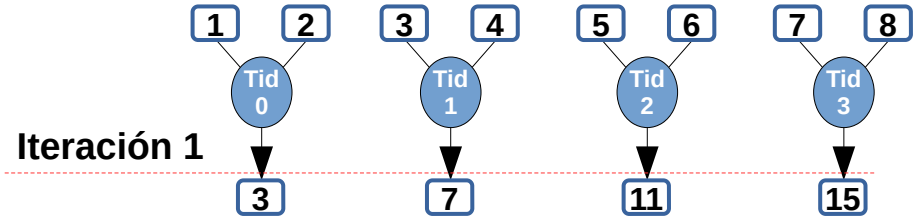
Suma por reducción
Estrategia

- Dado un vector de N elementos, la solución de una **suma por reducción** en GPU consiste en una implementación basada en árbol.
- Se requieren varias iteraciones, en cada iteración se realiza una nueva llamada al kernel y se aprovecha el hecho de que los valores de las variables en memoria global no cambian entre llamados al kernel.

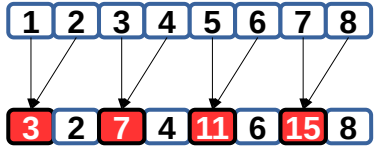


Suma por reducción
Estrategia

- Para la primera iteración se crean $N/2$ hilos.
- Cada hilo suma su posición y la siguiente.

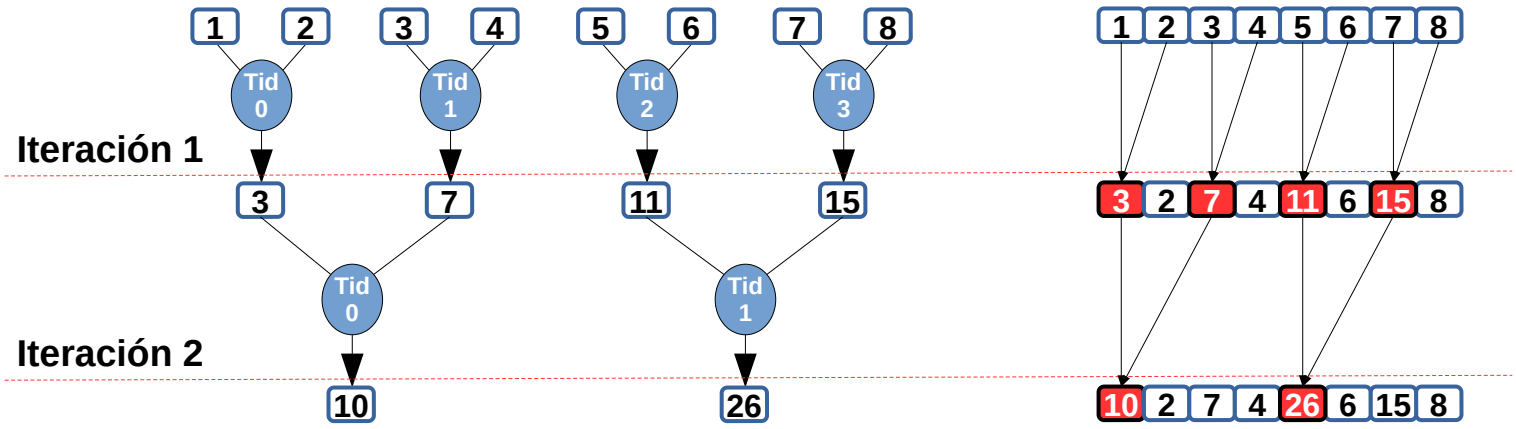


- Para minimizar el espacio de almacenamiento, el resultado de cada suma parcial se almacena en la posición del primer operando.



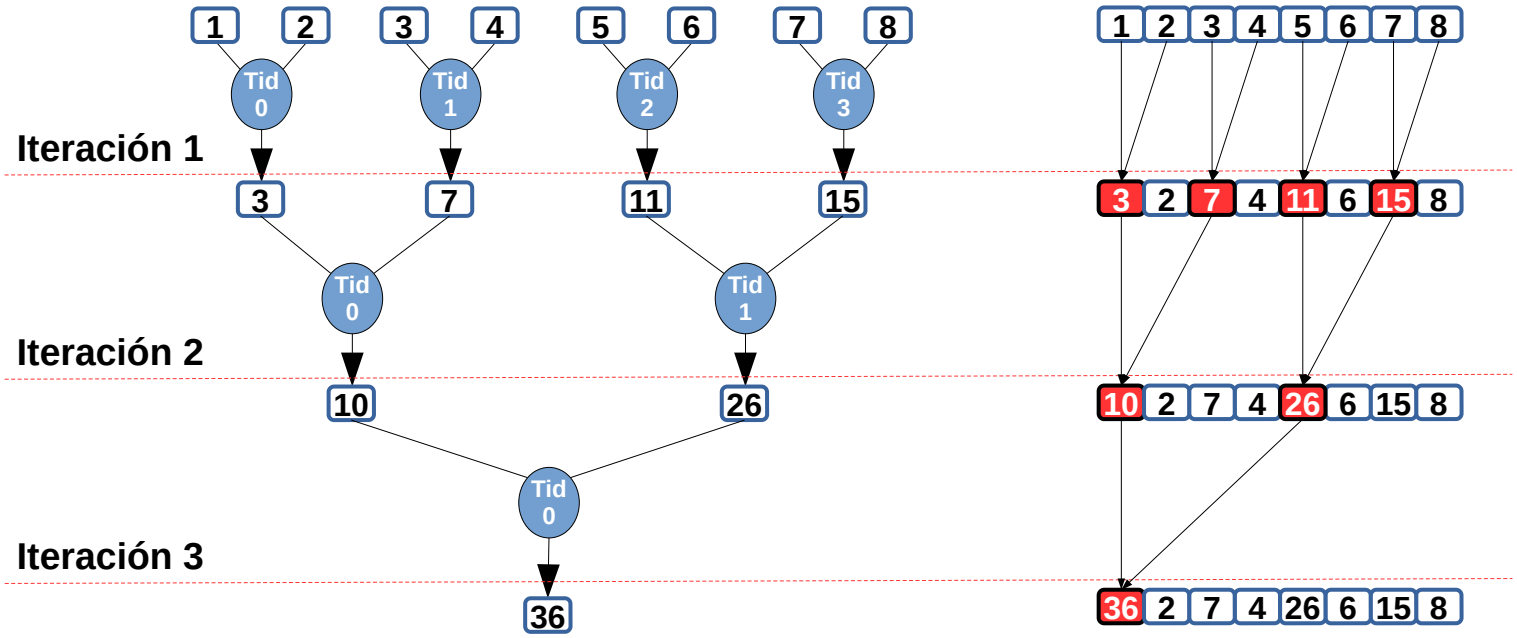
Suma por reducción
Estrategia

- Para la siguiente iteración se invoca nuevamente al kernel con la mitad de hilos de la iteración anterior y se sigue con la misma estrategia.



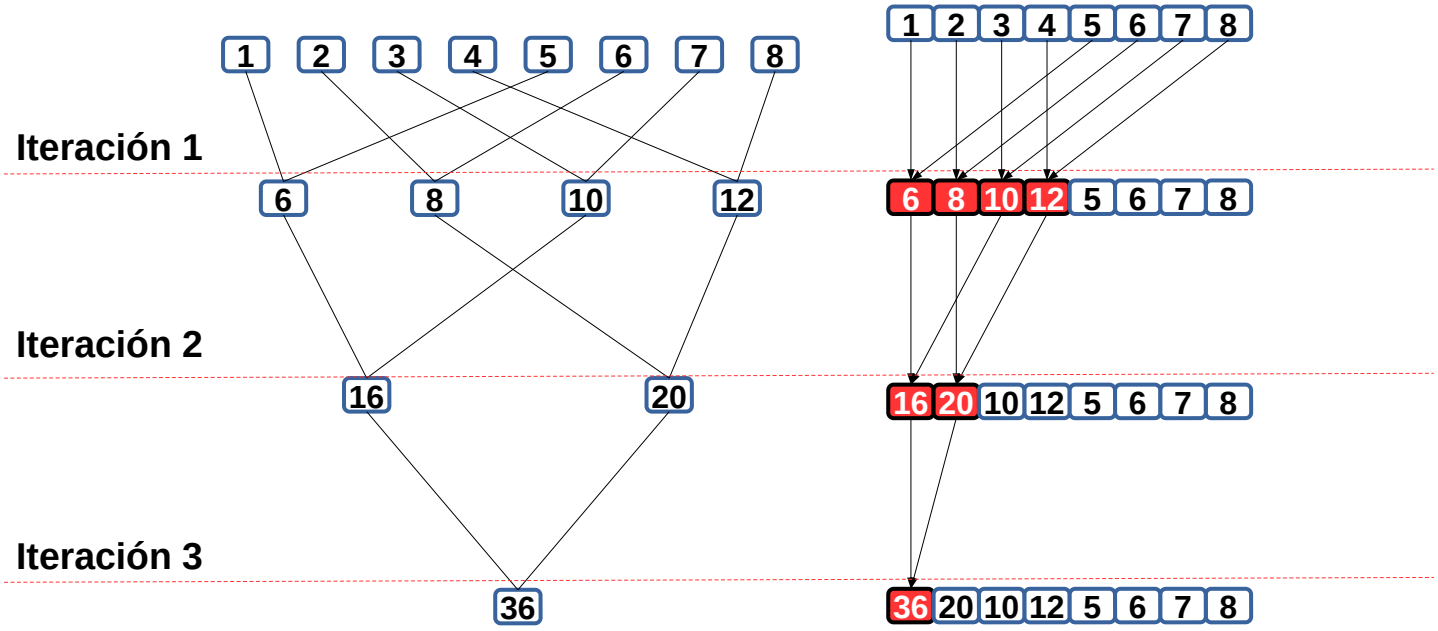
Suma por reducción
Estrategia

- En la última iteración se invoca al kernel con un sólo hilo, el resultado final queda en la primer posición del vector.



Suma por reducción
Alternativa para mejora de acceso a memoria

- Una alternativa, que mejora el acceso a memoria, es utilizar un patrón de acceso a posiciones contiguas.



Suma por reducción
Código CUDA

- Pseudocódigo:


```
__global__ kernelReduccion("parámetros"){
 int idx = blockDim.x*blockIdx.x + threadIdx.x;
 If (idx < "límite"){
  vglobal[idx] += vglobal[idx + "distancia"];
 }
}
```

```
int main(int argc, char* argv[]){
 int blockSize = 256;
 dim3 dimBlock(blockSize);
…
 for ("nro iteraciones"){
   dim3 dimGrid("en función de N y dimBlock")
   kernelReduccion<<<dimGrid, dimBlock>>>("parámetros");
   cudaDeviceSynchronize();
 }
...
}
```

Suma por reducción
Código CUDA

- Pseudocódigo:

```
__global__ kernelReduccion("parámetros"){
 int idx = blockDim.x*blockIdx.x + threadIdx.x;
 If (idx < "límite"){
  vglobal[idx] += vglobal[idx + "distancia"];
 }
}
```

```
int main(int argc, char* argv[]){
 int blockSize = 256;
 dim3 dimBlock(blockSize);
…
 for ("nro iteraciones"){
   dim3 dimGrid("en función de N y dimBlock")
   kernelReduccion<<<dimGrid, dimBlock>>>("parámetros");
   cudaDeviceSynchronize();
 }
...
}
```

El tamaño del problema puede ser menor a la cantidad de hilos por bloque.
Un hilo podría leer una posición de memoria más allá de los límites del problema.

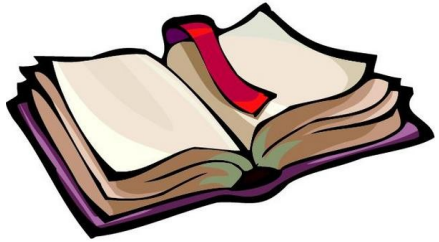
Agenda

I. Reducción

II. Suma por reducción

   i. Estrategia y solución en GPU

   ii. Optimización usando memoria compartida

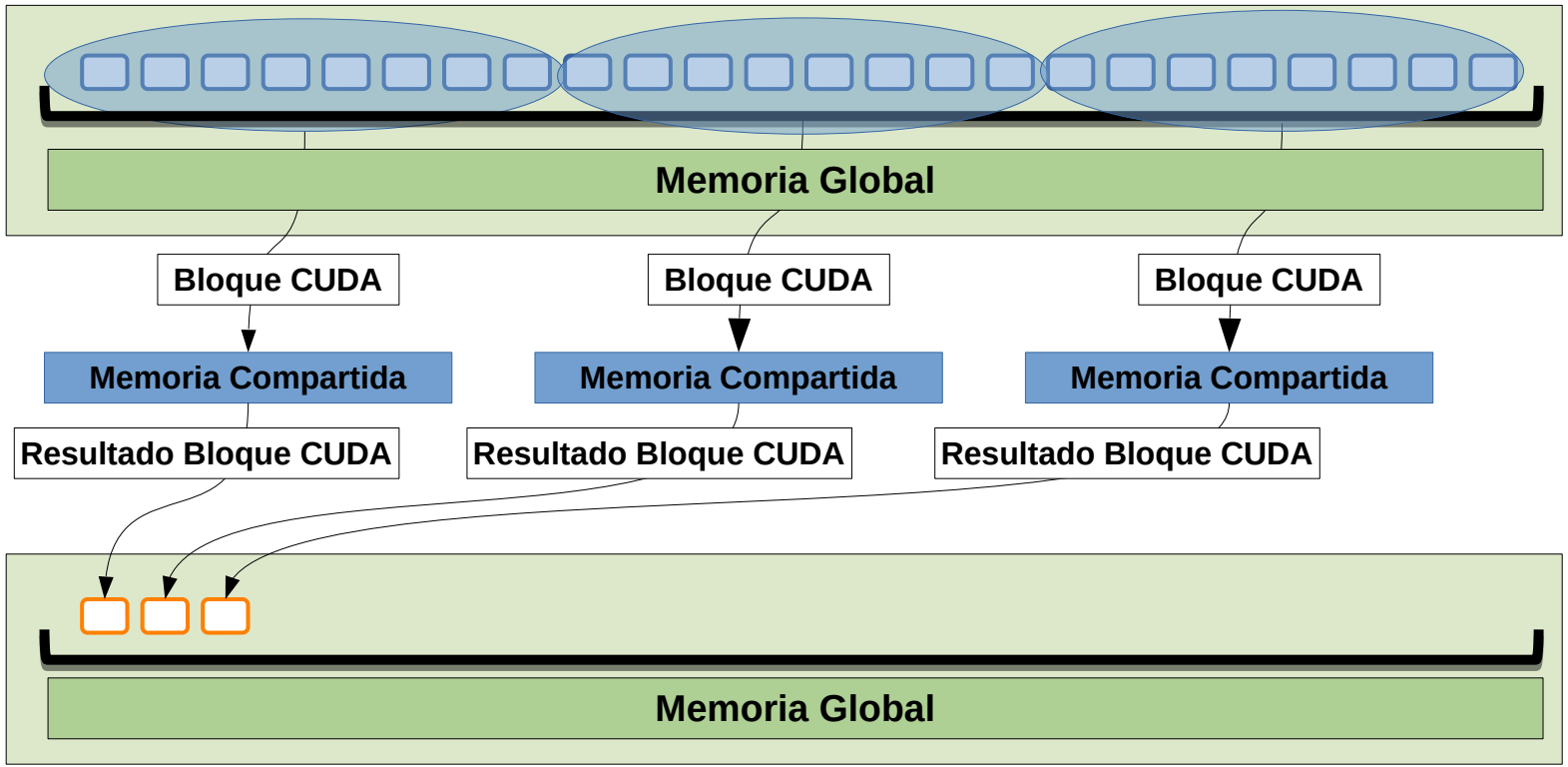


Suma por reducción
Mejoras por el uso de memoria compartida

- Podemos mejorar el tiempo de ejecución utilizando una memoria más rápida como la memoria compartida.
- La estrategia es la siguiente:
  - Cada bloque trae una porción de los datos, de forma coalescente, desde memoria global a la memoria compartida.
  - Cada bloque realiza la reducción en memoria compartida.
  - Cada bloque almacena el resultado que reside en memoria compartida en la memoria global.
- Cada bloque dejará un valor. Luego, los resultados de todos los bloques deben reducirse nuevamente.

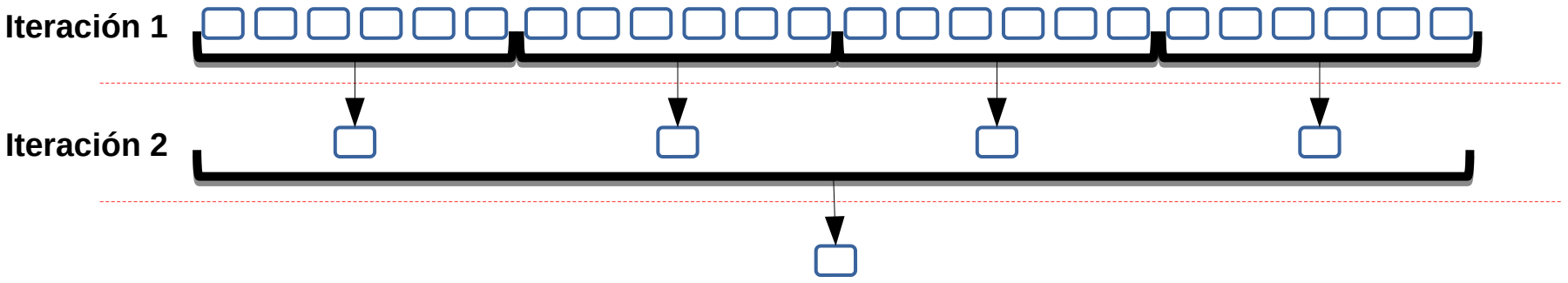
Suma por reducción
Mejoras por el uso de memoria compartida

- Una iteración consiste de:



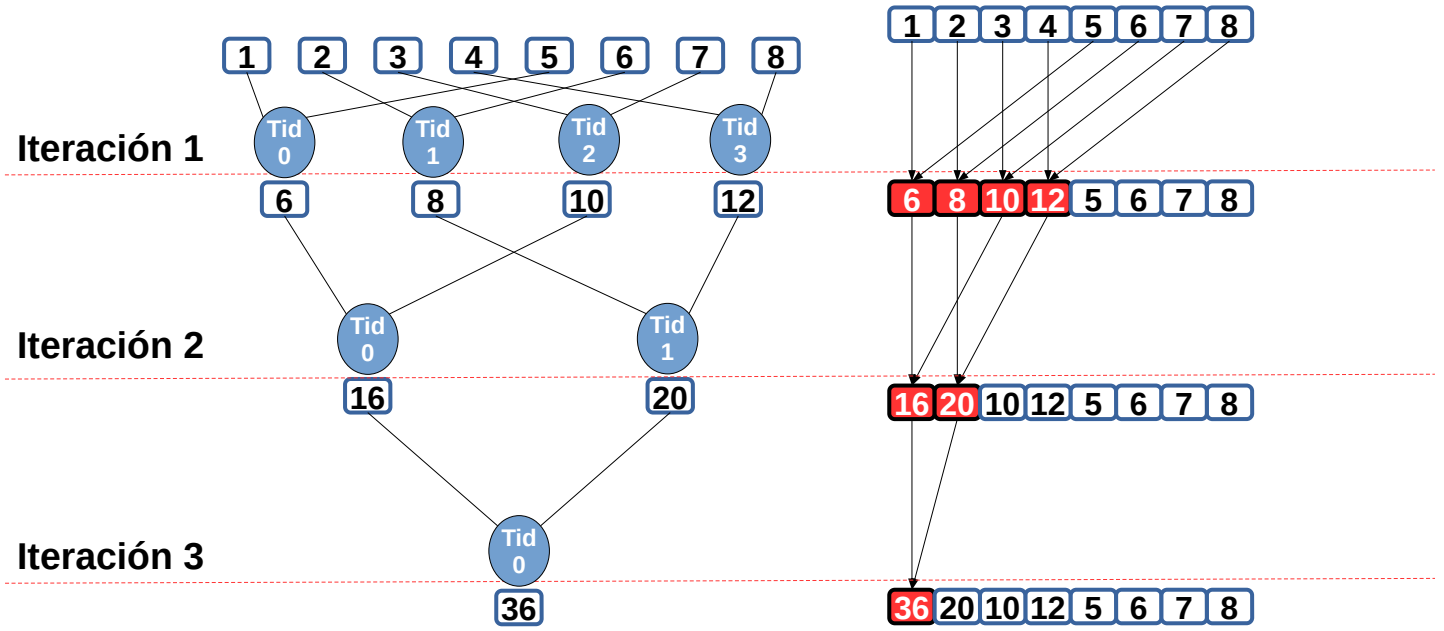
Suma por reducción
Mejoras por el uso de memoria compartida

- Las siguientes iteraciones reducen los resultados de los bloques de la iteración anterior.



Suma por reducción
Mejoras por el uso de memoria compartida

- Usamos el siguiente patrón de acceso que asegura un direccionamiento secuencial libre de conflictos.



Suma por reducción
Implementación

```
__global__ void reduce(int *g_idata, int *g_odata) {
 extern __shared__ int sdata[];
  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

      sdata[threadIdx.x] = g_idata[tid];
      __syncthreads();

      for(unsigned int s=blockDim.x/2; s >0; s >>= 1) {
            if (threadIdx.x < s)
                  sdata[threadIdx.x] += sdata[threadIdx.x + s];

            __syncthreads();
      }

      if (threadIdx.x == 0)
            g_odata[blockIdx.x] = sdata[0];
}
```

if con bajo grado de divergencia