

COMPUTE UNIFIED DEVICE ARCHITECTURE

BEST PRACTICES



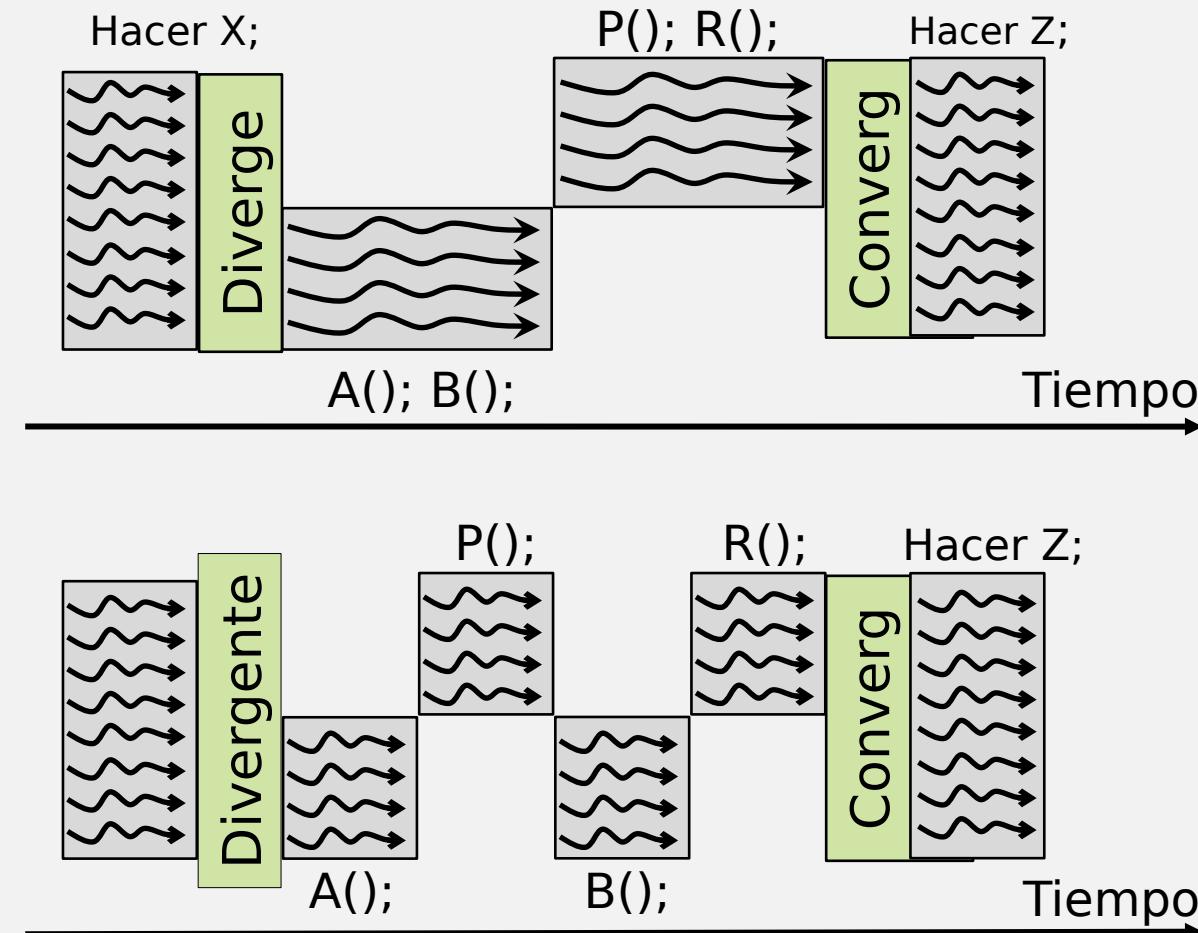
SALTOS DIVERGENTES (*BRANCH DIVergENCE*)

- La importancia de los *Saltos Divergentes* es a nivel de *warp*.
(Un warp es un grupo de 32 hilos (también hay *half-* y *quarter-warps*)).
- Todos los hilos de un warp comienzan en la misma dirección de programa, pero tienen su propio contador de direcciones de instrucción por lo que tienen independencia de ejecución.
- Un warp ejecuta una instrucción común a cada instante de tiempo:
 - La mayor productividad se alcanza cuando todos los hilos del warp ejecutan el camino programado.
 - Si los hilos del warp divergen (p. ej. salto condicional) el warp ejecuta *secuencialmente* todos los caminos.
 - En cada camino los hilos que no estén en él son deshabilitados.
 - Ejecutados todos los posibles caminos, los hilos convergen en su ejecución.
- Evitar los saltos divergentes por la pérdida de rendimiento que implican (en CUDA todo puede cambiar).

SALTOS DIVERGENTES (BRANCH DIVERSION)

- PreVolta: todas las sentencias de un camino se ejecutan antes de cualquier sentencia del otro camino de ejecución.

```
Hacer X;
if (threadIdx.x < 4) {
    A();
    B();
} else {
    P();
    R();
}
Hacer Z;
```



- Volta: permiten la *interleaved execution* de las sentencias de los distintos caminos de ejecución.

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- Secuencial en CPU:

```
...
    suma=0.0;
    for (i=0; i<n; i++)
        suma += v[i];
...
...
```

- Paralelo OpenMP CPU:

```
...
    suma=0.0;
#pragma omp parallel for reduction(+: suma) num_threads(ncores)
    for (i=0; i<n; i++)
        suma += v[i];
...
...
```

- Explota localidad espacial

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- Paralelo OpenMP CPU “*handmade localidad espacial CPU*”:

```
...
    ncores = omp_get_num_procs();
    for (i=0; i<ncores; i++) { tmpVec[i]=0.0; }
    #pragma omp parallel num_threads(ncores) private(hilo)
    {
        hilo=omp_get_thread_num();
        for (i=0; i<(n/ncores); i++)
            tmpVec[hilo] += v[hilo*(n/ncores)+i];
    }
    suma=0.0;
    for (i=0; i<ncores; i++) { suma += tmpVec[i]; }
...
}
```

- ¿Mejor, peor o igual que el anterior?

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- Paralelo OpenMP CPU “*handmade rompiendo localidad espacial CPU*”

```
...
    ncores = omp_get_num_procs();
    for (i=0; i<ncores; i++) { tmpVec[i]=0.0; }
    #pragma omp parallel num_threads(ncores) private(hilo)
    {
        hilo=omp_get_thread_num();
        for (i=0; i<n; i+=ncores)
            tmpVec[hilo] += Vector[hilo+i];
    }
    suma=0.0;
    for (i=0; i<ncores; i++) { suma += tmpVec[i]; }
...
}
```

- ¿Mejor, peor o igual que los anteriores?

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- 16 cores CPU:

<code>./Coalescente (n=2²³= 8388608)</code>	
Tiempo secuencial	1.6161119E-02
Tiempo paralelo reduction	4.1111679E-02
Tiempo paralelo hand made	2.7210464E-02
Tiempo paralelo bad	3.0115871E-02
<code>./Coalescente (n=2²⁴=16777216)</code>	
Tiempo secuencial	3.2298271E-02
Tiempo paralelo reduction	3.6256226E-02
Tiempo paralelo hand made	2.6870016E-02
Tiempo paralelo bad	4.4180351E-02
<code>./Coalescente (n=2²⁵=33554432)</code>	
Tiempo secuencial	4.6381504E-02
Tiempo paralelo reduction	3.0885887E-02
Tiempo paralelo hand made	2.8962816E-02
Tiempo paralelo bad	7.5936546E-02

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- CUDA “*localidad espacial CPU*”:

```
localidad<<<1, 32>>>(v, n);
...
__global__ void localidad(double *Vector, const int n) {
    __shared__ double parciales[32];
    int   i, chunk=n/32;
    double suma=0.0;
    for (i=0; i<chunk; i++)
        suma += Vector[threadIdx.x*chunk+i];
    parciales[threadIdx.x]=suma;  suma=0.0;
    __syncthreads();
    if (threadIdx.x == 0) {
        for (i=0; i<32; i++) {suma += parciales[i]; }
        Vector[0] = suma;
    }
}
```

COALESCENCIA

Sea v un vector de tamaño n . Resolver $\text{suma} = \sum_{i=0}^{n-1} v[i]$

- CUDA “*rompiendo localidad espacial CPU*”:

```
sin_localidad<<<1, 32>>>(v, n);
...
__global__ void sin_localidad(double *Vector, const int n) {
    __shared__ double parciales[32];
    int   i;
    double suma=0.0;
    for (i=0; i<n; i+=32)
        suma += Vector[threadIdx.x+i];
    parciales[threadIdx.x]=suma;  suma=0.0;
    __syncthreads();
    if (threadIdx.x == 0) {
        for (i=0; i<32; i++) {suma += parciales[i]; }
        Vector[0] = suma;
    }
}
```

COALESCENCIA

Sea v un vector de tamaño n . Resolver suma = $\sum_{i=0}^{n-1} v[i]$

- 16 cores CPU y kernels GPU con 1 bloque de 32 hilos:

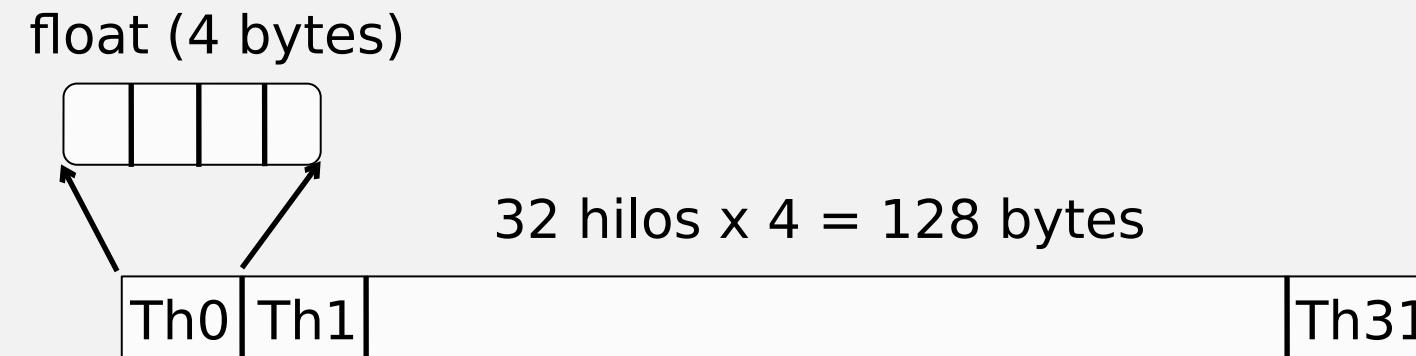
./Coalescente ($n=2^{23}= 8388608$)	
Tiempo secuencial	1.6161119E-02
Tiempo paralelo reduction	4.1111679E-02
Tiempo paralelo hand made	2.7210464E-02
Tiempo paralelo bad	3.0115871E-02
Tiempo GPU localidad	3.7153248E-02
Tiempo GPU sin_localidad	1.4544736E-02
./Coalescente ($n=2^{25}=33554432$)	
Tiempo secuencial	4.6381504E-02
Tiempo paralelo reduction	3.0885887E-02
Tiempo paralelo hand made	2.8962816E-02
Tiempo paralelo bad	7.5936546E-02
Tiempo GPU localidad	1.4267780E-01
Tiempo GPU sin_localidad	5.0520161E-02

COALESCENCIA

- *Localidad Espacial* CPU: Orientada a reducir el número de accesos a memoria central a nivel de hilo.
- El enfoque de la localidad espacial en CPU no funciona bien en GPU, como se ha visto en el ejemplo anterior.
- Coalescencia es la *Localidad Espacial* GPU: Orientada a reducir el número de accesos a nivel de warp.
- Cuando un *warp* accede a memoria global junta (**Coalescencia**) los accesos de los hilos dentro del *warp*.
- Una operación de acceso a memoria global puede tener que *repetirse* múltiples veces dependiendo de la distribución de los datos (distribución de los accesos a memoria) entre los hilos del *warp*.
- El acceso a la memoria global se realiza en transacciones de 32, 64 ó 128 bytes.
- Los segmentos de memoria que estén alineados (cuya primera dirección sea múltiplo del tamaño del dispositivo) pueden ser leídos o escritos por transacciones de memoria.
- Lo anterior afecta al rendimiento, más o menos, en función del tipo de memoria al que se esté accediendo.
- A mayor dispersión de las direcciones mayor pérdida de rendimiento.
- Se debe maximizar la coalescencia para un mayor rendimiento en los accesos a memoria global.

COALESCENCIA

- Un acceso se traduce en una sola instrucción de acceso a memoria global si y solo si el tamaño del tipo es 1, 2, 4, 8 o 16 bytes y los datos están alineados de forma natural.
- Los requisitos de alineación se cumplen automáticamente para los tipos incorporados (*char, short, int, long, float, doble, etc.*).
- Para que estos accesos sean totalmente coalescentes la anchura del bloque de hilos y de la estructura de datos deben ser múltiplos de *warp*.

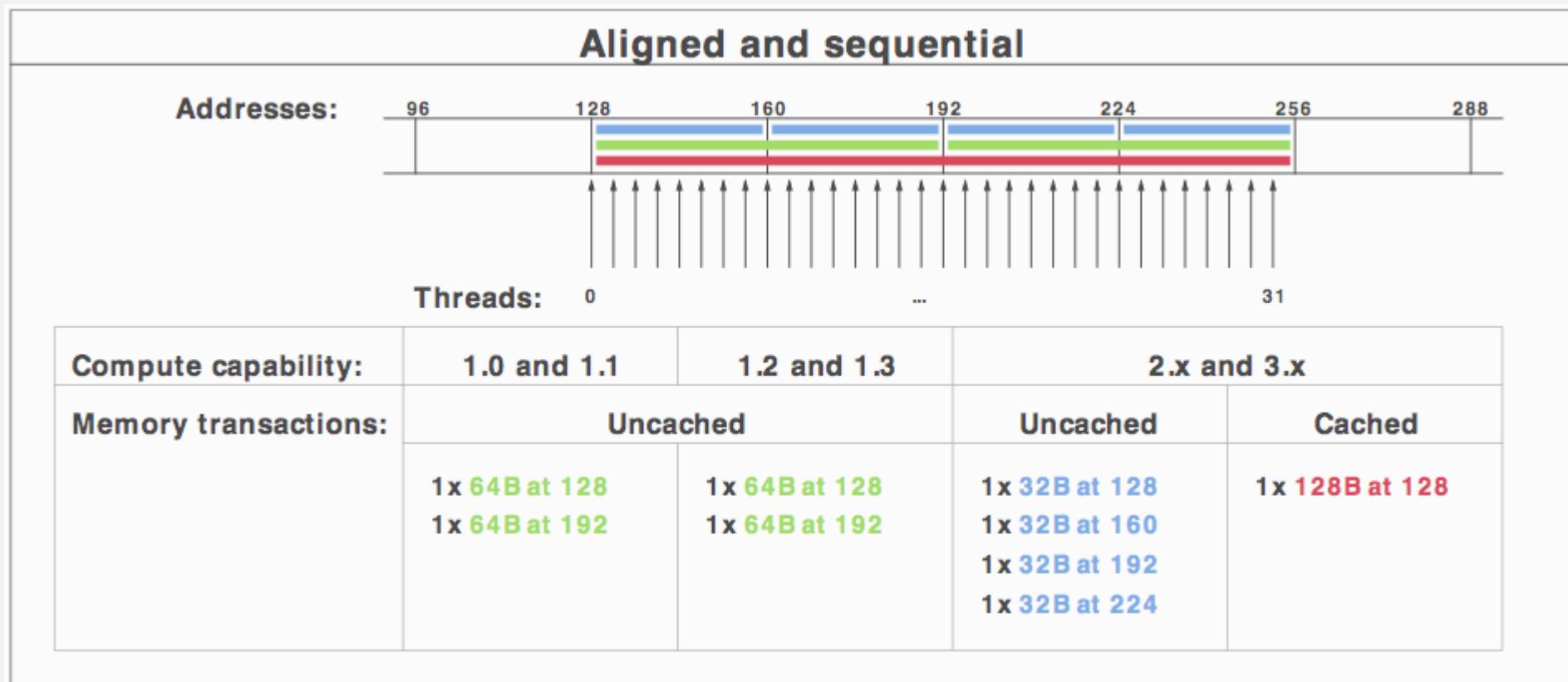


- Un patrón de acceso a memoria *global* para una matriz 2D *adecuado* es aquel donde un hilo genérico (tx, ty) usa $\text{BaseAddress} + \text{width} * ty + tx$ para acceder a su elemento, siendo *width* la anchura de la matriz.

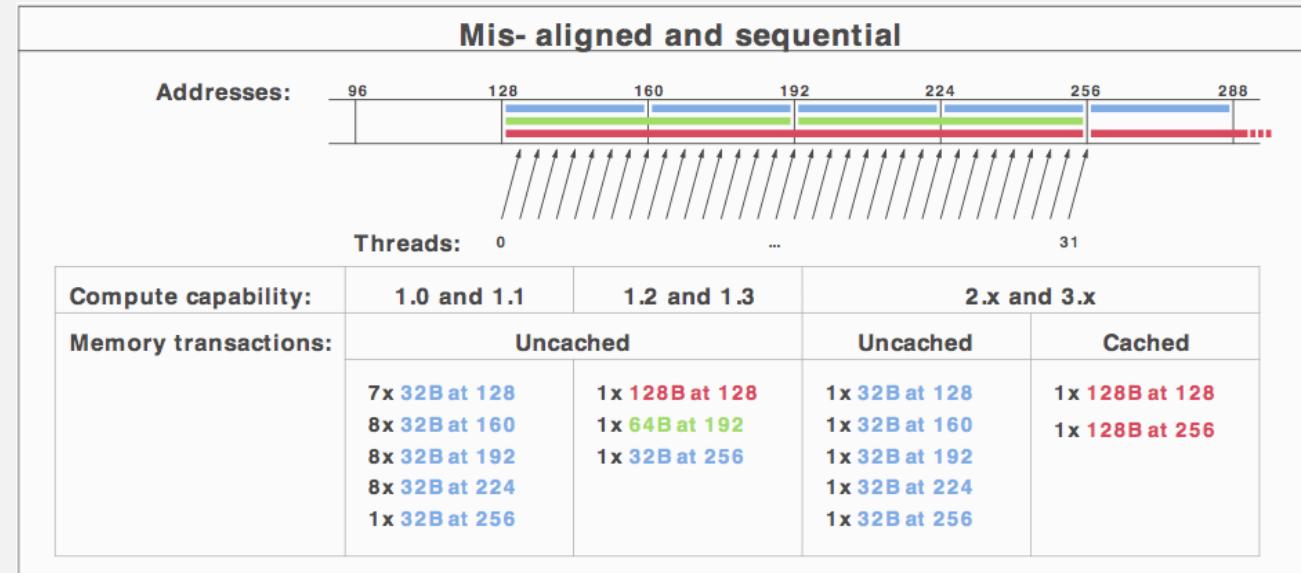
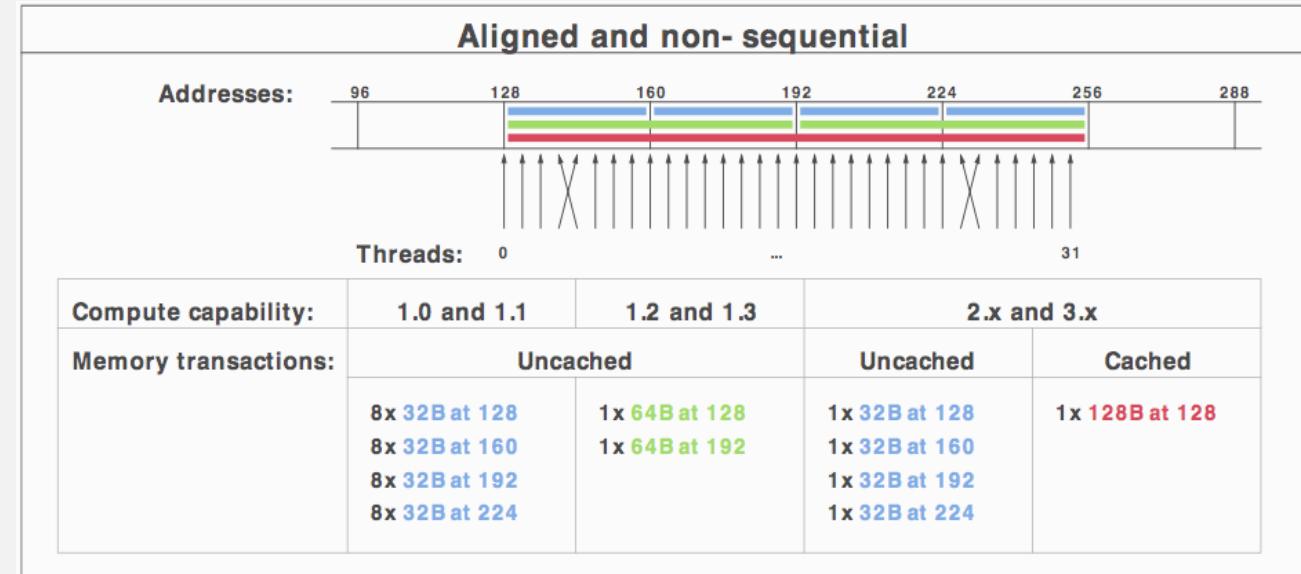
COALESCENCIA

Guía para maximizar la productividad en el acceso a memoria:

- Usar los patrones de acceso óptimos para cada arquitectura (CUDA Compute Capability).
- Usar tipos de datos que cumplan con el requisito de alineación a 32, 64 ó 128 bytes.
- *Padding, shared memory* y la existencia de la caché ayudan a mitigar el problema de la coalescencia.



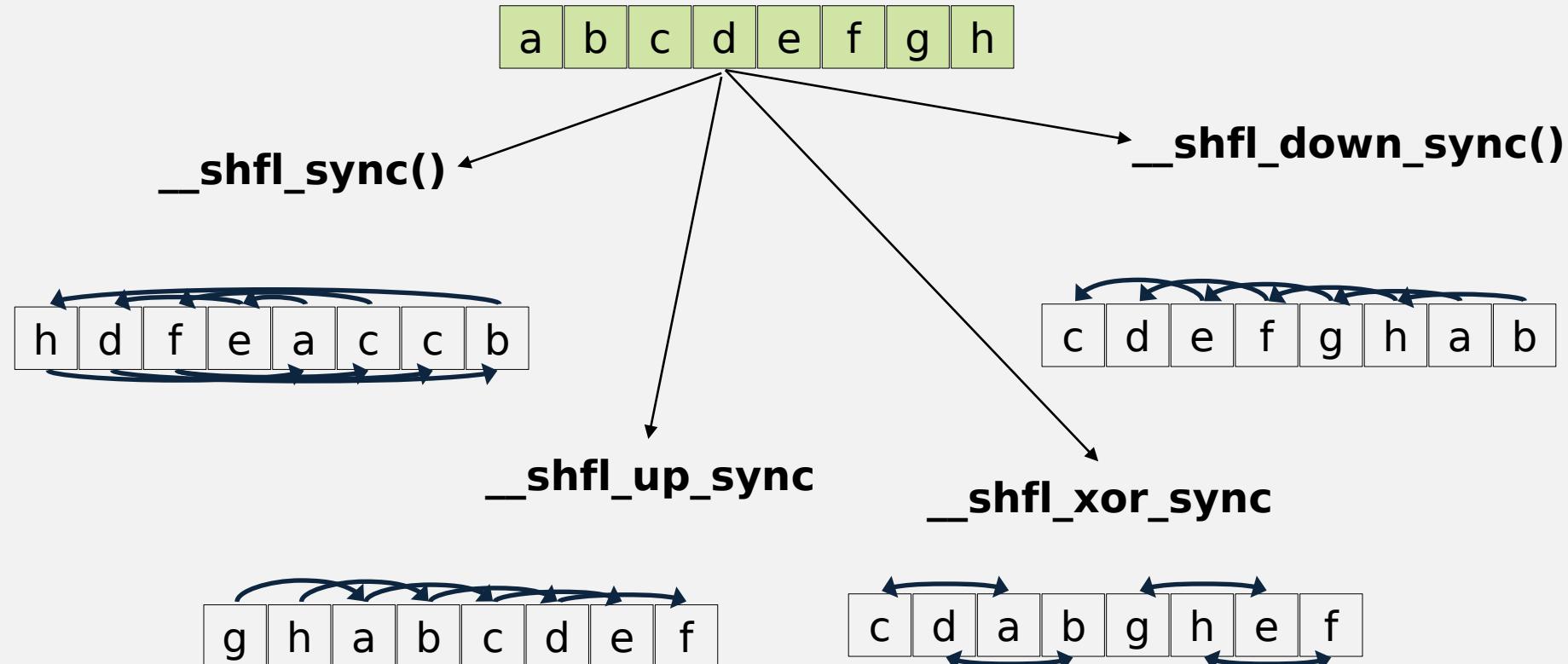
COALESCENCIA



WARP-LEVEL PRIMITIVES

Para gestionar la interacción entre los hilos de un warp a *bajo nivel*.

- Tres tipos: intercambio, máscaras (`_activemask`) y sincronización (`_syncwarp`)
- Evita el uso de *shared memory* y pueden mejorar el rendimiento.

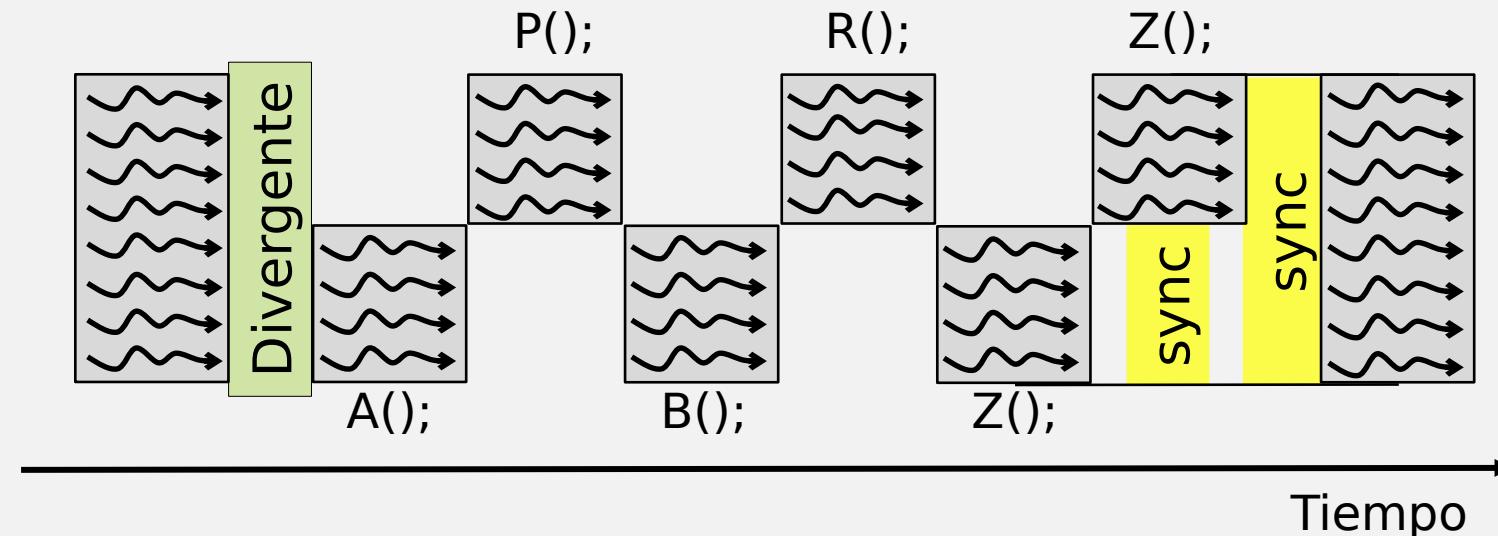


WARP-LEVEL PRIMITIVES

El caso `_syncwarp()`

- En Volta no es necesario para otras *warp-level primitives*, pero puede serlo para otro tipo de operaciones.

```
X();  
if (threadIdx.x < 4) { Hacer A(); Hacer B(); }  
else { Hacer P(); Hacer R(); }  
Z();  
_syncwarp();
```



WARP-LEVEL PRIMITIVES

El caso `_shfl_down_sync()`

```
int __shfl_down_sync(unsigned mask, int v, unsigned s, int width=warpSize)
```

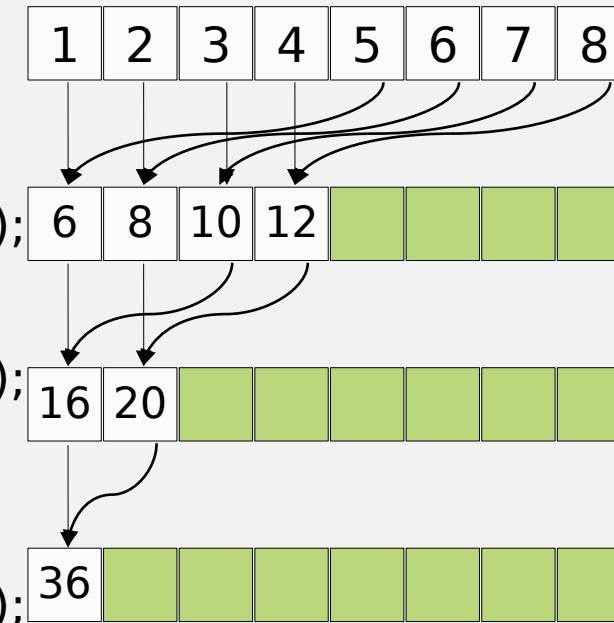
- **mask** Mascara de 32-bit que denota los hilos del warp que participan en la operación. Lo primero que hace la primitiva es sincronizar los hilos que participan (si no están ya sincronizados).

- `unsigned m=0xffffffff`

```
v += __shfl_down_sync(m, v, 4);
```

```
v += __shfl_down_sync(m, v, 2);
```

```
v += __shfl_down_sync(m, v, 1);
```



WARP-LEVEL PRIMITIVES

Para ajustar la máscara si el número de hilos es menor que el tamaño del warp

```
#define MASK 0xffffffff  
__inline__ __device__ double ReduceWarpFull(double val, const  
unsigned int N) {  
    unsigned mask = __ballot_sync(MASK, threadIdx.x < N);  
    if (threadIdx.x < N)  
        for (unsigned int offset = 16; offset > 0; offset /= 2)  
            val += __shfl_down_sync(mask, val, offset);  
}
```

En Volta (y posteriores) funcionan con saltos divergentes

```
if (threadIdx.x % 2) {  
    ....  
    val += __shfl_sync(MASK, val, 0);  
} else {  
    val += __shfl_sync(MASK, val, 0);  
    ...
```

WARP-LEVEL PRIMITIVES

Resolver

- CUDA “coalescente”:

```
__global__ void sin_localidad(double *Vector, const int n) {  
    __shared__ double parciales[32];  
    double suma = 0.0;  
    for (unsigned int i=0; i<n; i+=32)  
        suma += Vector[threadIdx.x+i];  
    parciales[threadIdx.x]=suma;  suma=0.0;  
    __syncthreads();  
  
    if (threadIdx.x == 0) {  
        for (unsigned int i=0; i<32; i++)  {summa += parciales[i]; }  
        Vector[0] = summa;  
    }  
}
```

WARP-LEVEL PRIMITIVES

Resolver

```
#define MASK 0xffffffff
__inline__ __device__ double ReduceWarp(double val) {
    for (unsigned int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(MASK, val, offset, 32);
    return val;
}

__global__ void warp_prim(double *V, const unsigned int N)
{
    double suma=0.0;
    for (unsigned int i=0; i<N; i+=32)
        suma += V[threadIdx.x+i];
    suma=ReduceWarp(suma);
    if (threadIdx.x == 0)    Vector[0] = suma;
}
```

WARP-LEVEL PRIMITIVES

Resolver

- <<<1, 32>>>(Vector, n);

./Coalescente (n=2 ²³ =8388608)	
Tiempo GPU localidad	2.9422655E-02
Tiempo GPU sin_localidad	1.4593856E-02
Tiempo GPU warp_prim	1.4574624E-02
./Coalescente (n=2 ²⁴ =16777216)	
Tiempo GPU localidad	5.8586655E-02
Tiempo GPU sin_localidad	2.9044737E-02
Tiempo GPU warp_prim	2.9019840E-02
./Coalescente (n=2 ²⁵ =33554432)	
Tiempo GPU localidad	1.1697629E-01
Tiempo GPU sin_localidad	4.7669121E-02
Tiempo GPU warp_prim	4.1792000E-02

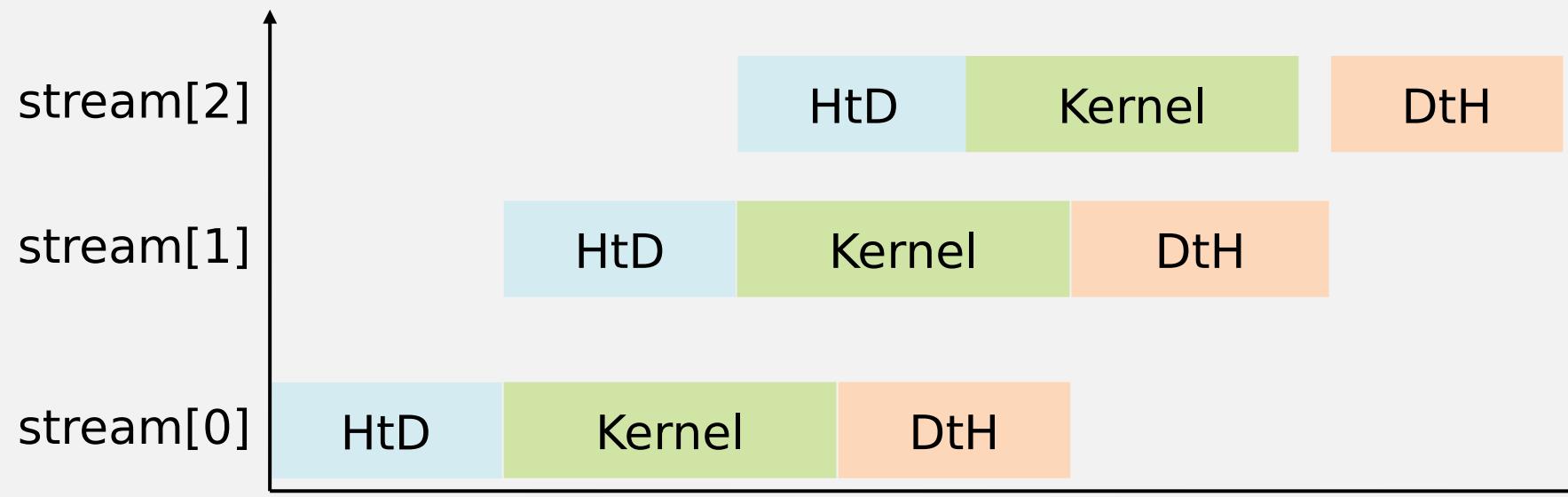
STREAMS

- Un *Stream* es una secuencia de comandos que se ejecutan en orden (equivalencia: cola FIFO).
- Si no se especifica los comandos van al *stream por defecto*, que siempre existe.
- Los *streams* se ejecutan concurrentemente entre sí.
- Cada hilo CPU puede tener su “*stream por defecto*” independiente (con “nvcc --default-stream=per-thread”).

Ejemplo de uso

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
...  
cudaMalloc (&dev1, size);  
cudaMallocHost (&host1, size); //tiene que ser pinned  
...  
cudaMemcpyAsync(dev1, host1, size, dir, stream1);  
kernel2<<<grid, block, 0, stream2>>>(..., dev2, ...);  
...  
alguna_función_cpu()
```

STREAMS



- `cudaDeviceSynchronize()` Espera a que todos los comandos anteriores en todas las secuencias de todos los hilos de *Host* estén completados.
- `cudaStreamSynchronize()` Espera hasta que todos los comandos anteriores en la secuencia dada como parámetro estén completados.
- `cudaStreamWaitEvent()` Partiendo de un *stream* y un evento hace que todo el flujo de ese *stream* pare su ejecución hasta que el evento dado se haya completado.
- `cudaStreamQuery()` Permite saber si todos los comandos anteriores de un *stream* han finalizado.

STREAMS

Plantilla

// Creando

```
cudaStream_t *streams=(cudaStream_t*)malloc(nstr*sizeof(cudaStream_t));           //nstr = número de streams  
for (int i=0; i<nstr; i++)  
    cudaStreamCreate(&(streams[i]));
```

// Lanzamiento asíncrono de los *kernel*s cada uno con sus propios datos

```
for (int i=0; i<nstr; i++)  
    kernel<<<blocks, threads, 0, streams[i]>>>(dA + (i*n/nstr), dB, niters);
```

// Lanzamiento asíncrono de las copias. No empiezan hasta el fin de su *kernel*

```
for (int i=0; i<nstr; i++)  
    cudaMemcpyAsync(hA + (i*n/nstr), dA + (i*n/nstr), size/nstr, cudaMemcpyDeviceToHost, streams[i]);
```

// Liberando recursos

```
for (int i=0; i<nstr; i++)  
    cudaStreamDestroy(streams[i]);
```

STREAMS

$C = \beta C + \alpha AB$; usando el *modo estándar*

```
NBlk.x = NBlk.y = (int)ceil((float)n/32);  
TpBlk.x = TpBlk.y = 32; NBlk.z = TpBlk.z = 1;
```

```
cudaEventRecord(start, 0);  
cudaMemcpy(devA, cpuA, n*n*sizeof(double), cudaMemcpyHostToDevice));  
cudaMemcpy(devB, cpuB, n*n*sizeof(double), cudaMemcpyHostToDevice));  
cudaMemcpy(devC, cpuC, n*n*sizeof(double), cudaMemcpyHostToDevice));  
  
MatMult<<<NBlk, TpBlk>>>(devC, devA, devB, alfa, beta, n, n);  
  
cudaMemcpy(cpuC, devC, n*n*sizeof(double), cudaMemcpyDeviceToHost));  
CHECKLASTERR();  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
cudaEventElapsedTime(&time, start, stop);
```

STREAMS

C=bC+aAB; usando 1 hilo CPU

```
chunk=n/nstreams; stride=chunk*n;
NBlk.x=(int)ceil((float) n/32);
NBlk.y=(int)ceil((float)chunk/32);
NBlk.z=TpBlk.z=1; TpBlk.x=TpBlk.y=32;

cudaEventRecord(start, 0);
cudaMemcpy(devB, cpuB, n*n*sizeof(double), cudaMemcpyHostToDevice);
for(i=0; i<nstreams; i++) {
    cudaMemcpyAsync(&devA[stride*i], &cpuA[stride*i], stride*sizeof(double), cudaMemcpyHostToDevice, streams[i]);
    cudaMemcpyAsync(&devC[stride*i], &cpuC[stride*i], stride*sizeof(double), cudaMemcpyHostToDevice, streams[i]);

    MatMult<<<NBlk, TpBlk, 0, streams[i]>>>(&devC[stride*i], &devA[stride*i], devB, alfa, beta, chunk, n);

    cudaMemcpyAsync(&cpuC[stride*i], &devC[stride*i], stride*sizeof(double), cudaMemcpyDeviceToHost, streams[i]);
}
CHECKLASTERR();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

STREAMS

C=bC+aAB; usando tantos hilos CPU como streams

```
chunk=n/nstreams; stride=chunk*n;
NBlk.x=(int)ceil((float)n/32);
NBlk.y = (int)ceil((float)chunk/32);
NBlk.z=TpBlk.z=1; TpBlk.x=TpBlk.y=32;

cudaEventRecord(start, 0);
cudaMemcpy(...);
#pragma omp parallel for num_threads(nstreams)
for(i=0; i<nstreams; i++) {
    cudaMemcpyAsync(...);
    cudaMemcpyAsync(...);
    MatMult<<<NBlk, TpBlk, 0, streams[i]>>>(...);
    cudaMemcpyAsync(...);
    cudaStreamSynchronize(streams[i]);
}
CHECKLASTERR();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
```

DINAMISMO

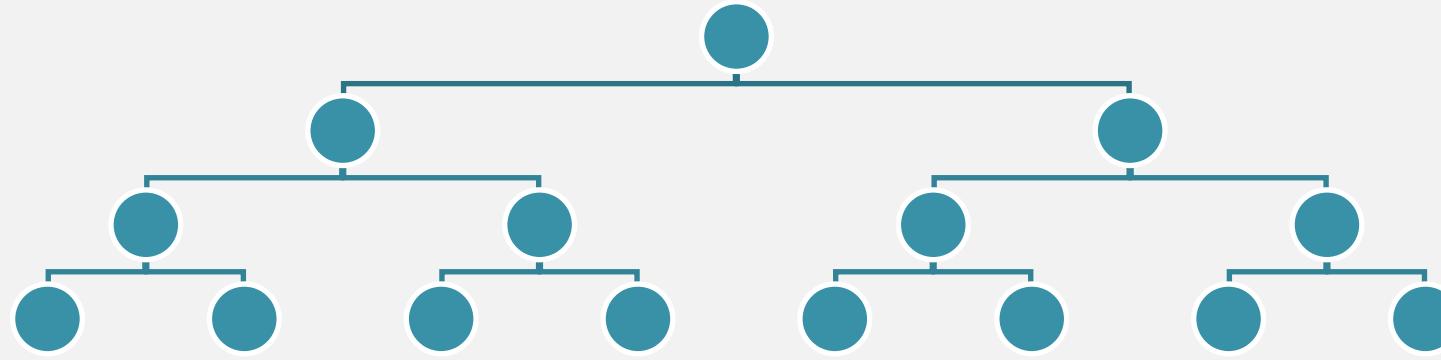
Los kernels puede lanzar, a su vez, la ejecución de otros kernels.

Tomado de una aplicación propia

```
_global_ void kernel_doGPU(...) {  
    ...  
    kernel_Cfreq<<<NMIDI, sizeWarp>>>(...);  
    if (BETA == 0.0)  
        kernel_CompDisB0<<<Grid2D, TPBI2D>>>(...);  
    else if (BETA == 1.0) {  
        kernel_Reduction<<<1, sizeWarp>>>(...);  
        kernel_CompDisB1<<<Grid2D, TPBI2D>>>(...);  
    } else {  
        kernel_PowToReal<<<GridNMID32, sizeWarp>>>(...);  
        kernel_CompDisBG<<<Grid2D, TPBI2D>>>(...);  
    }  
}
```

REDUCCIÓN PARALELA EN CUDA

- Muy común. Importante que sea eficiente.
- Enfoque basado en divide y vencerás a nivel de bloque de hilo.



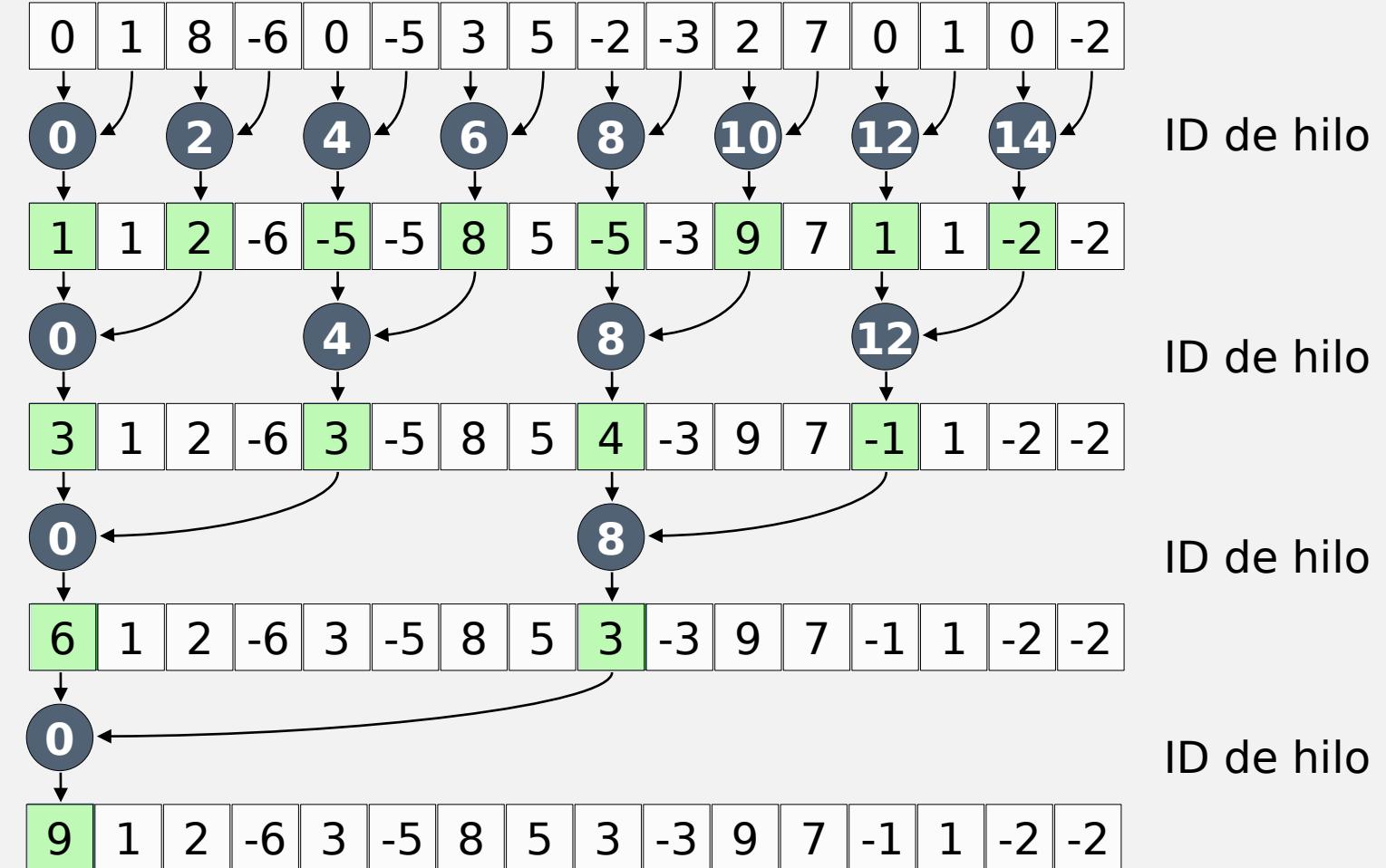
- Importante poder lanzar múltiples bloques:
 - Para procesar estructuras de datos muy grandes.
 - Para mantener ocupados todos los SMs de la GPU.
 - Cada bloque reduce una parte de la estructura.
- La GPU usada en el siguiente ejemplo tiene una tasa de transferencia de memoria teórica de 86.4 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1:

Valores en compartida

1 *Stride 1*



REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1:

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la
    memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1:

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) → Saltos divergentes, bajo rendimiento
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    ...
}
```

Rendimiento: 2.083 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 2: stride como índice – sin divergencia

```
__global__ void reduce2(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

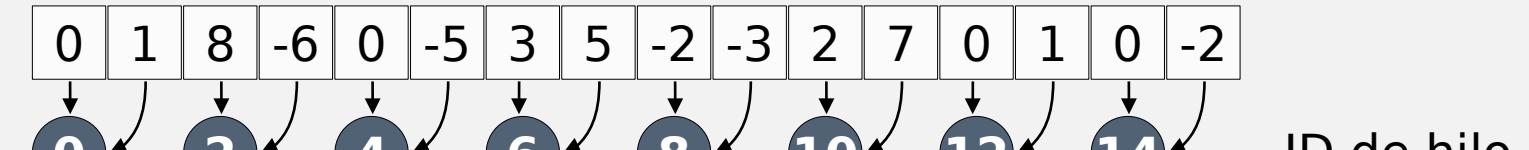
    // Haciendo la reducción en compartida
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;
        if (index < blockDim.x) sdata[index] += sdata[index + s];
        __syncthreads();
    }
    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 1: (repetido aquí para poder comparar fácilmente)

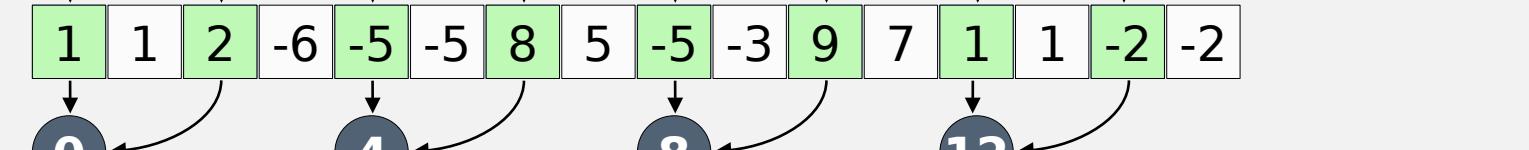
Valores en compartida

1 *Stride 1*



Valores en compartida

2 *Stride 2*



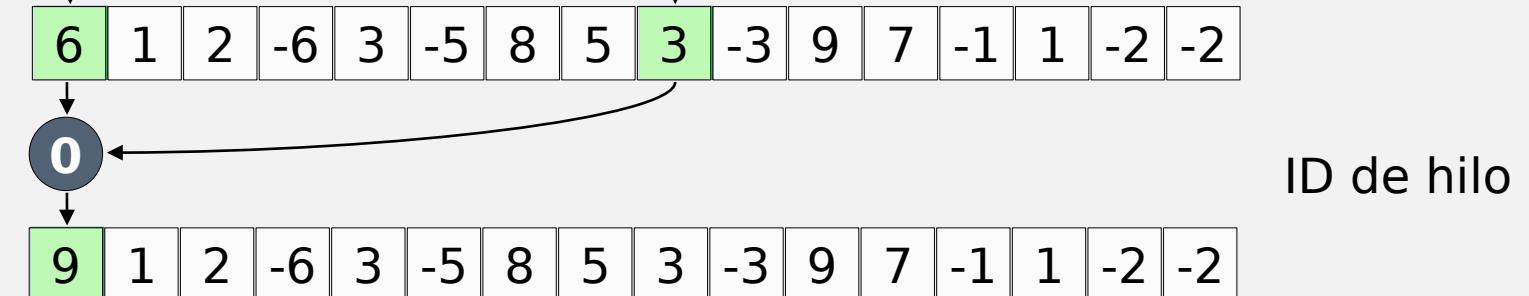
Valores en compartida

3 *Stride 4*



Valores en compartida

4 *Stride 8*



ID de hilo

ID de hilo

ID de hilo

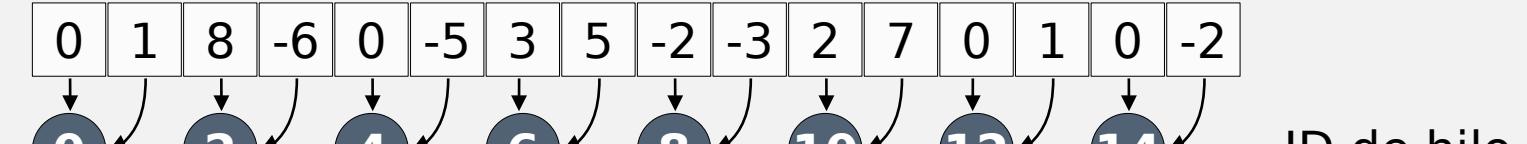
ID de hilo

REDUCCIÓN PARALELA EN CUDA

Reducción Nº 2: stride como índice – sin divergencia

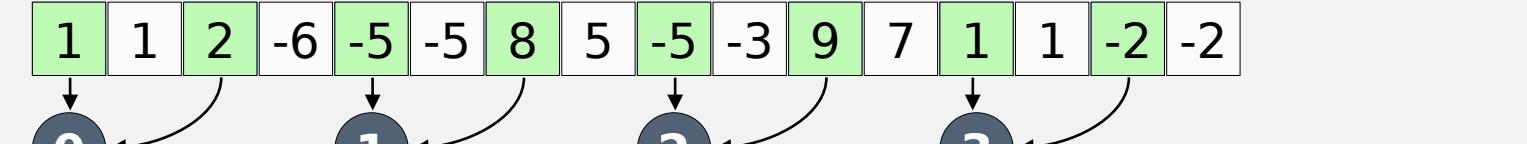
Valores en compartida

1 *Stride 1*



Valores en compartida

2 *Stride 2*



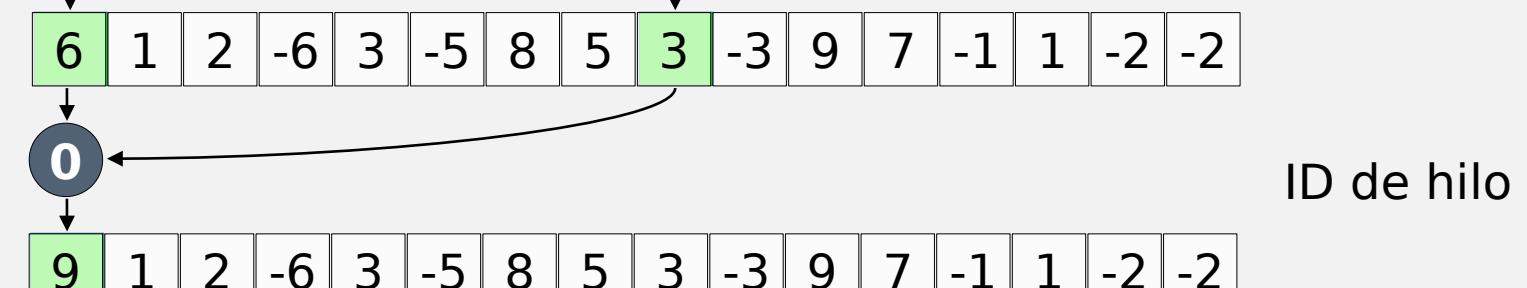
Valores en compartida

3 *Stride 4*



Valores en compartida

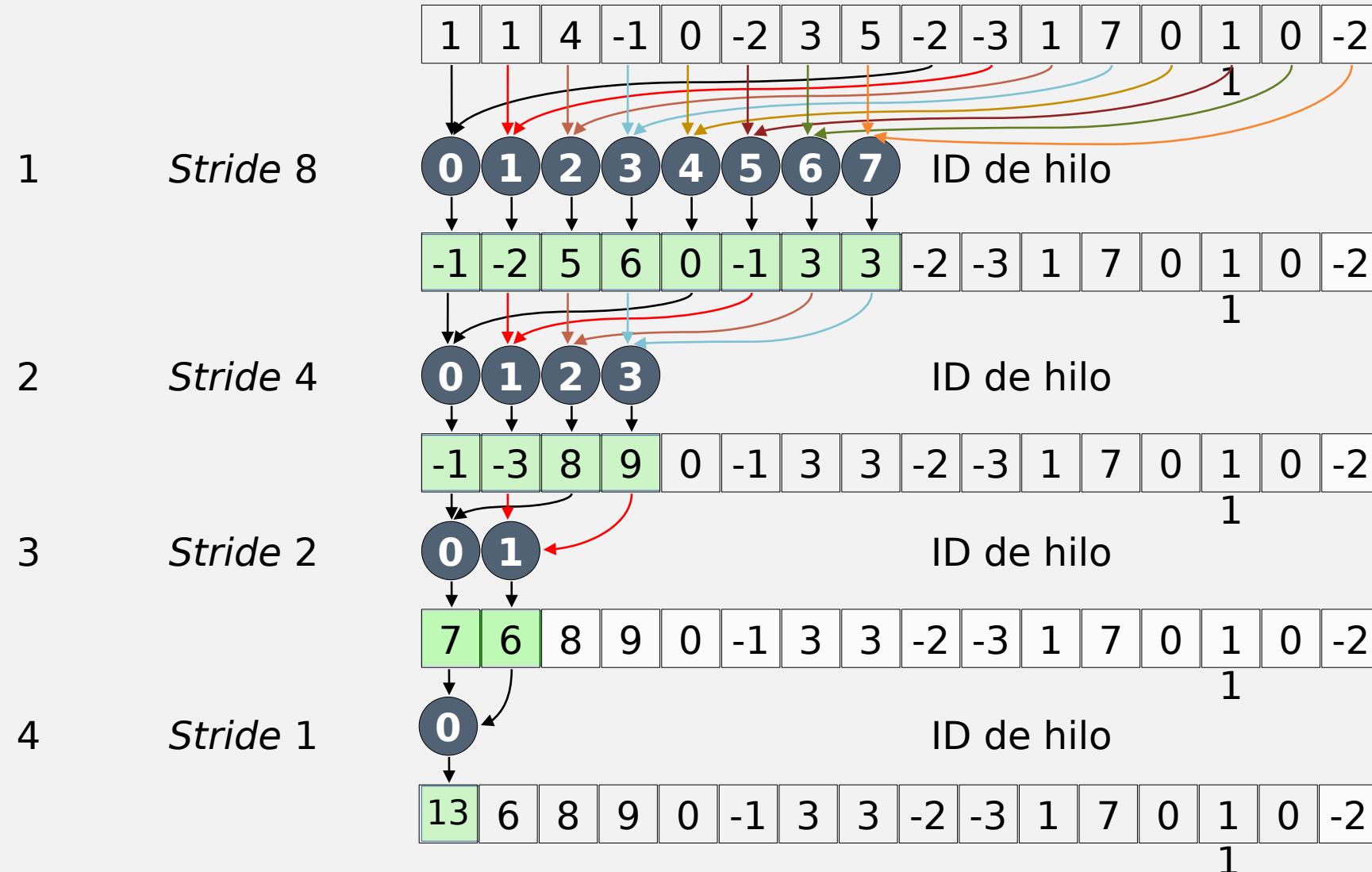
4 *Stride 8*



Rendimiento: 4.854 GB/s. El problema: Conflicto de acceso a bancos de memoria compartida

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 3: direccionamiento secuencial



REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 3: direccionamiento secuencial

```
__global__ void reduce3(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int blockDim.x/2; s>0; s>>=1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 3: direccionamiento secuencial

```
__global__ void reduce3(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Haciendo la reducción en compartida
    for(unsigned int blockDim.x/2; s>0; s>>=1) { //La mitad de los hilos idle
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    ...
}
```

Rendimiento: 9.741 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 4: suma durante la carga

```
__global__ void reduce4(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Cada hilo carga en memoria compartida un elemento de la memoria global
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();
}

...
```

Rendimiento: 17.377 GB/s

REDUCCIÓN PARALELA EN CUDA

- A medida que avanza la reducción el número de hilos activos disminuye.
 - Cuando $s \leq 32$ solo queda un *warp* activo.
- Las instrucciones son SIMD sincrónicas dentro del warp.
- Eso significa que cuando $s \leq 32$:
 - No es necesario `_syncthreads()`
 - No es necesario "*if (tid < s)*"
- Se puede, por tanto, recurrir al desenrollado de las últimas iteraciones del bucle interno.

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 5: desenrollado del último warp

```
__global__ void reduce5(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    ...

    // Haciendo la reducción en compartida
    for(unsigned int blockDim.x/2; s>>=1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }
    if (tid < 32) {
        sdata[tid] += sdata[tid + 32];  sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];  sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];  sdata[tid] += sdata[tid +  1];
    }
    ...
}
```

Rendimiento: 17.377 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Nº 6: múltiples sumas por carga

```
__global__ void reduce6(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) {
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;                                // stride para mantener coalescencia
    }
    __syncthreads();
}
```

...

Rendimiento: 17.377 GB/s

REDUCCIÓN PARALELA EN CUDA

- Reducción Final:

```
__global__ void reduce(int *g_idata, int *g_odata, unsigned int n) {  
    extern __shared__ int sdata[];  
  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x*2 + tid;  
    unsigned int gridSize = blockSize*2*gridDim.x;  
    sdata[tid] = 0;  
  
    while (i < n) {  
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];  
        i += gridSize;  
    }  
    __syncthreads();  
  
    ...
```

REDUCCIÓN PARALELA EN CUDA

- Reducción Final:

```
...
    if (blockSize >= 512) {
        if (tid < 256) sdata[tid] += sdata[tid + 256];
        __syncthreads();
    }

    if (blockSize >= 256) {
        if (tid < 128) sdata[tid] += sdata[tid + 128];
        __syncthreads();
    }

    if (blockSize >= 128) {
        if (tid < 64) sdata[tid] += sdata[tid + 64];
        __syncthreads();
    }

...
}
```

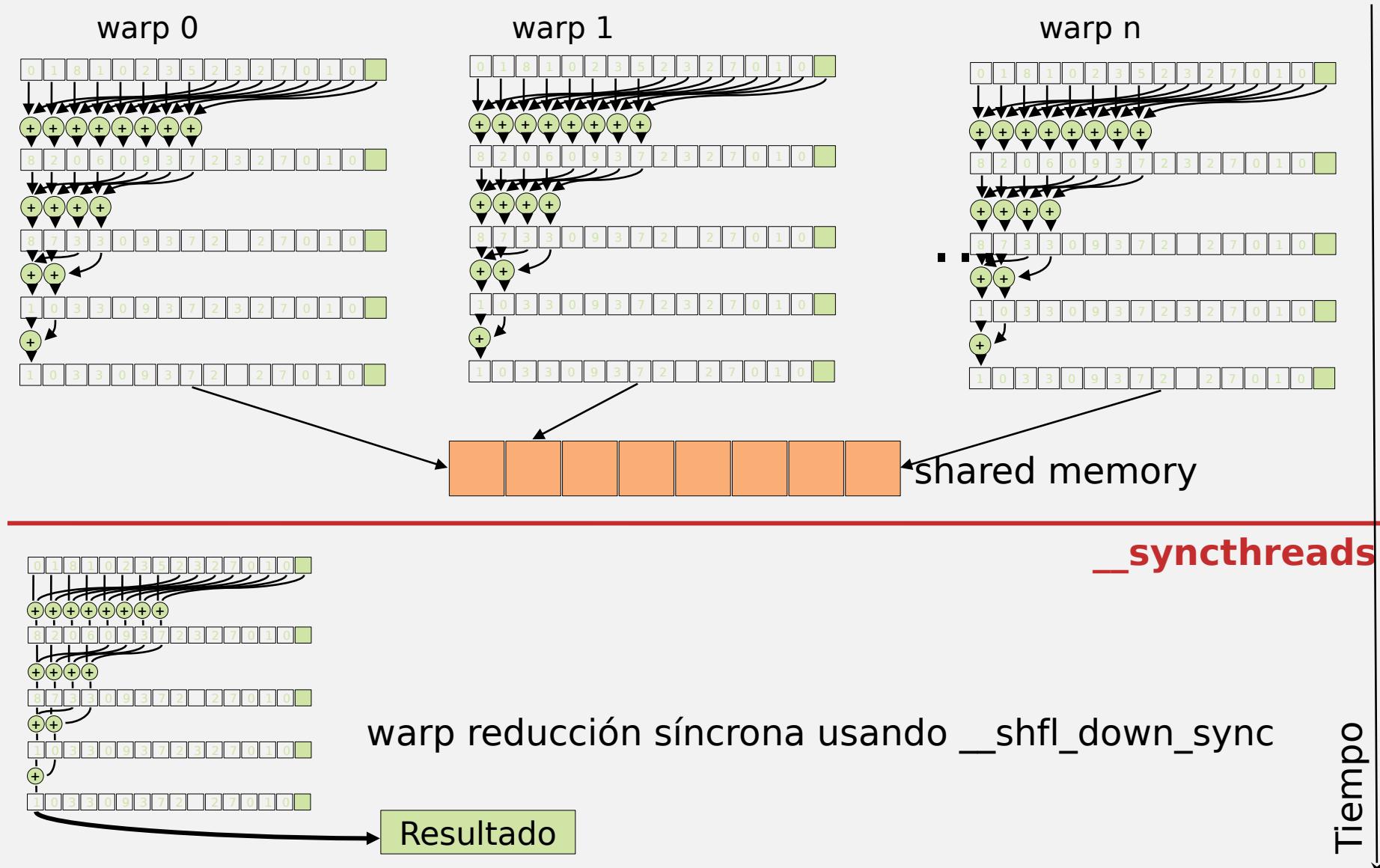
REDUCCIÓN PARALELA EN CUDA

- Reducción Final:

```
...
    if (tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
    }

    // Escribiendo resultado a la memoria global
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

REDUCCIÓN PARALELA EN CUDA

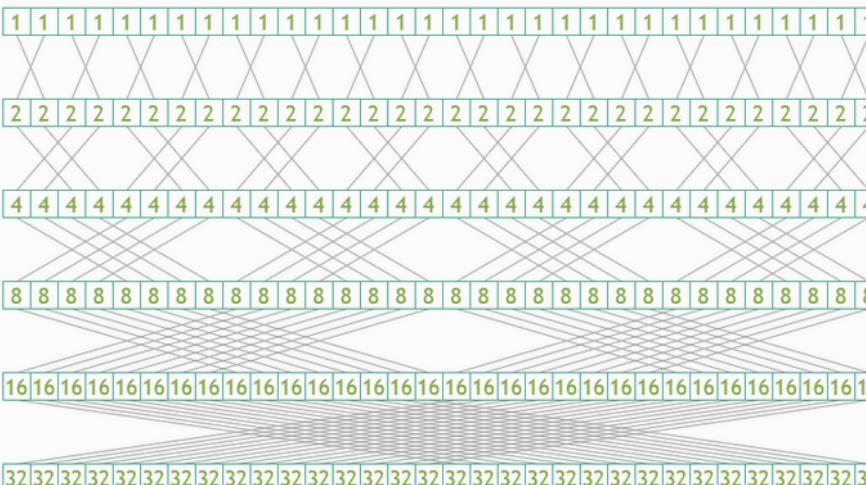


JUNTANDO CONCEPTOS: BUSCAR EL ÚLTIMO MÍNIMO

```
int Onelmin(double *odata, int *opos, double *idata, int maxGrid, int N) {  
    int nBlocks=0, nThreads=0, sShared=0, S;  
  
    BlocksAndThreads(&nBlocks, &nThreads, &sShared, maxGrid, N);  
  
    kernel_Onelmin<<<nBlocks, nThreads, 2*sShared>>>(odata, opos, idata, nThreads,  
    IsPow2(N), N);  
  
    S = nBlocks;  
    while (S > 1) {  
        BlocksAndThreads(&nBlocks, &nThreads, &sShared, maxGrid, S);  
        kernel_OnelminLast<<<nBlocks, nThreads, 2*sShared>>>(odata, opos, odata, opos,  
        nThreads, IsPow2(S), S);  
        S = (S + (nThreads*2-1)) / (nThreads*2);  
    }  
  
    cudaDeviceSynchronize();  
    return opos[0];  
}
```

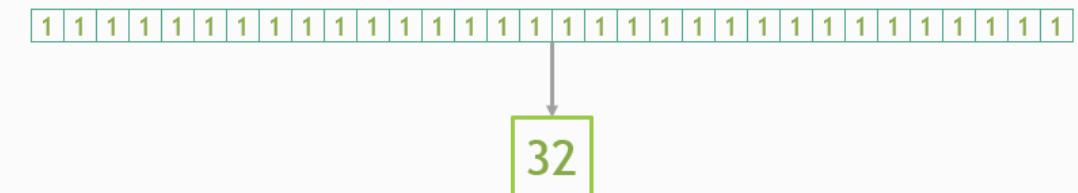
WARP-WIDE REDUCTION

- Usando la nueva instrucción de reducción de la A100 se puede realizar operaciones de reducción en un solo paso. Antes requería 5 pasos con operaciones SHFL como muestra la figura.



```
__device__ int reduce(int value) {
    value += __shfl_xor_sync(0xFFFFFFFF, value, 1);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 2);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 4);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 8);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 16);

    return value;
}
```



```
int total = __reduce_add_sync(0xFFFFFFFF, value);
```

CUDA TASK GRAPH

- La ejecución del trabajo en la GPU se divide en tres etapas: lanzamiento, inicialización del grid y ejecución del kernel. Para kernels con tiempos de ejecución reducidos los costes *generales* pueden ser una fracción significativa del tiempo total de ejecución.
- Muchas aplicaciones intensivas en GPU tienen una estructura iterativa en la que el mismo *trabajo* se ejecuta repetidamente. El uso de CUDA Streams requiere que la CPU vuelva a enviar el trabajo a la GPU con cada iteración lo que consume tiempo y recursos de la CPU.
- Los grafos de tareas CUDA proporcionan un modelo más eficiente para enviar trabajo a la GPU.
- Un grafo de tareas consta de una serie de operaciones, como copias de memoria y lanzamientos de kernel, conectadas por dependencias y se define por separado de su ejecución.
- Los grafos de tareas permiten definir una vez y ejecutar repetidamente flujos de trabajo.
- Un grafo de tareas permite el lanzamiento de cualquier número de kernels en una sola operación, mejorando enormemente la eficiencia y el rendimiento de la aplicación.
- Separar la definición del grafo de tareas de su ejecución reduce significativamente los costes en CPU del lanzamiento del kernel. Además, permiten que CUDA realice optimizaciones porque todo el flujo de trabajo es visible para el driver.

CONCLUSIONES

- “Cuando aprendemos a manejar un martillo, sólo vemos clavos”. No usar GPUs para menos de 10^7 datos.
- Primero buscar en internet. Segundo artesanía para adaptar código. Lema de Google Scholar:....
- Ojo a las dependencias: calcula el propietario. Si no, `_syncthreads` dentro de un bloque. Entre bloques no hay control. También tenemos `atomic`, que atomiza la celda, no toda la variable.
- Matlab: democratización de la programación matemática. CUDA: Quien quiera peces...
- Grano fino a la GPU. Grano grueso/tareas a la CPU.
- No es grave si los hilos tienen más carga: lo que importa es tener los cores al 100%.
- 70% programas son “compute bound”, mientras 30% son “memory bound” → ¡¡Usar memoria!!
- No hacerse el/la listx: la fuerza bruta es caballo ganador.
- Empezar lanzando los kernels con `<<<N/256,256>>>`, luego ir variando arriba/abajo (no menos de 64) en potencias de 2.