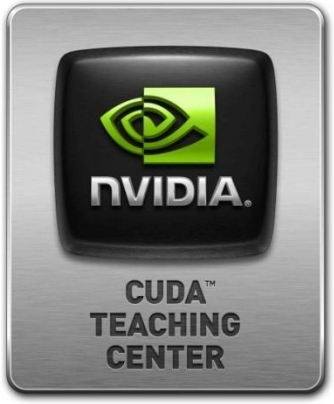
Programación masivamente paralela en procesadores gráficos (GPUs)

E. Dufrechou , M. Freire, P. Ezzatti y M. Pedemonte



Clase 7
Patrones de cómputo 1

Contenido

- Histograma
  - Acceso coalesced
  - Operaciones atómicas
  - Privatization
- Reduce

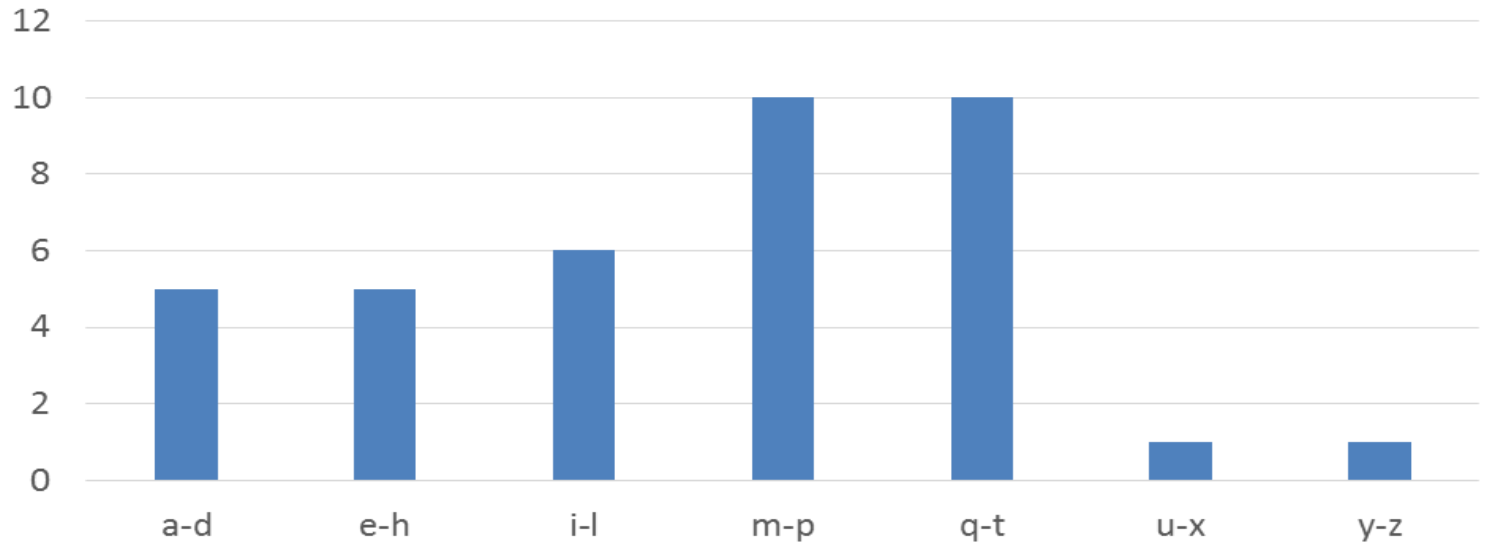
Histogram

Histogram

- Es una representación gráfica de una variable en forma de barras.
- La superficie de la barra es proporcional a la frecuencia de los valores representados.
- Se clasifican los valores de acuerdo a su frecuencia en cubetas.
- Sirven para tener una primera aproximación a la distribución de la muestra.
- Cuando los valores de la variable son discretos se conocen con el nombre de diagrama de frecuencias.

Histogram

- Veamos un ejemplo con texto.
- Se definen las cubetas: a-d, e-h, i-l, m-p, …
- Para cada carácter se debe incrementar la cubeta correspondiente.
- El histograma de la frase “Programming Massively Parallel Processors” sería:

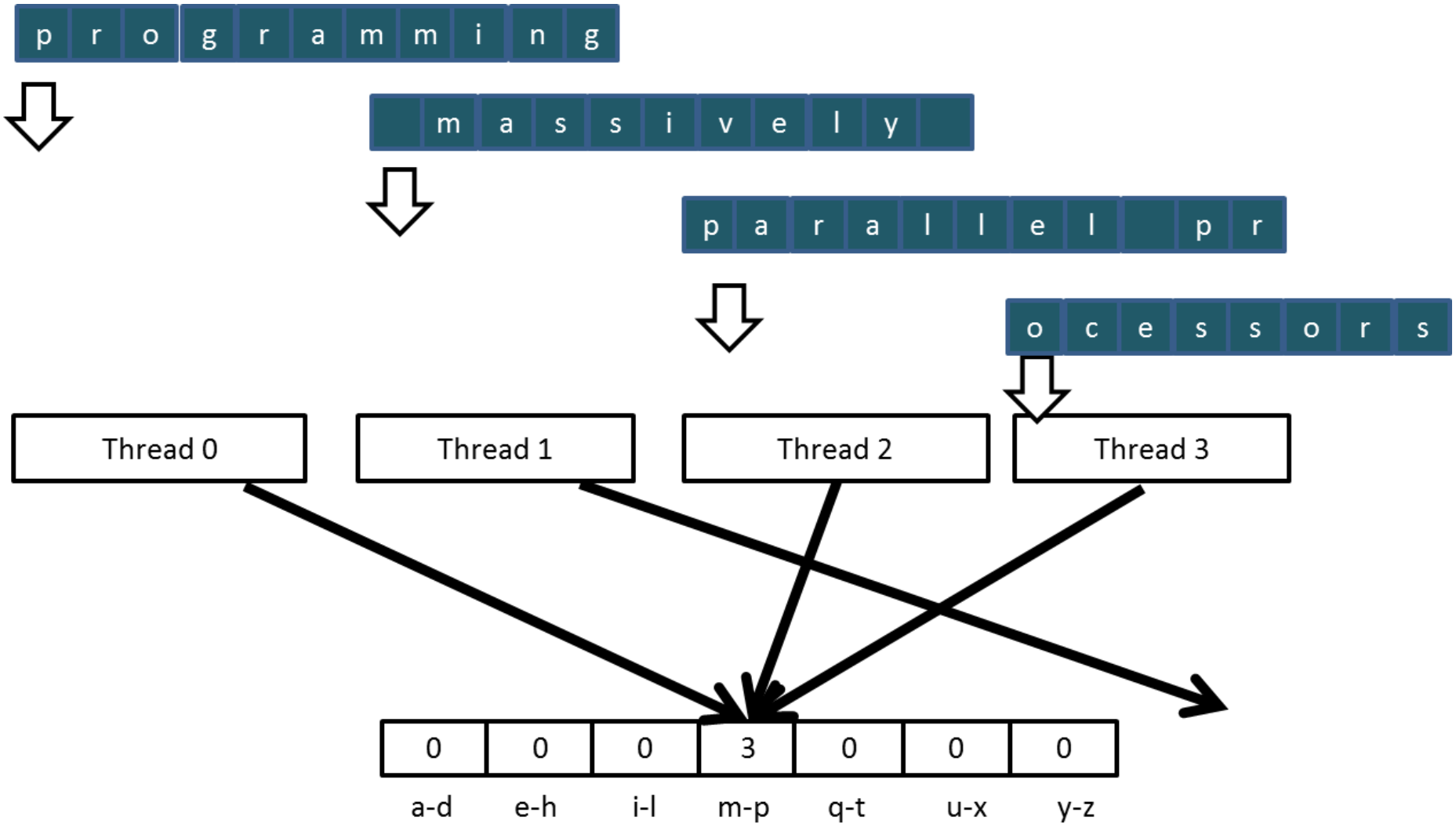


Histogram

- **Primera aproximación sencilla:**
  - **La entrada se particiona en secciones.**
  - **Cada hilo procesa una sección de la entrada**
  - **Cada hilo itera procesando los elementos de su sección**
  - **Para cada letra se incrementa la cubeta correspondiente.**

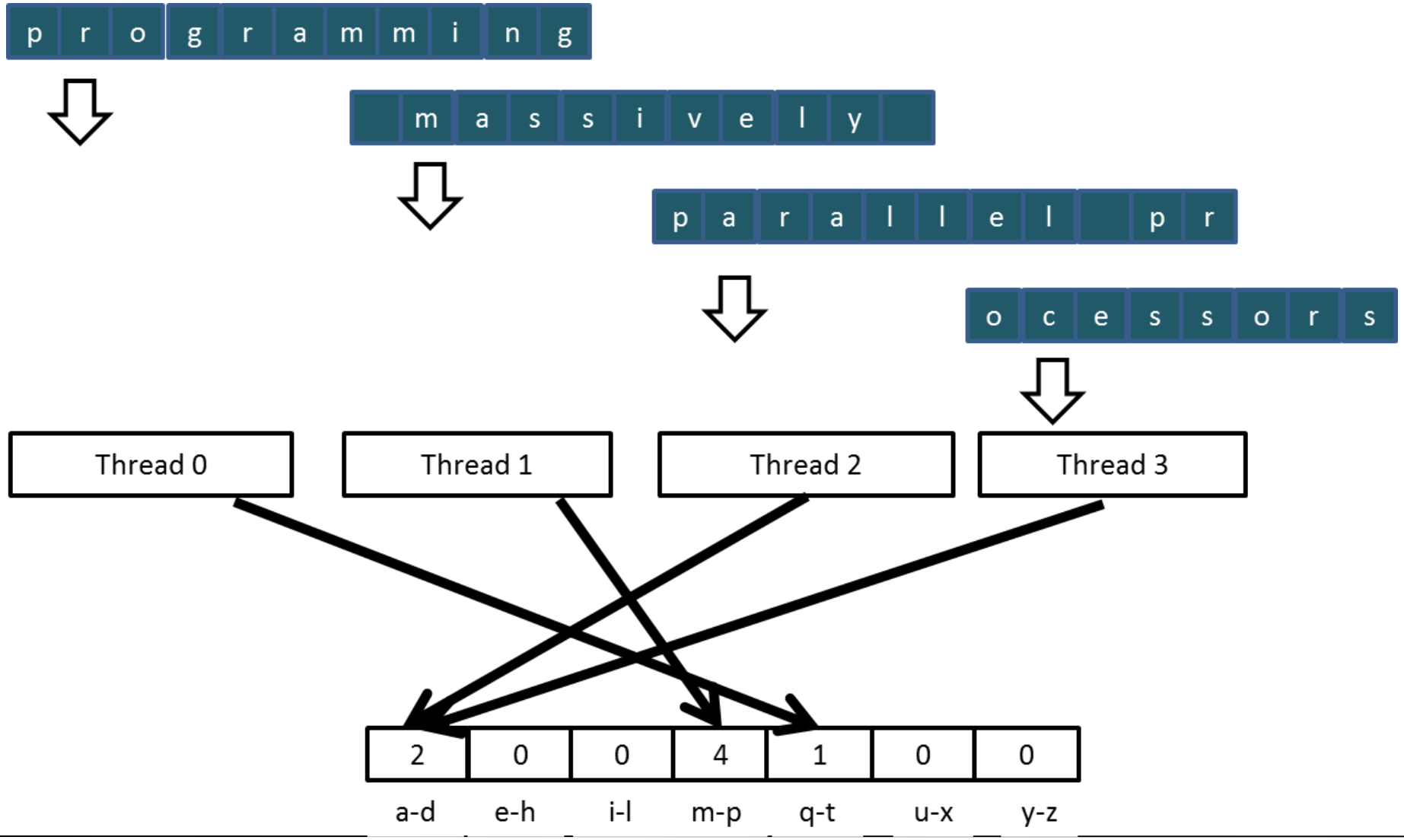
Histogram

Primera iteración



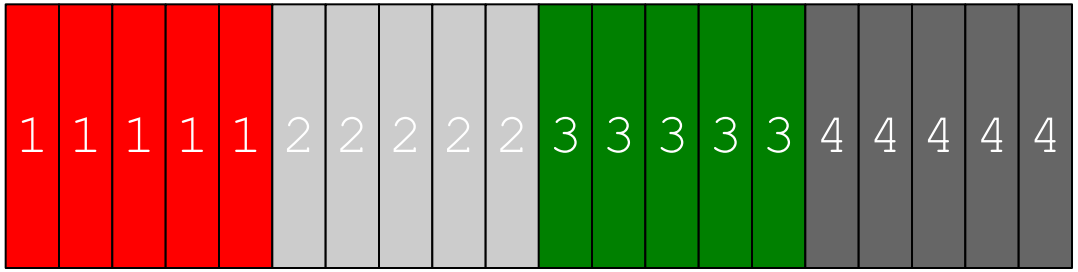
Histogram

Segunda iteración



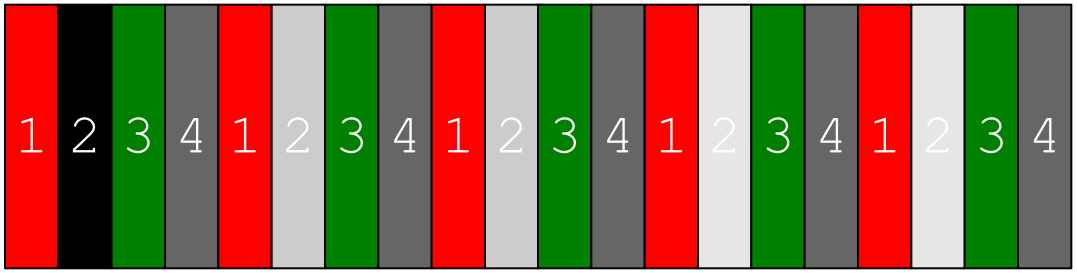
Histogram

- **Los accesos a memoria son ineficientes cuando se particiona por secciones:**
  - **Hilos adyacentes no acceden a posiciones adyacentes.**
  - **Los accesos no son coalesced.**
  - **El ancho de banda de acceso a la RAM se desaprovecha.**



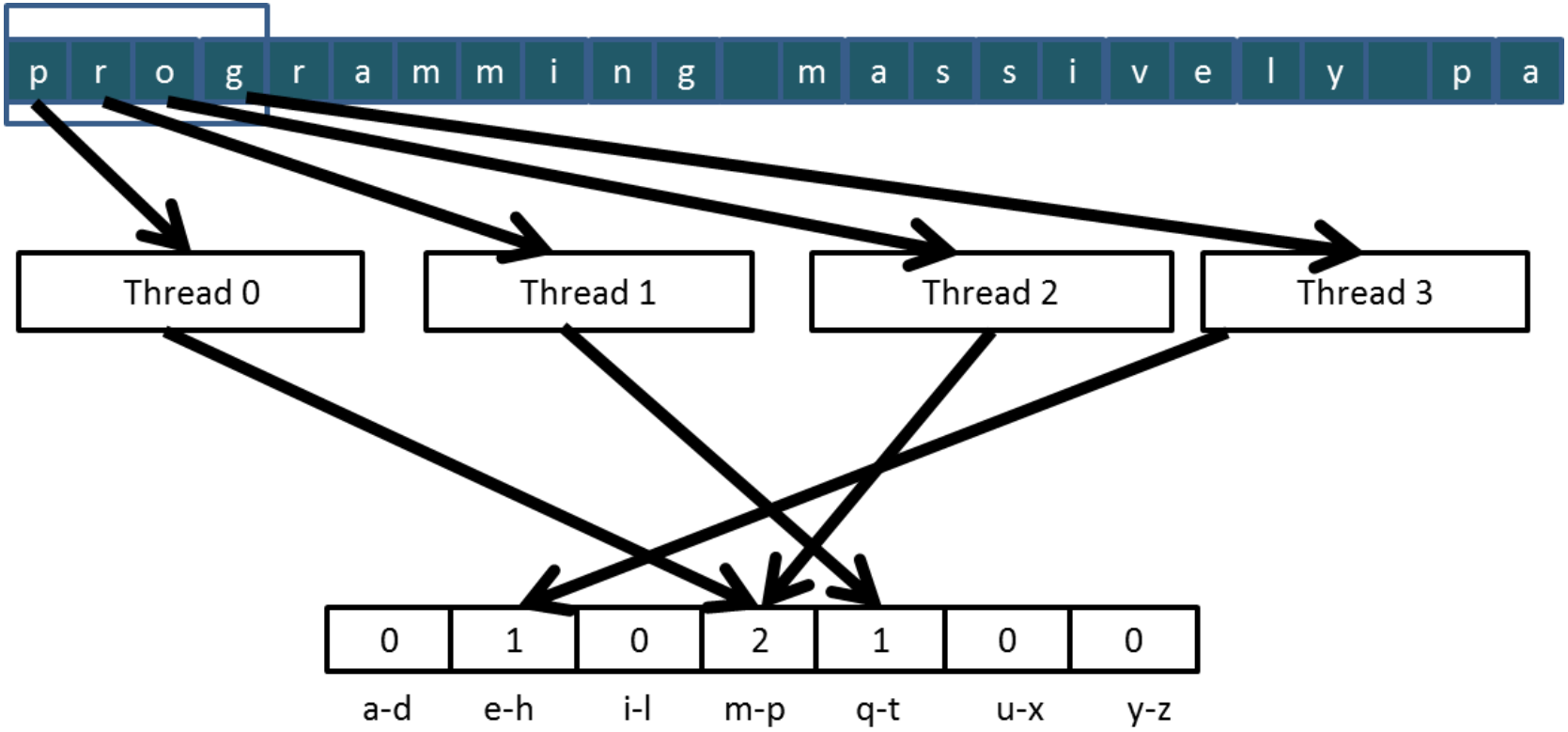
Histogram

- **El acceso intercalado da un mejor patrón de acceso a los datos:**
  - **Los hilos en su conjunto procesan una sección contigua de elementos.**
  - **Después de procesar una sección, se mueven a la siguiente y repiten el procedimiento.**
  - **Los accesos a memoria son coalesced.**



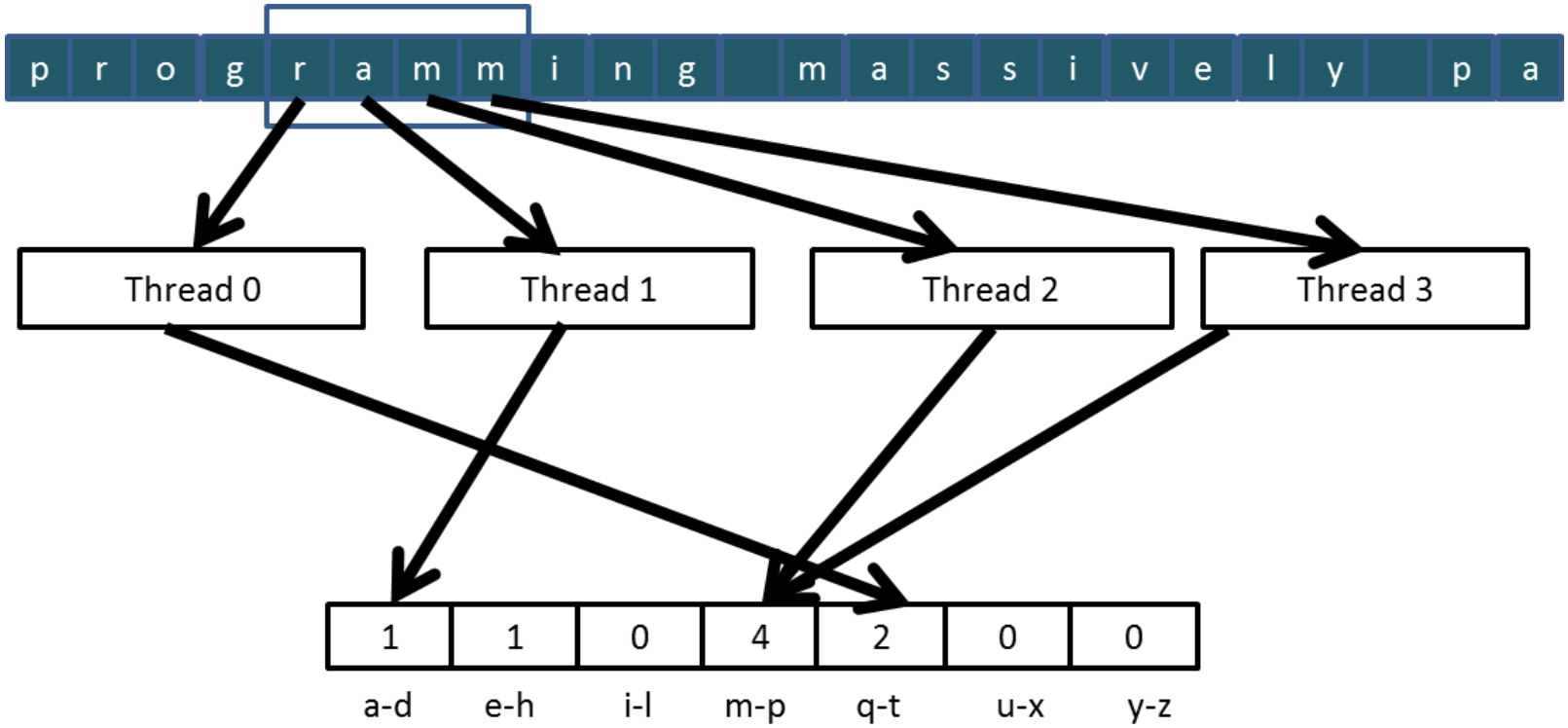
Histogram

Primera iteración



Histogram

Segunda iteración



Histogram

- **Condiciones de carrera:**
  - **Pueden surgir cuando se realizan operaciones del tipo read-modify-write.**
  - **Pueden producir errores muy difíciles de reproducir.**
  - **Las operaciones atómicas permiten evitar las condiciones de carrera.**

Histogram

Thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

- **Old and New son registros propios de los hilos.**
- **¿Qué valor tendrá Mem[x] después de haber completado la ejecución de los hilos.**
- **¿Qué valor almacenan en Old cada hilo?**
- **Los valores pueden variar de acuerdo al orden relativo de ejecución.**
- **A esto se conoce con el nombre de condición de carrera.**

Histogram

Ejemplo de ejecución 1:

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

Thread 1 Old = 0

Thread 2 Old = 1

Mem[x] = 2 después de la ejecución

Histogram

Ejemplo de ejecución 2:

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

Thread 1 Old = 1

Thread 2 Old = 0

Mem[x] = 2 después de la ejecución

Histogram

Ejemplo de ejecución 3:

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 después de la ejeución

Histogram

Ejemplo de ejecución 4:

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 después de la ejecución

Histogram

Las operaciones atómicas garantizan que los resultados sean los correctos.

Thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Histogram

- **Operaciones atómicas:**
  - **Una operación read-modify-write es ejecutada por una única instrucción del hardware en una dirección de memoria.**
  - **Una sola operación lee el valor viejo, calcula el nuevo valor y escribe el nuevo valor en la memoria.**
  - **Es el hardware el que garantiza que ningún otro hilo pueda realizar una operación read-modify-write en la misma dirección de memoria hasta que la ejecución de la operación atómica se haya completado.**
  - **Cualquier otro hilo que intente realizar una operación atómica en la misma dirección será puesta en espera.**
  - **Es decir que los hilos ejecutan las operaciones atómicas en una dirección de forma secuencial.**

Histogram

- **Operaciones atómicas:**
  - **Se ejecutan llamando a funciones como: Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)**
  - **Están explicadas en la CUDA programming guide**

- **Atomic Add:**
  - **int atomicAdd (int* address, int val);**
  - **Lee la palabra de 32 bits apuntada por la dirección address en memoria global o compartida, le suma val y almacena el resultado en la misma dirección.**
  - **Ojo: La función retorna el valor leído no el almacenado.**

Histogram

- **Una implementación básica:**
  - **- Unsigned int de 32 bits:**
  - *unsigned int atomicAdd(unsigned int* address, unsigned int val);*
  - **- Unsigned int de 64 bits:**
  - *unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);*
  - **- Simple precisión:**
  - *float atomicAdd(float* address, float val);*

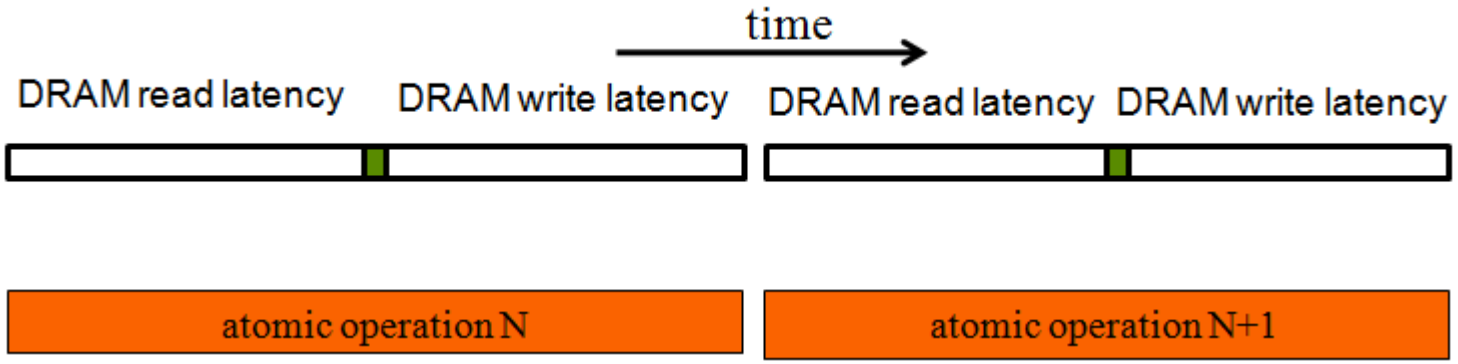
Histogram

- **Por ejemplo:**
  - **El kernel recibe un puntero al buffer.**
  - **Cada hilo procesa los elementos separados por un paso (stride)**

```
__global__ void histo_kernel(unsigned char *buffer,
        long size, unsigned int *histo)
{
   int i = ...
   int stride = ...

// consecutive elements
   while (i < size) {
       atomicAdd( &(histo[buffer[i]]), 1);
       i += stride;
   }
}
```

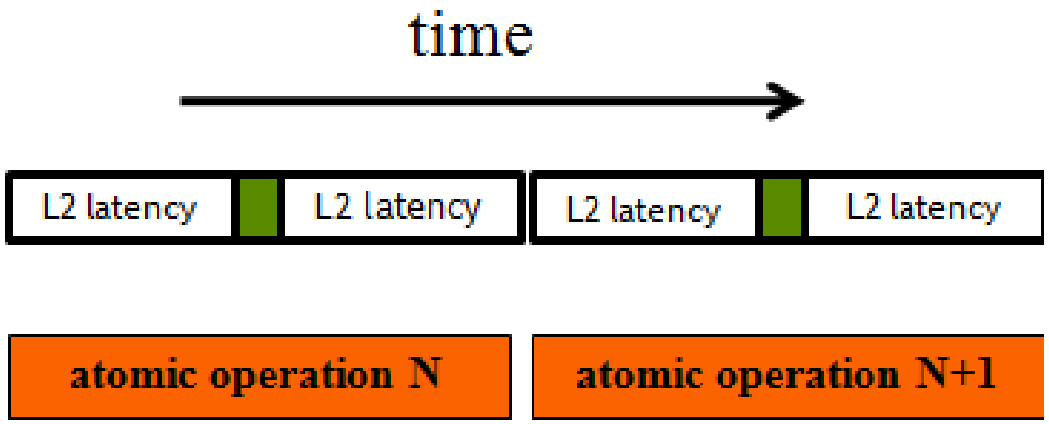
Histogram

- **Operaciones atómicas en memoria global:**
  - **La lectura tiene una latencia de algunos cientos de ciclos.**
  - **La escritura también tiene una latencia similar.**
  - **Mientras se realiza la operación nadie puede acceder a la dirección.**



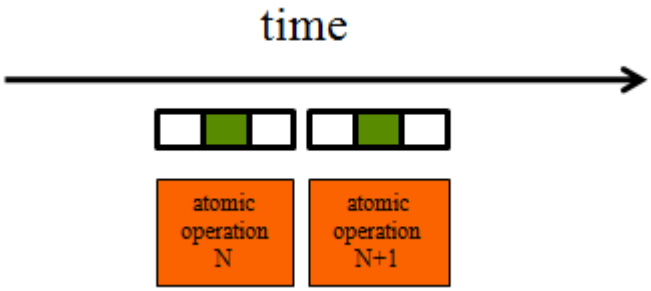
Histogram

- **Operaciones atómicas en cache L2:**
  - **Latencia media, alrededor de 1/10 de la latencia de la RAM.**
  - **Se comparte entre todos los bloques.**
  - **Es una mejora con respecto a los atomics sobre memoria global.**



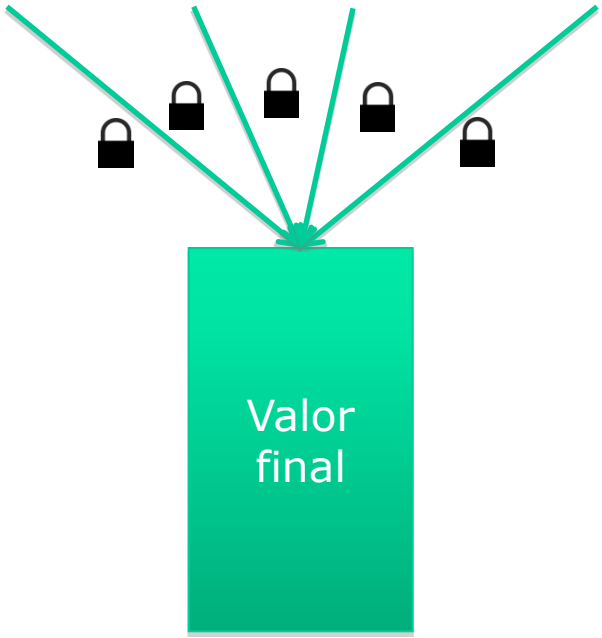
Histogram

- **Operaciones atómicas en memoria compartida:**
  - **La latencia es muy corta.**
  - **Es restringido a cada bloque.**
  - **Requiere esfuerzo por parte del programador.**



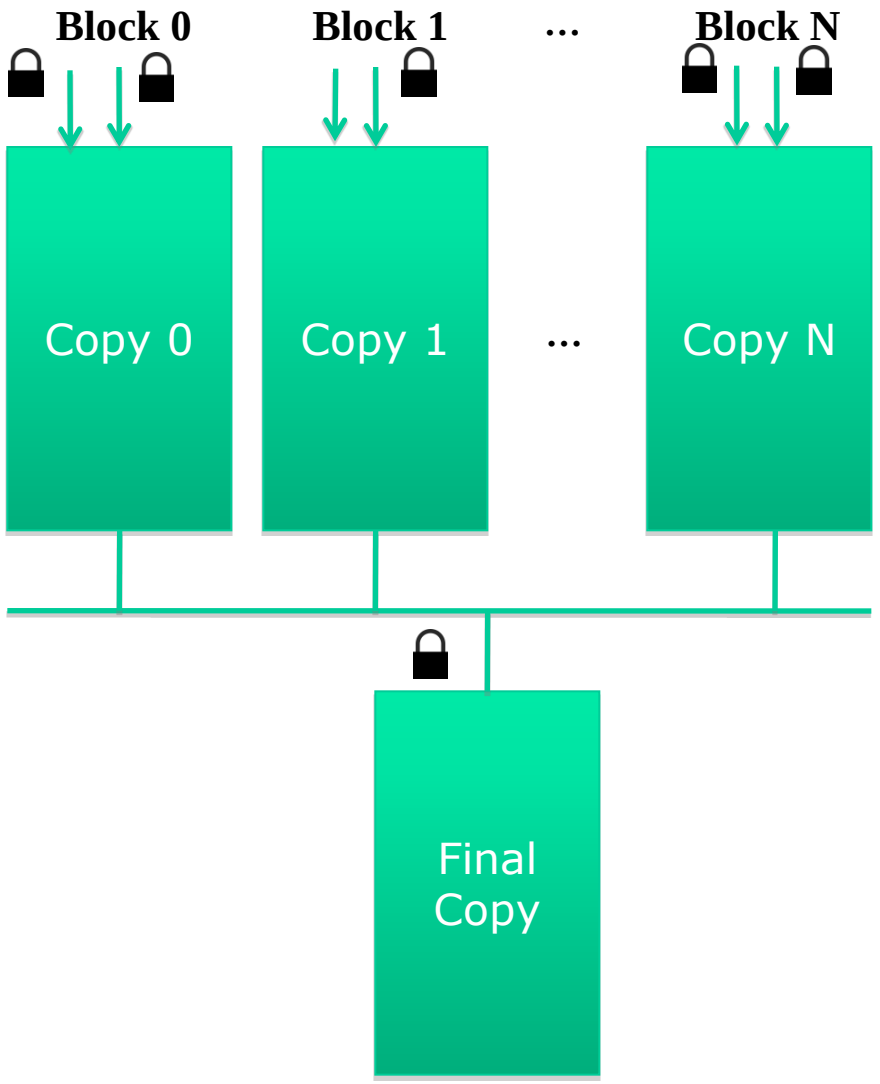
Histogram

- **Privatization:**
  - **Es una técnica que se usa para reducir la latencia, aumentar el rendimiento y reducir la serialización.**



Histogram

- **Privatization:**



Reduce

Reduce

- Reducción en paralelo de la suma.
- Los datos se copian inicialmente de la memoria global a la memoria compartida.
- Solamente se realiza la primera etapa de la reducción.
- La suma de subtotales podría ser en:
  - GPU: haciendo más etapas de reducción.
  - CPU: transfiriendo a CPU los resultados intermedios y haciendo una iteración en CPU.

Reduce

- **Para declarar la memoria compartida se puede usar una constante:**

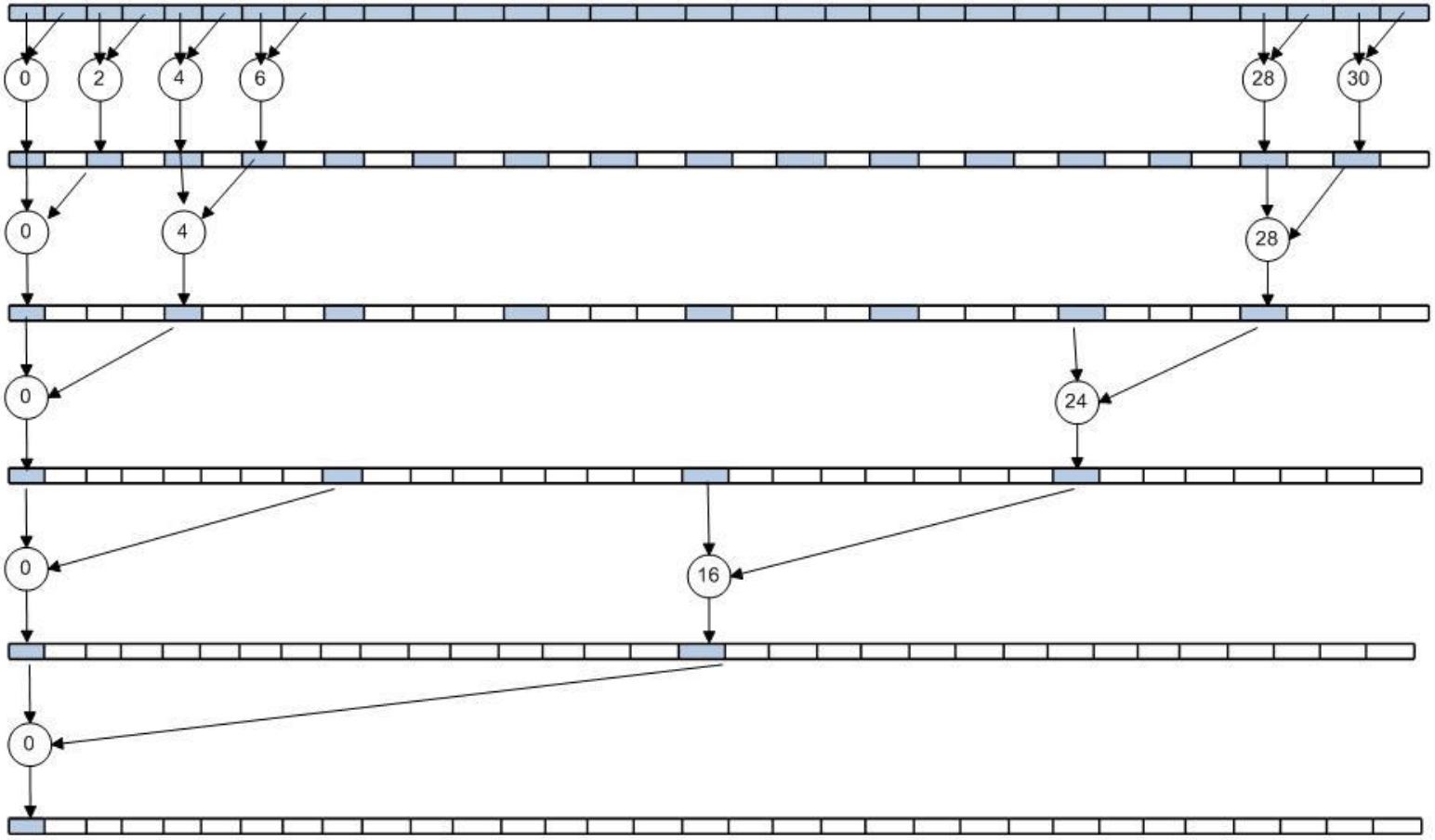
```
#define CANT_HILOS 128

__global__ void reduction(float * output, float * input) {
  __shared__ float intermedio[CANT_HILOS];
  …
};

reduction <<<N_BLOCK,CANT_HILOS>>> (output,input);
```

- **Para declarar la memoria compartida se puede hacer en forma dinámica con extern:**

```
__global__ void reduction(float * output, float * input) {
  extern __shared__ float intermedio[];
  …
};

reduction <<<N_BLOCK,CANT_HILOS,
             CANT_HILOS*sizeof(float)>>> (output,input);
```

Reduce

Primer enfoque



Reduce

Primer enfoque

```
__global__ void reduction(float * output, float * input){

   extern __shared__ float intermedio[];
   int step = 1;

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   intermedio[threadIdx.x] = input[idx];

    __syncthreads();

    while (step < blockDim.x) {
        if (threadIdx.x%(2*step)==0)
             intermedio[threadIdx.x] =
             intermedio[threadIdx.x] + intermedio[threadIdx.x + step];
        __syncthreads();
        step = step * 2;
    }

    __syncthreads();
    if (threadIdx.x==0)
        output[blockIdx.x] = intermedio[0];
}
```

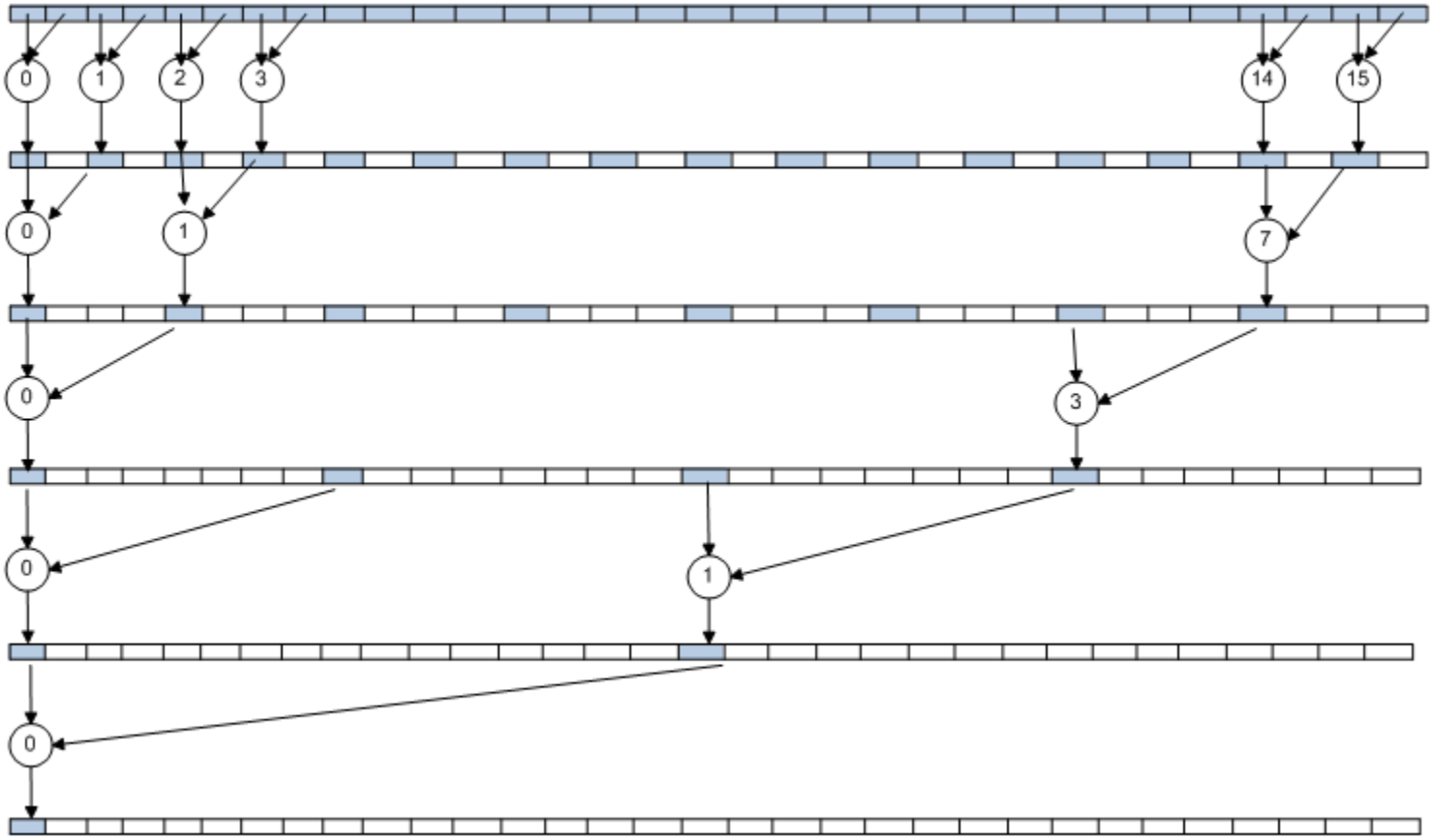
Reduce

Primer enfoque

- Bloques 5000
- Hilos 128
- Ejecutado en loop 5000 x 128 veces en una 9800 GTX+ (Comp. Cap 1.1, 512 MB, 128 CUDA cores), el tiempo de ejecución es aproximadamente 505 s.
- La implementación tiene varios problemas.
- El más importante es la divergencia de warps.

Reduce

Segundo enfoque



Reduce

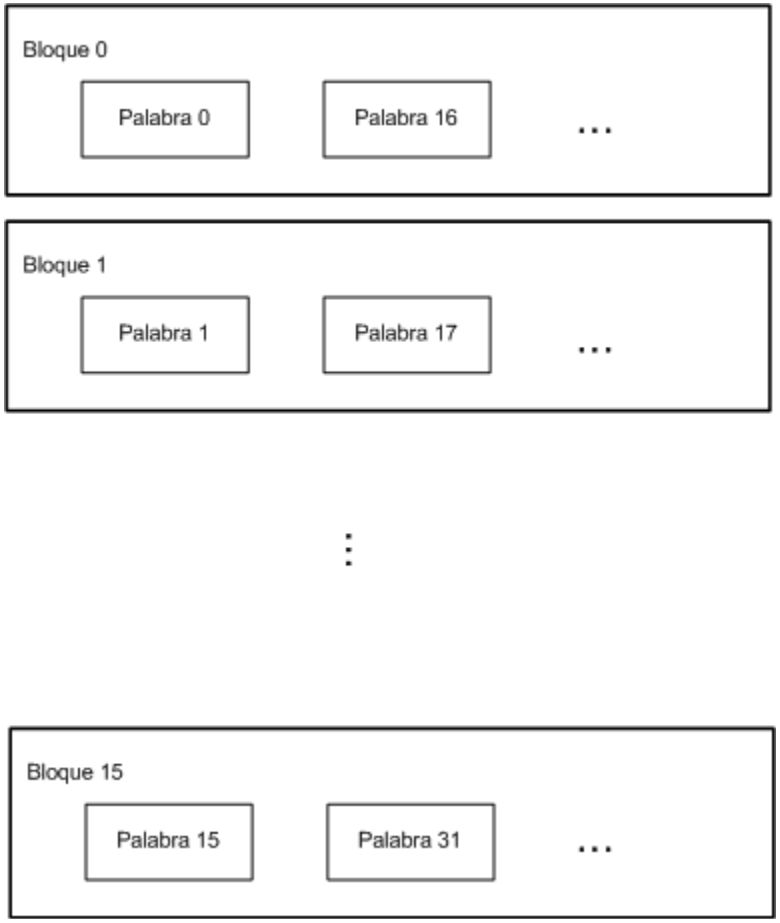
Segundo enfoque

```
__global__ void reduction(float * output, float * input){

   extern __shared__ float intermedio[];
   int step = 1;

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   intermedio[threadIdx.x] = input[idx];

   __syncthreads();

   while (step < blockDim.x) {
        if (threadIdx.x < blockDim.x/(2*step))
          intermedio[2*step*threadIdx.x] =
   intermedio[2*step*threadIdx.x]+intermedio[2*step*threadIdx.x+step];
        __syncthreads();
        step = step * 2;
    }

    __syncthreads();
    if (threadIdx.x==0)
        output[blockIdx.x] = intermedio[0];
}
```

Reduce

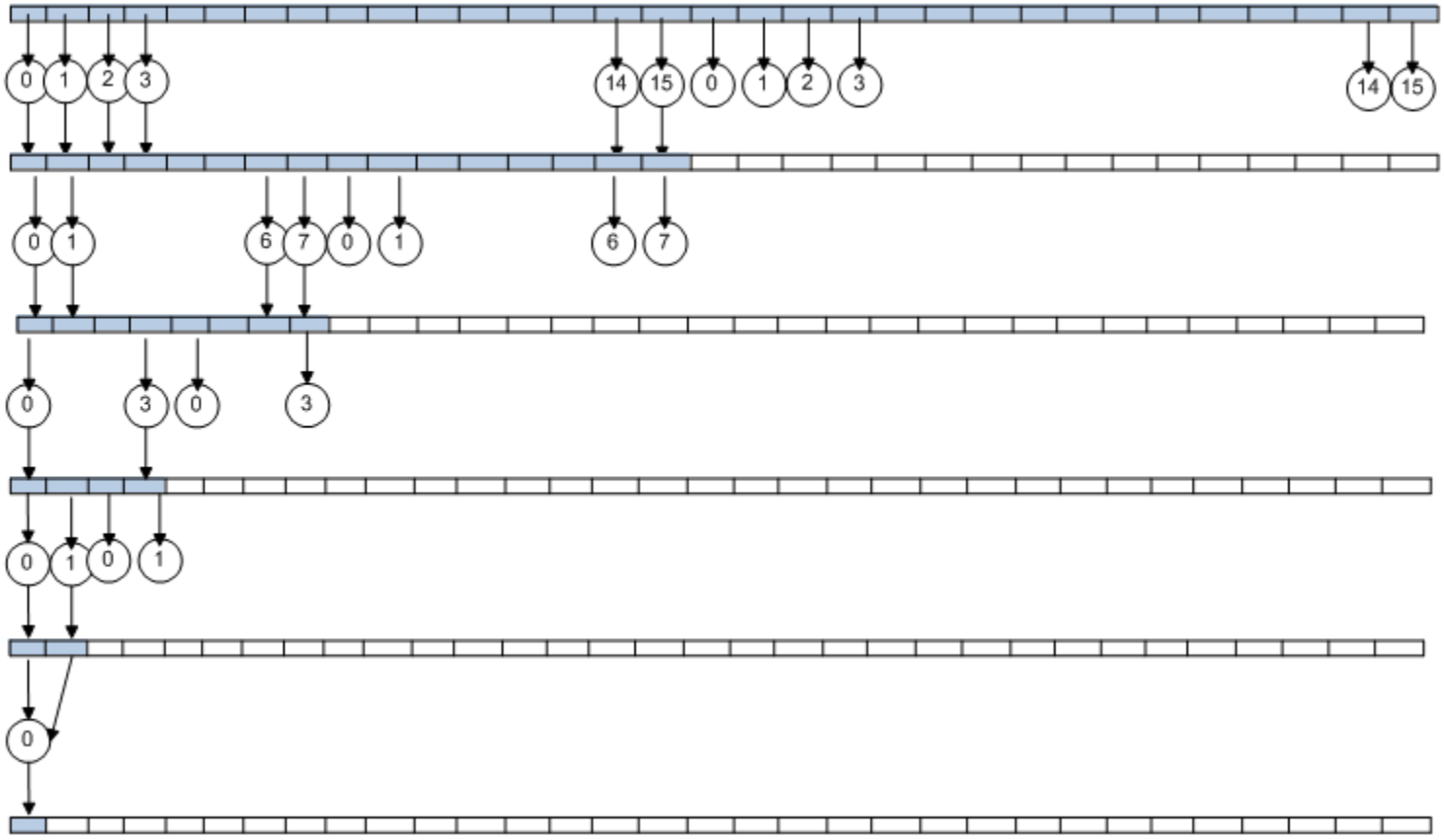
Segundo enfoque

- Bloques 5000
- Hilos 128
- Ejecutado en loop 5000 x 128 veces en una 9800 GTX+ (Comp. Cap 1.1, 512 MB, 128 CUDA cores), el tiempo de ejecución es aproximadamente 182 s. (mejora 2.77)
- Hay conflictos de bancos en el acceso a memoria compartida.



Reduce

Tercer enfoque



Reduce

Tercer enfoque

```
__global__ void reduction(float * output, float * input){

   extern __shared__ float intermedio[];
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   intermedio[threadIdx.x] = input[idx];
   __syncthreads();

   int i = blockDim.x/2;
   while (i != 0) {
        if (threadIdx.x < i)
             intermedio[threadIdx.x] =
                intermedio[threadIdx.x] + intermedio[threadIdx.x + i];
        __syncthreads();
        i = i / 2;
    }

    __syncthreads();
    if (threadIdx.x==0)
        output[blockIdx.x] = intermedio[0];
}
```

Reduce

Tercer enfoque

- **Bloques 5000**
- **Hilos 128**
- **Ejecutado en loop 5000 x 128 veces en una 9800 GTX+ (Comp. Cap 1.1, 512 MB, 128 CUDA cores), el tiempo de ejecución es aproximadamente 153 s. (mejora 3.30).**
- **Y si movemos el primer `syncthreads()` para adentro del if, así solamente sincronizamos los hilos que suman??**

  - Ojo!!! Son todos los hilos!!!

Reduce

Otras optimizaciones …

- **loop unrolling**
- **leer más de una palabra de la memoria global**
- **lower occupancy**