GPU Join

Team Members

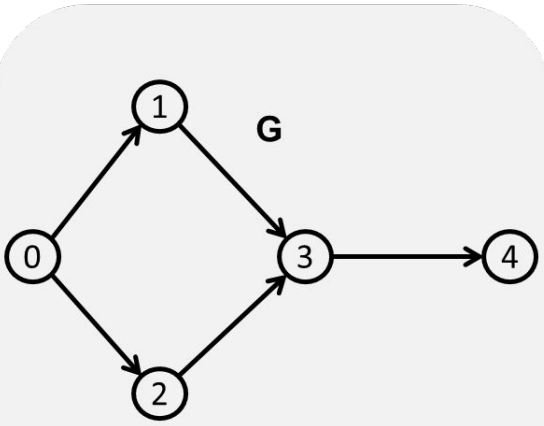
Ahmedur Rahman Shovon
Landon Dyken
Sidharth Kumar
(University of Alabama at Birmingham)

Mentors
Thomas Applencourt, ANL
Oded Green, NVIDIA

GPUJoin

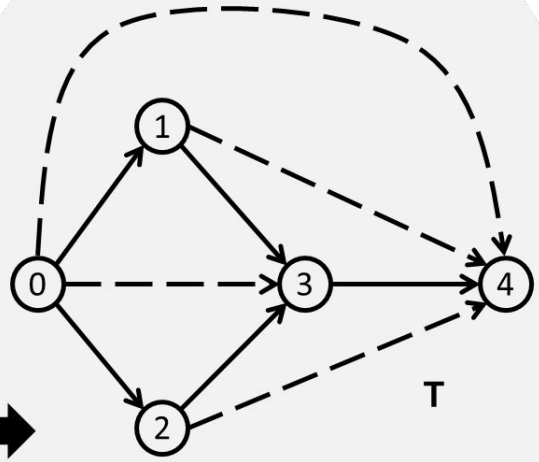
- Declarative logical inference at scale
- Scientific driver for the chosen algorithm
  - Graph mining
  - program analysis
  - deductive databases
- What's the algorithmic motif?
  - Implement relational algebra (RA) backend to support declarative analysis
- What parts are you focusing on?
  - Getting a prototype implementation of RA

Path finding: Logical Inference for Graphs



0	1
1	3
3	4
0	2
2	3

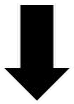
T (x, y) <- G (x, y)
T (x, z) <- T (x, y), G (y, z)



0	1
1	3
3	4
0	2
2	3
0	4
0	3
2	4
1	4

Pipeline of our work

Datalog



Relational Algebra

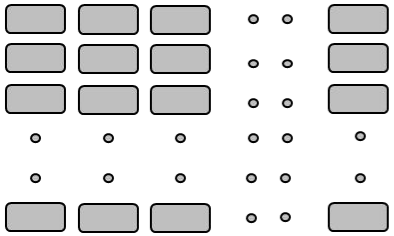
Datalog rule for computing transitive closure

```
T(x,y) <- G(x,y).
T(x,z) <- T(x,y), G(y,z).
```

Operationalized as a fixed-point iteration using F_G

$$F_G(T) \triangleq G \cup \Pi_{1,2}(\rho_{0/1}(T) \bowtie_1 G)$$

Compute resources



Evolution and Strategy

- What was your goal coming here?
  - Goal was too ambitious, but we were able to realign after suggestion from our mentors
  - We wanted to develop multi-GPU multi-node implementation of relational algebra backend
- What was your initial strategy?
  - To implement optimized join algorithms for single GPU in CUDA, then utilize this for MPI + CUDA in the distributed relational algebra system

Evolution and Strategy

- How did this strategy change?
  - We realized we were not knowledgeable enough of GPU programming yet to create our own very fast join implementations from scratch in CUDA
  - Our mentors set us up with tools/libraries which provided optimized abstractions (Rapids CuDF, HashGraph, CuCollections)
  - We finished making a python prototype of transitive closure that utilizes rapids for fast iterated GPU joins

Evolution and Strategy

- Where are you heading towards?
  - We were linked an NVIDIA course (Fundamentals of Accelerated Computing with CUDA C/C++) that we completed to improve our CUDA and GPU programming skills
  - Although rapids has good performance, we want to create our own CUDA backend for an MPI + CUDA prototype using CUDA aware MPI so that we can optimize by combining operations specific to our purpose
  - Before this though, want to create a dask-cudf baseline for multi-gpu joins

Results and Final Profile

- What were you able to accomplish?
- Created rapids TC prototype run on Theta
- Optimized CUDA nested loop join operation on Theta

- What did you learn?
  - Many GPU programming best practices i.e. preventing branch diversion, utilizing multiple compute passes to improve parallelism, and reducing page faults
  - Gained knowledge of many technologies for multi-node multi-gpu join implementation in the future (NVSHMEM, HashGraph and CuCollections for optimized CUDA backend, etc.)

Benchmarks on Theta GPU (NVIDIA A100 - 40536MiB)

Number of rows	CUDF time (s)	Pandas time (s)
100000	0.034978	0.294990
150000	0.023296	0.655962
200000	0.031487	1.152136
250000	0.036887	1.785033
300000	0.039247	2.559597
350000	0.057324	3.469027
400000	0.073785	4.536874
450000	0.088625	5.866922
500000	0.107790	7.016265
550000	0.125129	8.476868

Join using CUDF and Pandas

Number of rows	#Blocks	#Threads	#Result rows	Pass 1	Offset calculation	Pass 2	Total time
100000	98	1024	20000986	0.00728293	0.00155869	0.0287826	0.0376242
150000	147	1024	44995231	0.0200516	0.00146178	0.0721115	0.0936249
200000	196	1024	80002265	0.025782	0.00148748	0.105717	0.132986
250000	245	1024	125000004	0.0378728	0.00147928	0.149159	0.188511
300000	293	1024	179991734	0.045733	0.00149265	0.197326	0.244552
350000	342	1024	245006327	0.0704528	0.00152981	0.258077	0.330059
400000	391	1024	319977044	0.0807149	0.00183633	0.333223	0.415774
450000	440	1024	404982983	0.112455	0.00172126	0.395609	0.509785
500000	489	1024	499965209	0.125456	0.00176208	0.47409	0.601308
550000	538	1024	605010431	0.138507	0.00185872	0.554933	0.695299

Nested loop join (non atomic)

Number of rows	#Blocks	#Threads	#Result rows	Pass 1	Offset calculation	Pass 2	Total time
100000	3125 x 3125	32 x 32	20000986	0.0326351	3.676e-05	0.0912985	0.12397
150000	4688 x 4688	32 x 32	44995381	0.0668192	2.085e-05	0.175562	0.242402
200000	6250 x 6250	32 x 32	80002288	0.10098	2.2623e-05	0.311574	0.412577
250000	7813 x 7813	32 x 32	125000004	0.157802	2.129e-05	0.486814	0.644637
300000	9375 x 9375	32 x 32	179991734	0.232257	5.3171e-05	0.769122	1.00143
350000	10938 x 10938	32 x 32	245006327	0.307065	3.0778e-05	0.955728	1.26282
400000	12500 x 12500	32 x 32	319977044	0.402273	3.6008e-05	1.25115	1.65346
450000	14063 x 14063	32 x 32	404982983	0.508222	4.6889e-05	1.58239	2.09066
500000	15625 x 15625	32 x 32	499965209	0.626641	4.8672e-05	1.95498	2.58167
550000	17188 x 17188	32 x 32	605010431	0.757276	6.2097e-05	2.33137	3.08871

Nested loop join (atomic)

Benchmarks on Theta GPU (NVIDIA A100 - 40536MiB)

Number of rows	#Blocks	#Threads	#Result rows	Pass 1	Offset calculation	Pass 2	Total time
550000	8594	64	605010431	0.167814	0.00177992	0.554116	0.72371
550000	4297	128	605010431	0.181085	0.00168487	0.549306	0.732076
550000	2149	256	605010431	0.173532	0.00166867	0.545732	0.720933
550000	1075	512	605010431	0.177913	0.00177223	0.549719	0.729405
550000	538	1024	605010431	0.17582	0.00166037	0.561509	0.738989

Different number of threads and blocks configuration for nested loop join

Number of rows	TC size	Iterations	Time (s)
333	55611	333	1.546973
990	490545	990	11.516639
2990	4471545	2990	48.859073
4444	9876790	4444	98.355756
4990	12452545	4990	121.888416
6990	24433545	6990	263.082299
8990	40414545	8990	536.293174

Transitive closure calculation for string graph datasets

What problems you encountered

- Did not have a unified development environment
  - Some using Windows, linux, older gaming GPUs to run code
  - Difficult to get libraries to work in each environment
  - Started taking increased advantage of Theta interactive jobs!
- Creating working conda environment on Theta

Wishlist

- What do you wish existed to make your life easier?
  - Documentation on running Conda on theta
  - Improved documentation for technologies we used (CuDF->dask-cudf in particular)

Was it worth it?

- Was this worth it?
- YES
- **The DLI course with hands on was very helpful**
- We applied the error handling, prefetching, unified memory, and different configurations to get better performance
- Will you continue development?
  - We will develop CUDA aware MPI for our Datalog backend
  - We will implement multi GPU TC versions using Rapids dask-cudf