Pass Function Pointers to Kernels in CUDA Programming

Introduction

Ever since I started to learn to CUDA, my impression of CUDA kernels is that it is a very isolated piece of code in the program and has lots of different restrictions. Because of this, I used to write CUDA kernel functions that have code duplications and do similar jobs. Today let us take a look at how to use C++ templates and function pointers for CUDA kernels to reduce the code duplications.

It should be noted that to the best of my knowledge there is no similar tutorial on this. I experimented a lot and make the final program available to the public.

Tutorial

Code

The following is the code to compute the sum and the product of two values by passing different function pointers to the CUDA kernel. It also uses C++ template extensively. The code is also available on my GitHub Gist.

The key to passing function pointers to CUDA kernel is to use static pointers to device pointers followed by copying the pointers to the host side. Otherwise, I am sure you will get different kinds of weird errors.

```
1  #include <iostream>
2
3  // Since C++ 11
4  template<typename T>
5  using func_t = T (*) (T, T);
```

```
6
7   template <typename T>
8   __device__ T add_func (T x, T y)
9   {
10      return x + y;
11  }
12
13  template <typename T>
14  __device__ T mul_func (T x, T y)
15  {
16      return x * y;
17  }
18
19  // Required for functional pointer argument in kernel function
20  // Static pointers to device functions
21  template <typename T>
22  __device__ func_t<T> p_add_func = add_func<T>;
23  template <typename T>
24  __device__ func_t<T> p_mul_func = mul_func<T>;
25
26
27  template <typename T>
28  __global__ void kernel(func_t<T> op, T * d_x, T * d_y, T * result)
29  {
30      *result = (*op)(*d_x, *d_y);
31  }
32
33  template <typename T>
34  void test(T x, T y)
35  {
36      func_t<T> h_add_func;
37      func_t<T> h_mul_func;
38
39      T * d_x, * d_y;
40      cudaMalloc(&d_x, sizeof(T));
41      cudaMalloc(&d_y, sizeof(T));
42      cudaMemcpy(d_x, &x, sizeof(T), cudaMemcpyHostToDevice);
43      cudaMemcpy(d_y, &y, sizeof(T), cudaMemcpyHostToDevice);
44
45      T result;
46      T * d_result, * h_result;
47      cudaMalloc(&d_result, sizeof(T));
```

```
    h_result = &result;

    // Copy device function pointer to host side
    cudaMemcpyFromSymbol(&h_add_func, p_add_func<T>, sizeof(func_t<T>));
    cudaMemcpyFromSymbol(&h_mul_func, p_mul_func<T>, sizeof(func_t<T>));

    kernel<T><<<1,1>>>(h_add_func, d_x, d_y, d_result);
    cudaDeviceSynchronize();
    cudaMemcpy(h_result, d_result, sizeof(T), cudaMemcpyDeviceToHost);
    std::cout << "Sum: " << result << std::endl;

    kernel<T><<<1,1>>>(h_mul_func, d_x, d_y, d_result);
    cudaDeviceSynchronize();
    cudaMemcpy(h_result, d_result, sizeof(T), cudaMemcpyDeviceToHost);
    std::cout << "Product: " << result << std::endl;
}

int main()
{
    std::cout << "Test int for type int ..." << std::endl;
    test<int>(2.05, 10.00);

    std::cout << "Test float for type float ..." << std::endl;
    test<float>(2.05, 10.00);

    std::cout << "Test double for type double ..." << std::endl;
    test<double>(2.05, 10.00);
}
```

Compile

To compile the program, use `nvcc`.

```
$ nvcc main.cu -o main
```

Run

If the program compiles successfully, you should be able to see the following message when you run the program.

```
```

```
$ ./main
Test int for type int ...
Sum: 12
Product: 20
Test float for type float ...
Sum: 12.05
Product: 20.5
Test double for type double ...
Sum: 12.05
Product: 20.5
```

References