Universidad Católica San Pablo

CUDA **Shared Memory**

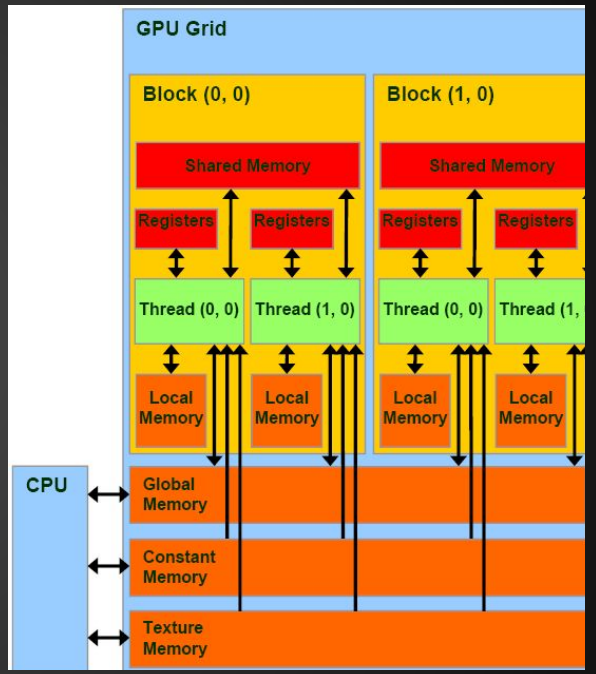
Marc-Antoine Le Guen

mleguen@ucsp.edu.pe

07/07/2017

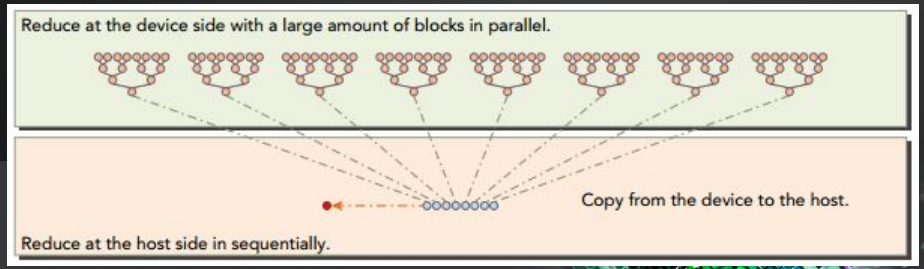
Shared memory

- __shared__
- Comunicación entre los threads de cada block
- High-Performance (low latency)
- Copia de la variable shared en cada block
- Cada thread del block comparte la variable
  - Sincronización



Shared memory - Producto escalar

- Ejemplo : $\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ...a_n b_n = \sum a_i \cdot b_i$
- Cada block contiene una tabla caché de tamaño blockDim.x
  - En esta tabla cada thread del block guardará los resultados de sus cálculos
  - Ejemplo
    - blockDim.x * gridDim.x = 100
    - N = 250
    - thread n°0 cache[0] = a0*b0 + a100*b100 + b200*b200
  - Se hará la suma de la tabla caché por cada block
  - Se retornará esta suma en una tabla C de tamaño GridDim.x
  - Se hará la suma en el CPU de esta tabla C



Shared memory - Producto escalar

- Ejemplo : $\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ... a_n b_n = \sum a_i \cdot b_i$

```
__global__ void dot( float *a, float *b, float *c ) {
        __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;
        }

        // set the cache values
        cache[cacheIndex] = temp;
}
```

- **tid** : idem que para la suma de dos vectores
- **cache** contiene la suma de las multiplicaciones de cada thread.
- **cacheIndex** = threadIdx.x el índice para el array **cache**
- **temp** es la suma de las multiplicaciones del thread actual.

Shared memory - Producto escalar

- Ejemplo : $\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ... a_n b_n = \sum a_i \cdot b_i$

```
__global__ void dot( float *a, float *b, float *c ) {
        __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;
        }

        // set the cache values
        cache[cacheIndex] = temp;
}
```

- **tid** : idem que para la suma de dos vectores
- **cache** contiene la suma de las multiplicaciones de cada thread.
- **cacheIndex** = threadIdx.x el índice para el array **cache**
- **temp** es la suma de las multiplicaciones del thread actual.

Shared memory - Producto escalar

- Ejemplo : $\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ... a_n b_n = \sum a_i \cdot b_i$

```
__global__ void dot( float *a, float *b, float *c ) {
        __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;
        }

        // set the cache values
        cache[cacheIndex] = temp;
}
```

- **tid** : idem que para la suma de dos vectores
- **cache** contiene la suma de las multiplicaciones de cada thread.
- **cacheIndex** = threadIdx.x el índice para el array **cache**
- **temp** es la suma de las multiplicaciones del thread actual.

Shared memory - Producto escalar

- Ejemplo : $\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ... a_n b_n = \sum a_i \cdot b_i$

```
__global__ void dot( float *a, float *b, float *c ) {
        __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;
        }

        // set the cache values
        cache[cacheIndex] = temp;
}
```

- **tid** : idem que para la suma de dos vectores
- **cache** contiene la suma de las multiplicaciones de cada thread.
- **cacheIndex** = threadIdx.x el índice para el array **cache**
- **temp** es la suma de las multiplicaciones del thread actual.

Shared memory - Producto escalar

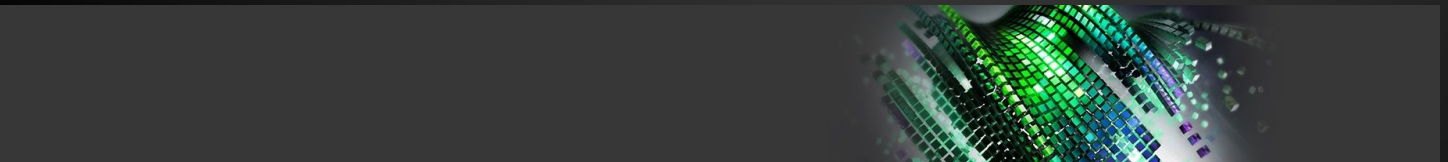
- Queremos hacer una operación sobre la variable en memoria compartida
  - Tenemos que asegurarnos que se terminó el trabajo previo sobre esta variable
  - Sincronizar los threads que comparten esta variable :

```
__global__ void dot( float *a, float *b, float *c ) {
        __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;
        }

        // set the cache values
        cache[cacheIndex] = temp;
      // synchronize threads in this block
      __syncthreads();

      // ready to go...
}
```

- **tid** : idem que para la suma de dos vectores
- **cache** contiene la suma de las multiplicaciones de cada thread.
- **cacheIndex** = threadIdx.x el índice para el array **cache**
- **temp** es la suma de las multiplicaciones del thread actual.

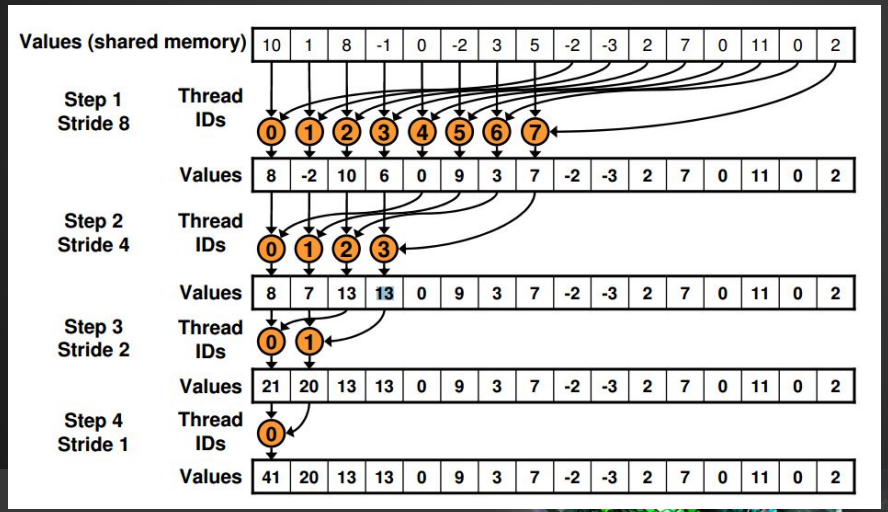
Shared memory - Producto escalar

- Nos queda hacer la suma del array cache para cada block
- Pasar de una tabla a una tabla b más pequeña tiene un nombre : reducción
- Una manera de hacer esto es dedicar un thread del block a este trabajo O(n)
- **Existen maneras de hacerlo de manera paralela.**
  - **Ejemplo con una cantidad de threads en potencia de 2.**



Shared memory - Reducción

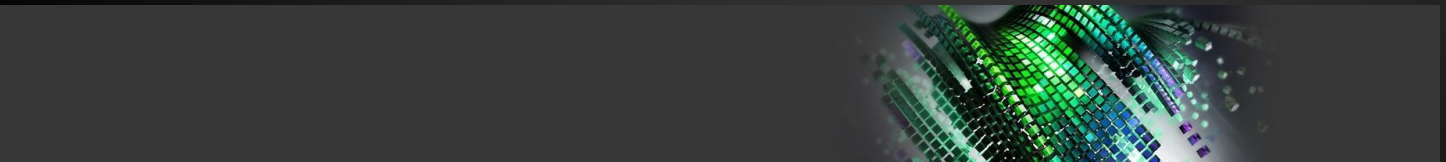
- **Cada thread sumará dos valores del caché[]**
  - **En una iteración estamos con la mitad del tamaño original**



Shared memory - Producto escalar

```
__global__ void dot( float *a, float *b, float *c ) {
        . . . . . .
         cache[cacheIndex] = temp;
       // synchronize threads in this block
       __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];

}
```

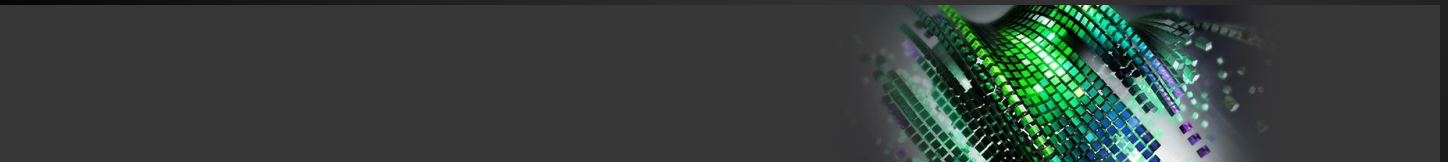
- Cada etapa necesita ser sincronizada
- El resultado de reducción se encuentra en el primer elemento de cache[]
- Se hará la suma final de `c` en el CPU
  - c contiene la suma hecha por cada block
  - `c.length = NbBlocks`



Shared memory - Producto escalar

```
__global__ void dot( float *a, float *b, float *c ) {
        . . . . . .
        cache[cacheIndex] = temp;
      // synchronize threads in this block
      __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];

}
```

- Cada etapa necesita ser sincronizada
- El resultado de reducción se encuentra en el primer elemento de cache[]
- Se hará la suma final de `c` en el CPU
  - c contiene la suma hecha por cada block
  - `c.length = NbBlocks`

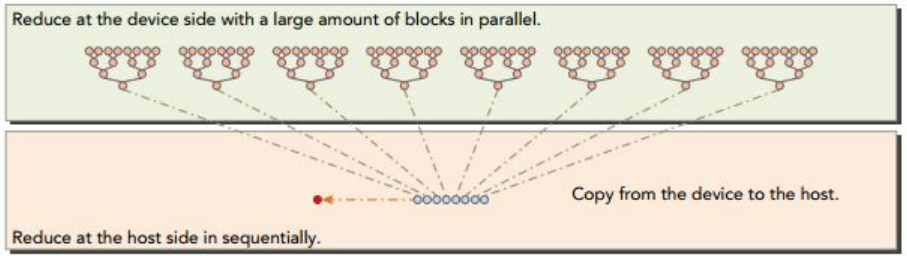


Shared memory - Producto escalar

- Cuidado con el uso de __syncthreads()

```
__global__ void dot( float *a, float *b, float *c ) {
        . . . . . .
        cache[cacheIndex] = temp;
      // synchronize threads in this block
      __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];

}
```

- **La tarjeta gráfica esperara para siempre que todos los threads pasen por** `__syncthreads()`



Shared memory - Producto escalar

- Ejemplo : $\mathbf{A}\cdot\mathbf{B} = (a_1, a_2, a_3, ..., a_n)\cdot(b_1, b_2, b_3, ..., b_n) = a_1b_1 + a_2b_2 + ...a_nb_n = \sum a_i \cdot b_i$

Main (1)

```
      float   *a, *b, c, *partial_c;
       float   *dev_a, *dev_b, *dev_partial_c;

       // allocate memory on the cpu side
       a = (float*)malloc( N*sizeof(float) );
       b = (float*)malloc( N*sizeof(float) );
       partial_c = (float*)malloc(
blocksPerGrid*sizeof(float) );
       // allocate the memory on the GPU
       cudaMalloc( (void**)&dev_a,
                           N*sizeof(float) );
       cudaMalloc( (void**)&dev_b,
                           N*sizeof(float) );
       cudaMalloc( (void**)&dev_partial_c,
                  blocksPerGrid*sizeof(float) );
       // fill in the host memory with data
       for (int i=0; i<N; i++) {
          a[i] = i;
          b[i] = i*2;
       }
```

Main(2)

```
       // copy the arrays 'a' and 'b' to the GPU
       cudaMemcpy( dev_a, a, N*sizeof(float),
                              cudaMemcpyHostToDevice );
       cudaMemcpy( dev_b, b, N*sizeof(float),
                     cudaMemcpyHostToDevice );

       dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a,
dev_b,        dev_partial_c );

       // copy the array 'c' back from the GPU to the CPU
       cudaMemcpy( partial_c, dev_partial_c,
                           blocksPerGrid*sizeof(float),
                           cudaMemcpyDeviceToHost );
       // finish up on the CPU side
       c = 0;
       for (int i=0; i<blocksPerGrid; i++) {
       c += partial_c[i];
       }
```

Shared memory - Producto escalar

- Ejemplo : $\mathbf{A}\cdot\mathbf{B} = (a_1, a_2, a_3, ..., a_n)\cdot(b_1, b_2, b_3, ..., b_n) = a_1b_1 + a_2b_2 + ...a_nb_n = \sum a_i \cdot b_i$

Main (1)

```
float   *a, *b, c, *partial_c;
  float   *dev_a, *dev_b, *dev_partial_c;

  // allocate memory on the cpu side
  a = (float*)malloc( N*sizeof(float) );
  b = (float*)malloc( N*sizeof(float) );
  partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
  // allocate the memory on the GPU
  cudaMalloc( (void**)&dev_a,
                  N*sizeof(float) );
  cudaMalloc( (void**)&dev_b,
                  N*sizeof(float) );
  cudaMalloc( (void**)&dev_partial_c,
            blocksPerGrid*sizeof(float) );
  // fill in the host memory with data
  for (int i=0; i<N; i++) {
     a[i] = i;
     b[i] = i*2;
  }
```

Main(2)

```
        // copy the arrays 'a' and 'b' to the GPU
        cudaMemcpy( dev_a, a, N*sizeof(float),
                        cudaMemcpyHostToDevice );
        cudaMemcpy( dev_b, b, N*sizeof(float),
                        cudaMemcpyHostToDevice );

        dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
dev_partial_c );

        // copy the array 'c' back from the GPU to the CPU
        cudaMemcpy( partial_c, dev_partial_c,
        blocksPerGrid*sizeof(float),
                        cudaMemcpyDeviceToHost);
        // finish up on the CPU side
        c = 0;
        for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
        }
```

Shared memory - Producto escalar

- Ejemplo : $\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ... a_n b_n = \sum a_i \cdot b_i$

Main (1)

```
float   *a, *b, c, *partial_c;
   float   *dev_a, *dev_b, *dev_partial_c;

   // allocate memory on the cpu side
   a = (float*)malloc( N*sizeof(float) );
   b = (float*)malloc( N*sizeof(float) );
   partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
   // allocate the memory on the GPU
   cudaMalloc( (void**)&dev_a,
                    N*sizeof(float) );
   cudaMalloc( (void**)&dev_b,
                    N*sizeof(float) );
   cudaMalloc( (void**)&dev_partial_c,
             blocksPerGrid*sizeof(float) );
   // fill in the host memory with data
   for (int i=0; i<N; i++) {
      a[i] = i;
      b[i] = i*2;
   }
```

Main(2)

```
        // copy the arrays 'a' and 'b' to the GPU
        cudaMemcpy( dev_a, a, N*sizeof(float),
                          cudaMemcpyHostToDevice );
        cudaMemcpy( dev_b, b, N*sizeof(float),
                        cudaMemcpyHostToDevice );

        dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
dev_partial_c );

        // copy the array 'c' back from the GPU to the CPU
        cudaMemcpy( partial_c, dev_partial_c,
        blocksPerGrid*sizeof(float),
cudaMemcpyDeviceToHost);
        // finish up on the CPU side
        c = 0;
        for (int i=0; i<blocksPerGrid; i++) {
             c += partial_c[i];
        }
```

Shared memory - Resumen

- Medio de comunicación entre los threads un block
- Reducir el uso del ancho de banda de la memoria