# Performance Analysis of CUDA Programs

February 12, 2018

## 1 Memory Coalescing

A memory transaction width of N refers to the fact that N consecutive elements can be retrieved in one memory transaction together. In most GPU architectures this transaction width is equal to either the warp size or half the warp size. Warps are basic units of thread scheduling for GPU architectures. The CUDA thread ids inside a warp are consecutive in nature. Global memory coalescing occurs when simultaneous memory accesses by CUDA threads inside a warp are also consecutive so that the total number of memory accesses can be combined into a single memory transaction. Consider the following two code snippets.

```
__global__
void mem_access(float* A){
        int tid = blockDim.x*blockIdx.x + threadIdx.x;
        A[tid]+=2;
}
```

```
__global__
void mem_access(float* A){
        int tid = blockDim.x*blockIdx.x + threadIdx.x;
        A[tid*2]+=2;
}
```

Let us assume we have a GPU architecture where the warp size is 16, the memory transaction width is 16. Let the size of the array be 2048 and the total number of CUDA threads launched is 1024. The total number of warps is therefore $1024/16 = 64$.
For the first code snippet, the array access expression is only $tid$, a single memory transaction is needed per warp to access all the required elements. Thus the total number of memory transactions for the corresponding program is the number of warps i.e. 64. After executing the code snippet the first 1024 elements of the array will get incremented. Note that the last 1024 elements are not touched by any thread.

For the second code snippet, the memory accesses are strided since the array expression is $tid * 2$. There exists a performance penalty in this case. Note that

every warp requires two memory transactions because the 16 threads executing in SIMT fashion require to access an overall memory block whose length is more than 16. For example, in warp1, we have threads with id "0,1,2,....,15" and they are accessing elements "A[0], A[2], A[4],... A[30]". Therefore, a minimum of two memory transactions are required by each warp for accessing the corresponding elements. This results in a total of $64*2 = 128$ memory transactions. As output of this code, every even index in A gets incremented by 2, while the odd indices are left untouched.

## 2 Branch Divergence

Branch divergence occurs in CUDA programs when the threads inside a warp encounter a branch instruction. Consider the following code snippet executing on a GPU with warp size as 16.

```
__global__
void divergence(float *M, float *N, float *P)
{
        int j=blockIdx.x*blockDim.x+threadIdx.x;
        if(j%2)
                M[j]+=2
        else
                M[j]*=2;

}
```

The conditional expression in the above code snippet is thread id dependent. The GPU is not capable of running both the `if` and `else` blocks at the same time and thus execution is serialized. This is simply because now we are having warps formed in such a way that all the threads inside a warp will not execute the same instruction. Assuming that each addition operation takes 2 clock cyles and each multiplication operation takes 4 clock cycles, the total number of cycles taken per warp is therefore $4 + 2 = 6$ cycles.
Now let us consider the following code segment.

```
__global__
void divergence(float *M, float *N, float *P)
{
        int j=blockIdx.x*blockDim.x+threadIdx.x;
        if((j/16)%2)
                M[j]+=2
        else
                M[j]*=2;

}
```

Note that in this code there will be no divergence !!!