CUDA - Performance Considerations

In this chapter, we will understand the performance considerations of CUDA.

A poorly written CUDA program can perform much worse than intended. Consider the following piece of code −

```
for(int k=0; k<width; k++) {
   product_val += d_M[row*width+k] * d_N[k*width+col];
}
```

For every iteration of the loop, the global memory is accessed twice. That is, for two floating-point calculations, we access the global memory twice. One for fetching an element of d_M and one for fetching an element of d_N. Is it efficient? We know that accessing the global memory is terribly expensive – the processor is simply wasting that time. If we can reduce the number of memory fetches per iteration, then the performance will certainly go up.

The CGMA ratio of the above program is 1:1. CGMA stands for **'Compute to Global Memory Access'** ratio, and the higher it is, the better the kernel performs. Programmers should aim to increase this ratio as much as it possible. In the above kernel, there are two floating-point operations. One is MUL and the other is ADD.

YEAR	CPU (in GFLOPS)	GPU (in GFLOPS)
2008	0-100	0-100
2009	0-100	300
2010	0-100	500
2011	0-100	800
2012	0-300	1200
2013	0-400	1500(K20)
2014	500	1800(K40)
2015	500-600	3000 (K80)
2016	800	4000 (Pascal)
2017	1300	7000 (Volta)

Let the DRAM bandwidth be equal to 200G/s. Each single-precision floating-point value is 4B. Thus, in one second, no more than 50G single-precision floating-point values can be delivered. Since the CGMA

ratio is 1:1, we can say that the maximum floating-point operations that the kernel will execute in 1 second is 50 GFLOPS (Giga Floating-point Operations per second). The peak performance of the Nvidia Titan X is 10157 GFLOPS for single-precision after boost. Compared to that 50GFLOPS is a minuscule number, and it is obvious that the above code will not harness the full potential of the card. Consider the kernel below −

```
__global__ void addNumToEachElement(float* M) {
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   M[index] = M[index] + M[0];
}
```

The above kernel simply adds M[0] to each element of the array M. The number of global memory accesses for each thread is 3, while the total number of computations is 1 (ADD in the second instruction). The CGMA ratio is bad: ⅓. If we can eliminate the global memory access for M[0] for each thread, then the CGMA ratio will improve to 0.5. We will attain this by caching.

If the kernel does not have to fetch the value of M[0] from the global memory for every thread, then the CGMA ratio will increase. What actually happens is that the value of M[0] is cached aggressively by CUDA in the constant memory. The bandwidth is very high, and hence, fetching it from the constant memory is not a high-latency operation. Caching is used generously wherever possible by CUDA to improve performance.

Memory is often a bottleneck to achieving high performance in CUDA programs. No matter how fast the DRAM is, it cannot supply data at the rate at which the cores can consume it. It can be understood using the following analogy. Suppose that you are thirsty on a hot summer day, and someone offers you cold water, on the condition that you have to drink it using a straw. No matter how fast you try to suck the water in, only a specific quantity can enter your mouth per unit of time. In the case of GPUs, the cores are ‘thirsty’, and the straw is the actual memory bandwidth. It limits the performance here.