



09/05/2023

# 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](mailto:ashov@uic.edu)



**Thomas Gilray**

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

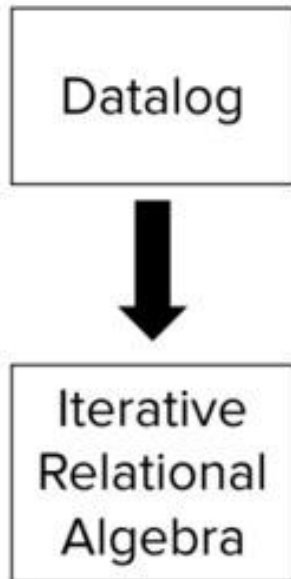


**Sidharth Kumar**

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

# Transitive Closure Computation using Iterative Relational Algebra

## Bottom-Up Logic Programming with Datalog



Datalog rule for computing **Transitive Closure (TC)**

$$\begin{aligned} T(x, y) &\leftarrow G(x, y) . \\ T(x, z) &\leftarrow T(x, y) , G(y, z) . \end{aligned}$$

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:

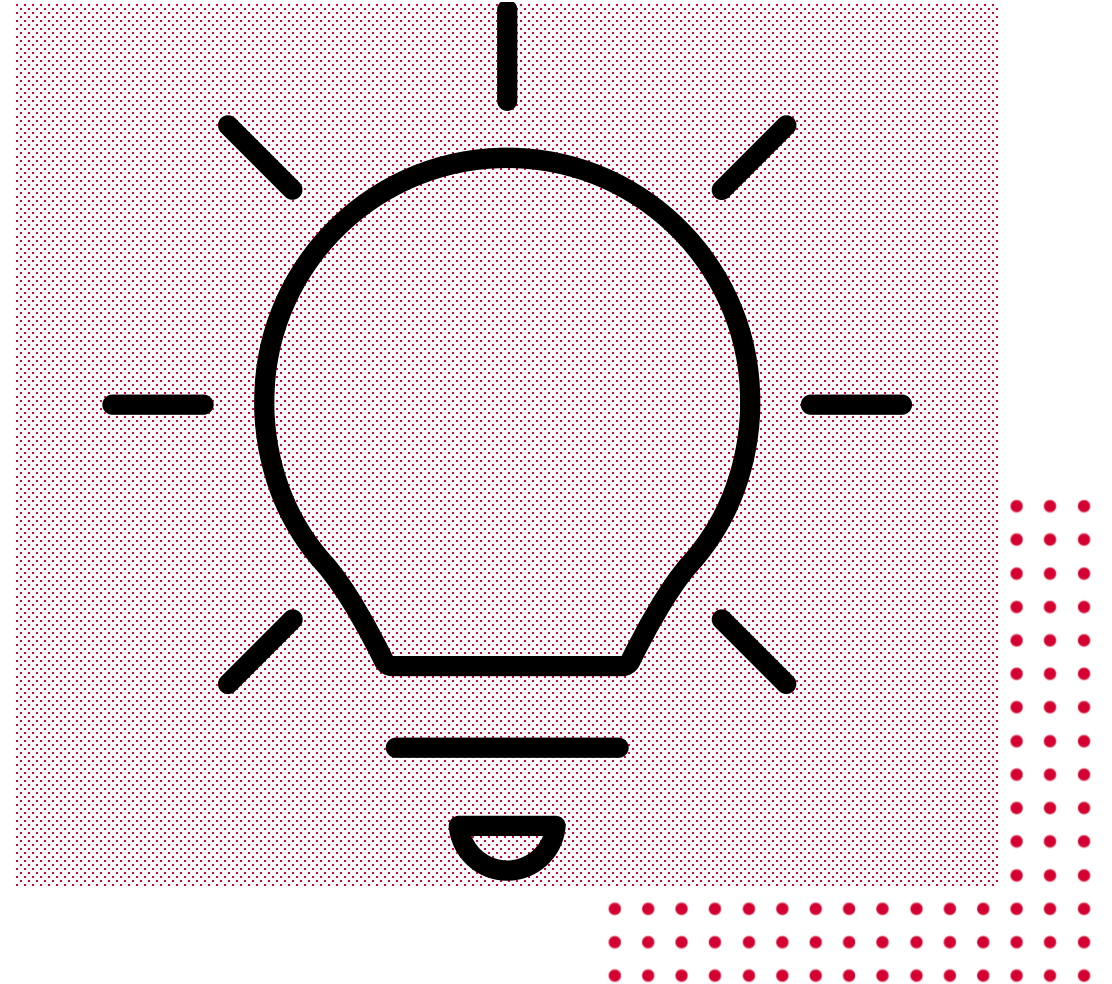
Union

Projection

Join

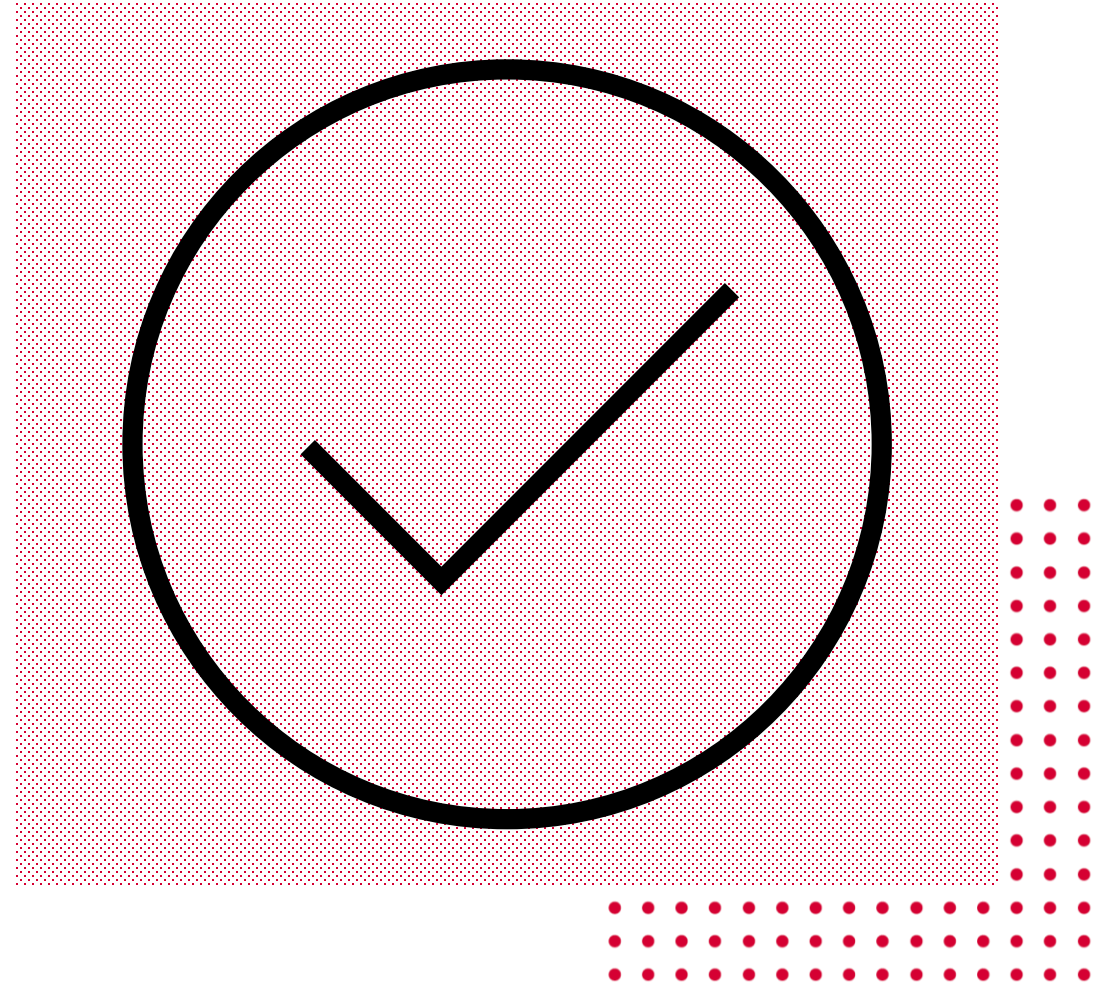
# Inspiration

- 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)



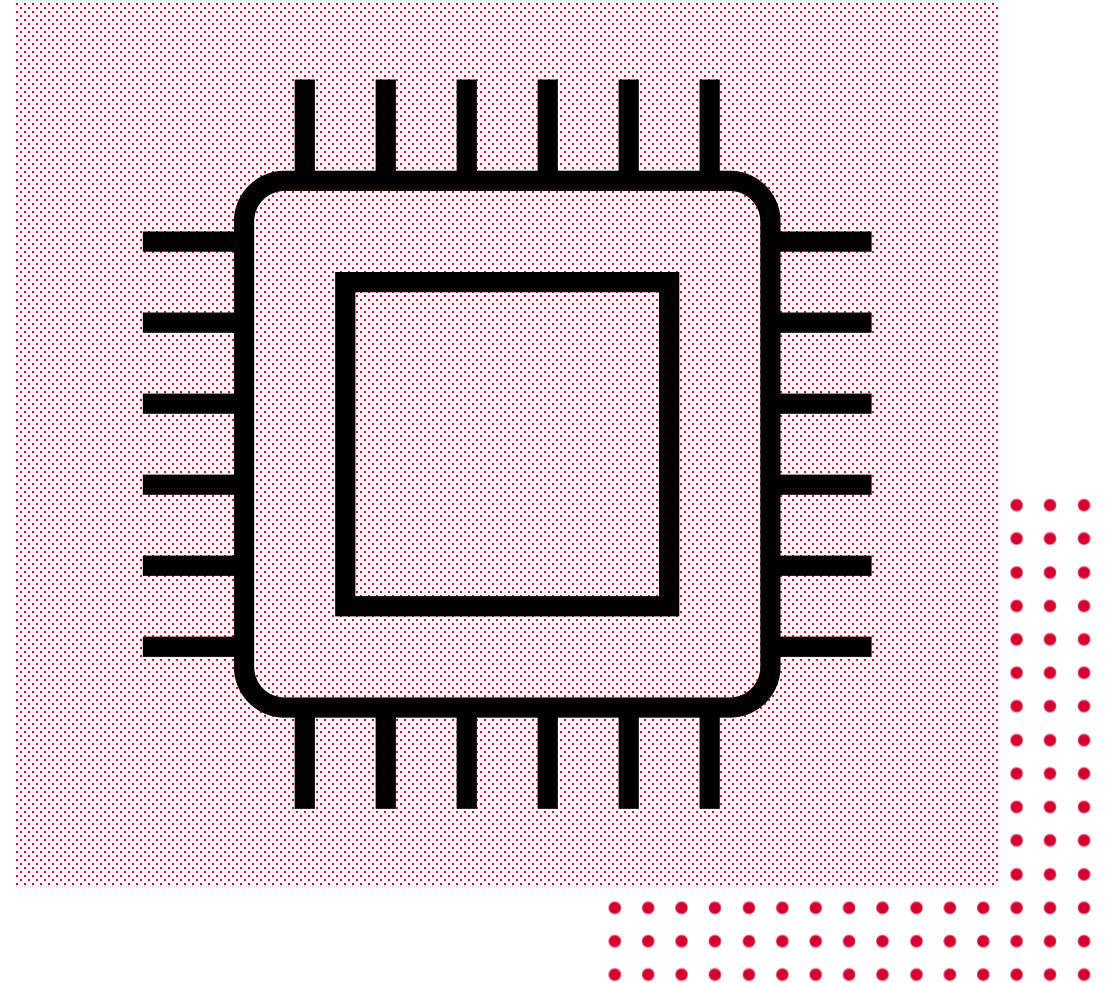
# What it does

- 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

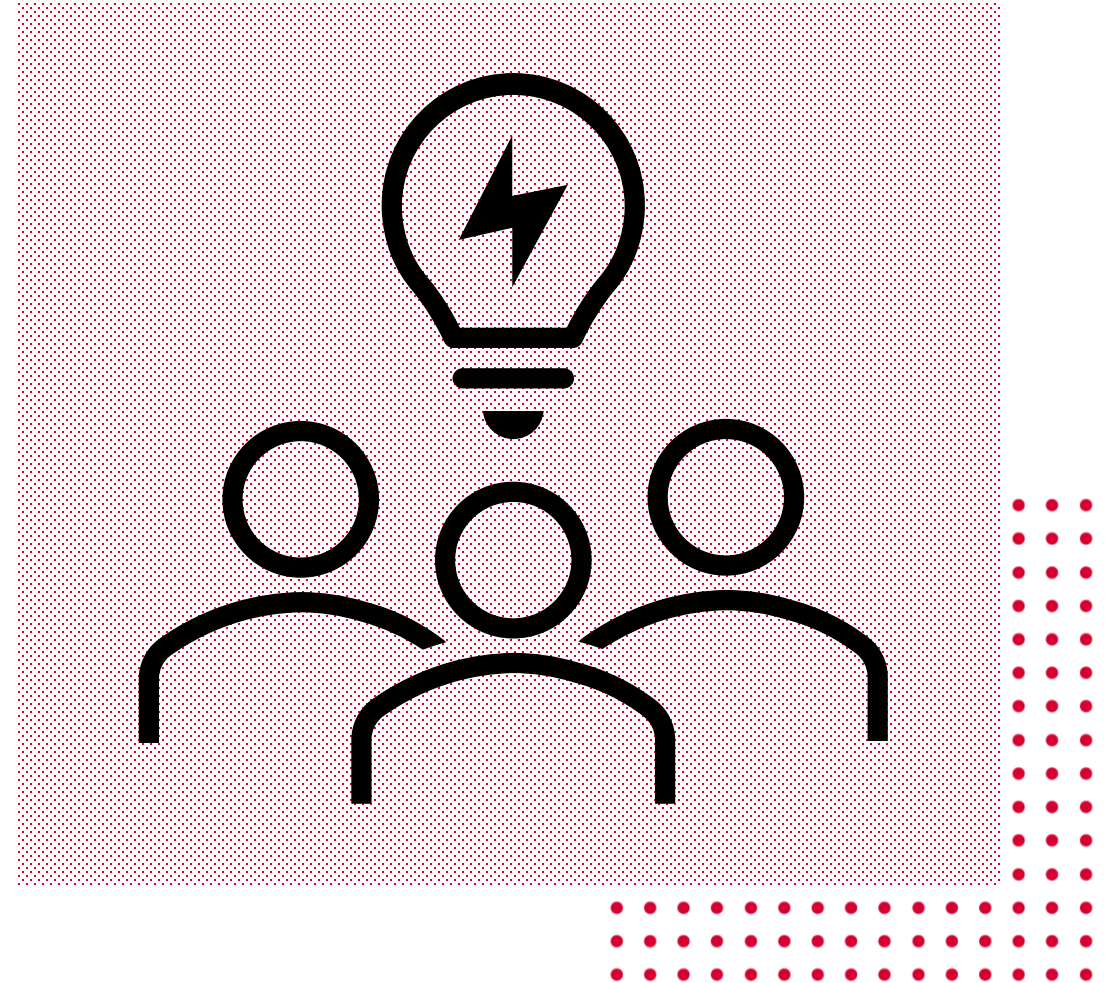
- Unlike C++, CUDA lacks efficient data structures, limiting our implementation's capabilities
- The available VRAM of a single GPU imposes constraints on our implementation's scalability
- Debugging kernel code posed significant challenges, turning it into a nightmarish experience





# What we learned

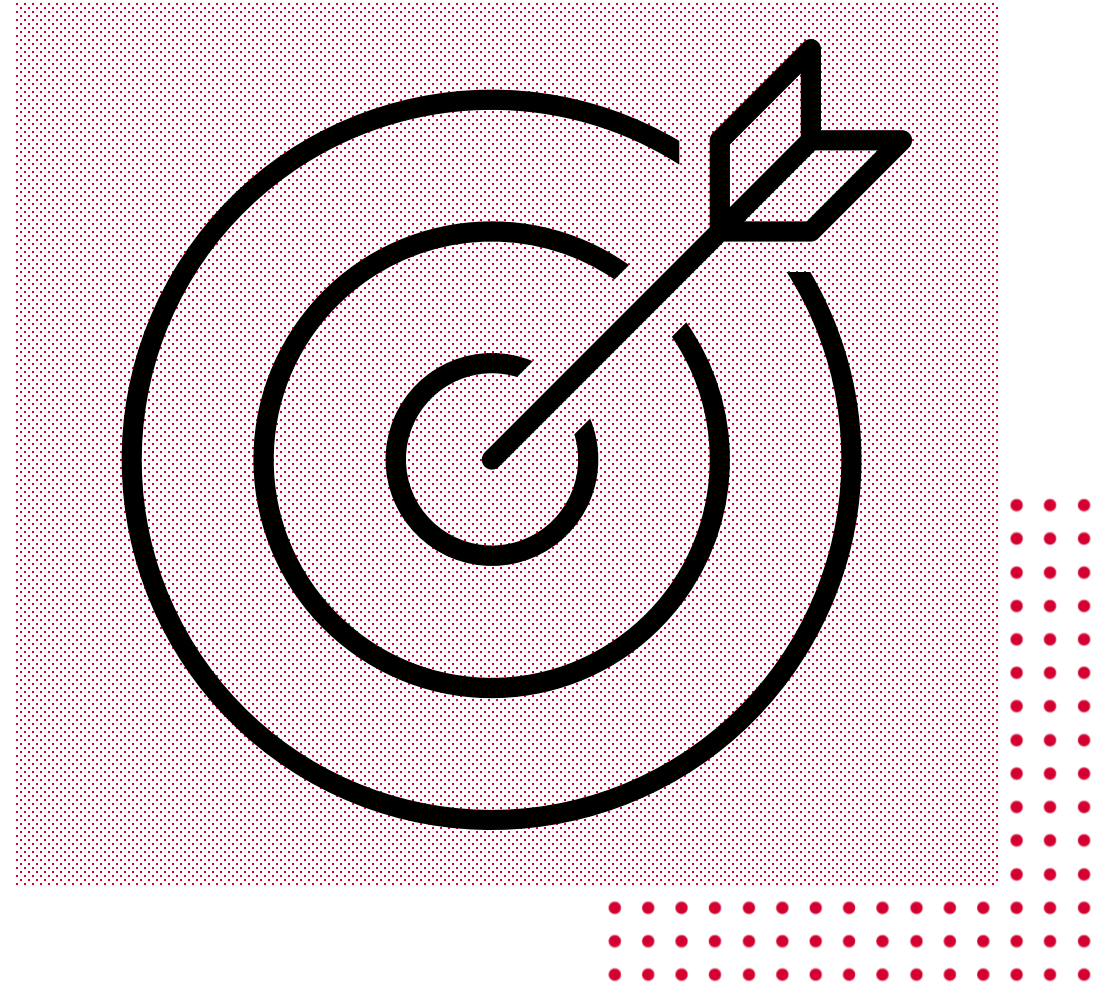
- Efficient memory management is crucial for successful CUDA implementations
- Handling the indeterministic result size per iteration in Iterative RA operations requires careful consideration
- While low-level GPU code allows optimization opportunities, it demands considerable time and effort compared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





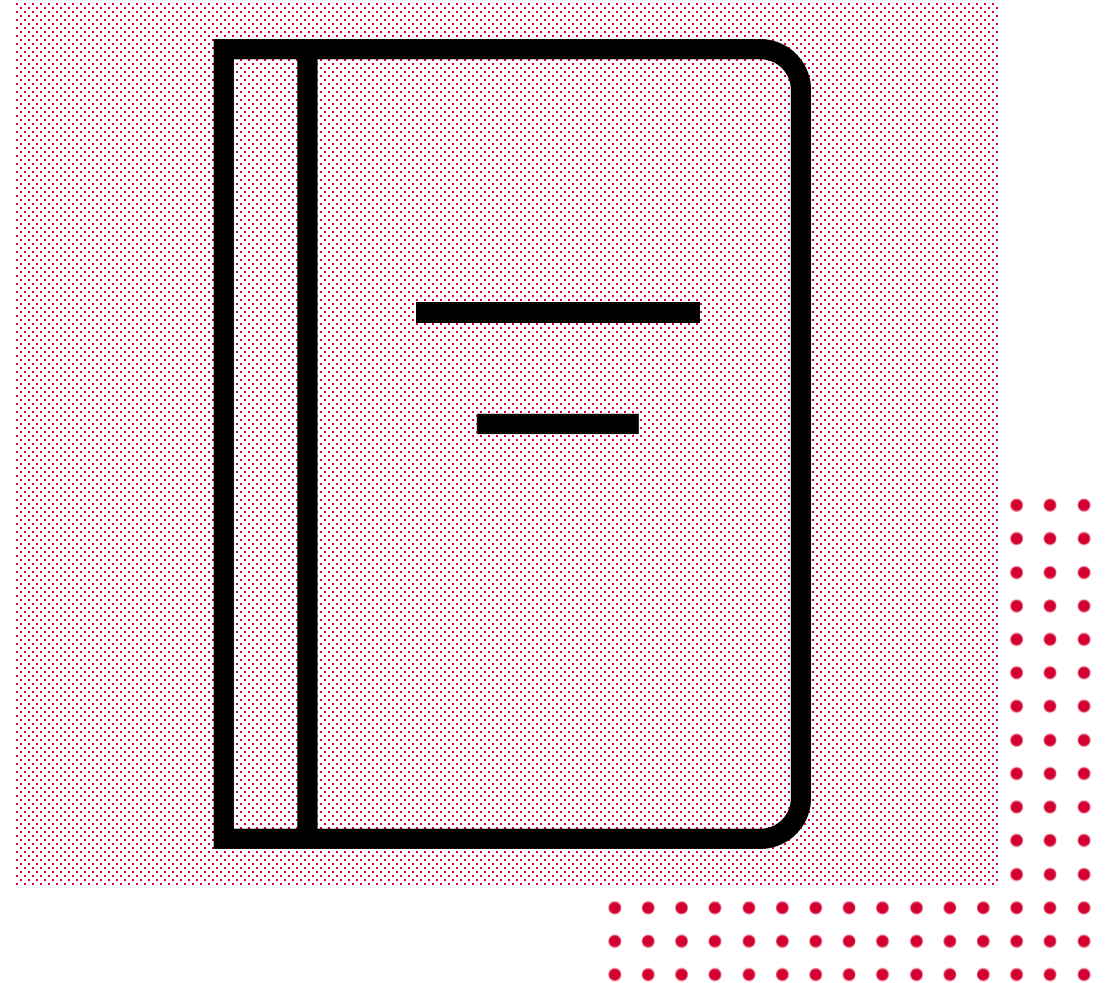
# Accomplishments

- 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



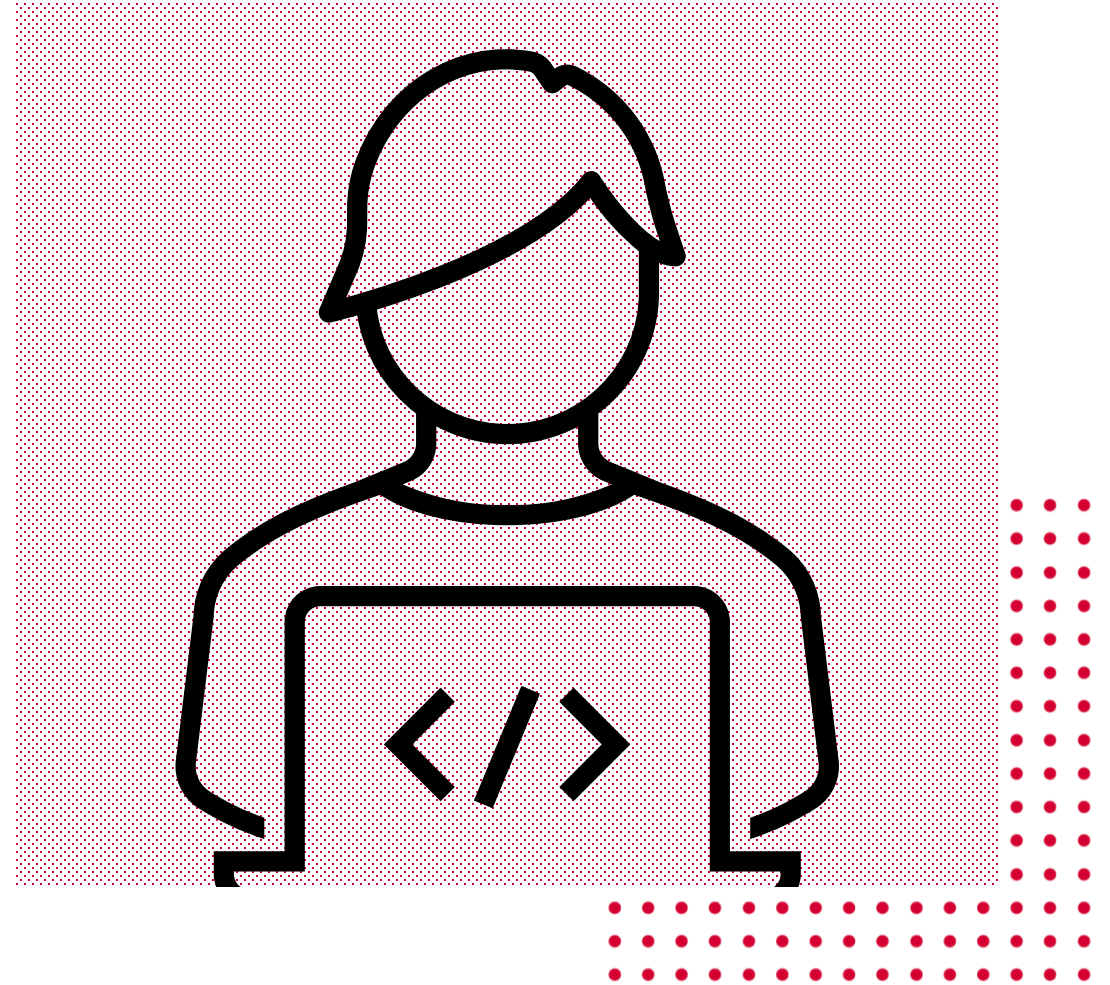
# Publications

- 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 auxiliary 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*

# 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\_7035.txt tc\_sycl*  
*tar -cvf tc\_sycl.tgz tc\_sycl*



# Check SYCL Converted Code

- We had error in converted SYCL code using SYCLomatic 20230725 release. In converted SYCL code, we needed to replace *std::reduces* to *std::reduce*
- This error did not appear when we used SYCLomatic 20230830 release

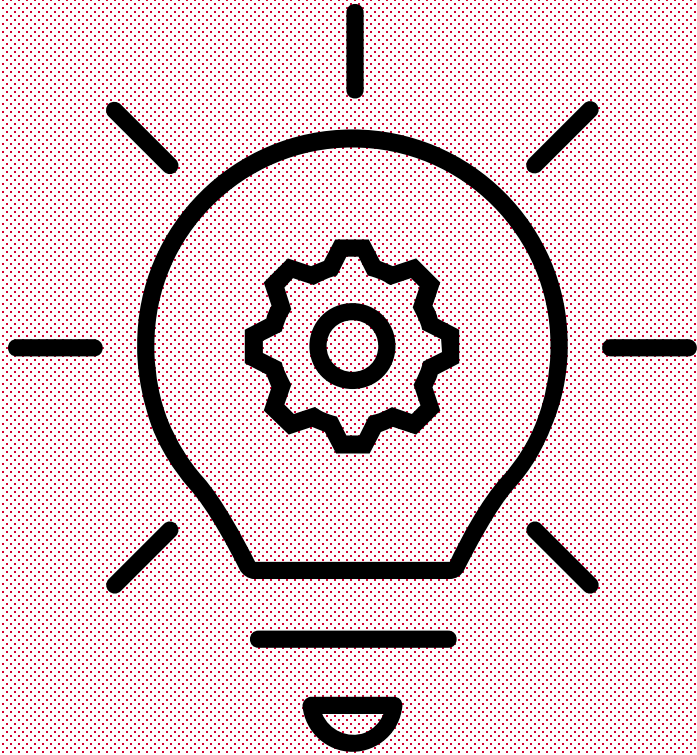
# Execute in Intel Dev Cloud

- 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*  
*srunc --pty bash*  
*source /opt/intel/oneapi/setvars.sh*
- Execute the SYCL code:  
*tar -xvf tc\_sycl.tgz*  
*cd tc\_sycl*  
*icpx -fsycl \*.cpp*



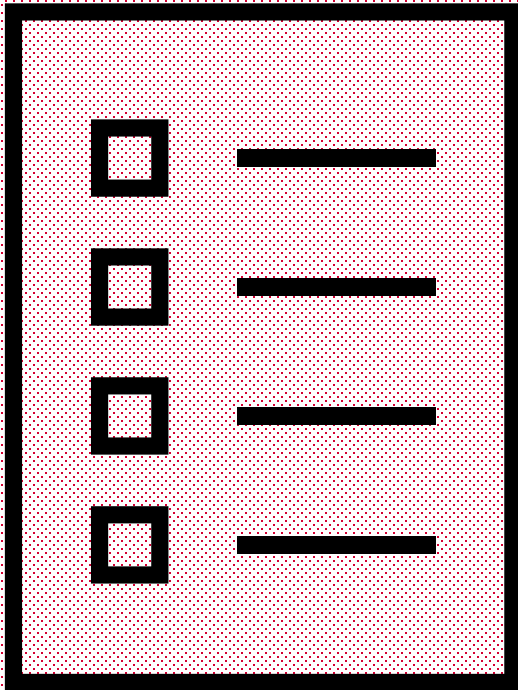
## Feedback

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



## Status

- We ported the CUDA transitive closure computation code to SYCL but it raised several errors on ported code; specially on Thrust CUDA code
- After closely investigate the errors, we found most of the errors are CUDA related and not related to SYCL
- As SYCL supports many standard data structures, we decided to implement SYCL implementation from scratch



## Future Direction

- Implement transitive closure computation using SYCL
- 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

**Thank You**