Error Management in CUDA

CUDA provides a way to handle errors that involve **exceeded GPU capacities** or **GPU malfunctioning**: the hardest errors to find out. These are not logic nor syntax errors.

- `cudaError_t` is a CUDA type given to handle errors, that is really an integer, and this number gives us a hint about the possible error. Every CUDA function returns an error that can be stored in this type. The only CUDA function that doesnt return a `cudaError_t` variable is a kernel itself: it must return void.

```
cudaError_t error;
error = cudaMalloc((void**)&ptr, size);
```

Types of Errors (Most Common)

- `cudaSuccess = 0`: The API call returned with no errors. In query calls, this also means the query is complete. Successful execution.
- `cudaErrorInvalidValue = 1`: This indicates that one or more parameters passed to the API function call is not within an acceptable range of values. An enum param in a CUDA function that you didnt match, or a different data type sent.
- `cudaErrorMemoryAllocation = 2`: the API call failed because it was unable to allocate enough memory to perform the requested operation. When you do not have/allocate enough space in kernel memory for a requested instruction: you would normally write on memory outside of your array, but if there is no more mem left, this happens.
- `cudaErrorInvalidConfiguration = 9`: This indicates that a kernel launch is requesting resources that can never be satisfied with the current device. Requesting too many shared memory per block than supported, as well as requesting too many threads or blocks. This happens when you have an invalid kernel config (grid/blocks): when you exceed the max num of blocks per grid or threads of the GPU card.
- `cudaErrorInvalidMemcpyDirection = 21`: the direction of the memcpy passed to API call is not one of the types specified by cudaMemcpyKind. You put another word, basically.

Process

```
cudaError_t error;
error = cudaMalloc((void**)&ptr, size);
cudaGetErrorString(error);

// after a kernel launch
```

```
error = cudaGetLastError();
cudaGetErrorString(error);
```

- `error` would be an intger, but in order to avoid checking in the docs, CUDA provides the function `cudaGetErrorString(error)` that, given an integer, it returns a string of the error details/explanation.
- To get a kernel error (otherwise a kernel is just void return), we can catch the last integer of error with `cudaGetLastError();`.

Open any project an type this funtion that will be used after any CUDA function call:

```
// host because every function call must be from host
__host__ void checkCUDAError(const char* msg){
    cudaError_t error;
    cudaDeviceSynchronize(); // avoid catching another error
       that is not the next we want
    error = cudaGetLastError(); // status of the last CUDA API
        call (maybe 0 or success, not error)
    if (error != cudaSuccess){
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}
```

Because the host/device execution is asynchronous (both at the same time), we need to take care of the sequence and sometimes you need to **pause** and wait for the kernel to finish in order to come back to the host: we need to synchronize.

Example

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h> /* srand, rand */
#include <time.h> /* time */

#include<iostream>
using namespace std;

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}
```

```

__global__ void idKernel(int* vecA, int* vecB, int* vecC) {
    int gId = threadIdx.x + blockDim.x * blockIdx.x;

    vecA[gId] = threadIdx.x;
    vecB[gId] = blockIdx.x;
    vecC[gId] = gId;
}

void printArray(int* arr, int size, char* msg) {
    cout << msg << ": ";
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main()
{
    const int vectorSize = 64;
    int* host_a = (int*)malloc(sizeof(int) * vectorSize);
    int* host_b = (int*)malloc(sizeof(int) * vectorSize);
    int* host_c = (int*)malloc(sizeof(int) * vectorSize);

    int* dev_a, * dev_b, * dev_c;

    cudaMalloc((void**)&dev_a, sizeof(int) * vectorSize);
    checkCUDAError("Error at cudaMalloc for dev_a");
    cudaMalloc((void**)&dev_b, sizeof(int) * vectorSize);
    checkCUDAError("Error at cudaMalloc for dev_b");
    cudaMalloc((void**)&dev_c, sizeof(int) * vectorSize);
    checkCUDAError("Error at cudaMalloc for dev_c");

    srand(time(NULL));

    for (int i = 0; i < vectorSize; i++) {
        host_a[i] = 0;
        host_b[i] = 0;
        host_c[i] = 0;
    }

    cudaMemcpy(dev_a, host_a, sizeof(int) * vectorSize,
       cudaMemcpyHostToDevice);
    checkCUDAError("Error at cudaMemcpy for host_a to dev_a");
    cudaMemcpy(dev_b, host_b, sizeof(int) * vectorSize,
       cudaMemcpyHostToDevice);
    checkCUDAError("Error at cudaMemcpy for host_b to dev_b");
    cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
       cudaMemcpyHostToDevice); // error 1
    checkCUDAError("Error at cudaMemcpy for host_c to dev_c");

    dim3 grid(1, 1, 1);
    dim3 block(2000, 1, 1); // max num is 1024, so here we
       will force an error
    idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
    checkCUDAError("Error at idKernel execution no. 1");
    cudaDeviceSynchronize(); // wait until kernel finishes and
        then come back to following code // not needed to
       check
    cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);
    //check also here
    cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);
    cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);

    printf("Execution 1: 1 block 64 threads \n");
    printArray(host_a, vectorSize, "threadIdx.x");
    printArray(host_b, vectorSize, "blockIdx.x");
    printArray(host_c, vectorSize, "globalId");

    grid.x = 64; // (64, 1, 1)
    block.x = 1; // (1, 1, 1)
    idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
    cudaDeviceSynchronize();
    cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);
    cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);
    cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);

    printf("\nExecution 2: 64 blocks 1 thread \n");
    printArray(host_a, vectorSize, "threadIdx.x");
    printArray(host_b, vectorSize, "blockIdx.x");
    printArray(host_c, vectorSize, "globalId");

    grid.x = 4;
    block.x = 16;
    idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
    cudaDeviceSynchronize();
    cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);
    cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);
    cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
       cudaMemcpyDeviceToHost);

    printf("\nExecution 3: 4 block 16 threads \n");
    printArray(host_a, vectorSize, "threadIdx.x");
    printArray(host_b, vectorSize, "blockIdx.x");
    printArray(host_c, vectorSize, "globalId");

    free(host_a);
    free(host_b);
    free(host_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    return 0;
}
```

Output

```
ERROR 9: invalid configuration argument (Error at idKernel execution no. 1)
Execution 1: 1 block 64 threads
threadIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
blockIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
globalId: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Execution 2: 64 blocks 1 thread
threadIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
blockIdx.x: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
globalId: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Execution 3: 4 block 16 threads
threadIdx.x: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
blockIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
globalId: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
```

ERROR 9: Kernel no. 1 is not really executed