

# PROCESADORES GRÁFICOS

## 09 OPTIMIZACIÓN Y MÁQUINA VIRTUAL PTX

**Máster Oficial  
en Informática  
Gráfica, Juegos y  
Realidad Virtual**

Escuela Técnica Superior  
de Ingeniería Informática



Universidad  
Rey Juan Carlos

# Índice

- Árbol de compilación
- Ensamblador PTX
- Control de flujo en CUDA
- Diferencias en el origen del problema de la ramificación CPU-GPU
- Ejemplos sobre las prácticas
- Predicación

*"Optimization hinders evolution"*  
--Anónimo

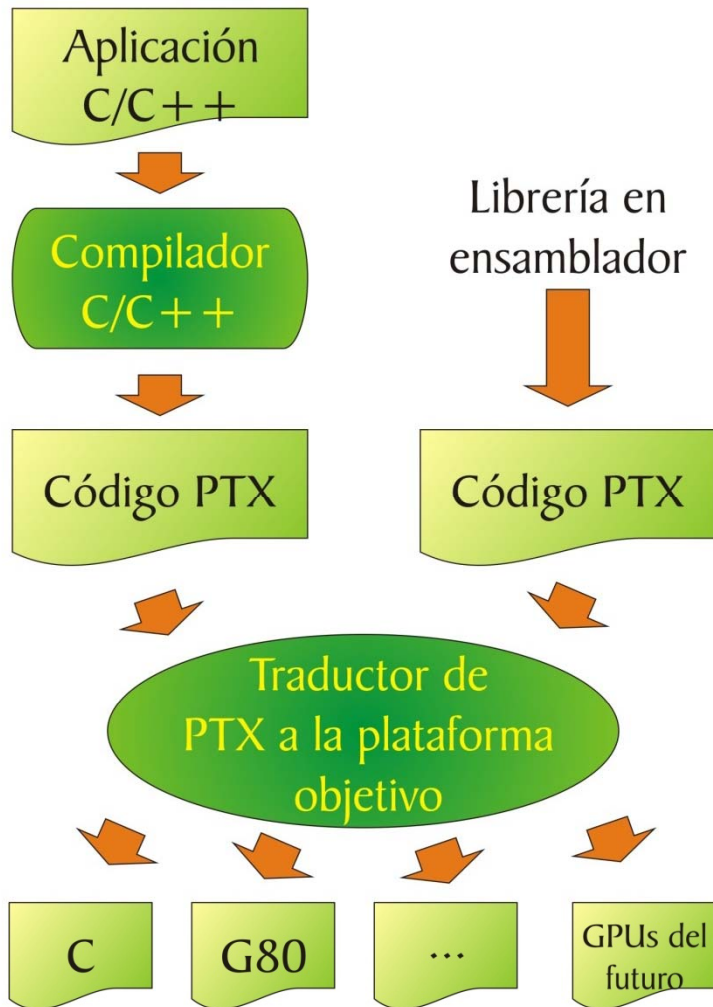
*"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."*  
---Donald Knuth

Aunque en esta clase me voy a apoyar en las transparencias de D. Kirk y WM Hwu la explicación será muy diferente

© Todas las marcas y productos mencionados en estas transparencias están registradas por sus respectivas compañías, y su uso es de carácter descriptivo con fines docentes.

Parte de las tablas y gráficos están basados en las presentaciones de GPGPU y streaming computing de NVidia y ATI-AMD, en los libros mencionados en la bibliografía, en el curso de David Kirk de la Universidad de Illinois, y el de Simon Green en ARCS08

# Máquina Virtual PTX e ISA



- Parallel Thread eXecution (PTX)
  - Máquina Virtual e ISA
  - Modelo de programación
  - Ejecución, recursos y estado
- ISA – Instruction Set Architecture
  - Declaración de variables
  - Instrucciones y operandos
- El traductor es un compilador optimizador
  - Traduce de PTX a código objetivo
- El driver implementa el runtime de la máquina virtual
  - Acoplado con el traductor

# Compilar CUDA a PTX

CUDA

```
float4 me = gx[gtid];  
me.x += me.y * me.z;
```

PTX

```
ld.global.v4.f32 {$f1, $f3, $f5, $f7}, [$r9+0];  
# 174      me.x += me.y * me.z;  
mad.f32      $f1, $f5, $f3, $f1;
```

# Ejemplo de función de CUDA

- CUDA

```
__device__ void interaction(  
    float4 b0, float4 b1, float3 *accel)  
{  
    r.x = b1.x - b0.x;  
    r.y = b1.y - b0.y;  
    r.z = b1.z - b0.z;  
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;  
    float s = 1.0f/sqrt(distSqr);  
    accel->x += r.x * s;  
    accel->y += r.y * s;  
    accel->z += r.z * s;  
}
```

- PTX

```
sub.f32    $f18, $f1, $f15;  
sub.f32    $f19, $f3, $f16;  
sub.f32    $f20, $f5, $f17;  
mul.f32    $f21, $f18, $f18;  
mul.f32    $f22, $f19, $f19;  
mul.f32    $f23, $f20, $f20;  
add.f32    $f24, $f21, $f22;  
add.f32    $f25, $f23, $f24;  
rsqrt.f32  $f26, $f25;  
mad.f32    $f13, $f18, $f26, $f13;  
mov.f32    $f14, $f13;  
mad.f32    $f11, $f19, $f26, $f11;  
mov.f32    $f12, $f11;  
mad.f32    $f9, $f20, $f26, $f9;  
mov.f32    $f10, $f9;
```

# Compilar un bucle que llama a una función

- CUDA

- sx es compartido
- mx, accel son locales

```
for (i = 0; i < K; i++) {  
    if (i != threadIdx.x) {  
        interaction(  
            sx[i], mx, &accel  
        );  
    }  
}
```

- PTX

```
mov.s32    $r12, 0;  
$Lt_0_26:  
setp.eq.u32    $p1, $r12, $r5;  
@$p1 bra    $Lt_0_27;  
mul.lo.u32    $r13, $r12, 16;  
add.u32    $r14, $r13, $r1;  
ld.shared.f32    $f15, [$r14+0];  
ld.shared.f32    $f16, [$r14+4];  
ld.shared.f32    $f17, [$r14+8];
```

***[El cuerpo de la función de la transparencia anterior iría aquí]***

```
$Lt_0_27:  
add.s32    $r12, $r12, 1;  
mov.s32    $r15, 128;  
setp.ne.s32    $p2, $r12, $r15;  
@$p2 bra    $Lt_0_26;
```

# Control de Flujo en CUDA

# Objetivo

- Entender las implicaciones del control de flujo en
  - La sobrecarga en la divergencia producida en la ramificación
  - La utilización de recursos en la ejecución del SM
- Aprender mejores formas de escribir el código cuando son necesarias estas estructuras de control
- Entender la **predicción** del compilador/HW que se han diseñado para reducir el impacto del control del flujo de ejecución
  - Hay un coste implícito

# Repaso de terminología

(desde el punto de vista del modelo de programación)

- *Hebra*: código concurrente que se ejecuta en el dispositivo CUDA y su estado asociado (en paralelo con otras hebras)
  - La unidad de paralelismo en CUDA
- *Warp*: un conjunto de hebras que se ejecuta en paralelo físicamente (en el mismo multiprocesador) en el G80
- *Bloque*: un grupo de hebras que se ejecutan juntas y forman una unidad de asignación de recursos
- *Grid*: un grupo de bloques de hebras que deben completar su tarea antes de que pueda continuar la siguiente fase del programa

# Cómo los bloques de hebras son repartidos

- Los bloques de hebras son divididos en warps
  - Los identificadores de hebras en un warp son consecutivos y en orden creciente
  - El warp 0 comienza con la hebra 0
- El reparto es siempre el mismo
  - De modo que se puede aprovechar este conocimiento para el diseño de nuestras estructuras de control
  - Aunque el tamaño exacto de los warps puede ir cambiando con el tiempo
- **No podemos presuponer ningún orden entre los warps**
  - Si hay dependencias entre hebras, es necesario utilizar `__syncthreads()` para obtener los resultados esperados.

# CPU vs. GPU

- La ramificación no reduce el rendimiento debido a la penalización derivada de un cauce segmentado en la ejecución de las instrucciones (como pasa en la CPU)
- Se debe a la implementación de los actuales multiprocesadores de CUDA → el concepto de Warp y la separación y compartición de ciertas unidades funcionales como la decodificación de instrucciones
- Las técnicas para resolverlo y aumentar las prestaciones tienen puntos en común

# Instrucciones de control de flujo

- Nuestra principal preocupación en las ramificaciones es la divergencia
  - Las hebras del mismo warp pueden tomar caminos diferentes
  - Y estos caminos se serializan en los multiprocesadores del G80
    - Los caminos tomados por las hebras en un warp son recorridos uno a uno hasta que no quedan más
- Estrategia típica: evitar estas divergencias si podemos hacer la ramificación función del identificador de las hebras
  - Ejemplo con divergencia
    - `If (threadIdx.x > 2) { }`
    - Esto crea dos caminos de control diferentes para las hebras de un bloque
    - La granularidad de la rama es inferior al tamaño de 1 warp; las hebra 0 y 1 toman un camino distinto al del resto de las hebras en el mismo warp
  - Ejemplo sin divergencia:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - También crea dos caminos de control diferentes en el mismo bloque
    - Pero esta vez la granularidad de los caminos de control es múltiplo del tamaño de warp; todas las hebras de un mismo warp siguen un único camino

# Reducción paralela

- Dado un array de valores, “reducirlo” a un único valor en paralelo
- Ejemplos
  - Reducción suma: suma todos los valores de un array
  - Cálculo del máximo: reduce a un único valor que ha de ser el máximo de los valores de los elementos
- Implementación paralela típica
  - Dividir recursivamente el número de hebras a la mitad y sumar dos valores por hebra
  - Requiere  $\log(n)$  pasos para  $n$  elementos y  $n/2$  hebras

Práctica 7

# Ejemplo de reducción de un vector

- El vector original está en memoria de vídeo
- La memoria compartida es utilizada para albergar la suma parcial del vector
- Cada iteración obtiene una suma parcial más cercana a la suma final
- La solución se coloca en el elemento cero

# Una posible implementación simple

- Suponemos que ya hemos cargado el array en

```
- __shared__ float partialSum[]
```

```
unsigned int t = threadIdx.x;
```

```
for (unsigned int stride = 1;
```

```
    stride < blockDim.x; stride *= 2)
```

```
{
```

```
    __syncthreads();
```

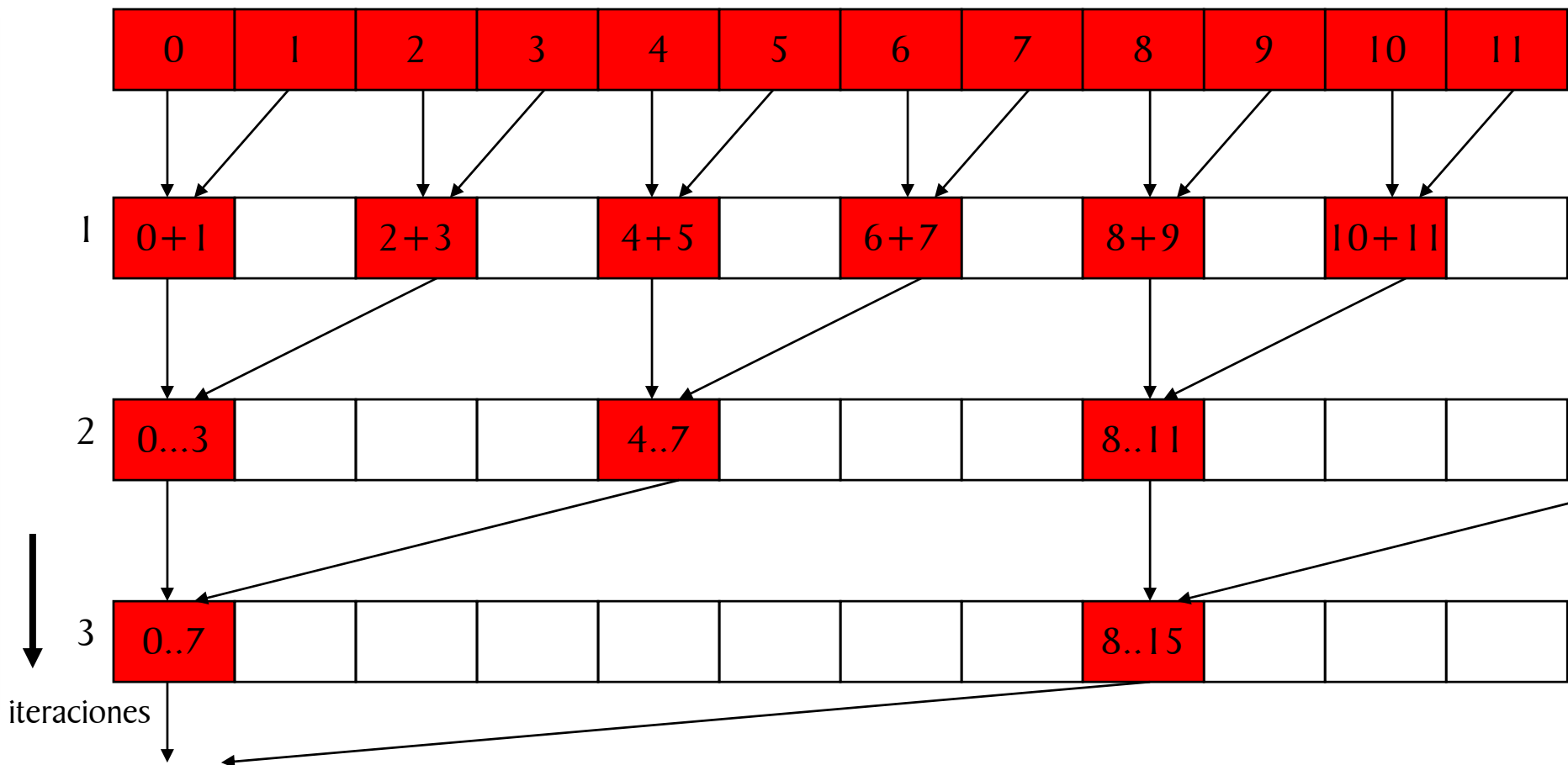
```
    if (t % (2*stride) == 0)
```

```
        partialSum[t] += partialSum[t+stride];
```

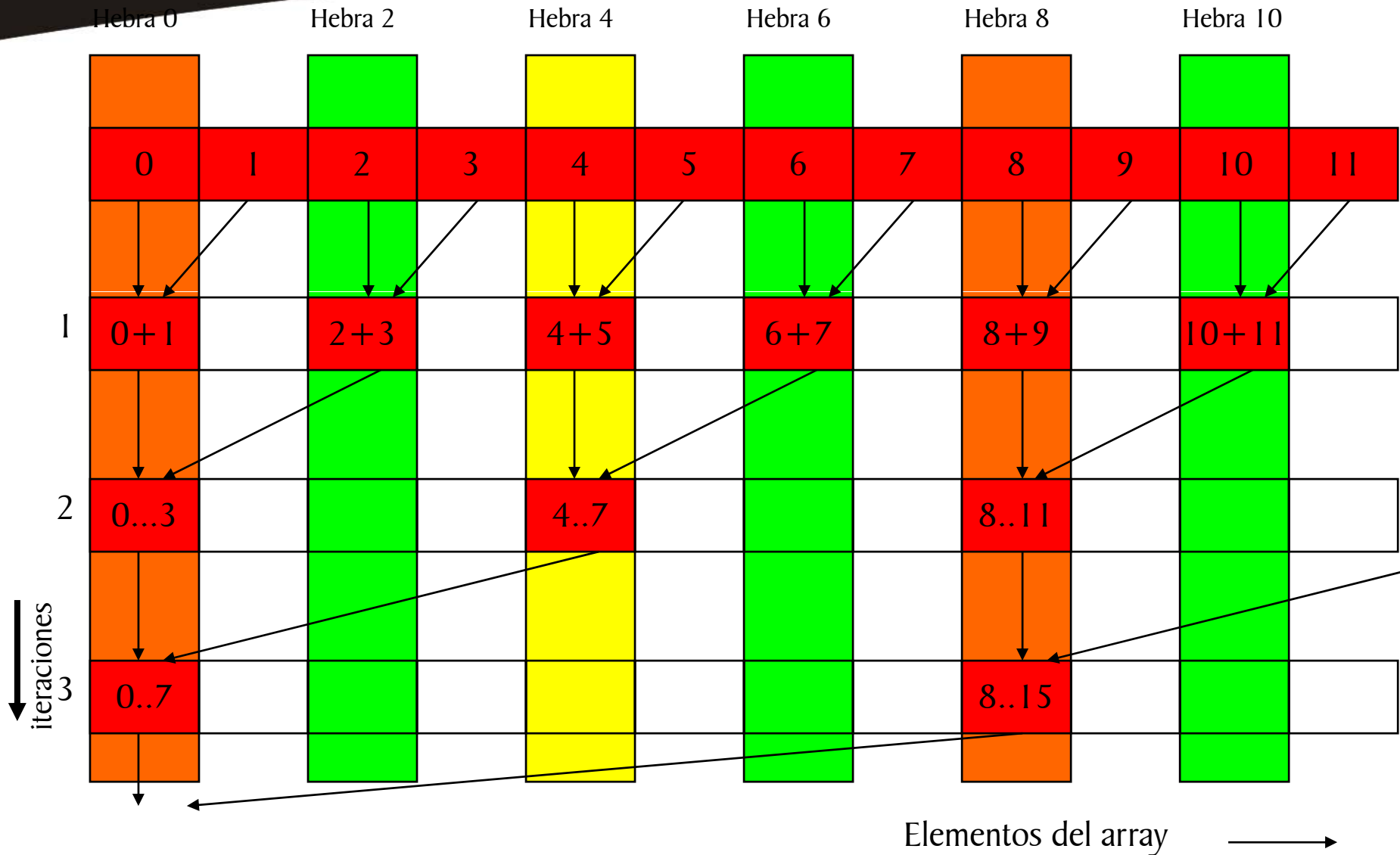
```
}
```

# Reducción de vectores con conflictos de bancos

Elementos del array →



# Reducción de vactores con divergencia entre ramas



# Puntualizaciones

- En cada iteración los dos caminos de control son atravesados secuencialmente por cada warp
  - Tenemos hebras que llevan a cabo sumas y otras que no
  - Las hebras que no realizan ninguna suma nos pueden costar ciclos extras en función de la implementación de la divergencia
- Nunca más de la mitad de las hebras serán ejecutadas a la vez
  - La mitad impar de las hebras están deshabilitadas desde el comienzo
  - En media, menos de una cuarta parte de las hebras de cada warp estarán activas a lo largo del tiempo
  - Tras la 5ª iteración, warps enteros estarán deshabilitados, no hay divergencia pero malgasta los recursos
    - Puede continuar y continuar hasta 4 iteraciones más ( $512/32 = 16 = 2^4$ ), donde cada iteración tiene una sola hebra por bloque activa hasta que todos los warps se retiran

# Atajos de la implementación

Divergencia debida a decisiones de ramificacion entremezcladas: NO MOLA

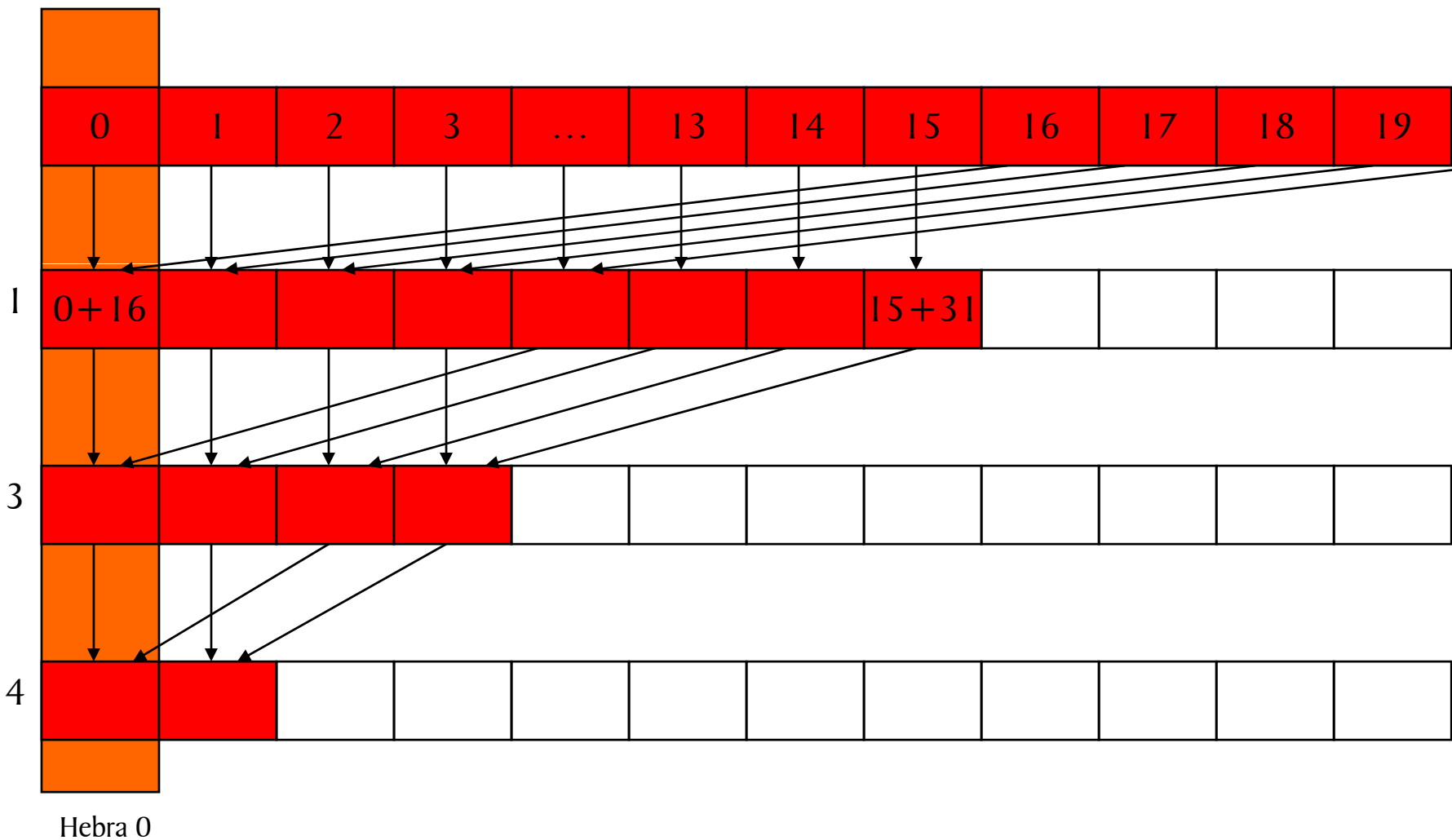
```
- __shared__ float partialSum[ ]  
  
unsigned int t = threadIdx.x;  
for (unsigned int stride = 1;  
     stride < blockDim.x; stride *= 2)  
{  
    __syncthreads();  
    if (t % (2*stride) == 0)  
        partialSum[t] += partialSum[t+stride];  
}
```

Conflictos entre bancos de mem. : NO MOLA

# Una mejor implementación

```
- __shared__ float partialSum[ ]  
  
unsigned int t = threadIdx.x;  
for (unsigned int stride = blockDim.x;  
     stride > 1;  stride >> 1)  
{  
    __syncthreads();  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];  
}
```

# Sin divergencia hasta < 16 sub-sumas



# Puntualizaciones sobre la nueva implementación

- Sólo las últimas 5 iteraciones tendrán divergencia
- Warps completos se “desactivarán” a medida que progresen las iteraciones
  - Para un bloque con 512 hebras, 4 iteraciones dejarán sólo una hebra en cada bloque
  - Para un uso más óptimo de los recursos nos conviene acabar con esos warps y bloques tan pronto como sea posible
- Evitamos los conflictos entre bancos

# Podríamos ir más allá con la optimización

Pero un refinamiento excesivo puede ser contraproducente

- Para los 6 últimos bucles tenemos sólo un warp activo (ej. tid's 0..31)

- Las lecturas y escrituras en memoria compartida se hacen de forma síncrona (SIMD) dentro de un mismo
- De manera que se puede saltar `__syncthreads()` y desenrollar las 5 últimas iteraciones

```
unsigned int tid = threadIdx.x;
for (unsigned int d = n>>1; d > 32; d >>= 1) {
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[tid + d];
}
__syncthreads();
if (tid <= 32) { // desenrollamos
    shared[tid] += shared[tid + 32];
    shared[tid] += shared[tid + 16];
    shared[tid] += shared[tid + 8];
    shared[tid] += shared[tid + 4];
    shared[tid] += shared[tid + 2];
    shared[tid] += shared[tid + 1];
}
```

Esto no funcionaría de la manera esperada si el tamaño de warp disminuye; se haría necesario un `__syncthreads()` entre cada instrucción! Y además tener `__syncthreads()` dentro de una estructura if puede dar muchos problemas !!

# Concepto de la ejecución de predicado

`<p1> LDR r1,r2,0`

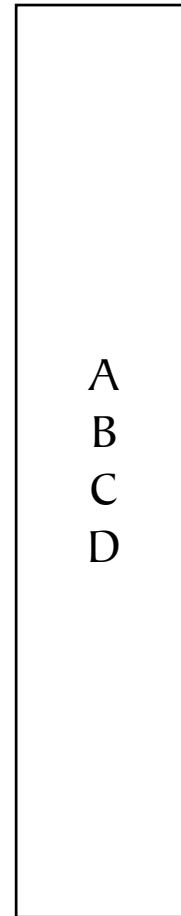
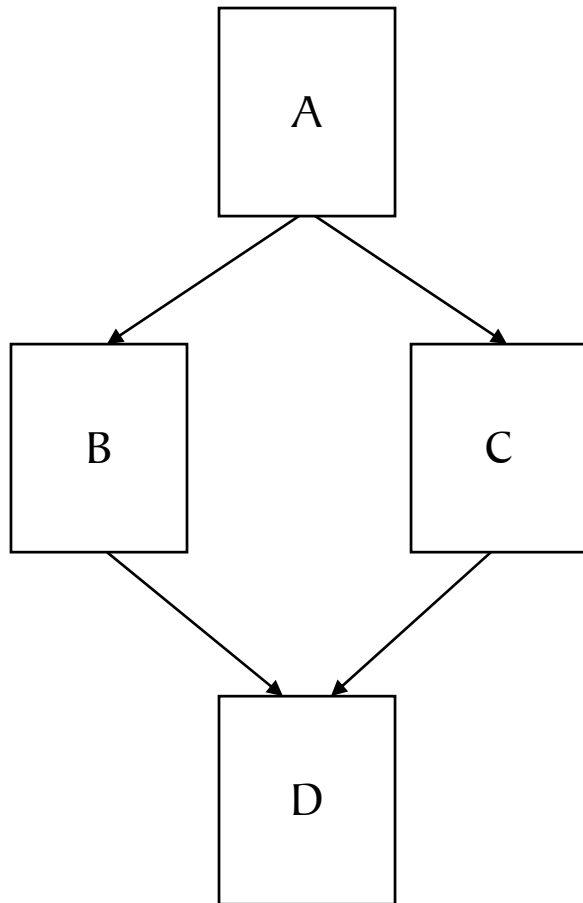
- Si `p1` es `TRUE`, la instrucción se ejecuta normalmente
- Si `p1` es `FALSE`, la instrucción se trata como un `NOP`

# Ejemplo de Predicación

```
:  
:  
if (x == 10)  
    c = c + 1;  
:  
:
```

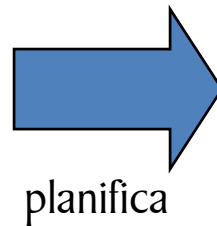
```
:  
:  
LDR r5, X  
p1 <- r5 eq 10  
<p1> LDR r1 <- C  
<p1> ADD r1, r1, 1  
<p1> STR r1 -> C  
:  
:
```

# En las estructuras IF-ELSE nos viene bien la predicación



# Ejemplo IF-ELSE

```
      :  
      :  
      p1,p2 <- r5 eq 10  
<p1> inst 1 from B  
<p1> inst 2 from B  
<p1> :  
      :  
<p2> inst 1 from C  
<p2> inst 2 from C  
      :  
      :
```



```
      :  
      :  
      p1,p2 <- r5 eq 10  
<p1> inst 1 from B  
<p2> inst 1 from C  
  
<p1> inst 2 from B  
<p2> inst 2 from C  
  
<p1> :  
      :
```

Tenemos un coste extra debido a un conjunto de instrucciones que tendrán que ser realizadas cada vez que se ejecute el código, pero evitamos la posible divergencia.

# Predicación de instrucciones en el G80



- Las instrucciones de comparación establecen códigos de comparación (CC)
- Se pueden “predicar” instrucciones y escribir sólo los resultados cuando CC cumple el criterio indicado (CC  $\neq$  0, CC  $\geq$  0, etc.)
- El compilador trata de predecir si la condición de una rama es propensa a producir muchos warps divergentes
  - Si tiene la certeza de que no va a divergir sólo genera “predicados” si el if tiene  $< 4$  instructions
  - Si no lo puede garantizar, sólo hace el predicado si el if tiene  $< 7$  instructions
- Puede reemplazar ramas con instrucciones de ramificación
- Todas las instrucciones “predicadas” requieren ciclos de ejecución
  - Aquellas con condiciones que no se cumplen (FALSE) no escriben su resultado
    - Ni invocan cargas o almacenamientos
  - Ahorra instrucciones de ramificación, por lo que puede ser más “barato” que serializar caminos divergentes

# Laboratorio con las soluciones

Si queréis después de los exámenes en Junio, podemos hacer un tutoría grupal en la que compartamos las distintas estrategias para optimizar el código de las prácticas de CUDA Que no he podido contar detalladamente en clase aplicadas a los casos de estudio porque se supone que era cosa vuestra ;-)

Y de paso comentamos las soluciones del examen.