

Cuda Kernels

A CUDA Kernel function is defined using the `__global__` keyword.

A Kernel is executed N times in parallel by N different threads on the device

Each thread has a unique ID stored in the built-in `threadIdx` variable, a struct with components x,y,z.

Each thread block has a unique ID stored in the built-in `blockIdx` variable, a struct with components x,y,z.

Kernel Configuration

Kernel `kernel Fun cti on< <<n um_ blocks, num_th reads> >`

Execution `>( params)`

Config-
uration

`num_blocks` The number of thread blocks along each dimension of the grid.

`num_th-reads` The number of threads along each dimension of the thread block

CUDA Thread Organization

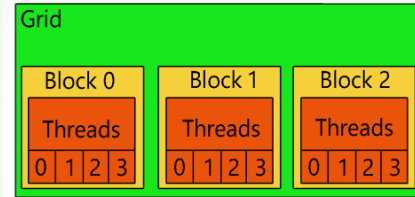
Thread are grouped in blocks and can be organized in 1 to 3 dimensions.

Blocks are grouped into grids which can be organized in 1 to 3 dimensions.

Blocks are executed independently.

1D Grid of 1D Blocks

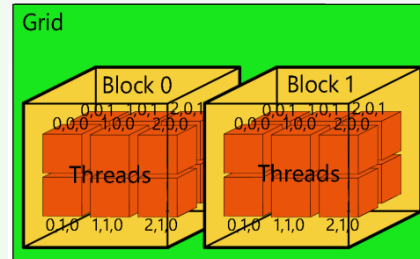
1D Grid of 1D Blocks



```
int index = blockIdx.x * blockDim.x + threadIdx.x
```

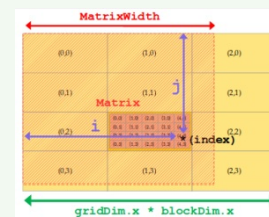
1D Grid of 3D Blocks

1D Grid of 3D Blocks



```
int index = blockIdx.x * blockDim.x * blockDim.y * blockDim.z + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

2D Grid of 2D Blocks applied on a Matrix



The index of each thread is identified by two coordinates *i* and *j*. We can find *i* applying the rule of 1D Grid of 1D Blocks over the x axis:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

And we can find *j* applying the rule of 1D Grid of 1D Blocks over the y axis:

```
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Thus, knowing that a row in the grid is large *GridDim.x times BlockDim.x*, we can calculate the index:

```
int index = j * gridDim.x * blockDim.x + i;
```

CUDA Events

Declaring a Cuda Event	<code>cudaEvent_t event;</code>
Allocating the event	<code>cudaEventCreate(&event);</code>
Recording the Event.	<code>cudaEventRecord(event);</code>
Synchronizing the event	<code>cudaEventSynchronize(event);</code>
Find elapsed time between two events	<code>cudaEventElapsedTime(&elapsed, a, b);</code>
Free event variables	<code>cudaEventDestroy(event);</code>

CUDA Streams

GPU operations on CUDA use execution queues called streams.
Operations pushed in a stream are executed according to a FIFO policy.
There is a default Stream, called <i>stream 0</i> .
Operations pushed in a non-default stream will be executed after all operations on default stream are emptied.
Operations assigned to default stream introduce implicit synchronization barriers among other streams.

CUDA Streams API

Create a stream	<code>cudaStreamCreate(&stream);</code>
Deallocate a stream	<code>cudaStreamDestroy(stream);</code>

CUDA Streams API (cont)

Block host until all operations on a stream are completed.	<code>cudaStreamSynchronize(stream);</code>
We can use stream to obtain the concurrent execution of the same kernel or different kernels.	

Synchronization operations

Explicit Synchronization	Implicit Synchronization
<i>cudaDeviceSynchronize()</i> blocks host code until all operations on device are completed	Operations assigned to default stream
<i>cudaStreamWaitEvent(stream, event)</i> blocks all operations assigned to a stream until event is reached.	Memory Allocations on device
	Settings operations on device
	Page-locked memory allocations



CUDA API

<https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>

Memory Workflow

First we allocate and "build" the input on the **host**.

Then we allocate dynamic memory on the **device**, obtaining pointers to the allocated memory areas.

Finally, we **initialize** the memory on the device and we **copy** the memory from the host to the device.

At the end of the computation, we may want to copy the memory from the device to the host.

Copy operation is *blocking*.

Memory Allocation API Functions

Dynamic memory allocation `cudaMalloc ((void **) &udev, N*sizeof( double ));`

Memory Initialization on device `cudaMemset (void *devPtr, int val, size_t count);`

Memory Allocation API Functions (cont)

Copying data from host to device `cudaMemcpy (void dst, void src, size_t size, cudaMemcpyDirection);`

Copying data from device to host `cudaMemcpy (void dst, void src, size_t size, cudaMemcpyDirection);`

After 4.0, CUDA supports **Unified Virtual Addressing** meaning that the system itself knows where the buffer is allocated. The *direction* parameter must be set to **cudaMemcpyDefault**.

Global Memory

Declaring a static variable `__device__ type variable_name;`

Declaring a dynamic variable `cudaMalloc ((void **) &ptr, size);`

Deallocating a dynamic variable `cudaFree (ptr)`

with the constant byte value *val*.

Global Memory (cont)

Allocating an aligned 2D buffer where elements are padded so that each row is aligned

```
cudaMallocPitch(&ptr, &pitch, width * sizeof(float), height);
```

cudaMallocPitch returns an integer pitch that can be used to access row element with stride access. For example:

```
float *row = devPtr + r * pitch;
```

Shared Memory

Static variable declaration inside the kernel.

```
__shared__ type shmem[SIZE];
```

Dynamic variable allocation outside the kernel

```
extern __shared__ type *shmem;
```

Constant memory

Declaring a static variable

```
__constant__ type variable_name;
```

Copy memory from host to device.

```
cudaMemcpyToSymbol(symbol_name, host_ptr, sizeof(type), 0, cudaMemcpyHostToDevice);
```

We cannot declare a dynamic variable on the constant memory

Texture Memory

Managing texture memory

Allocate global memory on device

```
cudaMalloc(&M, memsize);
```

Create a texture reference.

```
texture<data_type, dim> MtextureRef;
```

Create a channel descriptor

```
cudaChannelFormatDesc Mdesc = cudaCreateChannelFormatDesc(data_type);
```

Bind the texture reference to memory.

```
cudaBindTexture(0, MtextureRef, M, Mdesc);
```

Unbind at the end.

```
cudaUnbindTexture(MtextureRef);
```

In order to access the texture memory, we can use the texture reference

```
texture1D fetch(MtextureRef, address);
```

Accessing 2D cuda array.

```
texture2D fetch(MtextureRef, address);
```

Accessing 3D cuda array.

```
texture3D fetch(MtextureRef, address);
```



Asynchronous Data Transfers

Allocates `cudaMallocHost(buffer, size)`
page-locked memory on the host.

Frees `cudaFreeHost(buffer)`
page-locked memory.

Registers `cudaHostRegister()`
an existing host memory range for use by CUDA.

Unregisters `cudaHostUnregister()`
a memory range that was registered with `cudaHostRegister`.

Copies data `cudaMemcpyAsync(dest_buffer, src_buffer, dest_size, src_size, direction, stream)`
between host and device.

These operations must be queued into a non-default stream.

Page-locked Memory

Pageable memory is memory which is allowed to be paged in or paged out whereas **page-locked memory** is memory not allowed to be paged in or paged out.

Page out is moving data from RAM to HDD, while *page in* means moving data from HDD to RAM. These operations occurs when the main memory does not have enough free space.

Source: <https://leimao.github.io/blog/Page-Locked-Host-Memory--Data-Transfer/>

Error Handling

All CUDA API functions returns an error code of type `cudaError`.

The constant `cudaSuccess` means no error.

`cudaGetLastError` return the status of the internal error variable.

Calling this function resets the internal error to `cudaSuccess`.

Macro for Error Handling

```
#define CUDA_CHECK(X) {\
    cudaError_t _m_cudaStat = X;\
    if(cudaSuccess != _m_cudaStat) {\
        fprintf(stderr, "\nCUDA_ERROR: %s in file %s\n",\
            cudaGetErrorString(_m_cudaStat), __FILE__,\
            __LINE__);\
        exit(1);\
    }\
    ...
    CUDA_CHECK(cudaMemcpy(d_buf, h_buf, buffSize,\
        cudaMemcpyHostToDevice));
```