CME213: Parallel Computing using MPI, OpenMP and CUDA

Homework 2: CUDA Programming

Chi Zhang
Stanford University
SUID: 06116342
Spring 2018
Date: May 3, 2018

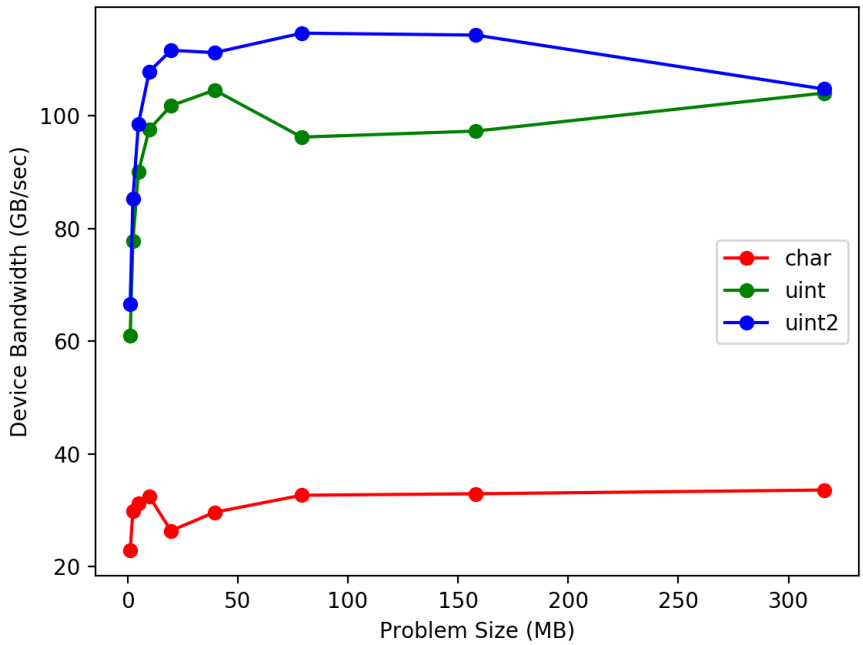
Problem 1: String Shift

1. see `shift.cu`
2. The table from the ouput is

OUTPUT

```
                    Device Bandwidth GB/sec
---------------------------------------------------------------------
                         char          uint         uint2
Problem Size MB
        1.23515       22.9548         60.977         66.549
         2.4703       29.8923        77.7802        85.3004
         4.9406       31.2033         89.973        98.4654
         9.8812       32.4783        97.5786        107.798
        19.7624       26.3943        101.751        111.576
        39.5248        29.669        104.514        111.165
        79.0496       32.6882        96.1842        114.612
        158.099       32.9524        97.2492        114.271
        316.198       33.6162        104.002        104.699
```

The plot generated from the table is shown as below



3. It should be pointed out first that the basic unit of threads in CUDA-enabled GPUs is a warp (32 threads) and data are read from cache lines residing in global memory.

   - For `char` kernel, each warp only touches `32 * sizeof(char) = 32` bytes at a time. However, a cache line (`128` bytes) has been fully loaded, which means `(128 - 32) = 96` bytes are wasted. This issue is resolved in `uint` kernel since each warp touches `32 * sizeof(uint) = 128` bytes (one full cache line) now.
   - The performance does not change much from `uint` to `uint2` kernels in that each warp in both cases is already performing coalesced accesses on global memory (`4` bytes per thread access, `128` bytes per warp access)

     `uint kernels: 32 * sizeof(uint) = 128 bytes ⇒ 1 full cache line`

     `uint2 kernels: 32 * sizeof(uint2) = 256 bytes ⇒ 2 full cache lines`

Problem 2: PageRank

1. see `pagerank.cu`
2. It is easier to analyze reads and writes following the code snippet here

---

```
float sum = 0.0f;
// iterate all edges
uint start = graph_indices[tid], end = graph_indices[tid+1];
for (uint i = start; i < end; ++i) {
        uint node = graph_edges[i];
        sum += 0.5f * inv_edges_per_node[node] * graph_nodes_in[node];
}
// add constant term
sum += 1.0f / (2 * num_nodes);
// update
graph_nodes_out[tid] = sum;
```

This is the update routine run by each thread/node, and the `for`-loop will iterate over all edges. Thus, the total bytes should be computed as

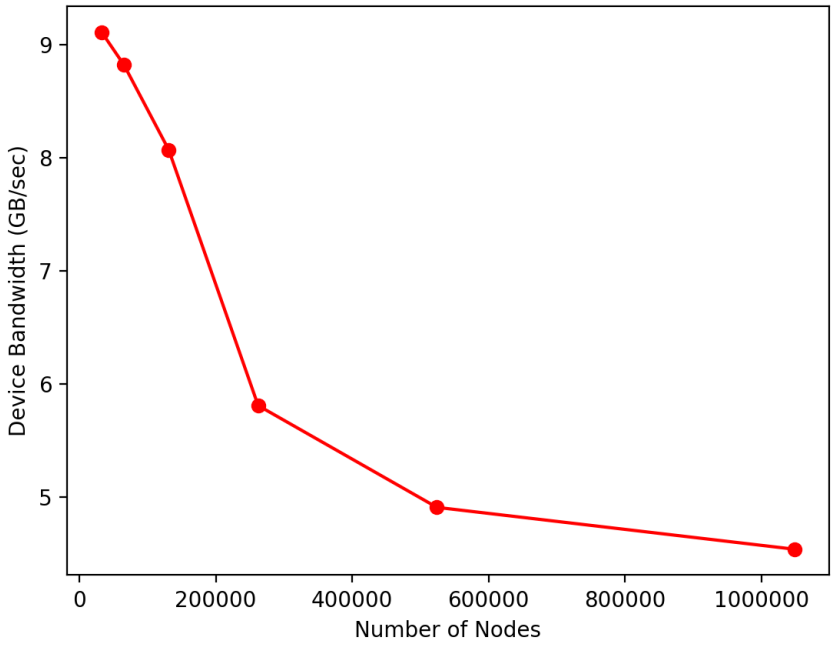
---

```
// 5 READs from memory
size_t index_read = 2 * (*nodes) * sizeof(uint);
size_t graph_edge_read = (*nodes) * (*edges) * sizeof(uint);
size_t graph_node_read = (*nodes) * (*edges) * sizeof(float);
size_t inv_edges_read = (*nodes) * (*edges) * sizeof(float);
size_t read_bytes = index_read + graph_edge_read + graph_node_read + inv_edges_read;
// 1 WRITE to memory
```

```
size_t write_bytes = (*nodes) * sizeof(float);
// total bytes READ and WRITE
size_t totalBytes = iterations * (read_bytes + write_bytes);
```

---

3. The plot is shown as below



4. The memory access pattern is *random access* in Problem 2. The pattern is reflected in operations such as `graph_edges[i]`. Due to random memory access, the bandwidth in Problem 2 is much lower than that in Problem 1. Also, randomness further hurts performance as the problem size increases since there is a higher probability that each thread in a warp might access different cache lines. In comparison, warps are accessing *continuous memory* in Problem 1 because array indices are sequential. As the problem size increases, warps are more likely to touch full cache lines and thus performance improves. To summarize, ensuring threads in a warp to access continuous memory which fits into exactly multiple cache lines will yield peek bandwidth.