JIT LTO USE CASES IN THE cuSPARSE LIBRARY

Federico Busato | Senior Software Engineer, Library Lead

AGENDA

Motivations and Challenges
The Need for Flexibility

Use Cases
Deep Learning and GraphBLAS

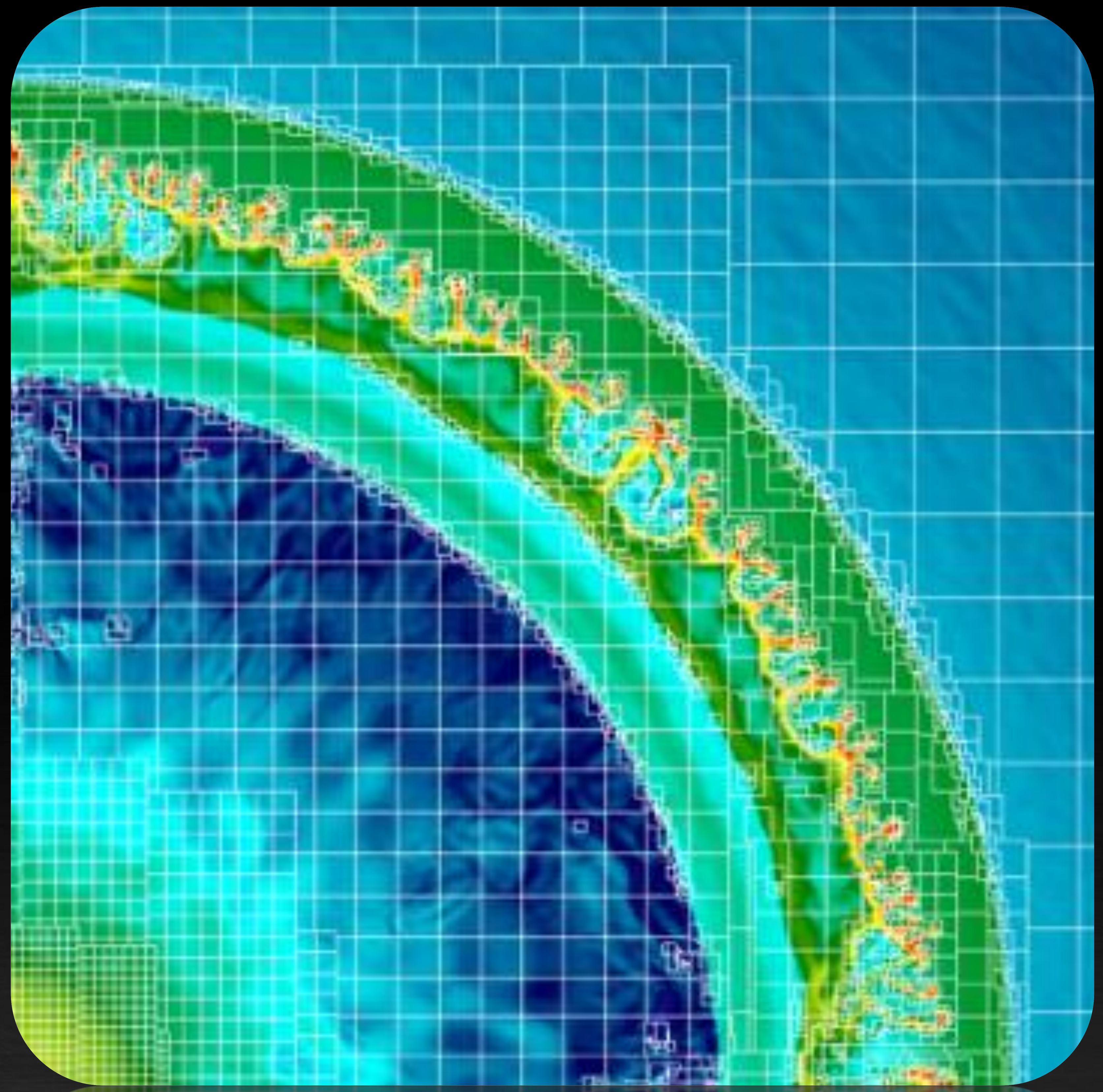
JIT LTO APIs
Workflow and Code Example

Performance Results and Conclusions
Comparison with the Hardwired Version

MOTIVATIONS AND CHALLENGES

The Need for Flexibility

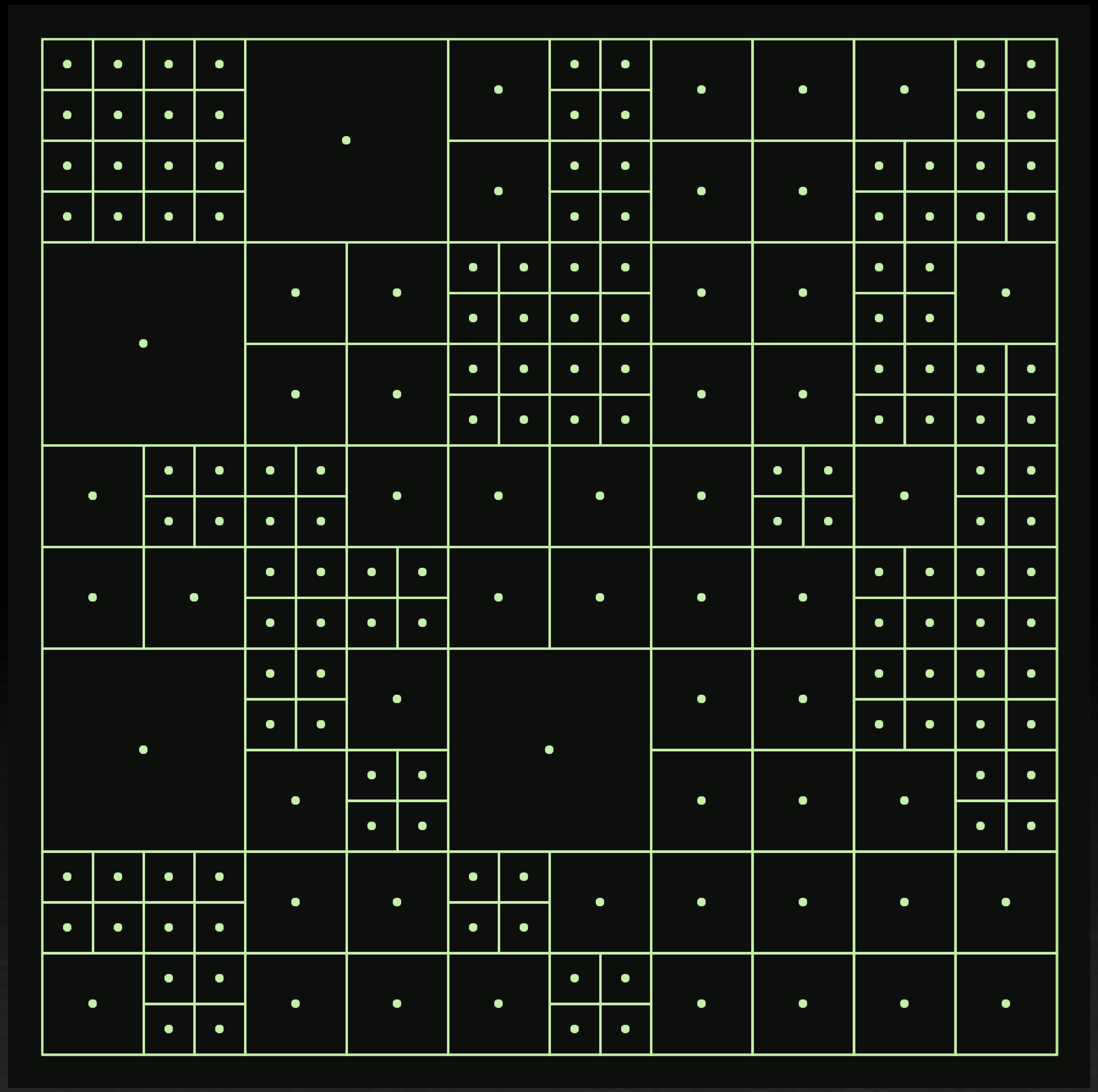
- In last decades, we saw a dramatic expansion of linear algebra and sparse linear algebra (LA) applications in industrial and academic contexts
  - New routines, requirements, data types, storage formats, etc.
  - Recently, linear algebra methods applied outside strict linear algebra
  - Generalize by replacing standard LA operations (i.e. addition, multiplication) with any operator
  - Black-box operators: users are free to perform arbitrary computation. Only input/output are fixed
- cuSPARSE is a closed-source GPU library
  - We cannot predict all potential uses
  - Relying on a fixed set of operators does not fix the problem → binary size constrains, requests for new operators, etc.
  - JIT is great for flexibility but it affects application performance



Adaptive Mesh Refinement Calculations, LBNL

Motivations and Challenges

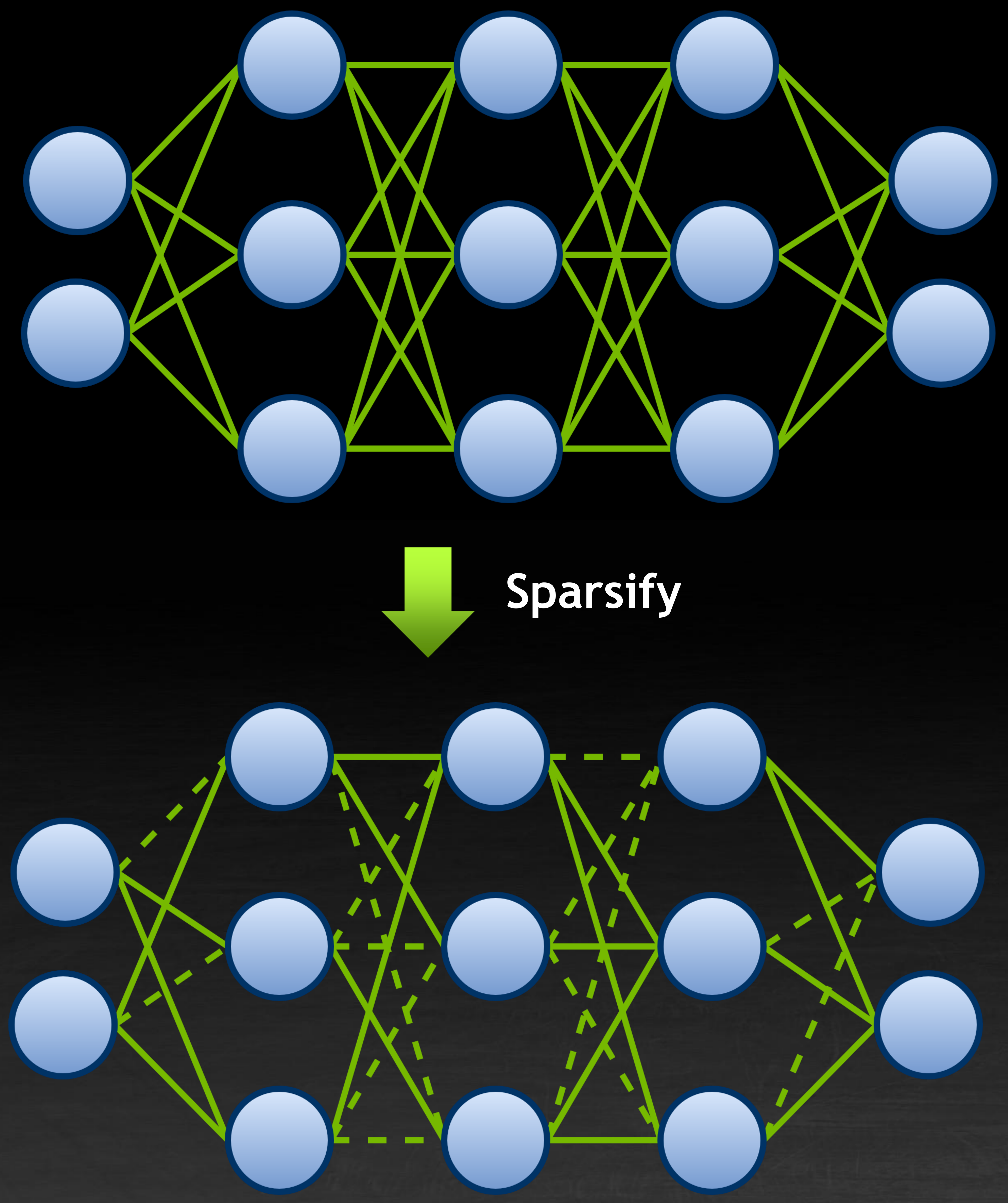
The Need for Flexibility



- JIT LTO provides both flexibility and performance
  - *Combine user code with library code*
  - Generate *highly optimized kernels* by substituting run-time parameters with constants
  - *Reduce the library binary size* by merging different parts at linking-time
  - *Run-time kernel tuning* by iterating among all template parameters
- JIT LTO downside:
  - Run-time overhead for compiling/linking the program
  - Adapt code for custom operators: atomicCAS or deterministic algorithm
  - Rely on NVRTC and Driver APIs
  - JIT LTO does not support CUDA Driver < 495.xx (for cuSPARSE)

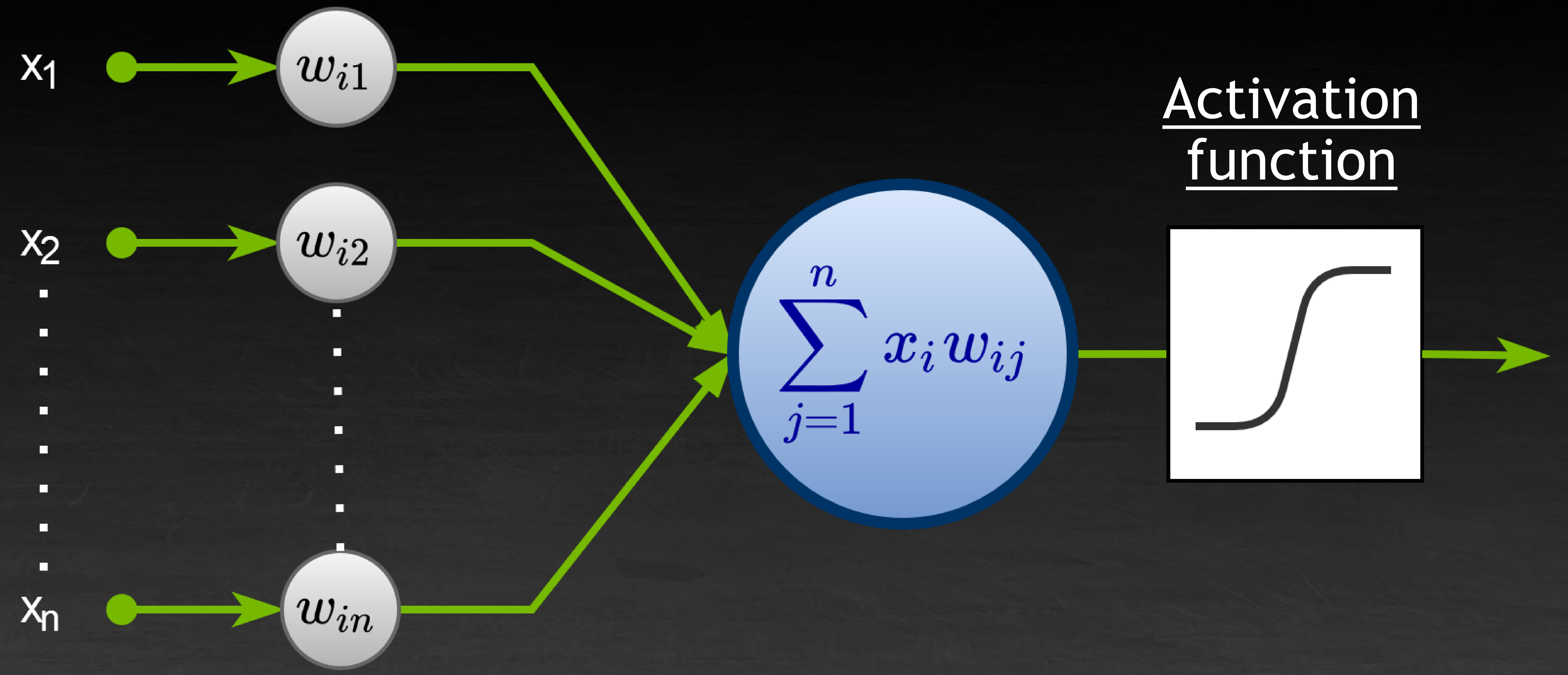
USE CASES

Neural Networks



Sparse computation is widely adopted

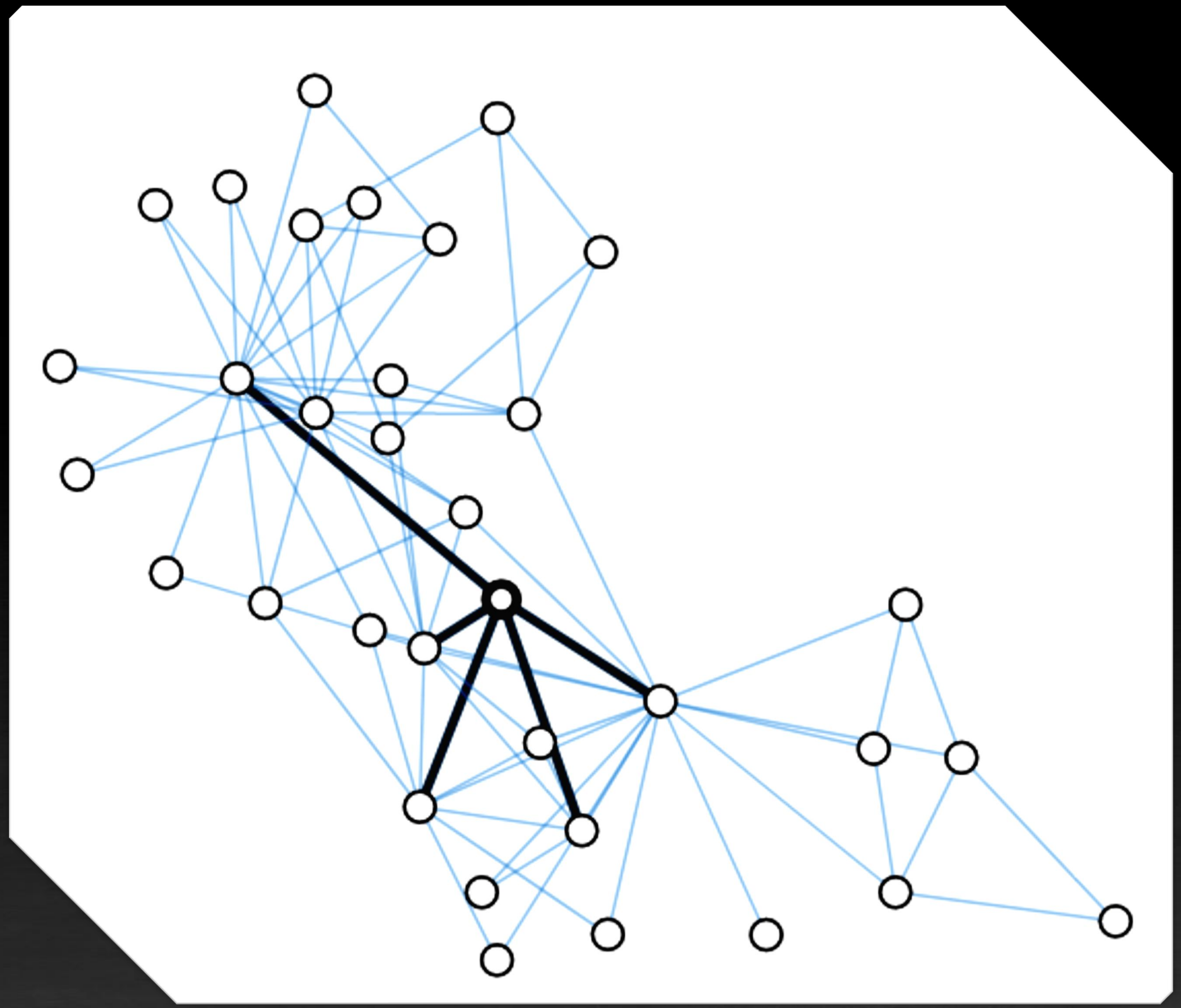
- Deep Graph Library (DGL)
- PyTorch Geometric
- K2 speech recognition library
- Facebook FBGEMM

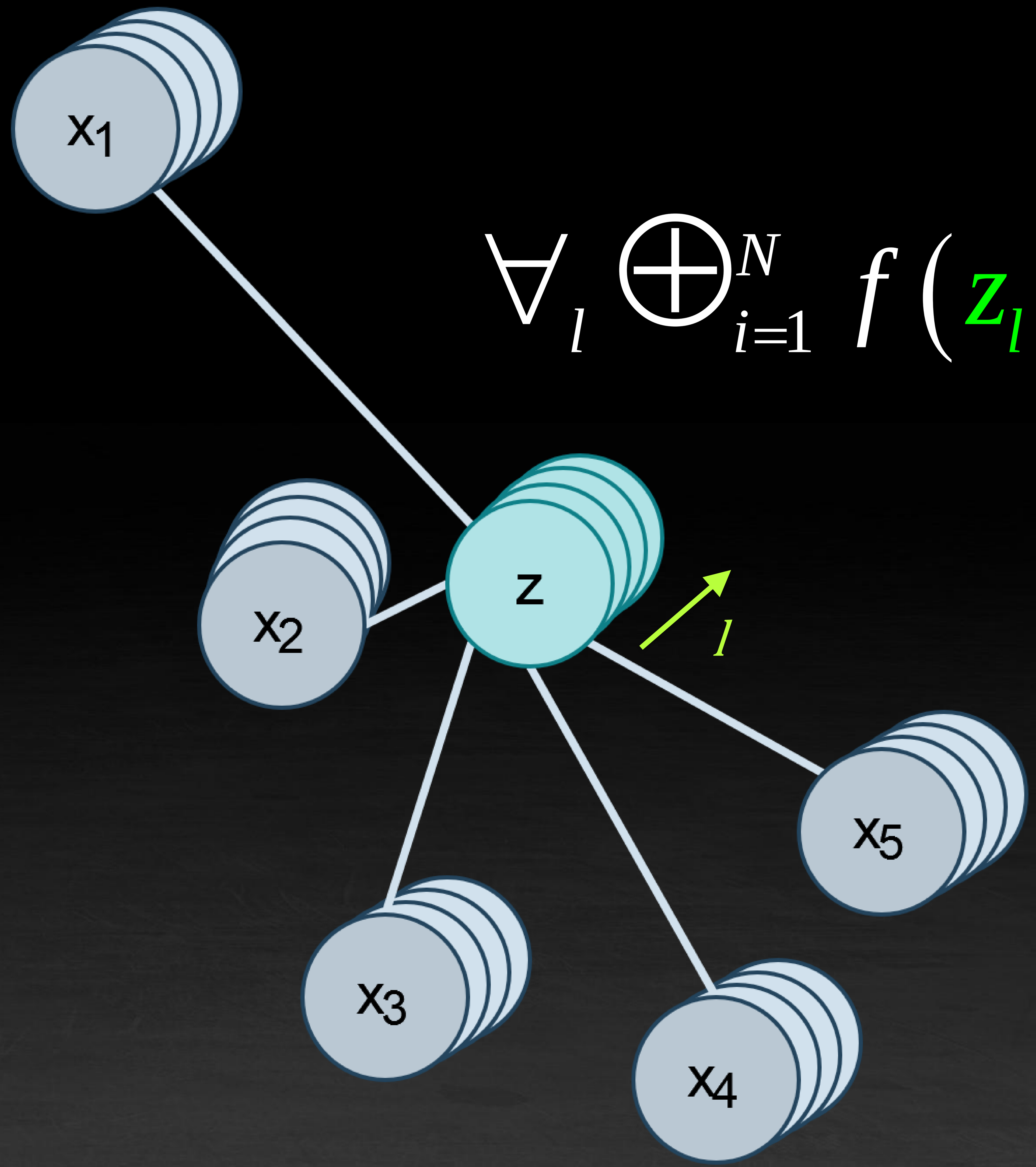


Survey: Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks, Hoefler et at.

USE CASES

Graph Neural Networks





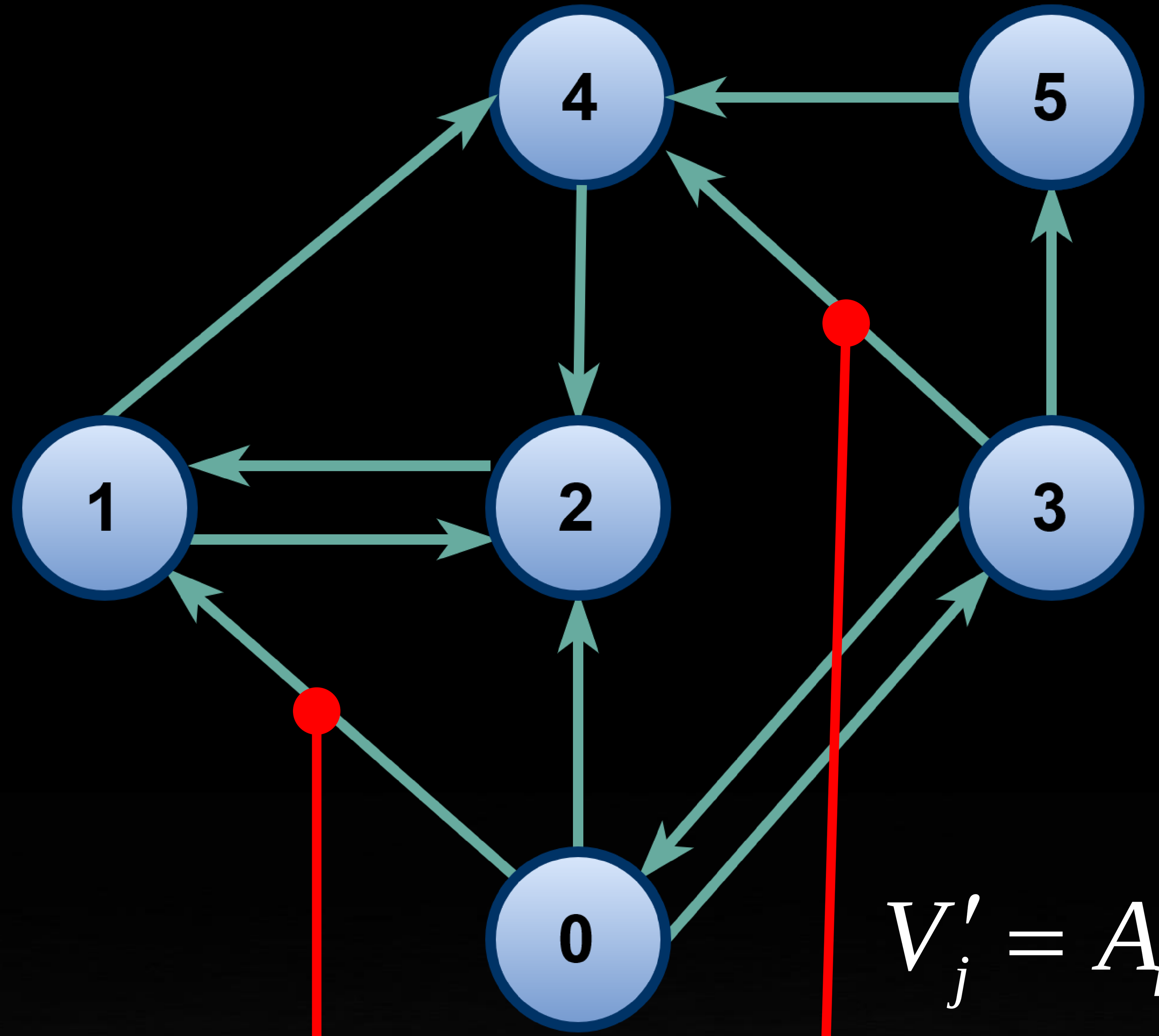
$$\forall_l \bigoplus_{i=1}^{N} f\left(z_l, x_{li}, z_l \leftrightarrow x_{li}\right)$$

USE CASES

GraphBLAS

Graphs and matrices are two sides of the same coin

- GraphBLAS standard defines standard building blocks for graph algorithms in the language of linear algebra
- This example shows Breath-First Search (BFS) graph traversal starting from vertex 0 by using linear algebra operations



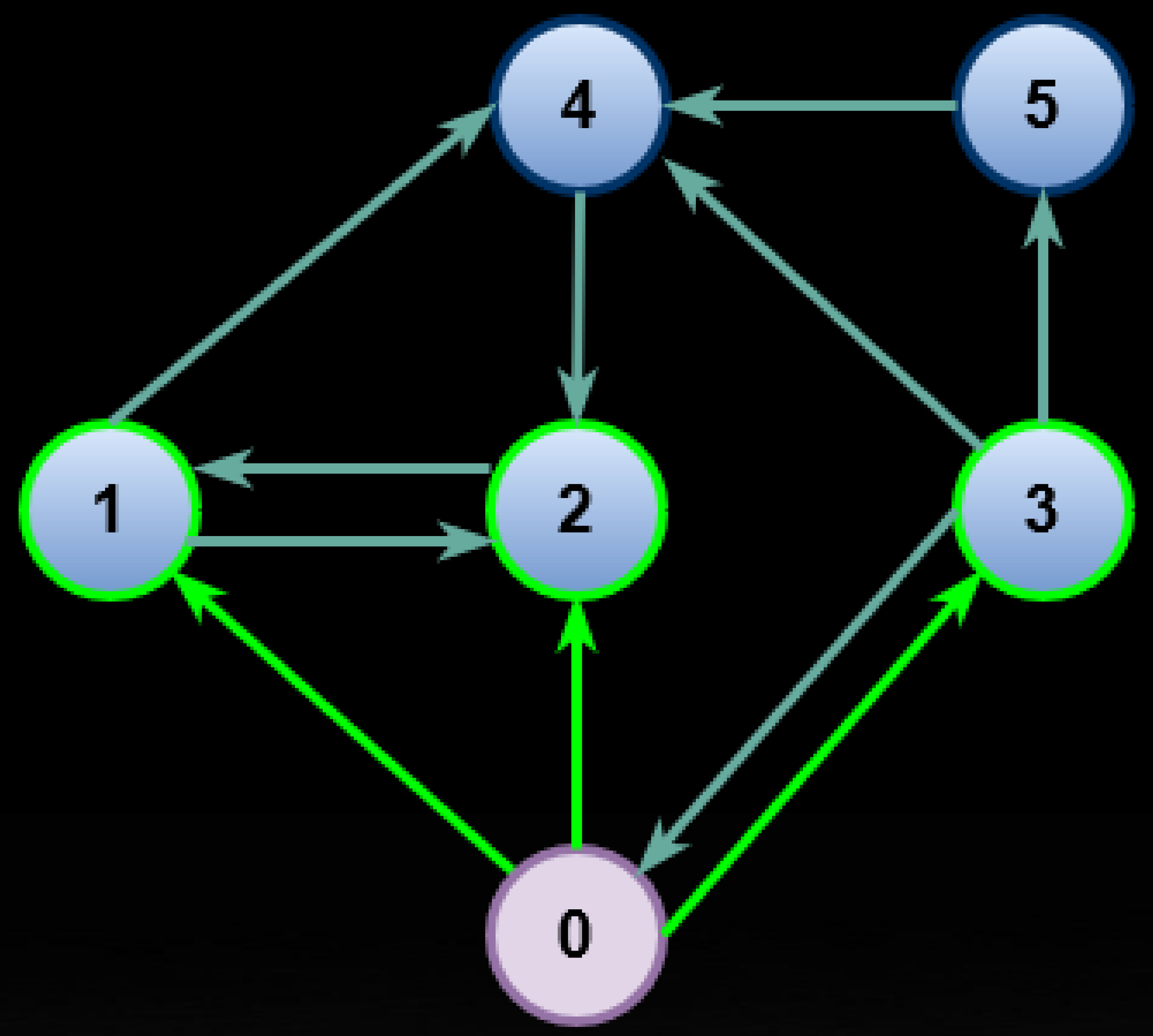
$$V'_j = A_{ij} \odot V_i \rightarrow \begin{cases} A_{ij} \text{ is set} & \min\left(V_i + 1, V_j\right) \\ A_{ij} \text{ is not set} & nop \end{cases}$$

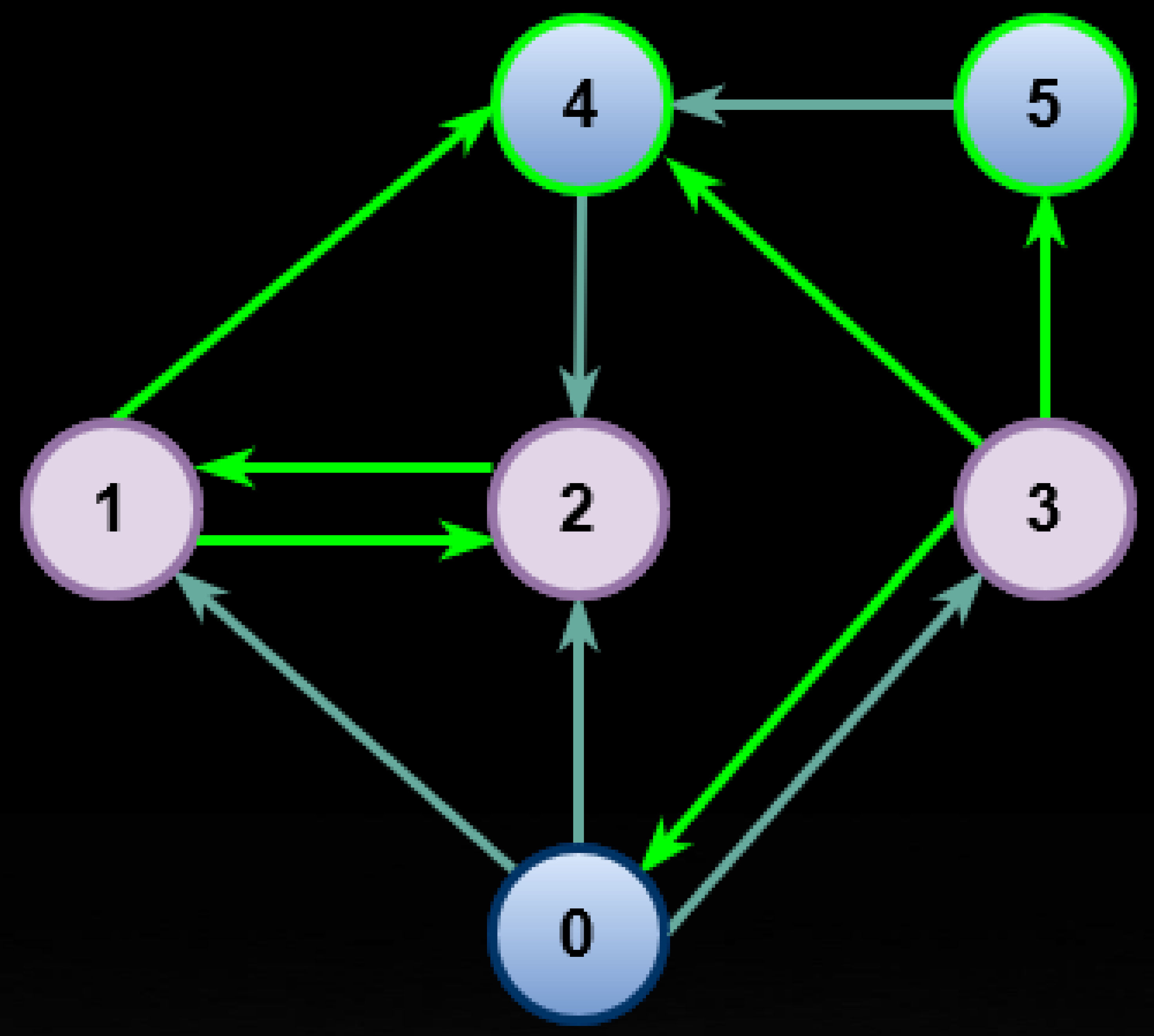
	0	1	2	3	4	5		
0		X	X	X			0	0
1			X		X		∞	?
2		X					∞	?
3	X				X	X	∞	?
4			X				∞	?
5					X		∞	?

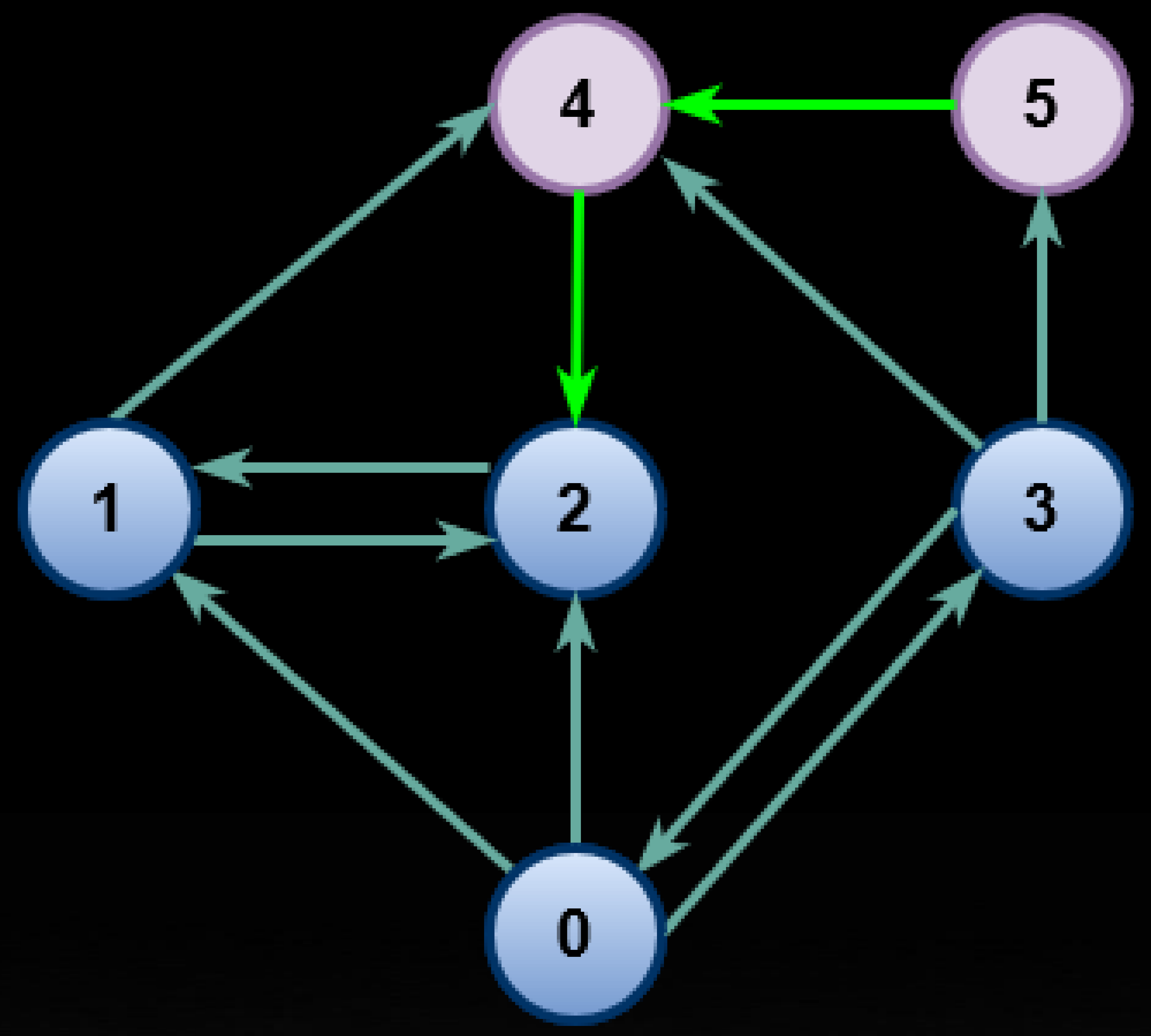
$$A^T \odot V = V'$$

Use Cases

GraphBLAS







	0	1	2	3	4	5		
0		X	X	X			0	0
1			X		X		∞	1
2		X					∞	1
3	X				X	X	∞	1
4			X				∞	∞
5					X		∞	∞

	0	1	2	3	4	5		
0		X	X	X			0	0
1			X		X		1	1
2		X					1	1
3	X				X	X	1	1
4			X				∞	2
5					X		∞	2

=

	0	1	2	3	4	5		
0		X	X	X			0	0
1			X		X		1	1
2		X					1	1
3	X				X	X	1	1
4			X				2	2
5					X		2	2

JIT LTO APIs

Overview

$$C = \alpha op(A) \cdot op(B) + \beta C$$

$$C'_{ij} = \text{epilogue}\left(\sum_k^{\oplus} op(A_{ik}) \otimes op(B_{kj}), C_{ij}\right)$$

Standard API

```
cusparseStatus_t
cusparseSpMM(cusparseHandle_t         handle,
             cusparseOperation_t      opA,
             cusparseOperation_t      opB,
             const void*              alpha,
             cusparseSpMatDescr_t     matA,
             cusparseDenseMatDescr_t  matB,
             const void*              beta,
             cusparseDenseMatDescr_t  matC,
             cudaDataType             computeType,
             cusparseSpMMAlg_t        alg,
             void*                    externalBuffer)
```

JIT LTO APIs

```
cusparseStatus_t
cusparseSpMMOp_createPlan(cusparseHandle_t       handle,
                          cusparseSpMMOpPlan_t*  plan,
                          cusparseOperation_t    opA,
                          cusparseOperation_t    opB,
                          cusparseSpMatDescr_t   matA,
                          cusparseDnMatDescr_t   matB,
                          cusparseDnMatDescr_t   matC,
                          cudaDataType           computeType,
                          cusparseSpMMOpAlg_t    alg,
                          const void*            addOperationNvvmBuffer,
                          size_t                 addOperationBufferSize,
                NVVM Data const void*            mulOperationNvvmBuffer,
                          size_t                 mulOperationBufferSize,
                          const void*            epilogueNvvmBuffer,
                          size_t                 epilogueBufferSize,
                          size_t*                SpMMWorkspaceSize)
```

```
cusparseStatus_t
cusparseSpMMOp(cusparseSpMMOpPlan_t plan,
               void*                externalBuffer)
```

JIT LTO APIs

cuSPARSE Workflow

Initialization

- Matrix allocation
- Matrix copy host to device
- Create cuSPARSE descriptors using the generic APIs

NVRTC

- Create a program
- Compile the program
- Get NVVM buffer

Execution

- Create the plan
- Run SpMM

```
int   *dA_csrOffsets, *dA_columns;
float *dA_values, *dB, *dC;
cudaMalloc(&dA_csrOffsets, (A_num_rows + 1) * sizeof(int));
cudaMalloc(&dA_columns,    A_nnz * sizeof(int));
…
cudaMemcpy(dA_csrOffsets, hA_csrOffsets,
           (A_num_rows + 1) * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dA_columns, hA_columns,
           A_nnz * sizeof(int), cudaMemcpyHostToDevice);
…
cusparseHandle_t handle;
cusparseCreate(&handle);
cusparseSpMatDescr_t matA;
cusparseDnMatDescr_t matB, matC;
cusparseCreateCsr(&matA, A_num_rows, A_num_cols, A_nnz,
                  dA_csrOffsets, dA_columns, dA_values,
                  CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I,
                  CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);
cusparseCreateDnMat(&matB, A_num_cols, B_num_cols, ldb, dB,
                    CUDA_R_32F, CUSPARSE_ORDER_ROW);
cusparseCreateDnMat(&matC, A_num_rows, B_num_cols, ldc, dC,
                    CUDA_R_32F, CUSPARSE_ORDER_ROW);
```

The full example will be provided in the [Nvidia GitHub Library Samples repository]

JIT LTO APIs

cuSPARSE Workflow

Initialization
- Matrix allocation
- Matrix copy host to device
- Create cuSPARSE descriptors using the generic APIs

NVRTC
- Create a program
- Compile the program
- Get NVVM buffer and buffer size

Execution
- Create the plan
- Run SpMM with custom operators

```
const char AddOp[] =
"__device__ float add_op(float value1, float value2) { \n\
    return value1 + value2;                            \n\
}";
nvrtcProgram prog;
nvrtcCreateProgram(&prog, AddOp, NULL, 0, NULL, NULL)
const char* nvrtc_options[] = {"-arch=compute_sm80", "-rdc=true",
                               "-dlto", "-std=c++14"};
int    num_options = 4;
void*  nvvm_add;
size_t nvvm_add_size;
nvrtcCompileProgram(prog, num_options, nvrtc_options);
nvrtcGetNVVMSize(prog, &nvvm_add_size);
nvrtcGetNVVM(prog, nvvm_add);
nvrtcDestroyProgram(&prog);
```

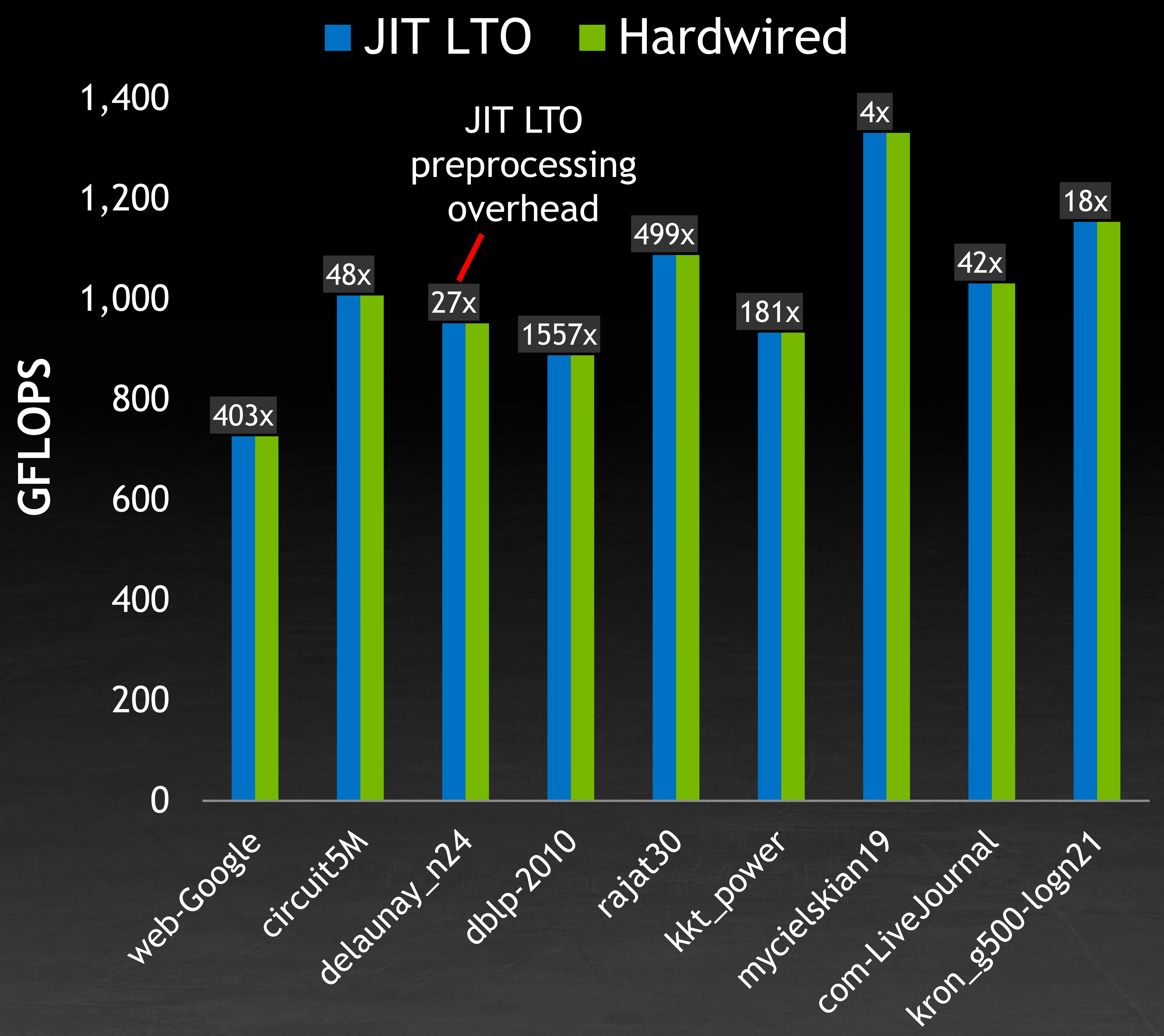
Repeat for MulOp and Epilogue strings

```
cusparseSpMMOpPlan_t plan;
cusparseSpMMOp_createPlan(handle, &plan, opA, opB,
                          matA, matB, matC, CUDA_R_32F,
                          CUSPARSE_SPMM_OP_ALG_DEFAULT,
                          nvvm_add, nvvm_add_size,
                          nvvm_mul, nvvm_mul_size,
                          nvvm_epilogue, nvvm_epilogue_size,
                          &bufferSize);
void* dBuffer;
cudaMalloc(&dBuffer, bufferSize);
cusparseSpMMOp(plan, dBuffer);
```

Performance Evaluation

SpMM JIT LTO vs. Hardwired

Identical code, Intrinsic instructions (Internal only)



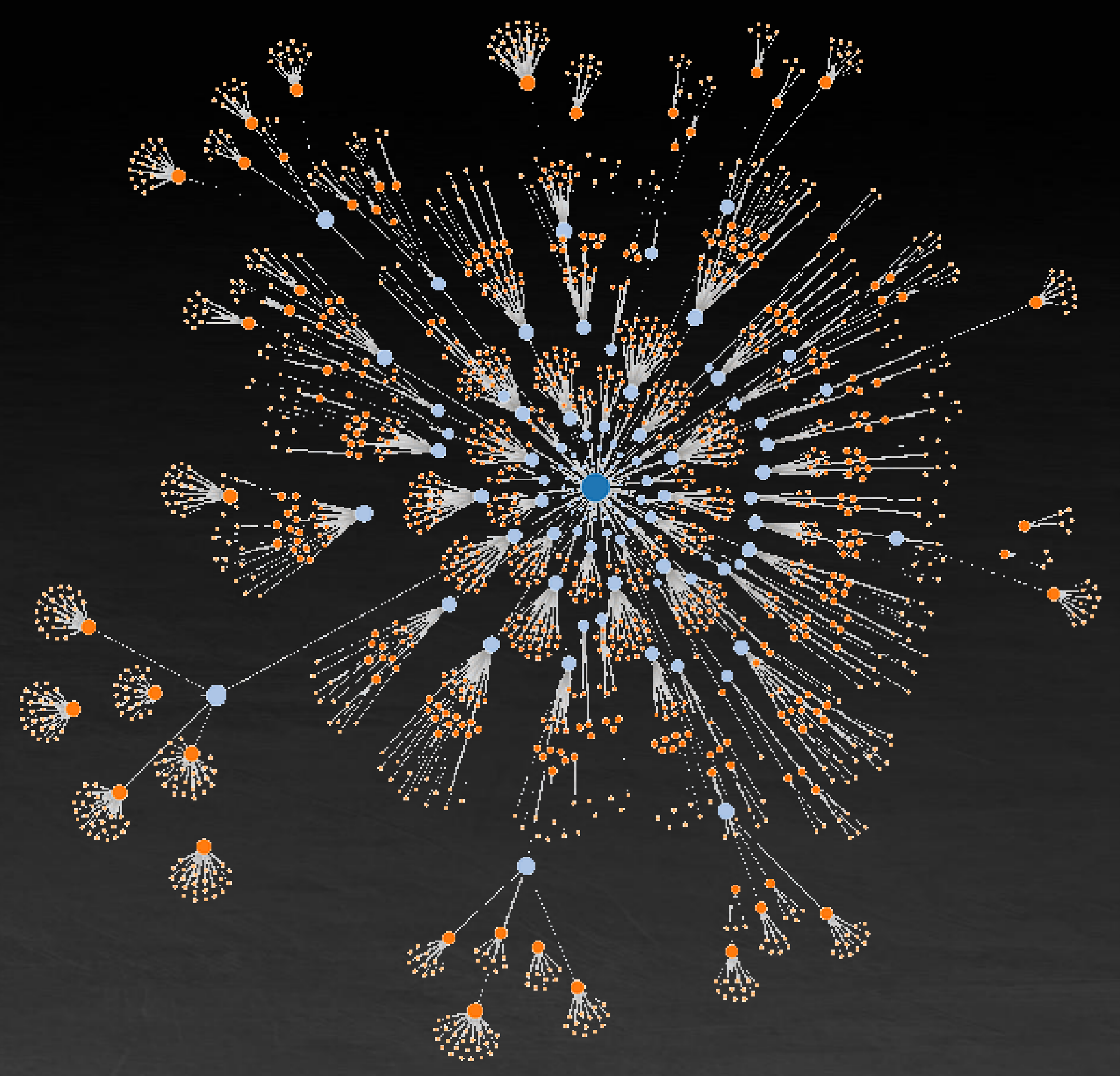
Public JIT LTO APIs

Conclusions and Future Work

cuSPARSE will provide JIT LTO capabilities starting from *CUDA 11.5u1*. The first release provides one routine (SpMM) with custom operators and one algorithm

Take home messages:

- JIT LTO provides zero-overhead compared to (the same identical) hardwired implementation
- Great flexibility improvement
- Need to amortize JIT LTO preprocessing time over multiple runs



Next steps:

- Extend JIT LTO to new routines, e.g. SpMV, SpGEMM
- One-to-one implementations compared to the hardcoded version
- Reduce the time spent in the run-time compile/link phases
- Persistence JIT cache for eliminating the preprocessing time