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 Execution Configuration	`kernel Fun cti on< <<n um_ blocks, num_th rea ds> >>( params)`
num_blocks	The number of thread blocks along each dimension of the grid.
num_threads	The number of threads along each dimension of the thread block

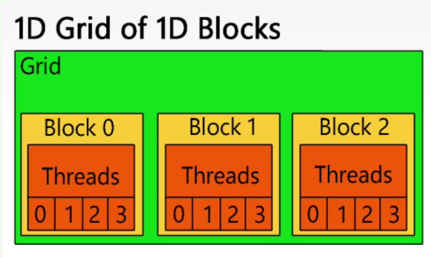
CUDA Thread Organization

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.

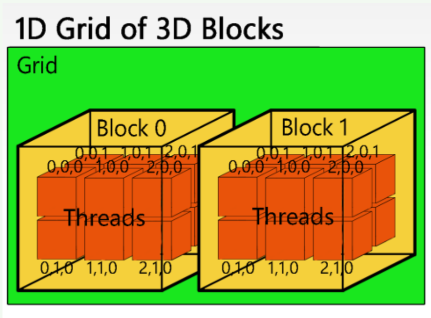
Blocks are executed independently.

1D Grid of 1D Blocks



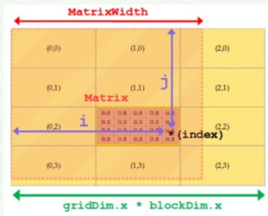

```
int index = blockIdx.x * blockDim.x + thread Idx.x ;
```

1D Grid of 3D Blocks




```
int index = blockIdx.x  blockDim.x  blockDim.y  blockDim.z + thread Idx.z  blockDim.y  blockDim.x + thread Idx.y  blockDim.x + thread Idx.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 + thread Idx.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 + thread Idx.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	`cudaEv ent_t event;`
Allocating the event	`cudaEv ent Cre ate (& event);`
Recording the Event.	`cudaEv ent Rec ord (ev ent);`
Synchronizing the event	`cudaEv ent Syn chr oni ze( event);`
Find elapsed time between two events	`cudaEv ent Ela pse dTi me( &e lapsed, a, b);`
Free event variables	`cudaEv ent Des tro y(e vent);`

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 *stream 0*.

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	`cudaSt rea mCr eat e(s tream1);`
Deallocate a stream	`cudaSt rea mDe str oy( stream)`

CUDA Streams API (cont)

Block host until all operations on a stream are completed.	`cudaSt rea mSy nch ron ize (st ream);`

We can use stream to obtain the concurrent execution of the same kernel or different kernels.

Synchronization operations

Explicit Synchronization	Implicit Synchronization
cudaDeviceSynchronize() blocks host code until all operations on device are completed	Operations assigned to default stream
cudaStreamWaitEvent(stream, event) 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*size of( dou ble ));`	*u_dev* is the pointer to the allocated variable
Memory Initialization on device	`cudaMe mse t(void *devPtr, int val, size_t count;`	*devPtr* is a pointer to the device address space. The function fills the first *count* bytes of the memory area with the constant byte value *val*.

Memory Allocation API Functions (cont)

Copying data from host to device	`cudaMe mCp y(void dst, void src, size_t size` ... `vice);`
Copying data from device to host	`cudaMe mCp y(void dst, void src, size_t size`

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

Global Memory

Declaring a static variable	`__device__ type variab le_ name;`
Declaring a dynamic variable	`cudaMa llo c((void **) &ptr, size );`
Deallocating a dynamic variable	`cudaFr ee(ptr)`

Global Memory (cont)

Allocating an aligned 2D buffer where elements are padded so that each row is aligned	`cudaMa llo cPi tch (&ptr, &p itch, width* siz eof (ty pe) , 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	`__cons tant__ type variab le_ name;`
Copy memory from host to device.	`cudaMe mcp yTo Sym bol (va ria ble _name, &h ost _src, sizeof (type), cudaMe mcp yH o stT oDe vice);`

We cannot declare a dynamic variable on the costant memory

Texture Memory

Managing texture memory	
Allocate global memory on device	`cudaMa llo c(&M, memsize)`
Create a texture reference.	`textur e<d ata type, dim> Mtextu reRef;`
Create a channel descriptor	`cudaCh ann elF orm atDesc Mdesc = cudaCr ea tat ype >();`
Bind the texture reference to memory.	`cudaBi ndT ext ure(0, Mtextu reRef, M, Mdes`
Unbind at the end.	`cudaUn bin dTe xtu re( MTe xtu reRef);`
In order to access the texture memory, we can use the texture reference, *Mtextu-reRef.*	`text1D fet ch( Mte xtu reRef, address);`
Accessing 2D cuda array.	`text2D fet ch( Mte xtu reRef, address);`
Accessing 3D cuda array.	`text3D fet ch( Mte xtu reRef, address);`

Asynchronous Data Transfers

Allocates page-locked memory on the host.	`cudaMa llo cHo st( buffer, size)`
Frees page-locked memory.	`cudaFr eeH ost (bu ffer)`
Registers an existing host memory range for use by CUDA.	`cudaHo stR egi ster()`
Unregisters a memory range that was registered with cudaHostRegister.	`cudaHo stU nre gis ter()`
Copies data between host and device.	`cudaMe mcp yAs ync (de st_ buffer, src_bu ffer, dest_size, src_size, direct ion ,st ream)`

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) {\
cudaEr ror_t _m_cud aStat = X;\
if(cud aSu ccess != _m_cud aStat) {\
fprint f(s tde rr, " \nC UDA _ERROR: %s in file %s
line %d\n",\
cudaGe tEr ror Str ing (_m _cu daS tat), __FILE__,
__LINE __);\
exit(1);\
} }
...
CUDA_C HECK( cudaMe mcp y(d _buf, h_buf, buffSize,
cudaMe mcp yHo stT oDe vice) );
```