

CUDA Getting started

Marc-Antoine Le Guen

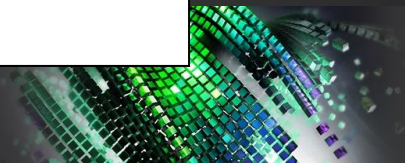
CUDA - Arquitectura de un programa

- Host Code == programación clásica C en CPU
- Device Code == Programación en GPU



- Ejemplo Host

```
int main( void ) {  
    printf( "Hello, World!\n"  
);  
    return 0;  
}
```



Kernel

- Función GPU
- `__global__` == Device (para el compilador)
- Ejemplo

```
__global__ void kernel (void)
{}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Kernel - Parámetros

- Clásico C
- **cudaMalloc** : Asignación memoria en el device
- **cudaMemcpy** : Copia datos memoria Host <-> Device
- **cudaFree** : Liberar memoria alocada con cudaMalloc
- Ejemplo

```
__global__ void add( int a, int b, int *c ) {  
    *c = a + b;  
}  
  
int main( void ) {  
    int c;  
    int *dev_c;  
    cudaMalloc((void**)&dev_c, sizeof(int));  
    add<<<1,1>>>( 4, 8, dev_c );  
    cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost );  
    printf( "2 + 7 = %d\n", c );  
    cudaFree( dev_c );  
    return 0;  
}
```

Kernel - cudaMalloc

- Qué puedes hacer con un puntero asignado con cudaMalloc ?
- Puedes pasarlo como parametro a una función que se ejecutará en el Device
- Puedes usarlo para leer o escribir en una función del Device
- Puedes pasarlo como parametro a una función que se ejecutará en el Host
- **No puedes** usarlo para leer o escribir en una función del Host



Kernel - cudaMemcpy

- **Standard C memcpy**
 - cudaMemcpyHostToDevice
 - cudaMemcpyDeviceToHost
 - cudaMemcpyDeviceToDevice
 - memcpy HostToHost



Propiedades del Device

- Cuantos devices y Como recuperar sus propiedades ?

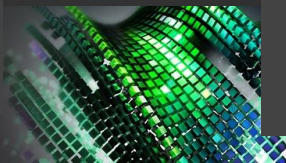
```
cudaDeviceProp prop ;  
int count ;  
cudaGetDeviceCount ( &count ) ;  
for (int i=0; i< count; i++) {  
    cudaGetDeviceProperties ( &prop, i);  
}
```



Propiedades del Device

```
struct cudaDeviceProp {  
    char name[256];  
    size_t totalGlobalMem;  
    size_t sharedMemPerBlock;  
    int regsPerBlock;  
    int warpSize;  
    size_t memPitch;  
    int maxThreadsPerBlock;  
    int maxThreadsDim[3];  
    int maxGridSize[3];  
    int clockRate;  
    size_t totalConstMem;  
    int major;  
    int minor;  
    size_t textureAlignment;  
    size_t texturePitchAlignment;  
    int deviceOverlap;  
    int multiProcessorCount;  
    int kernelExecTimeoutEnabled;  
    int integrated;  
    int canMapHostMemory;  
    int computeMode;  
    int maxTexture1D;  
    int maxTexture1DLinear;
```

```
    int maxTexture2D[2];  
    int maxTexture2DLinear[3];  
    int maxTexture2DGather[2];  
    int maxTexture3D[3];  
    int maxTextureCubemap;  
    int maxTexture1DLayered[2];  
    int maxTexture2DLayered[3];  
    int maxTextureCubemapLayered[2];  
    int maxSurface1D;  
    int maxSurface2D[2];  
    int maxSurface3D[3];  
    int maxSurface1DLayered[2];  
    int maxSurface2DLayered[3];  
    int maxSurfaceCubemap;  
    int maxSurfaceCubemapLayered[2];  
    size_t surfaceAlignment;  
    int concurrentKernels;  
    int ECCEnabled;  
        int pciBusID;  
    int pciDeviceID;  
    int pciDomainID;  
    int tccDriver;  
    int asyncEngineCount;  
    int unifiedAddressing;  
    int memoryClockRate;
```



Propiedades del Device

name[256]	is an ASCII string identifying the device;
totalGlobalMem	is the total amount of global memory available on the device in bytes
sharedMemPerBlock	is the maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor
regsPerBlock	is the maximum number of 32-bit registers available to a thread block
warpSize	is the warp size in threads;
maxThreadsPerBlock	is the maximum number of threads per block;
maxThreadsDim[3]	contains the maximum size of each dimension of a block
maxGridSize[3]	contains the maximum size of each dimension of a grid;
maxTexture2D[2]	contains the maximum 2D texture dimensions.

Propiedades del Device

- Exemplo de uso :

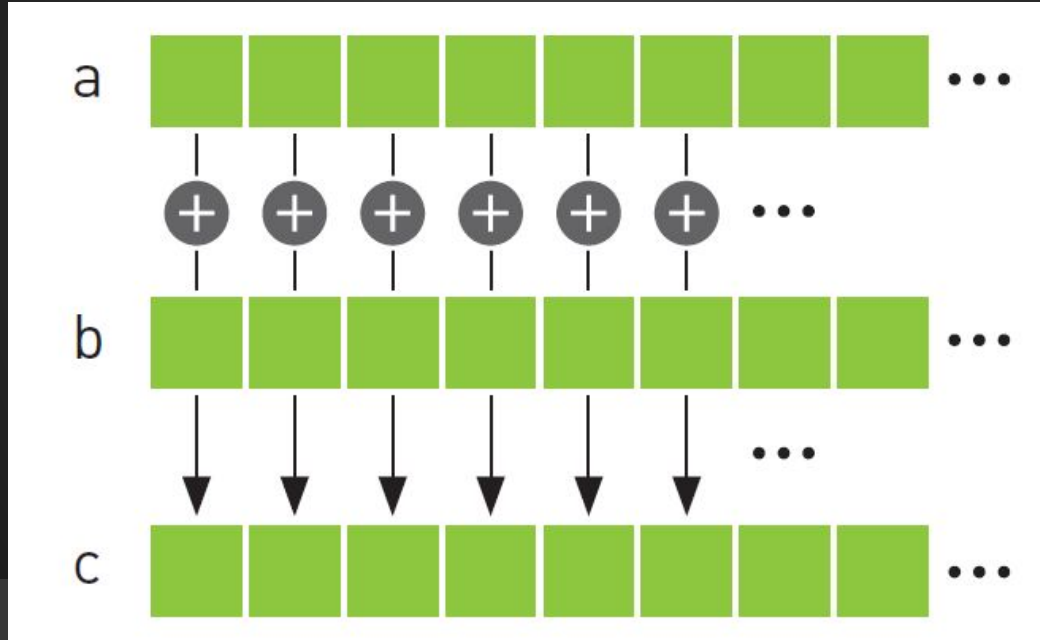
```
int main( void ) {
    cudaDeviceProp prop;
    int dev;
    cudaGetDevice( &dev );

    printf( "ID of current CUDA device: %d\n", dev );
    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 3;
    prop.minor = 0;
    cudaChooseDevice( &dev, &prop );
    printf( "ID of CUDA device closest to revision 3.0: %d\n", dev );
    cudaSetDevice( dev );
}
```



Primer programa

- Sumar dos vectores



Primer programa

- Sumar dos vectores
- CPU

```
void add( int *a, int *b, int *c ) {  
    int tid = 0; // this is CPU zero, so we start at zero  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        // we have one CPU, so we increment by one  
        tid += 1;    }  
}
```

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



Primer programa

- Sumar dos vectores
- 2 CPU

```
void add( int *a, int *b, int *c ) {  
    int tid = 0; // this is CPU zero, so we start at  
    zero  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += 2;  
    }  
}
```

CPU #1

```
void add( int *a, int *b, int *c ) {  
    int tid = 1; // this is CPU one, so we start at one  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += 2;  
    }  
}
```

CPU #2



Primer programa

- Sumar dos vectores
- GPU
 - `__global__`

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



Primer programa

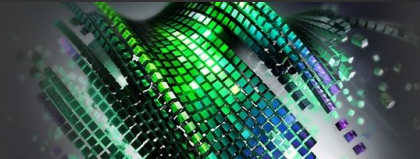
- Sumar dos vectores
- `<<N,1>>` N copias del kernel ($N < 65,535$)
- `blockIdx.x` el índice del block actual

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

```
add<<<N,1>>>>(vec_a,vec_b,vec_c) ;
```

GPU

main



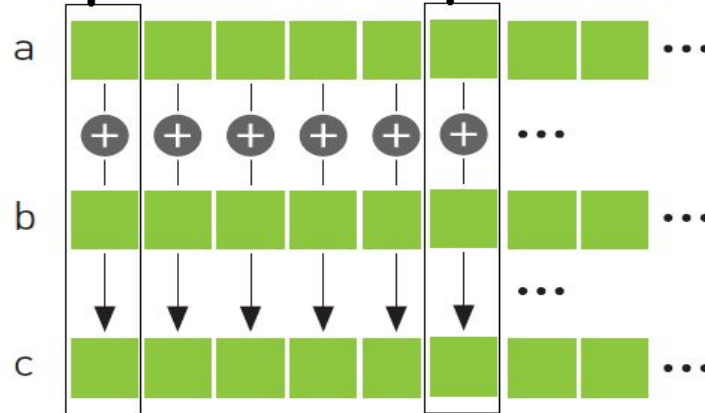
Primer programa

Kernel Copy n°0

`BlockIdx.x == 0`

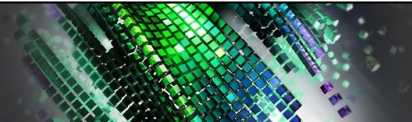
Kernel Copy n°5

`BlockIdx.x == 5`



Primer programa

```
int main( void ) {  
    // Host variables  
    int a[N], b[N], c[N];  
    // device variables  
    int *dev_a, *dev_b, *dev_c;
```



Primer programa

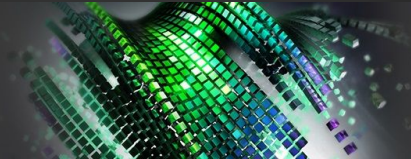
```
int main( void ) {  
    // Host variables  
    int a[N], b[N], c[N];  
    // device variables  
    int *dev_a, *dev_b, *dev_c;  
  
    // allocate the memory on the GPU  
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );  
  
    // fill the arrays 'a' and 'b' on the CPU  
    for (int i=0; i<N; i++) {  
        a[i] = -i;  
        b[i] = i * i;  
    }  
}
```



Primer programa

```
int main( void ) {  
    // Host variables  
    int a[N], b[N], c[N];  
    // device variables  
    int *dev_a, *dev_b, *dev_c;  
  
    // allocate the memory on the GPU  
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );  
  
    // fill the arrays 'a' and 'b' on the CPU  
    for (int i=0; i<N; i++) {  
        a[i] = -i;  
        b[i] = i * i;  
    }  
}
```

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

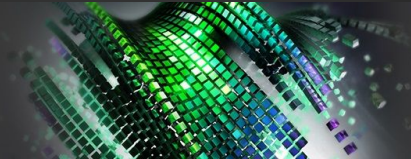


Primer programa

```
int main( void ) {  
    // Host variables  
    int a[N], b[N], c[N];  
    // device variables  
    int *dev_a, *dev_b, *dev_c;  
  
    // allocate the memory on the GPU  
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );  
  
    // fill the arrays 'a' and 'b' on the CPU  
    for (int i=0; i<N; i++) {  
        a[i] = -i;  
        b[i] = i * i;  
    }  
}
```

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

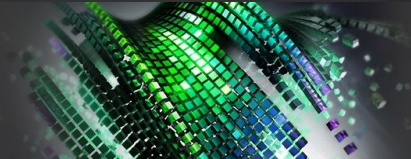
```
add<<<N,1>>>>( dev_a, dev_b, dev_c );
```



Primer programa

```
int main( void ) {  
    // Host variables  
    int a[N], b[N], c[N];  
    // device variables  
    int *dev_a, *dev_b, *dev_c;  
  
    // allocate the memory on the GPU  
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );  
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );  
  
    // fill the arrays 'a' and 'b' on the CPU  
    for (int i=0; i<N; i++) {  
        a[i] = -i;  
        b[i] = i * i;  
    }  
}
```

```
    // copy the arrays 'a' and 'b' to the GPU  
    cudaMemcpy( dev_a, a, N * sizeof(int),  
                );  
  
    cudaMemcpy( dev_b, b, N * sizeof(int),  
                cudaMemcpyHostToDevice );  
  
    add<<<N,1>>>>( dev_a, dev_b, dev_c );  
  
    // copy the array 'c' back from the GPU to the CPU  
    cudaMemcpy( c, dev_c, N * sizeof(int),  
                cudaMemcpyDeviceToHost );  
}
```



Primer programa

```
int main( void ) {
    // Host variables
    int a[N], b[N], c[N];
    // device variables
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
}
```

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

cudaMemcpy( dev_b, b, N * sizeof(int),
                                                    cudaMemcpyHostToDevice );

add<<<N,1>>>>( dev_a, dev_b, dev_c );

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy( c, dev_c, N * sizeof(int),
                                                    cudaMemcpyDeviceToHost );

// display the results
for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}

// free the memory allocated on the GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
```

```
}
```



Primer programa

1. **Asignación de memoria en GPU**
2. **Copiar memoria del CPU hacia el GPU**
3. **Invocar al kernel de CUDA**
4. **Copiar datos del GPU hacia el CPU**
5. **Liberar la memoria del GPU**



Dificultad de programar en CUDA

- Pensar en paralelo
- Entender la arquitectura de un GPU
- Entender la idea de proximidad “Locality” para reducir el tiempo de acceso a los datos
- Estructura jerárquica de memoria
- Estructura jerárquica de threads
- **Cuda Ambiente de Desarrollo**
 - NVIDIA Nsight™ integrated development environment
 - CUDA-GDB command line debugger
 - CUDA-MEMCHECK memory analyzer
 - GPU device management tools

