09/12/2023

Accelerating iterative relational algebra operations

oneAPI Hackathon: CUDA to SYCL Migration

UNIVERSITY OF ILLINOIS CHICAGO

Team Members

Ahmedur Rahman Shovon

Ph.D. Student (CS)
University of Illinois Chicago
Email: ashov@uic.edu

Thomas Gilray

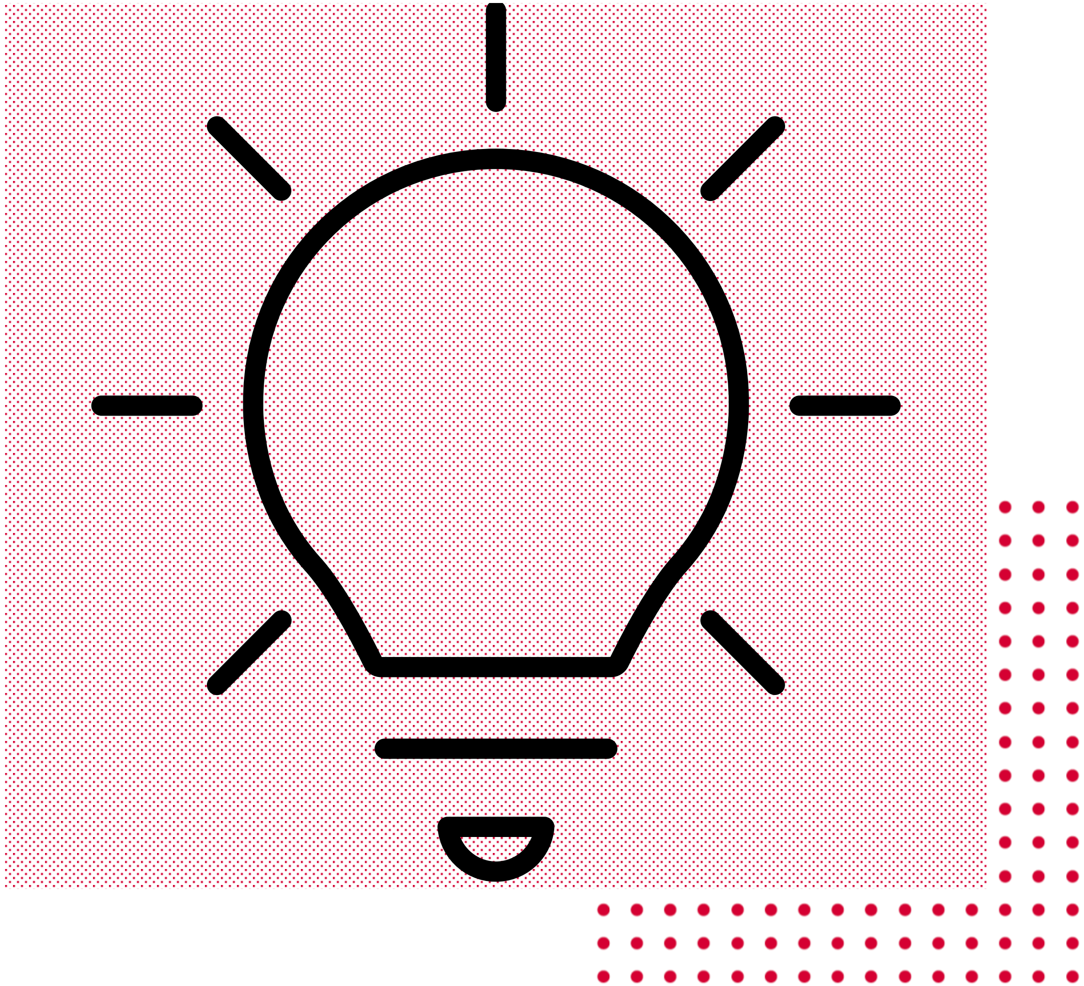
Assistant Professor (CS)
University of Alabama at Birmingham
Email: gilray@uab.edu

Sidharth Kumar

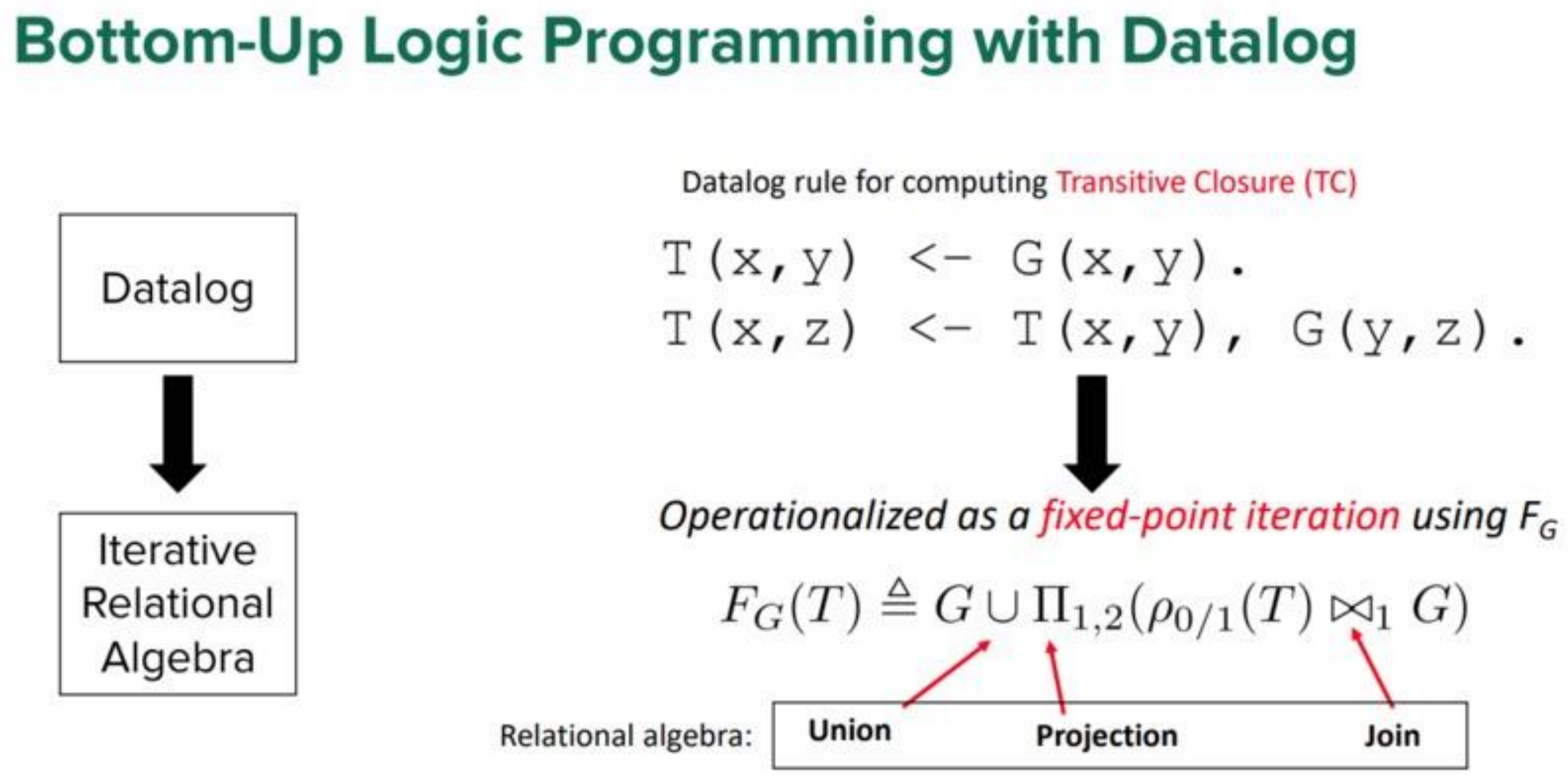
Assistant Professor (CS)
University of Illinois Chicago
Email: sidharth@uic.edu

Inspiration

- Iterative relational algebra (RA kernels in a fixed-point loop) enables bottom-up logic programming languages such as Datalog which can be implemented using relational algebra primitives (e.g., projections, reorderings, and joins)
- While much has explored standalone RA operations on the GPU, relatively less work focuses on iterative RA, which exposes new challenges (e.g., deduplication and memory management)



Transitive Closure Computation using Iterative Relational Algebra



Bottom-Up Logic Programming with Datalog

Datalog

Iterative Relational Algebra

Datalog rule for computing Transitive Closure (TC)

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

Operationalized as a fixed-point iteration using F_G

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

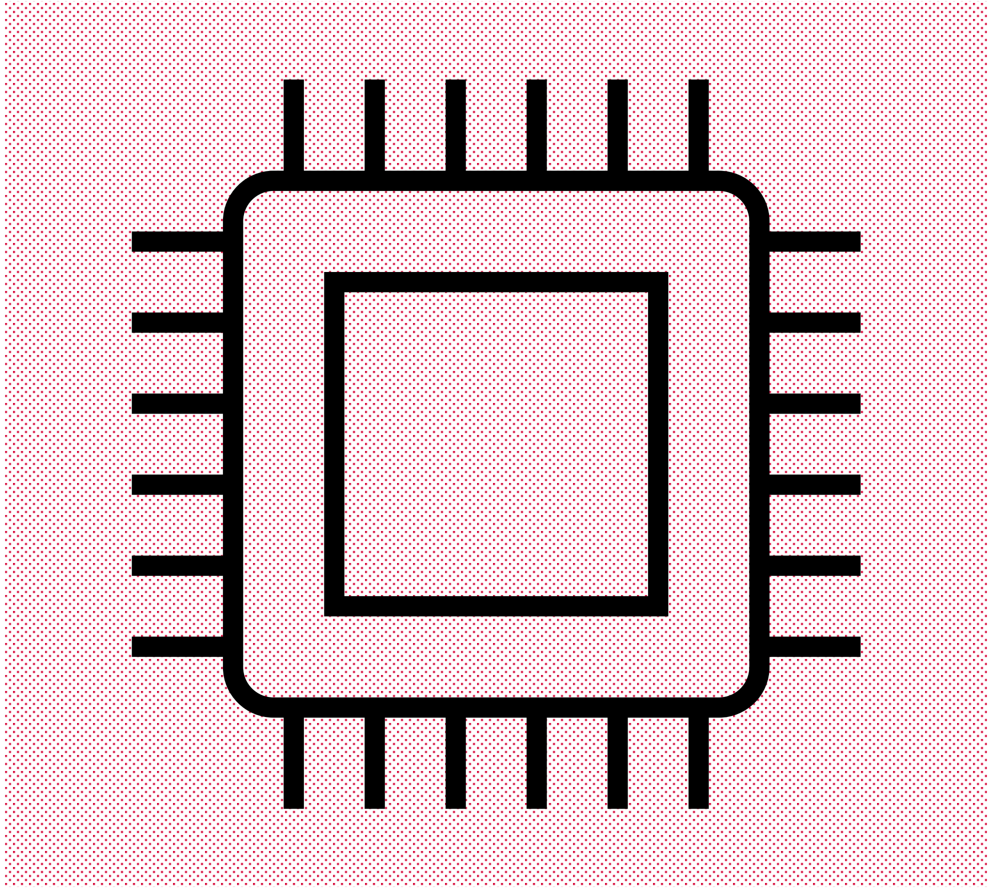
Relational algebra: **Union** **Projection** **Join**

What it does

- Developed a GPU-based hash-join implementation, leveraging
  - a novel open-addressing-based hash table implementation
  - operator fusing to optimize memory access
  - two variant implementations of deduplication
- Implemented transitive closure using our hash-join-based CUDA library and compared its performance against cuDF (GPU-based) and Souffle (CPU-based)

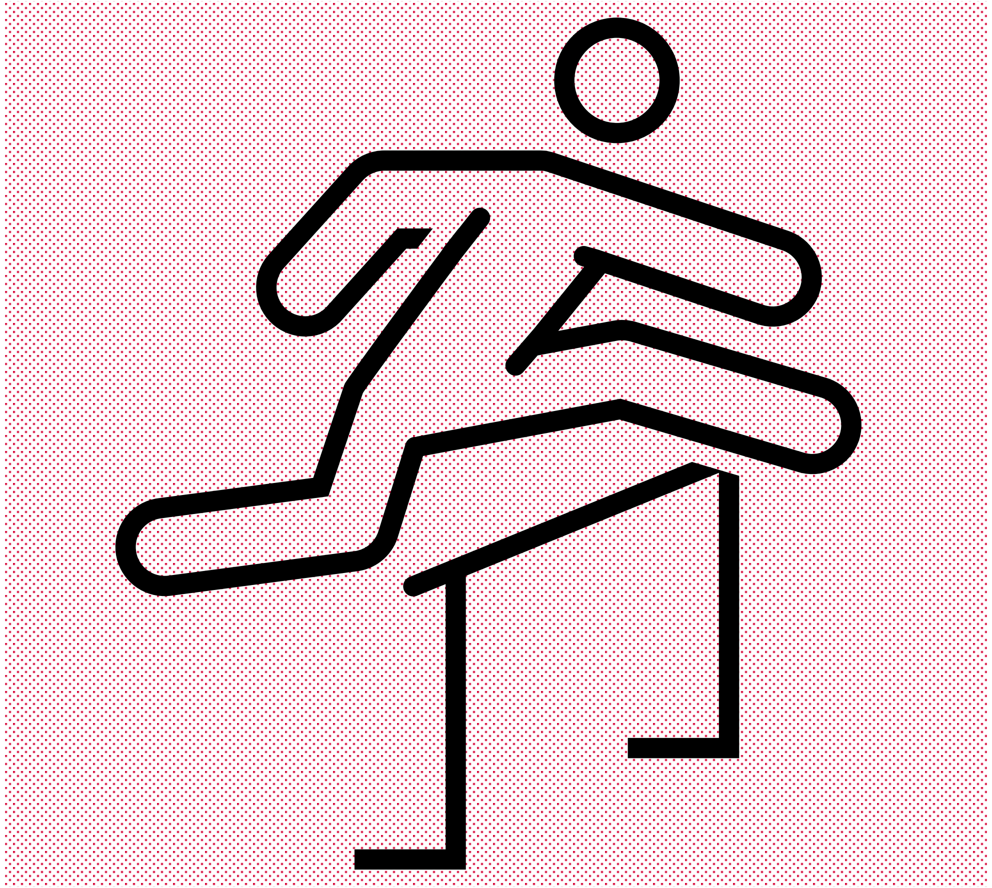
Environment and Datasets

- Benchmarked our experiments on the ThetaGPU supercomputer of Argonne National lab using a single nVidia A100 GPU
- CUDA kernel size: 3456 X 512
- CUDA version: 11.4
- Souffle version: 2.3 with 128 threads
- Datasets: Stanford large network dataset collection, SuiteSparse matrix collection, and road network real datasets collection



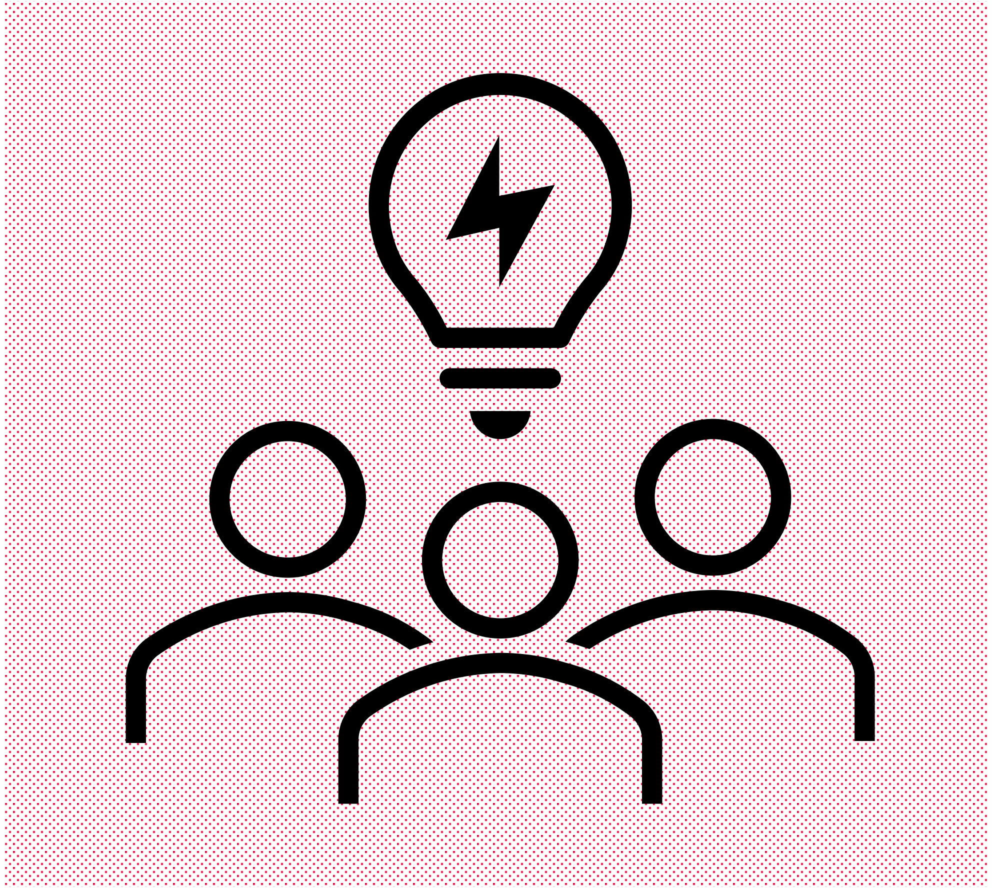
Challenges we ran into

- Unlike C++, CUDA lacks efficient data structures, limiting our implementation's capabilities
- The available VRAM of a single GPU imposes constraints on our implementation's scalability
- Debugging kernel code posed significant challenges, turning it into a nightmarish experience



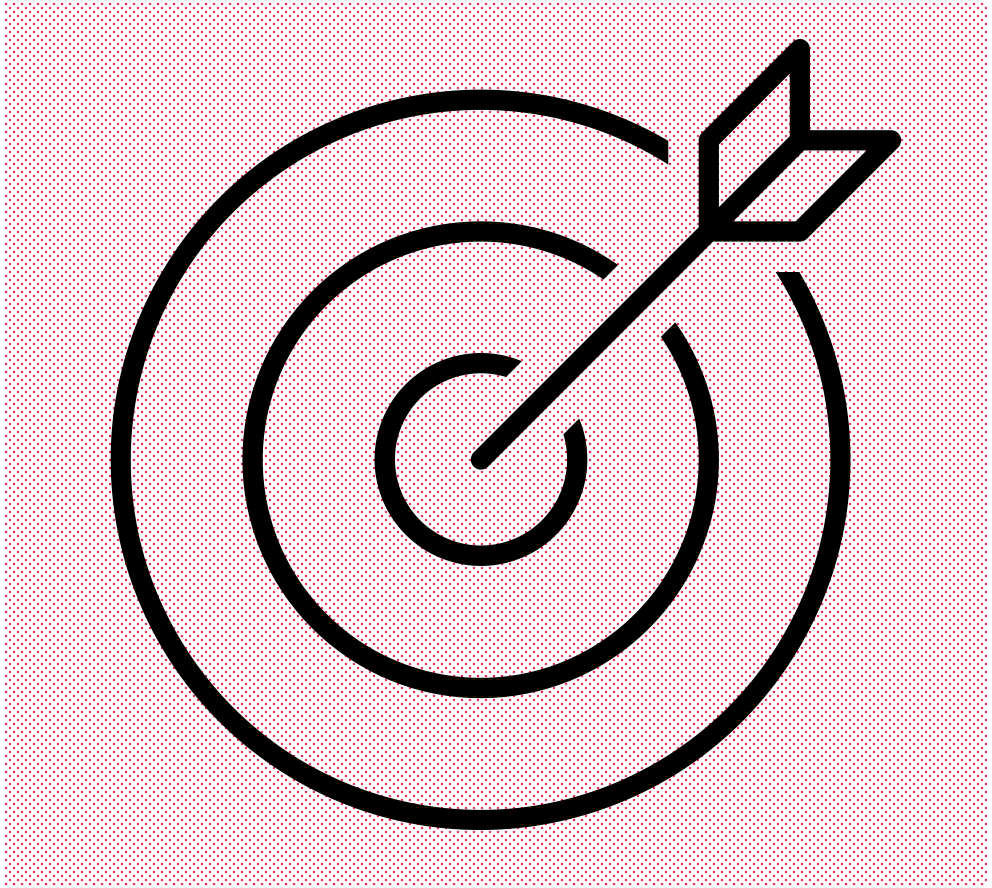
What we learned

- Efficient memory management is crucial for successful CUDA implementations
- Handling the indeterministic result size per iteration in Iterative RA operations requires careful consideration
- While low-level GPU code allows optimization opportunities, it demands considerable time and effort compared to using off-the-shelf libraries like cuDF



Accomplishments

- Our hash-join-based transitive closure computation shows favorable results against both cuDF and Souffle, with gains up to 10.8x against cuDF and 3.9x against Souffle

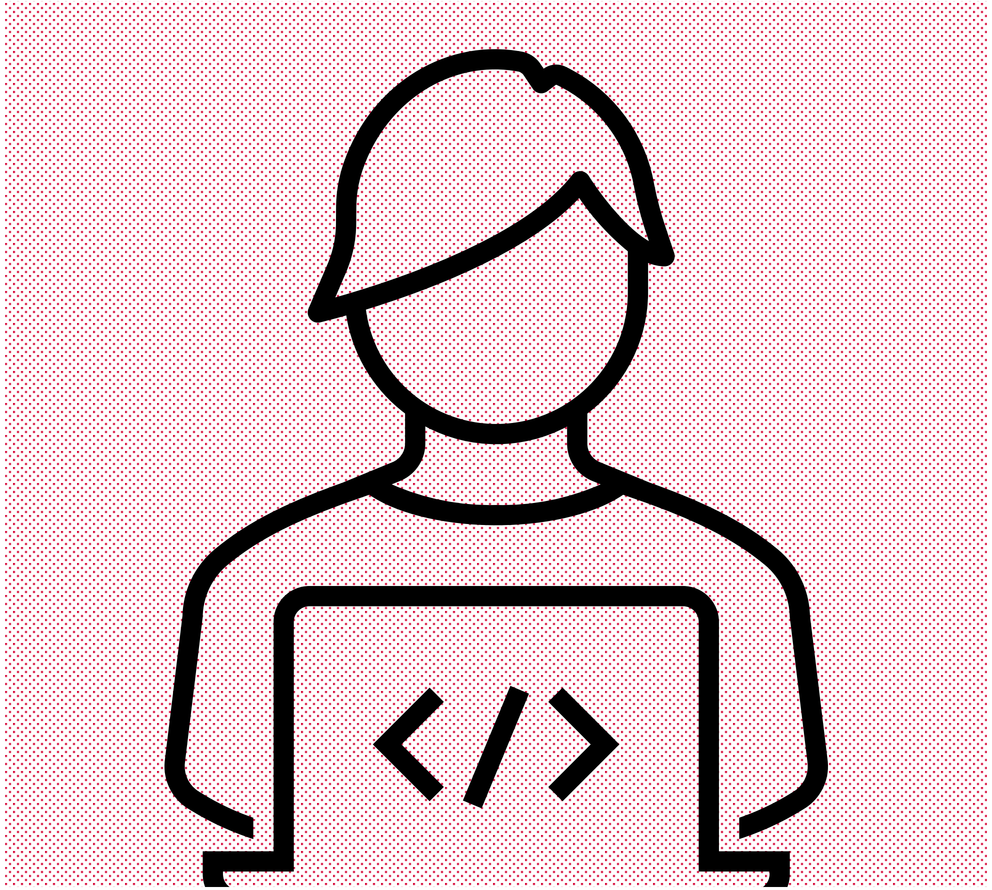


Publications

- Shovon, A. R., Gilray, T., Micinski, K., & Kumar, S. (2023). Towards Iterative Relational Algebra on the {GPU}. In 2023 USENIX Annual Technical Conference (USENIX ATC 23) (pp. 1009-1016).
- Shovon, A. R., Dyken, L. R., Green, O., Gilray, T., & Kumar, S. (2022, November). Accelerating Datalog applications with cuDF. In 2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3) (pp. 41-45). IEEE.

Project Repository

- Transitive closure computation using CUDA: https://github.com/harp-lab/usenixatc23
- Transitive closure computation using SYCL: https://github.com/arsho/tc

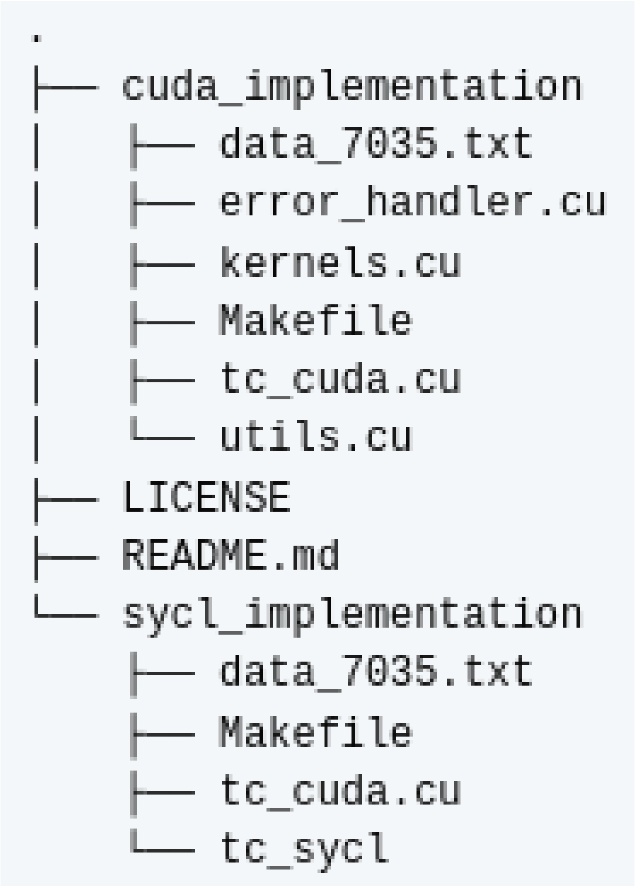


Porting TC computation CUDA implementation

Clean	Clean CUDA Project
Install	Install SYCLomatic
Convert	Convert CUDA code to SYCL
Check	Check the SYCL code and modify if necessary
Execute	Run in Intel Dev Cloud

Clean CUDA Project

- Our CUDA code has multiple files and a Makefile that has auxilary commands
- To port the CUDA project to SYCL first we cleaned the CUDA project
- We made one file that has CUDA code, simplified the Makefile, and kept one test dataset
- The folder *sycl_implementation* has the single cuda file



```
.
├── cuda_implementation
│   ├── data_7035.txt
│   ├── error_handler.cu
│   ├── kernels.cu
│   ├── Makefile
│   ├── tc_cuda.cu
│   └── utils.cu
├── LICENSE
├── README.md
└── sycl_implementation
    ├── data_7035.txt
    ├── Makefile
    ├── tc_cuda.cu
    └── tc_sycl
```

Install SYCLomatic

- Open a terminal and download SYCLomatic release:

```
cd ~/
mkdir syclomatic
cd syclomatic
wget https://github.com/oneapi-src/SYCLomatic/releases/download/20230725/linux_release.tgz
tar -xvf linux_release.tgz
```

- Add the bin path to .zshrc:

```
export PATH="~/syclomatic/bin:$PATH"
```

- Check *c2s* version:

```
c2s --version
```

SYCLomatic release: https://github.com/oneapi-src/SYCLomatic/tags

Convert CUDA code to SYCL

- Convert the CUDA code to SYCL and create a directory to store the SYCL code:
  intercept-build make
  c2s -p compile_commands.json --out-root tc_sycl
- Copy sample dataset to the SYCL code directory and create a compressed file:
  cp data_5.txt data_7035.txt tc_sycl
  tar -cvf tc_sycl.tgz tc_sycl

SYCLomatic doc: https://github.com/oneapi-src/SYCLomatic

Check SYCL Converted Code

- We had error in converted SYCL code using SYCLomatic *20230725* release. In converted SYCL code, we needed to replace *std::reducet* to *std::reduce*
- **This error did not appear when we used SYCLomatic *20230830* release**
- While converting Thrust's exclusive scan API, the SYCLomatic code was generating errors which was resolved by removing *(decltype(offset)::value_type)* from the following line: *std::exclusive_scan(oneapi::dpl::execution::make_device_policy(q_ct1), offset, offset + t_delta_rows, offset, 0);*
- For operator function, we needed to add *const* in the signature: *bool operator()(const Entity &lhs, const Entity &rhs) const*

SYCLomatic doc: https://github.com/oneapi-src/SYCLomatic

Execute in Intel Dev Cloud

- Upload the SYCL code folder to Intel Dev Cloud:

  ```
  scp tc_sycl.tgz idc:~/
  ```

- Connect to Intel Dev Cloud, start an interactive session, load the modules:

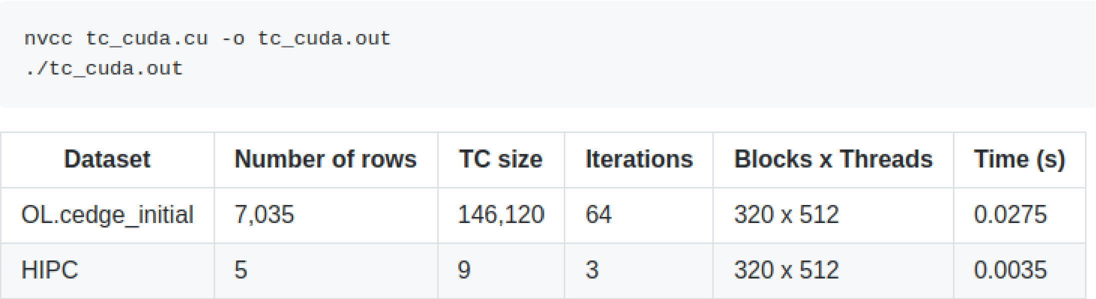
  ```
  ssh idc
  srun --pty bash
  source /opt/intel/oneapi/setvars.sh
  ```

- Execute the SYCL code:

  ```
  tar -xvf tc_sycl.tgz
  cd tc_sycl
  icpx -fsycl *.cpp
  ```

Intel Dev Cloud instances: https://scheduler.cloud.intel.com/#/systems

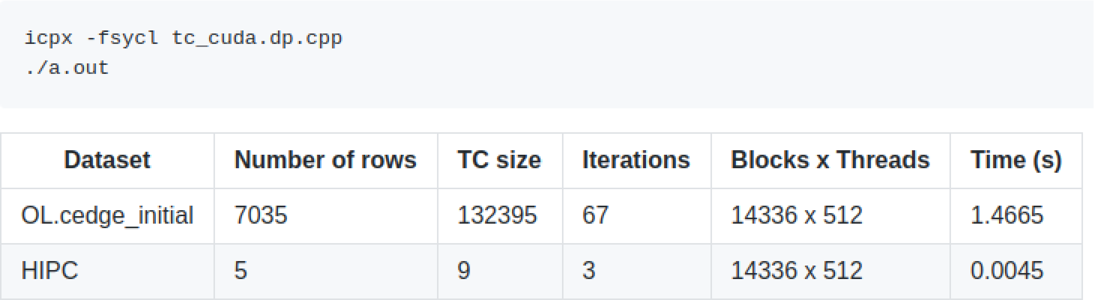
Result comparison: CUDA and SYCL



```
nvcc tc_cuda.cu -o tc_cuda.out
./tc_cuda.out
```

Dataset	Number of rows	TC size	Iterations	Blocks x Threads	Time (s)
OL.cedge_initial	7,035	146,120	64	320 x 512	0.0275
HIPC	5	9	3	320 x 512	0.0035

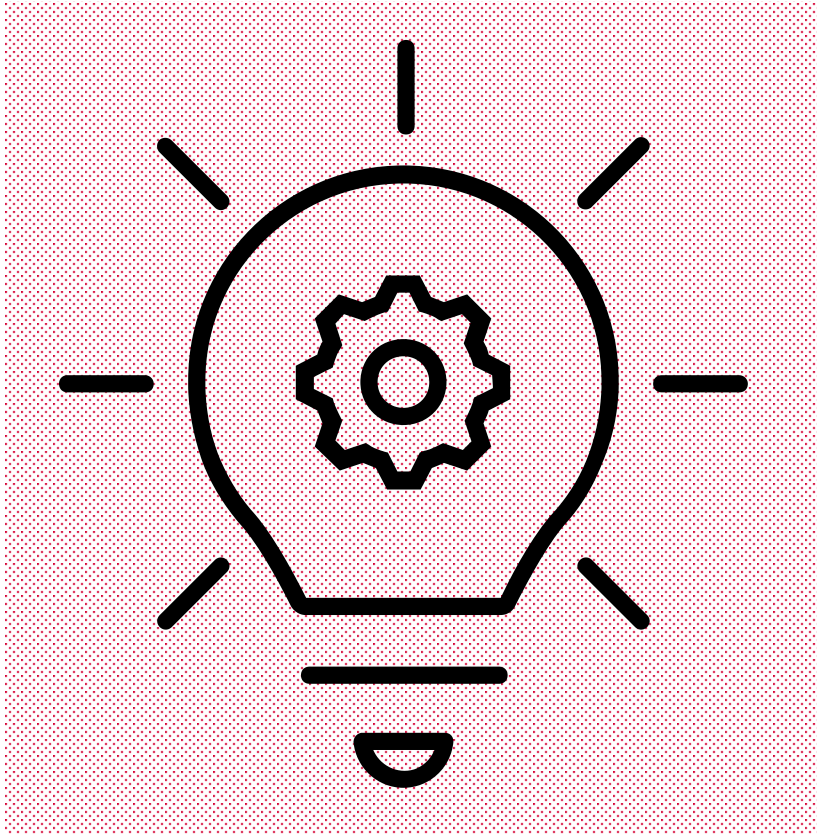
CUDA results



```
icpx -fsycl tc_cuda.dp.cpp
./a.out
```

Dataset	Number of rows	TC size	Iterations	Blocks x Threads	Time (s)
OL.cedge_initial	7035	132395	67	14336 x 512	1.4665
HIPC	5	9	3	14336 x 512	0.0045

SYCLomatic generated SYCL results



Status

- We ported the CUDA transitive closure computation code to SYCL using SYCLomatic
- We needed to manually change some of the converted code to resolve compilation error
- The SYCL results are correct for small datasets but incorrect for larger ones
- As SYCL supports many standard data structures, we decided to implement SYCL implementation from scratch removing the overheads of Thrust APIS

Future Direction

- Implement transitive closure computation using SYCL from scratch
- Compare the TC computation with CUDA, cuDF, and Souffle on single GPU
- Extend the computation to use multiGPU environment on intel GPU targeting the Aurora supercomputer

Feedback

- Examples on dynamic data structure implementations using SYCL would be helpful
- Automated generation of additional comments on ported code can explain the converted code

Thank You