

# Problemas GPU/CUDA

# Problema 1

```
__global__ void print( int i ) {  
    printf("Hola con i=%d\n",i);  
}  
int main(){  
    int i;  
    for(i=0;i<10;i++)  
        print<<< , >>>(i);  
    cudaDeviceSynchronize();  
    return 0;  
}
```

Cómo llamarlo para  
que pinte las  
siguientes 20 líneas:

Hola con i=0  
Hola con i=0  
Hola con i=1  
Hola con i=1  
Hola con i=2  
Hola con i=2  
Hola con i=3  
Hola con i=3  
...  
Hola con i=9  
Hola con i=9

# Problema 1

```
__global__ void print( int i ) {  
    printf("Hola con i=%d\n",i);  
}  
int main(){  
    int i;  
    for(i=0;i<10;i++)  
        print<<< 1 , 2 >>>(i);  
    cudaDeviceSynchronize();  
    return 0;  
}
```

Cómo llamarlo para  
que pinte las  
siguientes 20 líneas:

Hola con i=0  
Hola con i=0  
Hola con i=1  
Hola con i=1  
Hola con i=2  
Hola con i=2  
Hola con i=3  
Hola con i=3  
...  
Hola con i=9  
Hola con i=9

# Problema 1

```
__global__ void print( int i ) {  
    printf("Hola con i=%d\n",i);  
}  
int main(){  
    int i;  
    for(i=0;i<10;i++)  
        print<<< 2 , 1 >>>(i);  
    cudaDeviceSynchronize();  
    return 0;  
}
```

Cómo llamarlo para  
que pinte las  
siguientes 20 líneas:

Hola con i=0  
Hola con i=0  
Hola con i=1  
Hola con i=1  
Hola con i=2  
Hola con i=2  
Hola con i=3  
Hola con i=3  
...  
Hola con i=9  
Hola con i=9

# Problema 2

```
#include <stdio.h>
__global__ void print() {
    printf("Hilo %d del bloque %d\n",threadIdx.x,
blockIdx.x);
}
int main(){
    print<<< 1 , 64 >>>();
    cudaDeviceSynchronize();
    return 0;
}
```

¿En qué orden cabe esperarlos?

# Problema 2

```
#include <stdio.h>
__global__ void print() {
    printf("Hilo %d del bloque %d\n",threadIdx.x,
blockIdx.x);
}
int main(){
    print<<< 1 , 64 >>>();
    cudaDeviceSynchronize();
    return 0;
}
```

¿En qué orden cabe esperarlos?  
Seguro, de 32 en 32 en orden.  
Quizás, los 64.

# Problema 2\_2

```
#include <stdio.h>
__global__ void print() {
    printf("Hilo %d del bloque %d\n",threadIdx.x,
blockIdx.x);
}
int main(){
    print<<< 2 , 32 >>>();
    cudaDeviceSynchronize();
    return 0;
}
```

¿Y ahora?

# Problema 2\_2

```
#include <stdio.h>
__global__ void print() {
    printf("Hilo %d del bloque %d\n",threadIdx.x,
blockIdx.x);
}
int main(){
    print<<< 2 , 32 >>>();
    cudaDeviceSynchronize();
    return 0;
}
```

¿Y ahora?

Serialización en acceso a stdout dentro de cada warp. Concurrencia entre bloques.



# Problema 2\_2

```
#include <stdio.h>
__global__ void print() {
    printf("Hilo %d del bloque %d\n",threadIdx.x,
blockIdx.x);
}
int main(){
    print<<< 2 , 32 >>>();
    cudaDeviceSynchronize();
    return 0;
}
```

¿Y ahora?

Serialización en acceso a stdout dentro de cada warp. Concurrencia entre bloques. Pueden salir en desorden los bloques:

...

Hilo 29 del bloque 0

Hilo 30 del bloque 0

Hilo 31 del bloque 0

Hilo 0 del bloque 1

Hilo 1 del bloque 1

Hilo 2 del bloque 1

...

# Problema 3

```
void copy( int *a, int *b, int N) {
```

Copiar b en a

```
}
```

# Problema 3

```
void copy( int *a, int *b, int N) {  
    int tid;  
    for(tid=0;tid<N;tid++)  
        a[tid] = b[tid];  
}
```

Copiar b en a...  
¿En CUDA?

# Problema 3

```
__global__ void copy( int *a, int *b, int N ) {  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    if (tid < N)  
        a[tid] = b[tid];  
}
```

**¡OJO! a y b tienen que ser  
accesibles desde la GPU**

# Problema 4

```
void add( int *a, int *b, int *c , int N) {  
    int tid;  
    for(tid=0;tid<N;tid++)  
        c[tid] = a[tid] + b[tid];  
}
```

Partimos de C

¿En CUDA?

# Problema 4

```
__global__ void add( int *a, int *b, int *c, int N ) {  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

# Problema 4

```
__global__ void add( int *a, int *b, int *c, int N ) {  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

Pero... ¿y si N es muy muy grande?

Recordad: blockIdx.x máximo  $2^{16}$

# Problema 4

```
__global__ void add( int *a, int *b, int *c, int N ) {  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

Pero... ¿y si N es muy muy grande?

Recordad: blockIdx.x máximo  $2^{16}$ , más  
1024 hilos/bloque,  $2^{26}$  posiciones  $\sim 67\text{M}...$



# Problema 4\_2

```
__global__ void add( int *a, int *b, int *c, long N ) {  
    int tid = blockIdx.x*blockDim.x+threadIdx.x;  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += blockDim.x * blockDim.x;  
    }  
}
```

Para bucles más grandes

# Problema 4\_3

```
int i,j,r;
double s=0;
for (i=0;i<m;i++){
    for(j=0;j<n;j++){
        s=0;
        for(r=0;r<k;r++){
            s+=MatA[i*lda+r]*MatBt[j*ldb+r];
        }
        MatC[i*ldc+j]=beta*MatC[i*ldc+j]+alpha*s;
    }
}
```

Nuestro querido **dgemm**

¡¡A CUDA!!

# Problema 4\_3\_1

```
int i,j,r;
double s=0;
j=threadIdx.x+blockIdx.x*blockDim.x;
if(j<n)
    for(i=0;i<m;i++){
        s=0;
        for(r=0;r<k;r++){
            s+=MatA[i*lda+r]*MatBt[j*ldb+r];
        }
        MatC[i*ldc+j]=beta*MatC[i*ldc+j]+alpha*s;
    }
```

Nuestro querido **dgemm**

Cada kernel calcula una fila de C. **Ojo a las direcciones x e y**

# Problema 4\_3\_2

```
int i,j,r;
double s=0;
j=threadIdx.x+blockIdx.x*blockDim.x;
i=threadIdx.y+blockIdx.y*blockDim.y;
if(i<m && j<n){
    s=0;
    for(r=0;r<k;r++){
        s+=MatA[i*lda+r]*MatBt[j*ldb+r];
    }
    MatC[i*ldc+j]=beta*MatC[i*ldc+j]+alpha*s;
}
```

Nuestro querido **dgemm**

Cada kernel calcula una posición de C. **Ojo a las direcciones x e y**

Van desapareciendo for's...

Ahora ya tiene más sentido

# Problema 4\_3\_2

```
int i,j,r;
double s=0;
j=threadIdx.x+blockIdx.x*blockDim.x;
i=threadIdx.y+blockIdx.y*blockDim.y;
if(i<m && j<n){
    s=0;
    for(r=0;r<k;r++){
        s+=MatA[i*lda+r]*MatBt[j*ldb+r];
    }
    MatC[i*ldc+j]=beta*MatC[i*ldc+j]+alpha*s;
}
```

Nuestro querido **dgemm**

Cada kernel calcula una posición de C. **Ojo a las direcciones x e y**

Van desapareciendo for's...

¿Se puede mejorar aún más?

# Problema 4\_3\_2

```
int i,j,r;
double s=0;
j=threadIdx.x+blockIdx.x*blockDim.x;
i=threadIdx.y+blockIdx.y*blockDim.y;
if(i<m && j<n){
    s=0;
    for(r=0;r<k;r++){
        s+=MatA[i*lda+r]*MatBt[j*ldb+r];
    }
    MatC[i*ldc+j]=beta*MatC[i*ldc+j]+alpha*s;
}
```

Nuestro querido **dgemm**

Cada kernel calcula una posición de C. **Ojo a las direcciones x e y**

Van desapareciendo for's...

¿Se puede mejorar aún más?

**¡¡ES UNA REDUCCIÓN!!**

# Problema 5

```
#define LIMITE 1024
int numeros[LIMITE],i,j;
for(i=2;i<LIMITE;i++)
    numeros[i]=1;
```

¿Cómo se llama?

```
for (i=2;i<LIMITE;i++)
    if (numeros[i])
        for (j=2*i;j<LIMITE;j+=i)
            numeros[j] = 0;
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE){
```

```
        numeros[i]=1;
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
    }
```

```
}
```

¿Funciona?



# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE){
```

```
        numeros[i]=1;
```

Ojo, problema de sinc.

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
    }
```

```
}
```

¿Funciona?

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

Sigue sin haber  
barrera entre hilos

```
    if(i<LIMITE)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

¿Funciona?

```
}
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

¿Funciona?

Llamadas concurrentes acá,  
que no son un problema

```
}
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

¿Funciona?  
Casi...

Deadlock!!!

```
}
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE && i>1)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

¿Funciona?  
Parece que sí.

```
}
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE && i>1)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
}
```

¿Y cómo escribir desde el kernel las primeras 40 posiciones?

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE && i>1)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
    if(i<40)
```

```
        printf("GPU: %d es primo? %d\n",i,numeros[i]);
```

```
}
```

¿Y cómo escribir desde el kernel las primeras 40 posiciones?

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

Sale:

```
__global__ eras(...){  
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;  
    if(i<LIMITE)  
        numeros[i]=1;  
    __syncthreads();  
    if(i<LIMITE && i>1)  
        if (numeros[i])  
            for (j=2*i;j<LIMITE;j+=i)  
                numeros[j] = 0;  
    if(i<40)  
        printf("GPU: %d es primo? %d\n",i,numeros[i]);  
}
```

```
...  
GPU: 37 es primo? 1  
GPU: 38 es primo? 1  
GPU: 39 es primo? 1  
GPU: 2 es primo? 1  
GPU: 3 es primo? 1  
GPU: 4 es primo? 0  
GPU: 5 es primo? 1  
GPU: 6 es primo? 0  
GPU: 7 es primo? 1  
GPU: 8 es primo? 0  
GPU: 9 es primo? 0  
GPU: 10 es primo? 0  
GPU: 11 es primo? 1  
GPU: 12 es primo? 0  
GPU: 13 es primo? 1  
...
```



# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

Sale:

```
__global__ eras(...){  
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;  
    if(i<LIMITE)  
        numeros[i]=1;  
    __syncthreads();  
    if(i<LIMITE && i>1)  
        if (numeros[i])  
            for (j=2*i;j<LIMITE;j+=i)  
                numeros[j] = 0;  
    if(i<40)  
        printf("GPU: %d es primo? %d\n",i,numeros[i]);  
}
```

```
...  
GPU: 37 es primo? 1  
GPU: 38 es primo? 1  
GPU: 39 es primo? 1  
GPU: 2 es primo? 1  
GPU: 3 es primo? 1  
GPU: 4 es primo? 0  
GPU: 5 es primo? 1  
GPU: 6 es primo? 0  
GPU: 7 es primo? 1  
GPU: 8 es primo? 0  
GPU: 9 es primo? 0  
GPU: 10 es primo? 0  
GPU: 11 es primo? 1  
GPU: 12 es primo? 0  
GPU: 13 es primo? 1  
...
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

Sale:

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE && i>1)
```

```
        numeros[i]
```

¿Una barrera acá  
mejora las cosas?

```
        for (j=2*i;j<LIMITE;j+=i)
```

```
            numeros[j] = 0;
```

```
    if(i<40)
```

```
        printf("GPU: %d es primo? %d\n",i,numeros[i]);
```

```
}
```

# Problema 5\_1

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

Sale:

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
        __syncthreads();
```

```
        if(i<LIMITE && i>1)
```

```
            numeros[i]
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
        __syncthreads();
```

```
        if(i<40)
```

```
            printf("GPU: %d es primo? %d\n",i,numeros[i]);
```

```
}
```

Sólo sincroniza  
hilos DEL bloque

...

GPU: 37 es primo? 1

**GPU: 38 es primo? 1**

**GPU: 39 es primo? 1**

GPU: 2 es primo? 1

GPU: 3 es primo? 1

GPU: 4 es primo? 0

GPU: 5 es primo? 1

GPU: 6 es primo? 0

GPU: 7 es primo? 1

GPU: 8 es primo? 0

GPU: 9 es primo? 0

GPU: 10 es primo? 0

GPU: 11 es primo? 1

GPU: 12 es primo? 0

GPU: 13 es primo? 1

...

# Problema 5\_1

```
#define LIMITE 1024
```

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 1 , 1024 >>>(numeros);
```

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE)
```

```
        numeros[i]=1;
```

```
    __syncthreads();
```

```
    if(i<LIMITE && i>1)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
    if(i<40)
```

```
        printf("GPU: %d es primo? %d\n",i,numeros[i]);
```

```
}
```

Criba de Erastótenes

Solución 1: mejor  
decisión de hilos/bloque

Desorden en salida  
inevitable:

GPU: 32 es primo? 0

GPU: 33 es primo? 0

GPU: 34 es primo? 0

GPU: 35 es primo? 0

GPU: 36 es primo? 0

GPU: 37 es primo? 1

GPU: 38 es primo? 0

GPU: 39 es primo? 0

GPU: 2 es primo? 1

GPU: 3 es primo? 1

GPU: 4 es primo? 0

GPU: 5 es primo? 1

...

# Problema 5\_2

```
#define LIMITE 1024
```

Criba de Erastótenes

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
eras<<< 32 , 32 >>>(numeros);
```

Solución 2 (más flexible):  
dos kernels

```
__global__ eras(...){
```

```
    int i=threadIdx.x+blockIdx.x*BlockDim.x,j;
```

```
    if(i<LIMITE && i>1)
```

```
        if (numeros[i])
```

```
            for (j=2*i;j<LIMITE;j+=i)
```

```
                numeros[j] = 0;
```

```
    if(i<40)
```

```
        printf("GPU: %d es primo? %d\n",i,numeros[i]);
```

```
}
```

# Problema 5\_2

```
#define LIMITE 1024
```

```
int *numeros;
```

```
cudaMallocManaged(&numeros,LIMITE*sizeof(int));
```

```
init<<< 32 , 32 >>>(numeros);
```

```
eras<<< 32 , 32 >>>(numeros);
```

Criba de Erastótenes

Solución 2 (más flexible):  
dos kernels

```
__global__ void init(int* numeros) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x,j;  
    if(i<LIMITE)  
        numeros[i]=1;  
}
```

Desorden en salida  
inevitable:

GPU: 32 es primo? 0  
GPU: 33 es primo? 0  
GPU: 34 es primo? 0  
GPU: 35 es primo? 0  
GPU: 36 es primo? 0  
GPU: 37 es primo? 1  
GPU: 38 es primo? 0  
GPU: 39 es primo? 0  
GPU: 2 es primo? 1  
GPU: 3 es primo? 1  
GPU: 4 es primo? 0  
GPU: 5 es primo? 1  
...

# Problema 5\_2

```
#define LIMITE 1024
int *numeros;
cudaMallocManaged(&numeros,
init<<< 32 , 32 >>>(numeros);
eras<<< 32 , 32 >>>(numeros);
```

Criba de Erastótenes

¿Hace falta  
barrera acá?

Solución 2 (más flexible):  
dos kernels

```
__global__ void init(int* numeros) {
    int i=threadIdx.x+blockIdx.x*blockDim.x,j;
    if(i<LIMITE)
        numeros[i]=1;
}
```

Desorden en salida  
inevitable:

GPU: 32 es primo? 0  
GPU: 33 es primo? 0  
GPU: 34 es primo? 0  
GPU: 35 es primo? 0  
GPU: 36 es primo? 0  
GPU: 37 es primo? 1  
GPU: 38 es primo? 0  
GPU: 39 es primo? 0  
GPU: 2 es primo? 1  
GPU: 3 es primo? 1  
GPU: 4 es primo? 0  
GPU: 5 es primo? 1  
...

# Problema 6

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Llamada con un solo  
bloque.

Para un vector de  
valores:

[i]=i+1

a=1, b=2



# Problema 6

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Llamada con un solo  
bloque.  
Para un vector de  
valores:  
[i]=i+1  
a=1, b=2

¿Tiene sentido la  
shared?

# Problema 6

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Llamada con un solo bloque.

Para un vector de valores:

[i]=i+1

a=1, b=2

Saca:

[0]=1

[1]=5

[2]=8

[3]=11

...

[30]=92

[31]=95

[32]=0

[33]=0

[34]=0

[35]=0

...

¿Qué ha pasado?

# Problema 6

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Llamada con un solo bloque.

Para un vector de valores:

[i]=i+1

a=1, b=2

Saca:

[0]=1

[1]=5

[2]=8

[3]=11

...

[30]=92

**[31]=95**

**[32]=0**

[33]=0

[34]=0

[35]=0

...

¿Qué ha pasado?

¿Algo que ver con warps?

# Problema 6\_2

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
    __syncthreads();  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```



Barrera explícita

# Problema 6\_2

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
    __syncthreads();  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Llamada con un solo bloque.  
Para un vector de valores:  
[i]=i+1  
a=1, b=2

Saca:  
[0]=1  
[1]=5  
[2]=8  
[3]=11  
...  
[30]=92  
**[31]=95**  
**[32]=98**  
**[33]=101**  
**[34]=104**  
**[35]=107**  
...

Barrera explícita

# Problema 6\_2

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
    __syncthreads();  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Barrera explícita

Llamada con un solo bloque.

Para un vector de valores:

[i]=i+1

a=1, b=2

Saca:

[0]=1

[1]=5

[2]=8

[3]=11

...

[30]=92

**[31]=95**

**[32]=98**

**[33]=101**

**[34]=104**

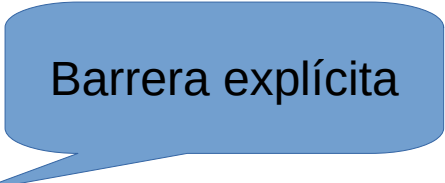
**[35]=107**

...

Barrera sobre TODO  
EL BLOQUE

# Problema 6\_2

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(i==0)  
        buf[0]=a;  
    if(i==1)  
        buf[1]=b;  
    __syncthreads();  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```



Barrera explícita

Llamada con un solo bloque.

Para un vector de valores:

[i]=i+1

a=1, b=2

¿Y si lo hubiéramos llamado con **más de un bloque**?

# Problema 6\_3

```
__global__ void vec(int* vect, int* v, int n, int a, int b) {  
    int i=threadIdx.x+blockIdx.x*blockDim.x;  
    __shared__ int buf[2];  
    if(threadIdx.x==0)  
        buf[0]=a;  
    if(threadIdx.x==1)  
        buf[1]=b;  
    __syncthreads();  
    if(i>0 && i<n)  
        v[i]=buf[0]*vect[i-1]+buf[1]*vect[i];  
}
```

Para un vector de  
valores:  
[i]=i+1  
a=1, b=2



# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

Condiciones para que  
funcione

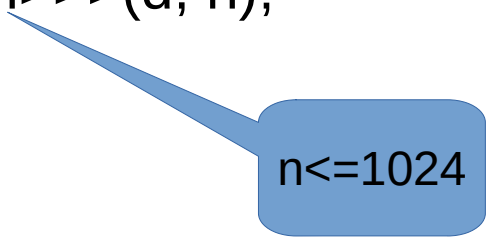
```
...
inv<<<1,n>>>(d, n);
...
```

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

¿Qué hace?

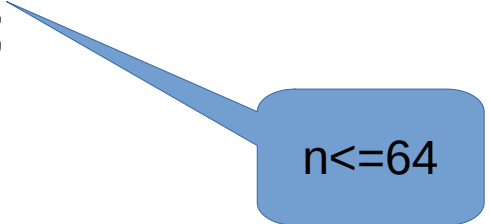
```
...
inv<<<1,n>>>(d, n);
...
```



$n \leq 1024$

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```



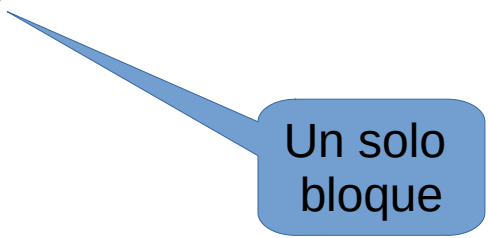
n<=64

¿Qué hace?

```
...
inv<<<1,n>>>(d, n);
...
```

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```



Un solo  
bloque

¿Qué hace?

```
...
inv<<<1,n>>>>(d, n);
...
```

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

¿Qué hace?

¿Tiene sentido?

```
...
inv<<<1,n>>>(d, n);
...
```

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

...
inv<<<1,n>>>(d, n);
...
```

¿Qué hace?

¿Tiene sentido?

No sólo por velocidad,  
sino como **buffer**.

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

...
inv<<<1,n>>>(d, n);
...
```

¿Qué hace?

¿Tiene sentido?

No sólo por velocidad,  
sino como buffer.

**OJO: ¡acceso  
coalescente donde  
más importa (en d[t])!**  
El acceso a shared no  
tiene esas restricciones

# Problema 7

```
__global__ void inv(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

...
inv<<<1,n>>>(d, n);
...
```

¿Qué hace?

¿Tiene sentido?

No sólo por velocidad,  
sino como buffer.

OJO: ¡acceso  
coalescente donde más  
importa (en d[t])!  
El acceso a shared no  
tiene esas restricciones

**¿Se puede hacer sin  
shared?**



# Problema 7

```
__global__ void inv(int *d, int n)
{
    int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

En memoria  
De hilo,  
no de bloque

```
...
inv<<<1,n>>>(d, n);
...
```

¿Qué hace?

¿Tiene sentido?

No sólo por velocidad,  
sino como buffer.

OJO: ¡acceso  
coalescente donde más  
importa (en d[t])!  
El acceso a shared no  
tiene esas restricciones

¿Se puede hacer sin  
shared? ¡¡¡NO!!!!

# Problema 7\_2

```
__global__ void dininv(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

```
...
dininv<<<1,n,n*sizeof(int)>>>(d, n);
...
```

Lo mismo, pero  
dinámicamente.

# Problema 7\_2

```
__global__ void dininv(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

Paréntesis  
vacíos

```
...
dininv<<<1,n,n*sizeof(int)>>>(d, n);
...
```

Lo mismo, pero  
dinámicamente.

# Problema 7\_2

```
__global__ void dininv(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

Paréntesis  
vacíos

En bytes

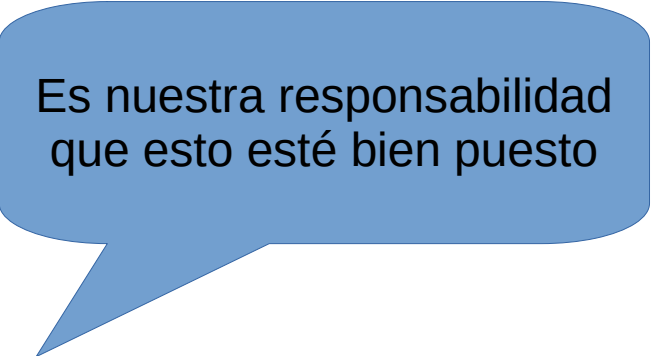
```
...
dininv<<<1,n,n*sizeof(int)>>>(d, n);
...
```

Lo mismo, pero  
dinámicamente.

# Problema 7\_2

```
__global__ void dininv(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

Lo mismo, pero  
dinámicamente.



Es nuestra responsabilidad  
que esto esté bien puesto

```
...
dininv<<<1,n,n*sizeof(int)>>>(d, n);
...
```

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
  
    double tmp=0;  
    if (X<rows){  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*x[j];  
    }  
}  
...  
MatDotVecSh<<<1,n>>>(A, x,v, rows, cols);  
...
```

Usar shared:

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
  
    double tmp=0;  
    if (X<rows){  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*x[j];  
    }  
}  
...  
MatDotVecSh<<<1,n>>>(A, x,v, rows, cols);  
...
```

Usar shared:

n potencia de 2  $\geq$  rows

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){
```

```
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*x[j];  
    }  
}
```

```
...  
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);  
...
```

Usar shared: dinámica



# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){  
  
        for(j=0;j<cols;j++)  
            sh_x[j] = x[j];  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*sh_x[j];  
    }  
}
```

```
...  
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);  
...
```

Usar shared: dinámica

Cargar memoria shared

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){  
  
        for(j=0;j<cols;j++)  
            sh_x[j] = x[j];  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*sh_x[j];  
    }  
}
```

```
...  
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);  
...
```

Usar shared: dinámica

Cargar memoria shared

¿Funciona?

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){  
        for(j=0;j<cols;j++)  
            sh_x[j] = x[j];  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*sh_x[j];  
    }  
}
```

Redundante

Usar shared: dinámica

Cargar memoria shared

¿Funciona?

```
...  
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);  
...
```

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){  
        for(j=0;j<cols;j++)  
            sh_x[j] = x[j];  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*sh_x[j];  
    }  
}
```

Redundante

No hay sincronización

Usar shared: dinámica

Cargar memoria shared

¿Funciona?

```
...  
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);  
...
```

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){  
        if(threadIdx.x==0)  
            for(j=0;j<cols;j++)  
                sh_x[j] = x[j];  
  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*sh_x[j];  
    }  
}
```

Usar shared: dinámica

Cargar memoria shared

¿Funciona?

¿Ya no es redundante?

No hay sincronización

```
...  
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);  
...
```

# Problema 8

```
__global__ void MatDotVecSh(double *A,  
const double *x, double *v,  
const int rows, const int cols)  
{  
    int X = blockIdx.x * blockDim.x + threadIdx.x,j;  
    extern __shared__ double sh_x[];  
    double tmp=0;  
    if (X<rows){  
        if(threadIdx.x==0)  
            for(j=0;j<cols;j++)  
                sh_x[j] = x[j];  
        __syncthreads();  
        for(j=0;j<cols;j++)  
            v[X] += A[X*cols + j]*sh_x[j];  
    }  
}
```

...

```
MatDotVecSh<<<1,n,cols*sizeof(double)>>>(A, x,v, rows, cols);
```

...

Usar shared: dinámica

Cargar memoria shared

¿Funciona?

# Problema 9

```
__global__ void comp(int *d, int nI, int nF, int nC)
{
    extern __shared__ int s[];
    extern __shared__ float f[];
    extern __shared__ char c[];
    ...
}
```

¿Y si necesitamos más  
de un array  
compartido???

```
...
comp<<<1,n,nI*sizeof(int),nF*sizeof(float),nC*sizeof(char)>>>(d, n);
...
```

# Problema 9

```
__global__ void comp(int *d, int nI, int nF, int nC)
{
    extern __shared__ int s[];
    extern __shared__ float f[];
    extern __shared__ char c[];
    ...
}
```

¿Y si necesitamos más  
de un array  
compartido???

**NO COMPILA**

```
...
comp<<<1,n,nI*sizeof(int),nF*sizeof(float),nC*sizeof(char)>>>(d, n);
...
```



# Problema 9

```
__global__ void comp(int *d, int nI, int nF, int nC)
{
    extern __shared__ int s[];
    extern __shared__ float f[];
    extern __shared__ char c[];
    ...
}
```

¿Y si necesitamos más  
de un array  
compartido???

**NO COMPILA**

**Habría que meterlo  
todo en un solo  
vector**

```
...
comp<<<1,n,nI*sizeof(int),nF*sizeof(float),nC*sizeof(char)>>>(d, n);
...
```

# Problema 9\_2

```
__global__ void comp(int *d, int nI, int nF, int nC )  
{  
    extern __shared__ int s[];
```

¿Y si necesitamos más  
de un array  
compartido???

**Reservamos el total...**

```
    ...  
}
```

```
...  
comp<<<..., ..., nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);  
...
```

# Problema 9\_2

```
__global__ void comp(int *d, int nI, int nF, int nC )  
{  
    extern __shared__ int s[];
```

¿Y si necesitamos más  
de un array  
compartido???

**Reservamos el total...**

```
    ...  
}
```

Tamaño total que  
necesitamos

```
    ...  
comp<<<..., ..., nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);  
    ...
```

# Problema 9\_2

```
__global__ void comp(int *d, int nI, int nF, int nC )  
{  
    extern __shared__ int s[];  
  
    ...  
}
```

La magia de  
los punteros

¿Y si necesitamos más  
de un array  
compartido???

**Reservamos el total...**

```
...  
comp<<<..., ..., nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);  
...
```

# Problema 9\_2

```
__global__ void comp(int *d, int nI, int nF, int nC )
{
    extern __shared__ int s[];
    int *integerData = s;
    float *floatData = (float*)&integerData[nI];
    char *charData = (char*)&floatData[nF];
    ...
}
```

¿Y si necesitamos más  
de un array  
compartido???

**¡¡Ahora sí!!**  
(aunque sea un poco  
Frankenstein)

```
...
comp<<<..., ..., nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);
...
```

# Problema 10

```
__global__ void confl(int *d, int n )  
{  
    __shared__ int s[256];  
    s[threadIdx.x]=d[threadIdx.x];  
    __syncthreads();  
    ...  
}
```

¿Esto funciona?

```
...  
confl<<<..., 256>>>(...);  
...
```

# Problema 10

```
__global__ void confl(int *d, int n )  
{  
    __shared__ int s[256];  
    s[threadIdx.x]=d[threadIdx.x];  
    __syncthreads();  
    ...  
}
```

¿Esto funciona?

Sí

```
...  
confl<<<..., 256>>>(...);  
...
```

# Problema 10

```
__global__ void confl(int *d, int n )
{
    __shared__ int s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Esto funciona?

Sí

Pero, ¿funciona **BIEN**?  
Hablemos de  
**conflictos** en los  
bancos...



# Problema 10

```
__global__ void confl(int *d, int n )
{
    __shared__ int s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Esto funciona?

Sí

(int: 32 bits)

El hilo 0 accede a s[0],  
que está en el **primer**  
banco.

El hilo 1 a la posición  
s[1], en el **segundo**  
banco...

# Problema 10

```
__global__ void confl(int *d, int n )
{
    __shared__ int s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Esto funciona?

Sí

(int: 32 bits)

El hilo 0 accede a s[0],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[1], en el segundo  
banco...

**No hay conflictos.**

# Problema 10\_2

```
__global__ void confl(char *d, int n )  
{  
    __shared__ char s[256];  
    s[threadIdx.x]=d[threadIdx.x];  
    __syncthreads();  
    ...  
}
```

¿Y esto?

Sí, pero...

```
...  
confl<<<..., 256>>>(...);  
...
```

# Problema 10\_2

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

**(char: 8 bits)**

El hilo 0 accede a s[0],  
que está en el **primer**  
banco.

El hilo 1 a la posición  
s[1], en el **primer**  
banco...

# Problema 10\_2

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

**(char: 8 bits)**

El hilo 0 accede a s[0],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[1], en el primer  
banco...

**¿CONFLICTO?**

# Problema 10\_2

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

**(char: 8 bits)**

El hilo 0 accede a s[0],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[1], en el **primer**  
banco...

**¿CONFLICTO?**

**No: los hilos 1, 2, 3 y  
4 acceden al primer  
banco. ¿Pertenecen  
al mismo warp?**

# Problema 10\_2

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

(char: 8 bits)

El hilo 0 accede a s[0],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[1], en el **primer**  
banco...

**¿CONFLICTO?**

**No: los hilos 1, 2, 3 y  
4 acceden al primer  
banco. ¿Pertenecen  
al mismo warp?**

**Sí.**

# Problema 10\_2

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

(**char: 8 bits**)

El hilo 0 accede a s[0],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[1], en el **primer**  
banco...

¿**CONFLICTO?**

¿Y los hilos de la  
frontera?



# Problema 10\_2

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

**(char: 8 bits)**

El hilo 0 accede a s[0],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[1], en el **primer**  
banco...

**¿CONFLICTO?**

**El hilo 31 accede a la  
última palabra del  
banco 7.**

**El hilo 32 (nuevo  
warp), a la primera  
palabra del banco 8.  
¿Conflicto?**

# Problema 10\_3

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x+1]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **PERO...**



# Problema 10\_3

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x+1]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

(char: 8 bits)

El hilo 0 accede a s[1],  
que está en el **primer**  
banco.

El hilo 1 a la posición  
s[2], en el **primer**  
banco...

¿**CONFLICTO?**

# Problema 10\_3

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x+1]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, pero...

(char: 8 bits)

El hilo 0 accede a s[1],  
que está en el **primer**  
banco.

El hilo 1 a la posición  
s[2], en el **primer**  
banco...

¿CONFLICTO?

NO, pero...

# Problema 10\_3

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x+1]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

(**char: 8 bits**)

El hilo 0 accede a s[1],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[2], en el primer  
banco...

**¿CONFLICTO?**

El hilo 31 (**primer  
warp**) accede a la  
posición s[32] (**banco  
8**), y el hilo 32  
(**segundo warp**)...

# Problema 10\_3

```
__global__ void confl(char *d, int n )
{
    __shared__ char s[256];
    s[threadIdx.x+1]=d[threadIdx.x];
    __syncthreads();
    ...
}

...
confl<<<..., 256>>>(...);
...
```

¿Y esto?

Sí, **pero...**

(**char: 8 bits**)

El hilo 0 accede a s[1],  
que está en el primer  
banco.

El hilo 1 a la posición  
s[2], en el primer  
banco...

**¿CONFLICTO?**

El hilo 31 (**primer warp**) accede a la  
posición s[32] (**banco 8**), y el hilo 32  
(**segundo warp**) a la  
s[33], del **banco 8**  
**CONFLICTO** →  
**SERIALIZACIÓN** en el  
**acceso**

# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Coalescencia

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Si hay 4 hilos, qué posiciones leerá cada uno?

# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Coalescencia

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Dos opciones:

Th0: 0 1 2

Th1: 3 4 5

Th2: 6 7 8

Th3: 9 A B



# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Dos opciones:

Th0: 0 1 2

Th1: 3 4 5

Th2: 6 7 8

Th3: 9 A B

Th0: 0 4 8

Th1: 1 5 9

Th2: 2 6 A

Th3: 3 7 B

Coalescencia

¿Cuál es más rápida?  
¿Cuál respeta la  
coalescencia?

# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Dos opciones:

Th0: 0 1 2

Th1: 3 4 5

Th2: 6 7 8

Th3: 9 A B

Primer acceso: posiciones 0 3 6 9

Coalescencia

¿Cuál es más rápida?  
¿Cuál respeta la  
coalescencia?

# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Coalescencia

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Dos opciones:

Th0: 0 1 2

Th1: 3 4 5

Th2: 6 7 8

Th3: 9 A B

Primer acceso: posiciones 0 3 6 9

**NO CONSECUTIVAS → No coalescente**

¿Cuál es más rápida?

¿Cuál respeta la coalescencia?

# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Coalescencia

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Dos opciones:

Primer acceso: 0 1 2 3

Th0: 0 4 8

Th1: 1 5 9

Th2: 2 6 A

Th3: 3 7 B

¿Cuál es más rápida?

¿Cuál respeta la  
coalescencia?

# Problema 11

Tenemos una matriz de 3x4:

0 1 2 3  
4 5 6 7  
8 9 A B

Coalescencia

Almacenada en row-major:

0 1 2 3 4 5 6 7 8 9 A B

Dos opciones:

Primer acceso: 0 1 2 3

**CONSECUTIVAS** → **coalescente**

Th0: 0 4 8

Th1: 1 5 9

Th2: 2 6 A

Th3: 3 7 B

¿Cuál es más rápida?

¿Cuál respeta la  
coalescencia?

# Problema 11

Tenemos una matriz de 3x4:

```
0 1 2 3
4 5 6 7
8 9 A B
```

Coalescencia

Almacenada en row-major:

```
0 1 2 3 4 5 6 7 8 9 A B
```

Dos opciones:

Primer acceso: 0 1 2 3

**CONSECUTIVAS** → **coalescente**

Th0: 0 4 8

Th1: 1 5 9

Th2: 2 6 A

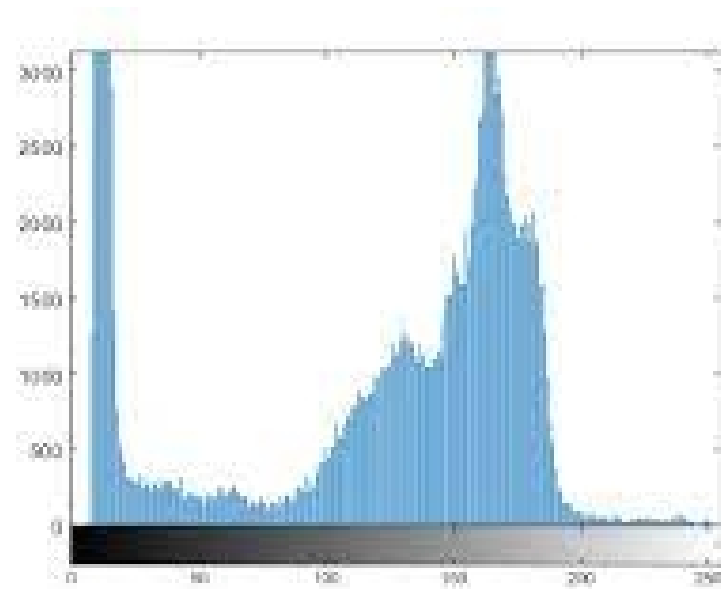
Th3: 3 7 B

¿Nos suena a lo de "stride"?

¿Cuál es más rápida?

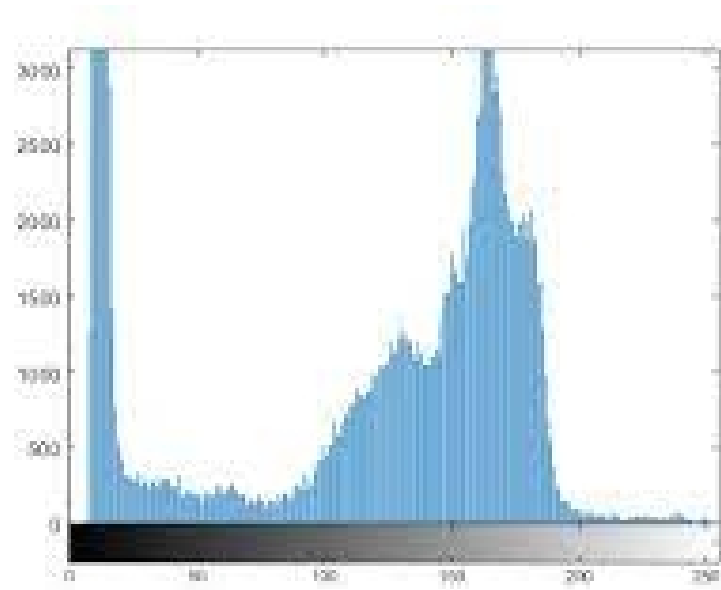
¿Cuál respeta la coalescencia?

# Problema 12



Histogramas...

# Problema 12



Histogramas...

¿Cómo se calcula en C?



# Problema 12

```
char image*={127,0,255,24,...};
```

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
int i,j;  
for(i=0;i<dimY;i++)  
    for(j=0;j<dimX;j++)
```

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
int i,j;  
for(i=0;i<dimY;i++)  
    for(j=0;j<dimX;j++)  
        hist[(unsigned char)image[i*dimX+j]]++;
```

Y ya está :)

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
int i,j;  
for(i=0;i<dimY;i++)  
    for(j=0;j<dimX;j++)  
        hist[(unsigned char)image[i*dimX+j]]++;
```

Y ya está :)  
Y... ¿en CUDA?

# Problema 12

```
char image*={127,0,255,24,...};
```

```
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){
```

```
int i,j;
```

```
for(i=0;i<dimY;i++)
```

```
    for(j=0;j<dimX;j++)
```

```
        hist[(unsigned char)image[i*dimX+j]]++;
```

```
}
```

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        hist[(unsigned char)image[i*dimX+j]]++;  
}
```

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        hist[(unsigned char)image[i*dimX+j]]++;  
}
```

Condición de  
carrera



# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        atomicAdd(&hist[(unsigned char)image[i*dimX+j]],1);  
}
```

Tiempo:  $2,1 \cdot 10^{-3}$   
Funciona, pero...

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        atomicAdd(&hist[(unsigned char)image[i*dimX+j]],1);  
}
```

Funciona, pero...  
¿Cuántos hilos entran en  
conflicto en ese atomic?

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        atomicAdd(&hist[(unsigned char)image[i*dimX+j]],1);  
}
```

Funciona, pero...  
¿Cuántos hilos entran en  
conflicto en ese atomic?  
...como mucho?

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        atomicAdd(&hist[(unsigned char)image[i*dimX+j]],1);  
}
```

Funciona, pero...  
¿Cuántos hilos entran en conflicto en ese atomic?  
...como mucho?  
¿**Shared** mejoraría algo?

# Problema 12

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ histogram(char* image, int dimX, int dimY, int[] hist){  
    int j = threadIdx.x+blockIdx.x*blockDim.x;  
    int i = threadIdx.y+blockIdx.y*blockDim.y;  
    if(i<dimY && j<dimX)  
        atomicAdd(&hist[(unsigned char)image[i*dimX+j]],1);  
}
```

Funciona, pero...  
¿Cuántos hilos entran en conflicto en ese atomic? ...como mucho?  
¿**Shared** mejoraría algo?  
Atomic sobre shared es mucho más rápida que sobre la memoria global.

# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < yres*xres) { atomicAdd(&sh[(unsigned char)A[i]],1); }  
  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]); }  
}
```

# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < yres*xres) { atomicAdd(&sh[(unsigned char)A[i]],1); }  
  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]);  
    }  
}
```



Falta  
sincronización

# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < yres*xres) { atomicAdd(&sh[(unsigned char)A[i]],1); }  
    __syncthreads();  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]); }  
}
```



# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < yres*xres) { atomicAdd(&sh[(unsigned char)A[i]],1); }  
    __syncthreads();  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]); }  
}
```

Error en los valores de hist  
¿Qué está fallando?

# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x +  
    if (i < yres*xres) { atomicAdd(&sh[(unsigned char)A[i]],1); }  
    __syncthreads();  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]); }  
}
```

La shared no tiene valores controlados en su inicio. ¿Sobre qué estamos sumando un 1?

Error en los valores de hist  
¿Qué está fallando?

# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if (threadIdx.x < 256) { sh[threadIdx.x] = 0; }  
    __syncthreads();  
    if (i < yres*xres) { atomicAdd(&sh[(unsigned char)A[i]],1); }  
    __syncthreads();  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]); }  
}
```

Error=0

Tiempo GPU:  $2,1 \cdot 10^{-3}$

Tiempo GPU con shared:  $1,4 \cdot 10^{-3}$

# Problema 12\_2

```
char image*={127,0,255,24,...};  
int hist[256]={0,.....0};
```

```
__global__ void kernelHistSh1D(int xres, int yres, double* A, int* hist)  
{  
    __shared__ int sh[256];  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if (threadIdx.x < 256) { sh[threadIdx.x] = 0; }  
    __syncthreads();  
    if (i < yres*xres) { atomicAdd(&hist[threadIdx.x], A[i]); }  
    __syncthreads();  
    if (threadIdx.x < 256) {  
        atomicAdd(&hist[threadIdx.x], sh[threadIdx.x]); }  
}
```

Estos dos "if" podrían quitarse si garantizamos que los bloques siempre tienen 256 hilos, pero no cambia los tiempos

Error=0

Tiempo GPU:  $2,1 \cdot 10^{-3}$

Tiempo GPU con shared:  $1,4 \cdot 10^{-3}$

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

Tamaño de cache igual  
que cada bloque.

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

Índice para movernos  
sobre a y b

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

Índice para movernos  
sobre cache

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```



# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

Acumulamos sobre temp

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsWithinBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

Acumulamos sobre temp  
...de malla en malla

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }
```

Guardamos en cache y sync.

```
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

¿Qué hace?

¿Qué tamaño útil tiene c?

Reducción N pasos 2 a 1  
(sólo la mitad del bloque  
trabaja en la primera  
iteración.  $\frac{1}{4}$  en la  
segunda,  $\frac{1}{8}$  en la  
tercera...)

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

¿Qué datos tiene cache  
acá?



```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

Un hilo por bloque guarda el dato en la pos “ID de bloque”.

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

# Problema x

```
__global__ void dot( float *a, float *b, float *c, int N ) {  
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

¿Qué hace?

¿Qué tamaño útil tiene c?

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
    cache[cacheIndex] = temp;  
    __syncthreads();  
    int i = blockDim.x/2;  
    while (i != 0) {  
        if (cacheIndex < i)  
            cache[cacheIndex] += cache[cacheIndex + i];  
        __syncthreads();  
        i /= 2;  
    }
```

¿Qué contenido tiene c al terminar?

```
    if (cacheIndex == 0)  
        c[blockIdx.x] = cache[0];  
}
```

