COMP 3031 Cuda Presentation

By LEUNG, Hang Kam (20425828, hkleungai)

https://hkust.zoom.us/rec/play/hCgdDf-wsGqOQwsEke7Osa0PMtkr2aZjFNvHbN9eJ3TEtoFkilB71wMoerP0BwFllH2lKdjR5klcANGK.FzlhBCRN6e3uU37y?autoplay=true&startTime=1607406860000

Task Overview

Task Overview

- Given the following,
  - `data.txt` containing columns of edges, with each row storing two vertex ID.
  - `main.cu` for processing I/O and calling the to-do function.
  - `tnt_counting.cu` with a function `tnt_counting()` to be filled.
- The task is,
  - Give all `tnt`, i.e. c6rings with 3 Ns and 6 Os (2 on each N).
  - Ns are to be fixed on $0, 2, 4$ entries of a cycle.
  - For each found `tnt`, there are $(3 \cdot 2) \cdot 2^3 = 48$ outputs.

Solving Approach With Cuda Parallelism

A Big Picture Of The Solution

- Four kernel functions applied in turns to compute for the given data.
  - `__global__ void global_get_six_cycles()`
  - `__global__ void global_get_cycles_duals()`
  - `__global__ void global_get_cycles_with_N()`
  - `__global__ void global_get_cycles_with_N_O2()`
- In each step, `__device__` functions may be applied for collecting similar logics.
- In particular `void eliminate_false_positive()` is used for filtering.
- In final steps, linearly deep-copying from the device pointer to `final_results`.

Step 0 — Pointer Allocation

- Need to allocate memory manually on a *host function* `void tnt_counting()`.
- Here lazily compute an upper bound of memory needed for intermediate pointers.

```
size_t max_result_size = c_c_size * 60 * sizeof(int);
```

  - By design, `final_results` takes the largest space among all pointers.
  - The worst case is every edge of `c-c` belongs to some c6rings.
  - Then for 15 vertices on each of the 48 outputs on k possible `tnt`, `c_c_size` is bounded below by $12k$, thus giving $15 \cdot 48 \div 12 = 60$.

Step $(m + 0.5)$ – Filtering Invalid Data

- Driver – `void eliminate_false_positive(int *out, int& out_size, ...)`
- On step m, map device pointer entry (by `tid`) to the output for later steps. Sometimes some entries are found not useful on step m. Then
  - Map such entry to a an `ERROR` int.
  - Passing the result device pointer to a host pointer `*h`.
  - Apply `eliminate_false_positive` to `*h` to reduce `ERROR` rows.
  - Obtain a new host pointer and an appropriate size for further computation.

Step 1 — Getting (Partial) Six-Cycles

- Driver — `__global__ void global_get_six_cycles()`
  - It takes `c-c` and its size and give an output on the first argument.
- **Assumption** — Each edge on `c-c` belongs to 0 or 2 cycles.
- Perform ~~fake~~ DFS on each edge on `c-c` to get c6rings on `c-c`.
  - Lazy Implemntation — each edge is only visited once without backtracking.
- Map invalid entries to `ERROR` if it does not form c6rings with N.

Step 2 — Getting All Six-Cycles

- Driver — `__global__ void global_get_cycles_duals()`.
- It takes Step 1's result (`*in` and `in_size`) and gives output `*out`.
- **Assumption** — Each edge on `c-c` belongs to 0 or 2 cycles.
  Even so, each edge can still induce cycles on two orientations.
- On each `tid`,
  - Map the cycles on `in` to `out[tid + in_size * (0..5 * 2)]`.
  - Compute a *dual* `{ a,f,e,d,c,b }` for `in[tid] = { a,b,c,d,e,f }`.
    Map it to `out[tid + in_size * (0..5 * 2 + 1)]]`.

Step 3 — Map Six-Cycles With N

- Driver — `__global__ void global_get_cycles_with_N()`.
  - It takes Step 2's `*in`, `*in_size` and `c-n`, `n-o` and and give `*out`.
- On each `tid`,
  - Locate the `0,2,4` entries for each `in[tid]`.
  - Scan `c-n` to find three Ns
  - Map `in[tid]` to `{ ...in[tid], N_1, N_2, N_3 }`;
    Or map entries to `ERROR` if any of the above fails.

Step 4 — Map Six-C-Plus-Three-N To TNT

- Driver — `__global__ void global_get_cycles_with_N_O2()`
  - It takes Step 3's `*in`, `*in_size` and `n-o` and and give `*out`.
- On each `tid`,
  - Locate the $6, 7, 8$ entries of `in[tid]` — `N_1`, `N_2`, `N_3`.
  - For each `N_i`, scans `n-o` to get `O_i1`, `O_i2`.
  - Form `{ ...in[tid], O_11, O_12, O_21, O_22, O_31, O_32}`.
    Assign it to `out[tid + in_size * (0..14 * 8 + 0)]`.
    Assign next combination to `out[tid + in_size * (0..14 * 8 + 1)]`, etc.

Cuda Solution's Performance

Time Cost In 4 Runs

Elapsed (mine)	Driver (mine)		Elapsed (given)	Driver (given)
0.092111604 s	0.055027969 s		0.469755584 s	0.407845215 s
0.092697601 s	0.055197983 s		0.430198318 s	0.377064850 s
0.093199651 s	0.055419682 s		0.451046886 s	0.406798187 s
0.092034762 s	0.055406975 s		0.427651141 s	0.377056152 s

- Mine works faster (by ~4 times on elapsed time, by ~7 times on Driver Time).
- Could possibly be enhanced if there are more time to develop a better `filter()`.

Comparison with Prolog Solution

Main Comparison

- *Clumsy Side* — Cuda is a less friendly mapper than Prolog.
    - Prolog has `include` for filter, `maplist` for map, `findall` for map-find.
    - Cuda's thread on each array entry computes to some output.
        - Technically cuda has `atomicadd()` or we can down-sweep for `filter()`
        - But either no better than current one, or make the code more chaotic.
- *Good Side* — Cuda is less abstract than Prolog.
    - Prolog uses cut, tail recusion, and multiple relation clauses for control-flow.
    - Cuda has more commonly-seen control-flow features (loop, if-else, etc).

The End