CUDA: VectorAddition

January 20, 2021

This CUDA program uses many of the CUDA's typedef, struct and built-in variables to perform the vectorAddition

[ ]:
```
#include <stdio.h>

// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>

#include <helper_cuda.h>
/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    // block dimension(along X axis)     block Index along X-axis   thread Index␣
 ↪along X-axis
    int i = blockDim.x                  * blockIdx.x                + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

/**
 * Host main routine
 */
int
main(void)
{
    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;

    // Print the vector length to be used, and compute its size
```

```
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    printf("[Vector addition of %d elements]\n", numElements);

    // Allocate the host input vector A
    float *h_A = (float *)malloc(size);

    // Allocate the host input vector B
    float *h_B = (float *)malloc(size);

    // Allocate the host output vector C
    float *h_C = (float *)malloc(size);

    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL || h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // Allocate the device input vector A
    float *d_A = NULL;
    err = cudaMalloc((void **)&d_A, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Allocate the device input vector B
    float *d_B = NULL;
    err = cudaMalloc((void **)&d_B, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
```

```
    }

    // Allocate the device output vector C
    float *d_C = NULL;
    err = cudaMalloc((void **)&d_C, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy the host input vectors A and B in host memory to the device input␣
↪vectors in
    // device memory
    printf("Copy input data from the host memory to the CUDA device\n");
    err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector A from host to device (error code␣
↪%s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector B from host to device (error code␣
↪%s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
    printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid,
           threadsPerBlock);
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
    err = cudaGetLastError();

    if (err != cudaSuccess)
    {
```

```
        fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy the device result vector in device memory to the host result vector
    // in host memory.
    printf("Copy output data from the CUDA device to the host memory\n");
    err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector C from device to host (error code␣
↪%s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Verify that the result vector is correct
    for (int i = 0; i < numElements; ++i)
    {
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
        {
            fprintf(stderr, "Result verification failed at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    printf("Test PASSED\n");

    // Free device global memory
    err = cudaFree(d_A);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector A (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaFree(d_B);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector B (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
```

```
    }

    err = cudaFree(d_C);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector C (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    printf("Done\n");
    return 0;
}
```

Here device is the GPU, host is the CPU

cudaMemcpy() -> Used to copy data from host to device and back to host after the computation has been made

Syntax Returns -> cudaError_t
Parameters:
void * dst const
void * src size_t count
enum cudaMemcpyKind kind
dst - Destination memory address
src - Source memory address
count - Size in bytes to copy
kind - Type of transfer -> (either from host->device | device->host) ::cudaSuccess of type cudaError_t ranging from 0-5 where 0->success

cudaMalloc is simular to malloc function of c, the memory will be allocated in the device and not in the host
Returns -> cudaError_t
Parameters:
void ** devPtr
size_t size
devPtr - Pointer to allocated device memory
size - Requested allocation size in bytes ::cudaSuccess,

cudaFree is simular to free in c

The above programs runs 196 blockPerGrids and 256 threadsPerBlock