CUDA Optimizations

Sathish Vadhiyar
Parallel Programming

SIMT Architecture and Warps

- Multi-processor creates, manages, schedules and executes threads in groups of 32 parallel threads called warps
- Threads in a warp start at the same program address
- They have separate instruction address counter and register state, and hence free to branch
- When a SM is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution

Warps and Warp Divergence

- A warp executes one common instruction at a time
- Hence max efficiency can be achieved when all threads in a warp agree on a common execution path
- If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path – branch divergence

Performance Optimization Strategies

- Maximize parallel execution to achieve maximum utilization
- Optimize memory usage to achieve maximum memory throughput
- Optimize instruction usage to achieve maximum instruction throughput

Maximize Parallel Execution

- Launch kernel with at least as many thread blocks as there are multiprocessors in the device
- The number of threads per block should be chosen as a multiple of warp size to avoid wasting computing resource with under-populated warps

Maximize Parallel Execution for Maximum Utilization

- At points when parallelism is broken and threads need to synchronize:
  - Use _syncthreads() if threads belong to the same block
  - Synchronize using global memory through separate kernel invocations
- Second case should be avoided as much as possible; computations that require inter-thread communication should be performed within a single thread block as much as possible

Maximize Memory Throughput

- Minimize data transfers between host and device
  - Move more data from host to device
  - Produce intermediate data on the device
  - Data can be left on the GPU between kernel calls to avoid data transfers
  - Batch many small host-device transfers into a single transfer
- Minimize data transfers between global (device) memory and device
  - Maximize usage of shared memory

Maximize Memory Throughput: Other performance Issues

- Memory alignment and coalescing : Make sure that all threads in a half warp access continuous portions of memory
  - (Refer to slides 30-34 of NVIDIA's Advanced CUDA slides)
- Shared memory divided into memory banks: Multiple simultaneous accesses to a bank can result in bank conflict
  - (Refer to slides 37-40 of NVIDIA's Advanced CUDA slides)

Maximize Instruction Throughput

- Minimize use of arithmetic instructions with low throughput
  - Trade precision for speed – e.g., use single-precision for double-precision if it does not affect the result much
- Minimize divergent warps
  - Hence avoid use of conditional statements that checks on threadID
  - e.g.: (instead of if(threadId >2), use if(threadId/warpSize) > 2)

Maximize Instruction Throughput: Reducing Synchronization

- _syncthreads() can impact performance
- A warp executes one common instruction at a time; Hence threads within a warp implicitly synchronize
- This can be used to omit _syncthreads() for better performance

Example with _syncthreads()

```
// myArray is an array of integers located in global or shared
// memory
__global__ void MyKernel(int* result) {
    int tid = threadIdx.x;
    ...
    int ref1 = myArray[tid];
    __syncthreads();
    myArray[tid + 1] = 2;
    __syncthreads();
    int ref2 = myArray[tid];
    result[tid] = ref1 * ref2;
    ...
}
```

Can be converted to….

Reducing Synchronization

```
// myArray is an array of integers located in global or shared
// memory
__global__ void MyKernel(int* result) {
    int tid = threadIdx.x;
    ...
    if (tid < warpSize) {
        int ref1 = myArray[tid];
        myArray[tid + 1] = 2;
        int ref2 = myArray[tid];
        result[tid] = ref1 * ref2;
    }
    ...
}
```

Threads are guaranteed to belong to the same warp. Hence no need for explit synchronization using _syncthreads()

Occupancy

- Occupancy depends on resource usage (register and shared memory usage)
- More the usage per thread, less number of threads can be simultaneously active
  - One cannot indiscriminately use registers in a thread - If (registers used per thread x thread block size) > N, the launch will fail; for Tesla, N = 16384
- Very less usage increases the cost of global memory access
- Maximizing occupancy can help cover latency during global memory loads

Occupancy Calculation

- Active threads per SM =
  - Active thread blocks per SM * ThreadsPerBlock
- Active warps per SM =
  - Active thread blocks per SM * my warps per block
- Active thread blocks per SM =
  - Min(warp-based-limit, registers-based-limit, shared-memory-based-limit)
- Occupancy = (active warps per SM) / (max warps per SM)