Kernels

Kernel launch, 1D-2D index space, Examples

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU Education Center of Óbuda University

KERNELS

CUDA language extensions

Write a kernel

Launch a kernel

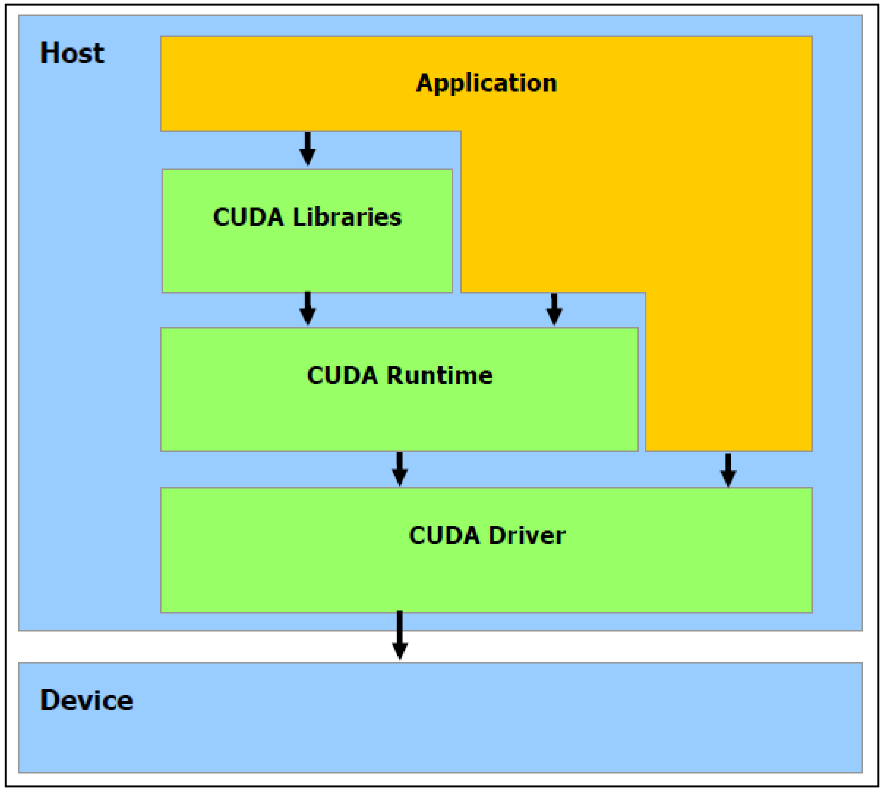
Synchronization

CUDA software stack

The CUDA software stack

- composed of the following layers
  - device driver
  - application programming interface (API) and it's runtime
  - additional libraries (two higher-level mathematical libraries of common usage)
- Programmers can reach all the three levels depending on simplicity/efficiency requirements
- It's not recommend to use only one of these levels in one component
- In these lessons we will always use the "CUDA Runtime" level. In this level we can utilize the features of the GPU (writing and executing kernels etc.) and the programming is quite simple

[i20]



Runtime libraries

Common component

- Built-in types
- Built-in variables
- New function/variable qualifiers

Host component

- Device handling functions
- Context handling functions
- Memory handling functions
- Program module handling functions
- Kernel handling functions

Device component

- Mathematical functions
- Synchronization functions
- Atomic functions

Common component – new variable types

Built-in vector types

- New built-in types for vectors:
  - char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4
  - short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4
  - int1, uint1, int2, uint2, int3, uint3, int4, uint4
  - long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4
  - float1, float2, float3, float4, double2
- For example int4, means a vector of 4 integers
- The components of the vectors are accessible via the x, y, z, w fields (according to the dimension of the vector)
- All of these vectors have constructor function named make_type. For example:
  int2 make_int2(int x, int y)

dim3 type

- The dim3 type is an integer vector type based on uint3 that is used to specify dimensions
- When defining a variable of type dim3, any component left unspecified is initialized to 1

Built-in types related to kernel calls

dim3 type

- In case of kernel start the size of the grid and the size of the blocks are stored in a dim3 variable. In case of the grid this is a 1 or 2 dimensional, in case of blocks this is a 1, 2 or 3 dimensional vector
- Example for usage of dim3 variables:

```
dim3 meret;
meret = 10;
meret = dim3(10, 20);
meret = dim3(10, 20, 30);
```

size_t type

- The size_t type is used to store memory sizes
- It is an unsigned integer

cudaStream_t type

- The cudaStream_t type identifies a stream
- In practice an unsigned integer value
- We will discuss streams later

Error handling in the host side

Result of cuda runtime functions

- All runtime functions return an error code (except the asynchronous functions). The result type of these is cudaError_t
- For example:
  *cudaError_t cudaEventCreate(cudaEvent_t *event)*
- It is highly recommend to check the result of these functions to avoid hidden critical errors

Error handling

- In case of asynchronous function calls, the method described above will not work. In these cases, we can read the error code from the synchronization function (described later), or one of the following global functions
- cudaError_t cudaGetLastError()
  Result is the error code of the last command
- cudaError_t cudaPeekAtLastError()
  Result is the error code of the last command. It does not reset the error variable to success
- const char* cudaGetErrorString(cudaError_t error)
  Result is the detailed description of an error code

Kernels

CUDA language extensions

Write a kernel

Launch a kernel

Synchronization

Kernel qualifiers

Kernel differences

- There are some special keywords
- There are some special available variables in the function's body (the previously mentioned threadIdx etc.)
- Directly not callable from the host code, there is a special kernel invocation syntax

CUDA keywords to sign functions

- __device__
  - Executed in: device
  - Callable from: device
- __global__
  - Executed in: device
  - Callable from: host
- __host__
  - Executed in: host
  - Callable from: host

Built-in variables available from kernels

gridDim

- Type: dim3
- Contains the dimensions of the grid

blockIdx

- Type : uint3
- Contains the block index within the grid

blockDim

- Type : dim3
- Contains the dimensions of the block

threadIdx

- Type : uint3
- Contains the thread index within the block

warpSize

- Type : int
- Contains the warp size in threads

Kernel implementation

Kernel example

- The following example shows a simple kernel implementation (multiply all values in the vector by 2):

```
__global__ void vectorMul(float* A)
{
      int i = threadIdx.x;
      A[i] = A[i] * 2;
}
```

- The __global__ keyword signs that the device will execute the function
- In the case of device functions, there must be not any result values
- The name of the kernel is *vectorMul*
- The function has one parameter: the address of the vector

Notes

- As it is clearly visible, the kernel don't have any information about the execution parameters (how many threads, how many blocks etc.)
- As discussed before, the kernel can use the threadIdx variable to determine which vector element to multiply

Available functions in kernels

Mathematical functions

- CUDA supports most of the C/C++ standard library mathematical functions. When executed in host code, a given function uses the C runtime implementation if available
    - basic arithmetic
    - sin/cos etc.
    - log, sqrt etc.

Time functions

- The clock() function should measure the runtime of the kernels. The signature of this function:
  clock_t clock()
- The return value is the actual value of a continuously incrementing counter (based on the clock frequency)
- Provides a measure for each thread of the number of clock cycles taken by the device to completely execute the thread, but not of the number of clock cycles the device actually spent executing thread instructions

Device component - functions

Fast mathematical functions

- For some of the functions, a less accurate, but faster version exists in the device runtime component
- It has the same name prefixed with __, like:
  __sinf, __cosf, __tanf, __sincosf, __logf, __log2f, __log10f, __expf, __exp10f, __powf
- The common C functions are also available, but it is recommended to use the functions above
  - Faster, based on the hardware units
  - Less accurate

Formatted output (≥CC2.0)

- int printf(const char *format[, arg, ...])
  prints formatted output from a kernel to a host-side output stream
- behaves in a similar way to the standard C-library *printf()* function
- executed as any other device-side function: per-thread, and in the context of the calling thread
- the output buffer for *printf()* is set to a fixed size before kernel launch
- get and set the printf buffer size
  - cudaDeviceGetLimit(size_t* size,cudaLimitPrintfFifoSize)
  - cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)

```
__global__ static void VectorShift(float *devA) {
        float tmp = devA[threadIdx.x];
        __syncthreads();
        printf("%f %f\n", threadIdx.x, tmp);
        devA[threadIdx.x+1] = tmp;
}
```

KERNELS

CUDA language extensions

Write a kernel

Launch a kernel

Synchronization

Kernel concept

Kernel in CUDA C

- A kernel is a C function, that, when called, are executed N times in parallel by N different CUDA threads
- Syntactically a kernel looks like a common C function (signed with __global__ qualifier), but it has several limitations
  - Cannot use some operators and functions (I/O routines, etc.)
  - The CPU process cannot call it directly

```
__global__ void kernelSample(float* A, int b)
{
        …
}
```

Kernel launch

- We do not need to start the CUDA threads one-by-one, we have to specify the name of the kernel and the number of necessary threads
- We can pass parameters to kernels, but all thread will give the same parameters

```
kernelSample<<<1, 200>>>(A, b);
```

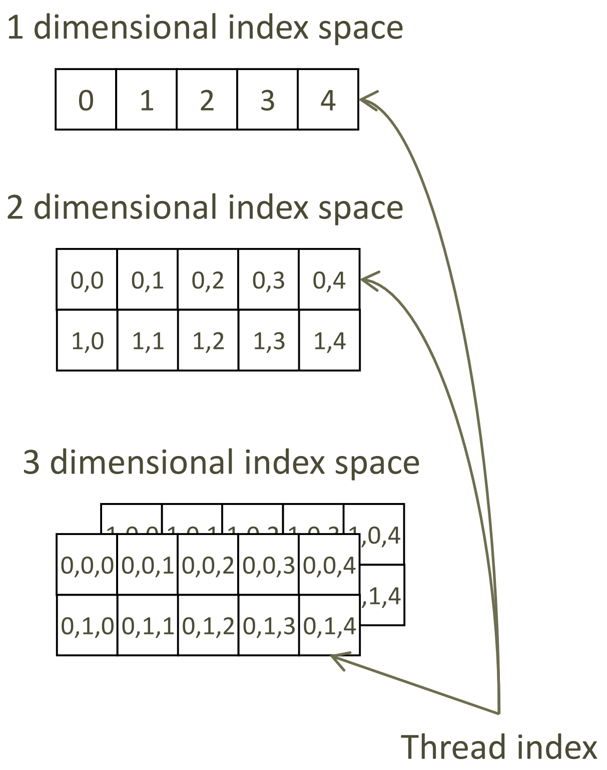
CUDA execution model - threads

Thread block

- Threads are organized into blocks
- One block can be
  - 1 dimensional
  - 2 dimensional
  - 3 dimensional

Thread index

- Each thread has a unique ID. So each thread can decide what data to work on
- Thread ID is available in the kernel via threadIdx variable
- In case of multidimensional index space, the *threadIdx* is a structure with the following fields:
  - *threadIdx.x*
  - *threadIdx.y*
  - *threadIdx.z*



Kernel invocation

Kernel invocation using 1 block

- If the size of the vector is not greater than the number of maximum threads, on block is enough to process the entire data space
- We use 1x1 grid size (first parameter)
- We use 200x1 block size (second parameter)

```
float*A = ...

... transfer data ...

vectorMul<<<1, 200>>>(A);

... Transfer results ...
```

Kernel execution

- With these execution parameters the device will create one block and 200 threads
- The local identifiers of the threads will be one dimensional numbers from 0 to 199
- The identifier of the block will be 0
- The block size will be 200

Full format of kernel calls

Any host function can call a kernel using the following syntax:

Kernel_name<<<Dg, Db, Ns, S>>>(parameters)

- where:
  - **Dg – grid size**
    A dim3 structure, that contains the size of the grid
    Dg.x * Dg.y = number of blocks
  - **Db – block size**
    A dim3 structure, that contains the size of the blocks
    Dg.x * Dg.y * Dg.z = number of thread within a single block
  - **Ns – size of the shared memory (optional parameter)**
    A size_t variable, that contains the size of the allocated shared memory for each blocks
  - **S – stream (optional parameter)**
    A cudaStream_t variable, that contains the stream associated to the command
- We will discuss Ns and S parameters later

Dynamic parallelism

Kernel can launch other kernel

- It is possible to launch kernels from kernel code (≥CC3.0)
- Programmer can use kernel launch <<< >>> in any kernel
- Launch is per-thread
- __synctreads() includes all launches by any thread in the block
- Example

```
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
        __syncthreads();
        if(tid == 0) {
            launch<<< 128, 256 >>>(buf);
            cudaDeviceSynchronize();
        }
        __syncthreads();
        cudaMemcpyAsync(data, buf, 1024);
        cudaDeviceSynchronize();
}
```

KERNELS

CUDA language extensions

Write a kernel

Launch a kernel

Synchronization

Synchronization – kernel level

Kernel (block) level synchronization

- void __syncthreads()
  - effect: synchronizes all threads in a block. Once all threads have reached this point, execution resumes normally
  - scope: threads in a single block

```
__global__ static void VectorShift(float *devA) {
      float tmp = devA[threadIdx.x];
      __syncthreads();
      devA[threadIdx.x+1] = tmp;
}
```

- This is a classical barrier synchronization
- *__syncthreads()* is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects
- It has no effect on threads in different blocks