# Thesis contributions

- APPy: Annotated Parallelism for Python on GPUs
  - [CC24] Parallelize Python loops and tensor expressions on GPUs
- ReACT: Redundancy-Aware Code Generation for Tensor Expressions
  - [PACT22] Redundancy elimination when fusing sparse/dense tensor operators
- Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels
  - [Onward!20] Compile Python/NumPy to C++ with high-level optimizations



Georgia Tech

# Problem statement: desired input and output

- Desired input: operator program in Python (can be sparse)

```python
def sddmm(sp_A, B, C):
    return sp_A * (B @ C)

def spmm_mm(sp_A, B, C):
    return sp_A @ (B @ C)

def norm_row(sp_A):
    return sp_A / sum(sp_A, axis=1)
```

- Desired output: fused CPU kernel with reduced redundant memory accesses and computations

```c
    #pragma omp parallel
  {

    auto T = new double [D2_dimension]();
    int jT = 0;
    #pragma omp for schedule(static)
    for (int32_t i = 0; i < C1_dimension; i++) {
        for (int32_t k = 0; k < D1_dimension; k++) {
            int32_t kC = i * C2_dimension + k;
            jT = 0;
            for (int32_t jB = B2_pos[i]; jB < B2_pos[(i + 1)]; jB++) {
                int32_t j = B2_crd[jB];
                int32_t jD = k * D2_dimension + j;
                T[jT] += C_vals[kC] * D_vals[k * D2_dimension + j];
                jT++;
            }
        }

        jT = 0;
        for (int32_t jB = B2_pos[i]; jB < B2_pos[(i + 1)]; jB++) {
            int32_t j = B2_crd[jB];
            A_vals[jB] += B_vals[jB] * T[jT];
            T[jT] = 0;
            jT++;
        }
      }

     delete T;
   }
```

# Limitations with State-of-the-art

- TACO
  - A code generator for arbitrary sparse/dense tensor algebra expressions
  - **maximal fusion** is implicit during code generation
- Limitations
  - Maximal fusion may introduce some types of redundant memory accesses and computations
  - Maximal fusion cannot properly fuse certain reduction expressions

```
def sddmm(sp_A, B, C):
    return sp_A * (B @ C)

def spmm_mm(sp_A, B, C):
    return sp_A @ (B @ C)

def norm_row(sp_A):
    return sp_A / sum(sp_A, axis=1)
```

Maximal fusion does not work because
it requires the “/” operator to be distributive over a summation

# Redundancy types identified

- **Type 1** (Reduction Redundancy): When multiple multiply-add operations are performed instead of multiple adds followed by a single multiply (distributive law).
- **Type 2** (Loop-Invariant Redundancy): When a loop invariant expression is introduced (could be invariant in a non-innermost loop) due to maximum fusion.
- **Type 3** (Load-Store Redundancy): When some values are stored and loaded in separate loops, and the loads/stores can be eliminated after fusion --- a classical benefit of loop fusion.
- **Type 4** (Dead-Value Redundancy): When some values are computed but not used later on (e.g., when multiplying with 0s in a sparse tensor) --- another classical benefit of loop fusion.

# (Type 1) Reduction redundancy

Input: c = b * sum(A, axis=1)

## With redundancy (due to maximal fusion)

```
for (int i = 0; i < NI; i++) {
  double s = 0;
  double bi = b[i];
  for (int j = 0; j < NJ; j++) {
    s += A[i,j] * bi;
    ...
  }
  ...
}
```

## Without redundancy

```
for (int i = 0; i < NI; i++) {
  double s = 0;
  for (int j = 0; j < NJ; j++) {
    s += A[i,j];
    ...
  }
  s = s * B[i];
  ...
}
```

Reduced number of multiplications in the innermost loop!

# (Type 2) Loop-Invariant redundancy

Input: A = (B + E) * (C @ D)

**With redundancy (due to maximal fusion)**

```
for (int i = 0; i < NI; i++)
  for (int k = 0; k < NK; k++)
    for (int j = 0; j < NJ; j++)
      A[i,j] += (B[i,j] + E[i,j]) * \ (C[i,k] * D[k,j]);
```

**Without redundancy**

```
double* T = new double[NJ];
for (int i = 0; i < NI; i++) {
  for (int j = 0; j < NJ; j++) {
    T[j] = B[i,j] + E[i,j];
  }
  for (int k = 0; k < NK; k++) {
    for (int j = 0; j < NJ; j++) {
      A[i,j] += T[j] * (C[i,k] * D[k,j]);
    }
  }
}
```

B[i,j] + E[i,j] is no longer repeatedly calculated for different *k* iterations!

# (Type 3) Load-Store redundancy

Input: s = sum(A, axis=1); B = A / s[:, None]

## With redundancy (due to no fusion)

```
double* s = new double[NI];
// Operator 1
for (int i = 0; i < NI; i++) {
  s[i] = 0;
  for (int j = 0; j < NJ; j++) {
    s[i] += A[i,j];
  }
}
// Operator 2
for (int i = 0; i < NI; i++) {
  for (int j = 0; j < NJ; j++) {
    B[i,j] = A[i,j] / s[i];
  }
}
```

## Without redundancy

```
// Operator 1 and 2 fused
for (int i = 0; i < NI; i++) {
  double s = 0;
  for (int j = 0; j < NJ; j++) {
    s += A[i,j];
  }

  for (int j = 0; j < NJ; j++) {
    B[i,j] = A[i,j] / s;
  }
}
```

A[i,j] and s[i] now have reduced reuse distance, which leads to better locality!

# (Type 4) Dead-Value redundancy

Input: B = where(A < 0, alpha * A, A)

## With redundancy (due to no fusion)

```
// Operator 1
double* tmp = new double[NI];
for (int i = 0; i < NI; i++) {
  tmp[i] = alpha * A[i];
}
// Operator 2
for (int i = 0; i < NI; i++) {
  if (A[i] < 0) {
    B[i] = tmp[i];
  }
  else {
    B[i] = A[i];
  }
}
```

Not all values in array tmp are useful!

## Without redundancy

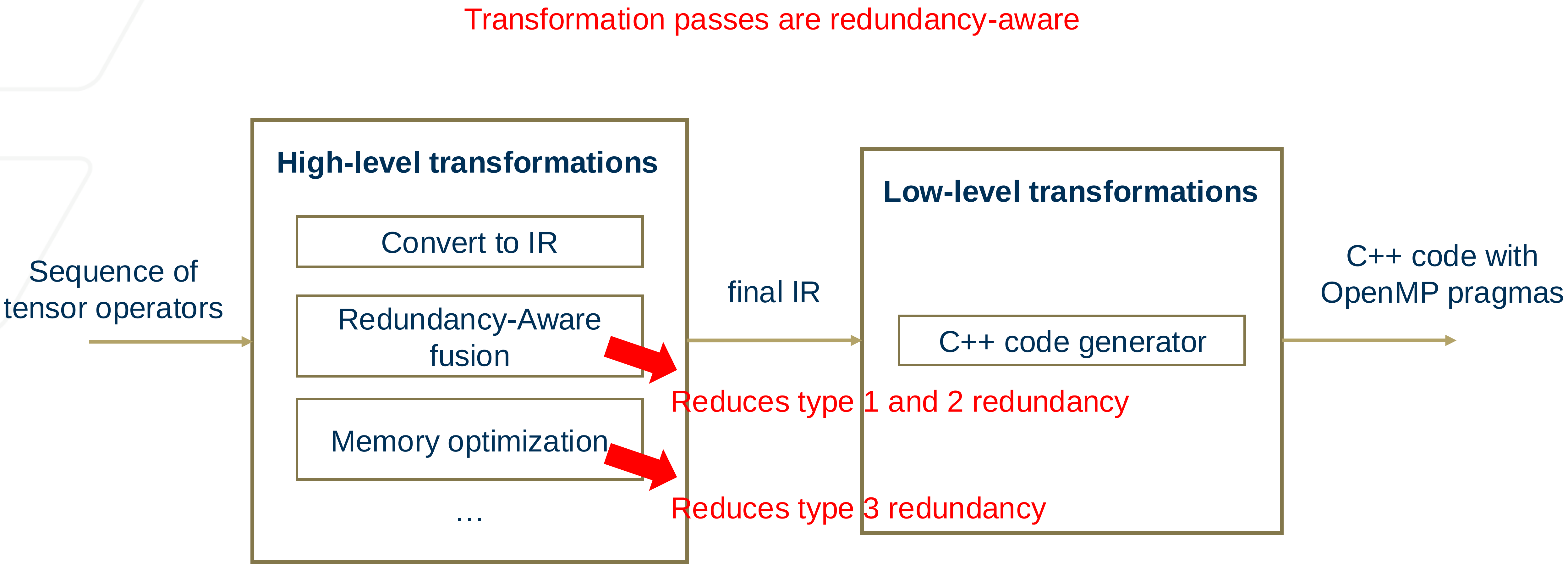
```
// Operator 1 and 2 fused
for (int i = 0; i < NI; i++) {
  if (A[i] < 0) {
    B[i] = alpha * A[i];
  }
  else {
    B[i] = A[i];
  }
}
```

The use of tmp is now eliminated, which reduces redundant computations and memory accesses!

# Redundancies eliminated by each approach

| Redundancy type | ReACT (this work) | TACO | SciPy |
|---|---|---|---|
| Reduction (type 1) | Yes | No | Yes |
| Loop invariant (type 2) | Yes | No | Yes |
| Load store (type 3) | Yes | Partially | No |
| Dead value (type 4) | Yes | Yes | No |

# How is ReACT able to reduce these redundancies?



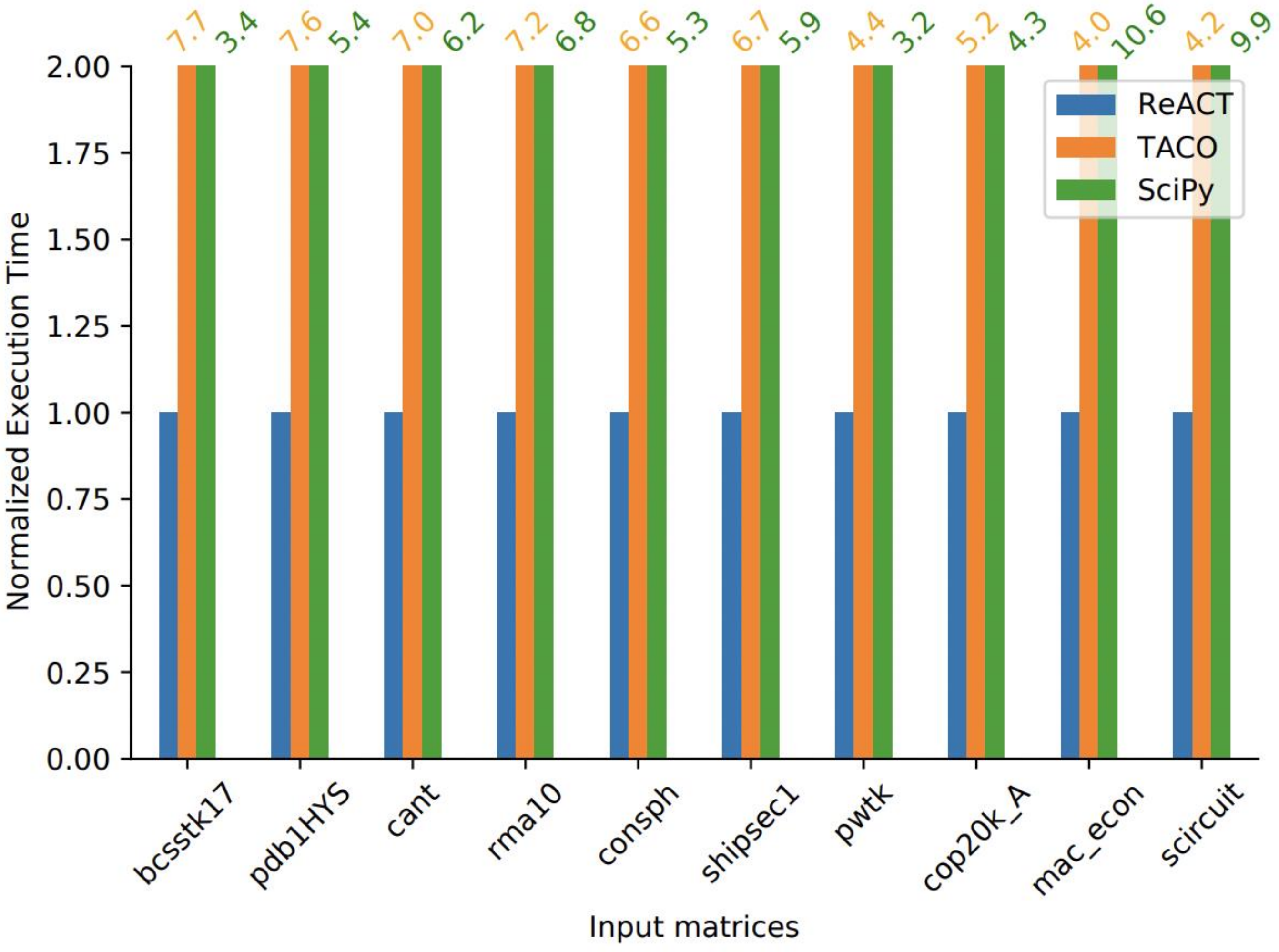
# Performance evaluation

- Test machine
  - 16-core Intel(R) Xeon(R) 2.20GHz CPU
  - OMP_NUM_THREADS is set to 16
- Kernels (all kernels have at least 2 operators)
  - SpMM-MM (sparse-dense matmul followed by dense matmul)
  - SDDMM/Masked MM (a dense matmul followed by a dense-sparse element-wise mul)
  - Sparse-softmax (row-wise softmax on a sparse matrix)
    - Expressed using basic operators such as exp, sum, divide etc
- Sparse matrices
  - A collection of real-world matrices from SuiteSparse
  - All sparse matrices are in CSR format
- Comparisons
  - ReACT (our approach)
  - TACO (SOTA compiler)
  - SciPy.sparse (SOTA library)



Georgia Tech

# SpMM-MM results – 5.9x faster than TACO




**(b) GNN-kernel1 (NH=256, NJ=16)**

"No" is good here!

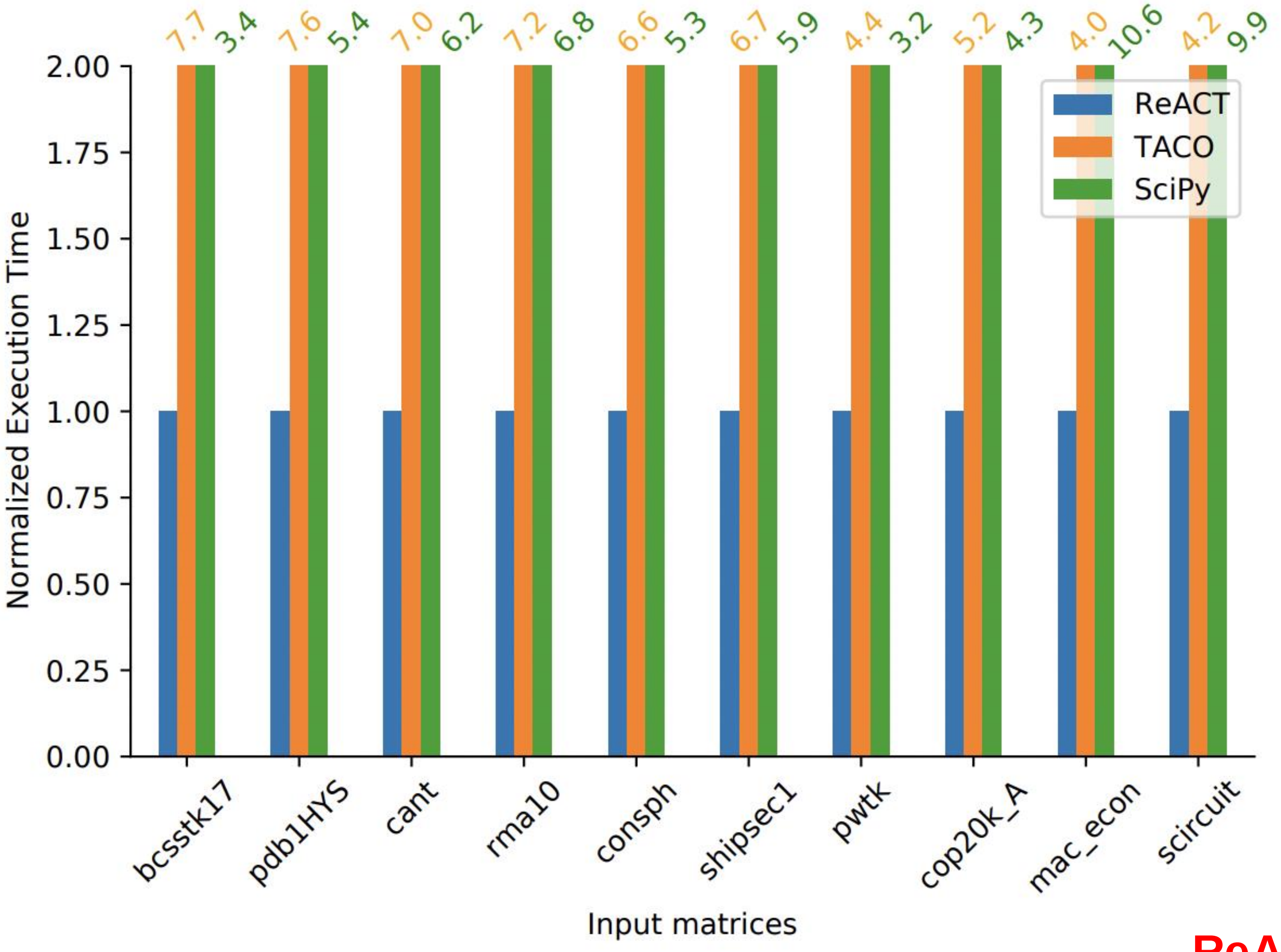
| Redundancy types present | TACO output | ReACT output |
|---|---|---|
| Type 1 | Yes | No |
| Type 2 | Yes | No |
| Type 3 | No | No |
| Type 4 | No | No |

**Code time complexity is reduced from $O(NNZ * NH * NJ)$ (TACO) to $O(NI * NH * NJ)$ (ReACT)**



Georgia Tech

# SpMM-MM results – 5.7x faster than SciPy




(b) GNN-kernel1 (NH=256, NJ=16)

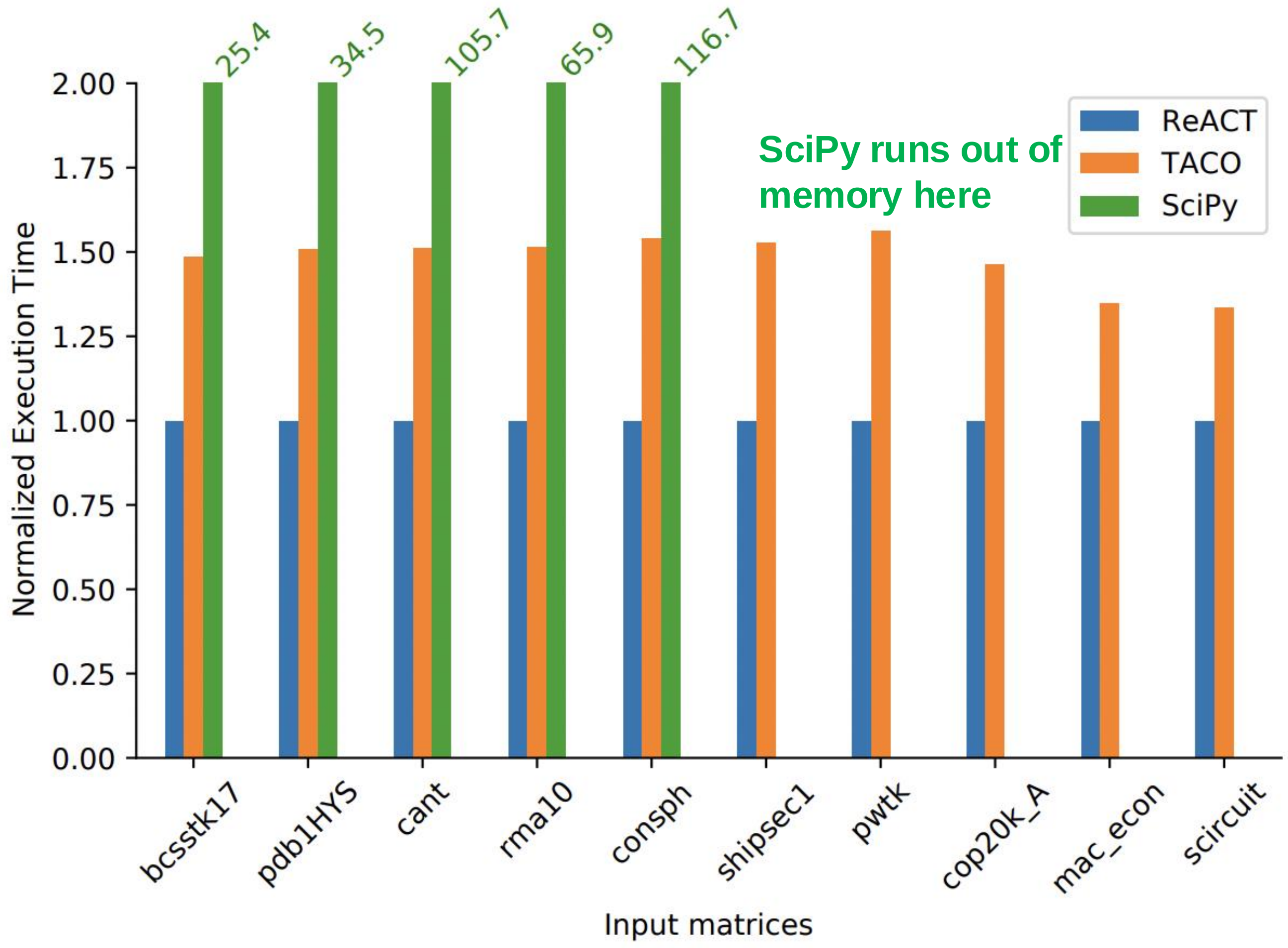
| Redundancy types present | SciPy | ReACT output |
|---|---|---|
| Type 1 | No | No |
| Type 2 | No | No |
| Type 3 | No | No |
| Type 4 | Yes | No |

**ReACT has better locality + more parallelism**
**Note: SciPy uses only a single thread for its SpMM implementation**



Georgia Tech

# SDDMM results – 1.5x faster than TACO

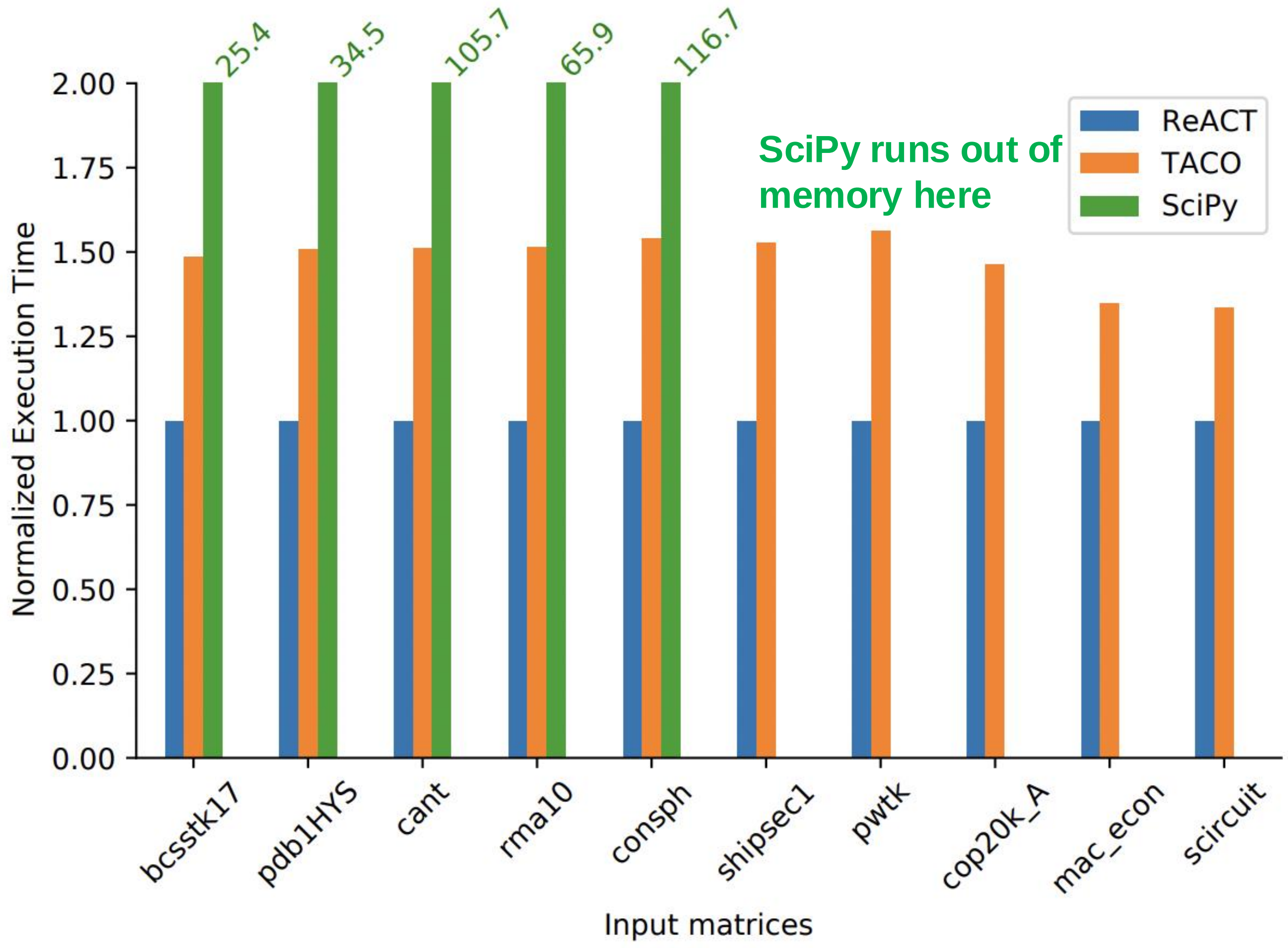



(a) SDDMM (NK=64)

| Redundancy types present | TACO output | ReACT output |
|---|---|---|
| Type 1 | Yes | No |
| Type 2 | No | No |
| Type 3 | No | No |
| Type 4 | No | No |

**Both the amount of memory accesses and computations are reduced by eliminating type 1 redundancy.**

# SDDMM results – 57.3x faster than SciPy




(a) SDDMM (NK=64)

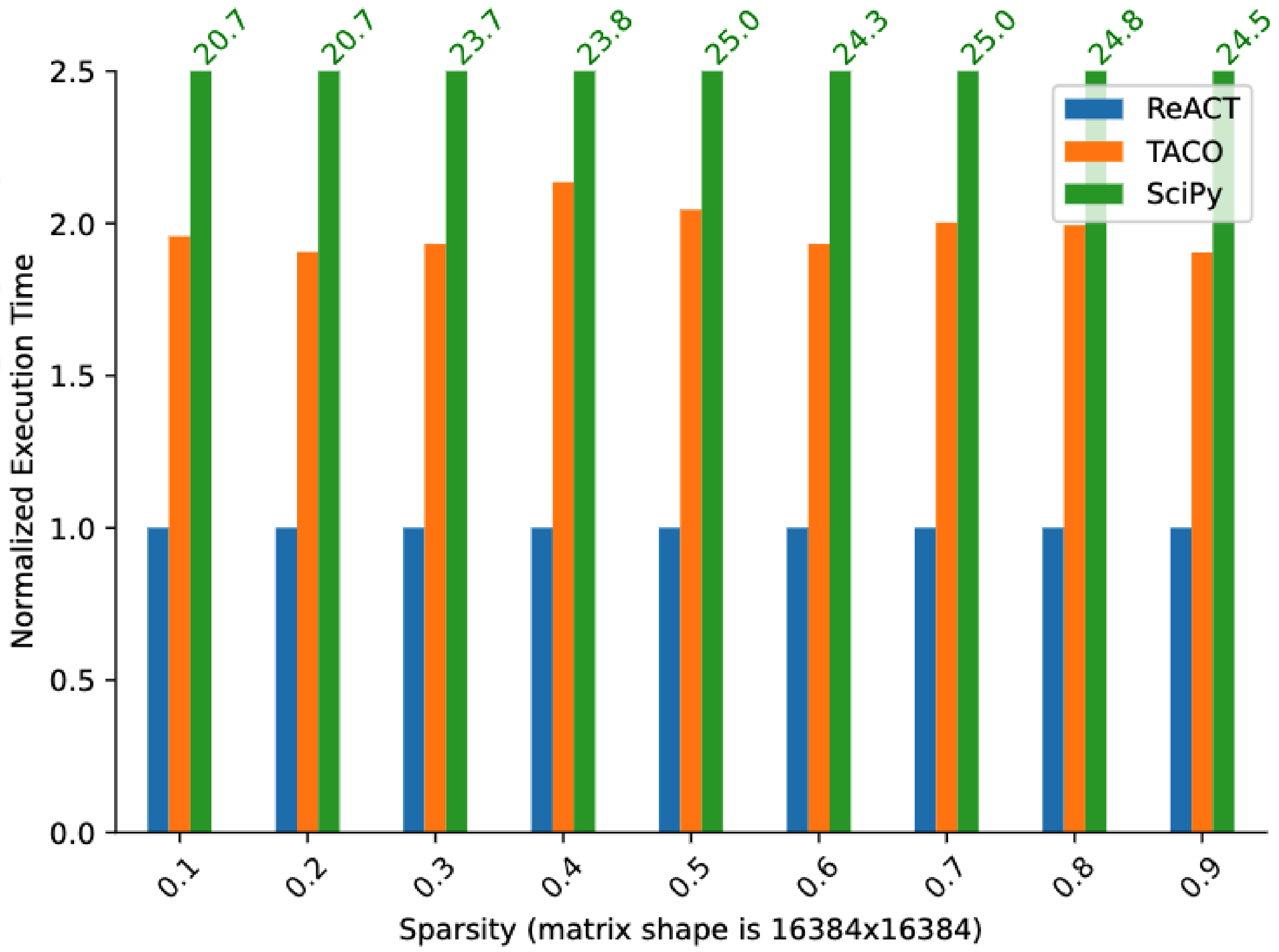
| Redundancy types present | SciPy | ReACT output |
|---|---|---|
| Type 1 | No | No |
| Type 2 | No | No |
| Type 3 | Yes | No |
| Type 4 | Yes | No |

**Many redundant computations are saved by eliminating type 4 (dead value) redundancies**



Georgia Tech

# Sparse-softmax results – 2.0x faster than TACO




| Redundancy types present | TACO output | ReACT output |
|---|---|---|
| Type 1 | No | No |
| Type 2 | No | No |
| Type 3 | Yes | No |
| Type 4 | No | No |

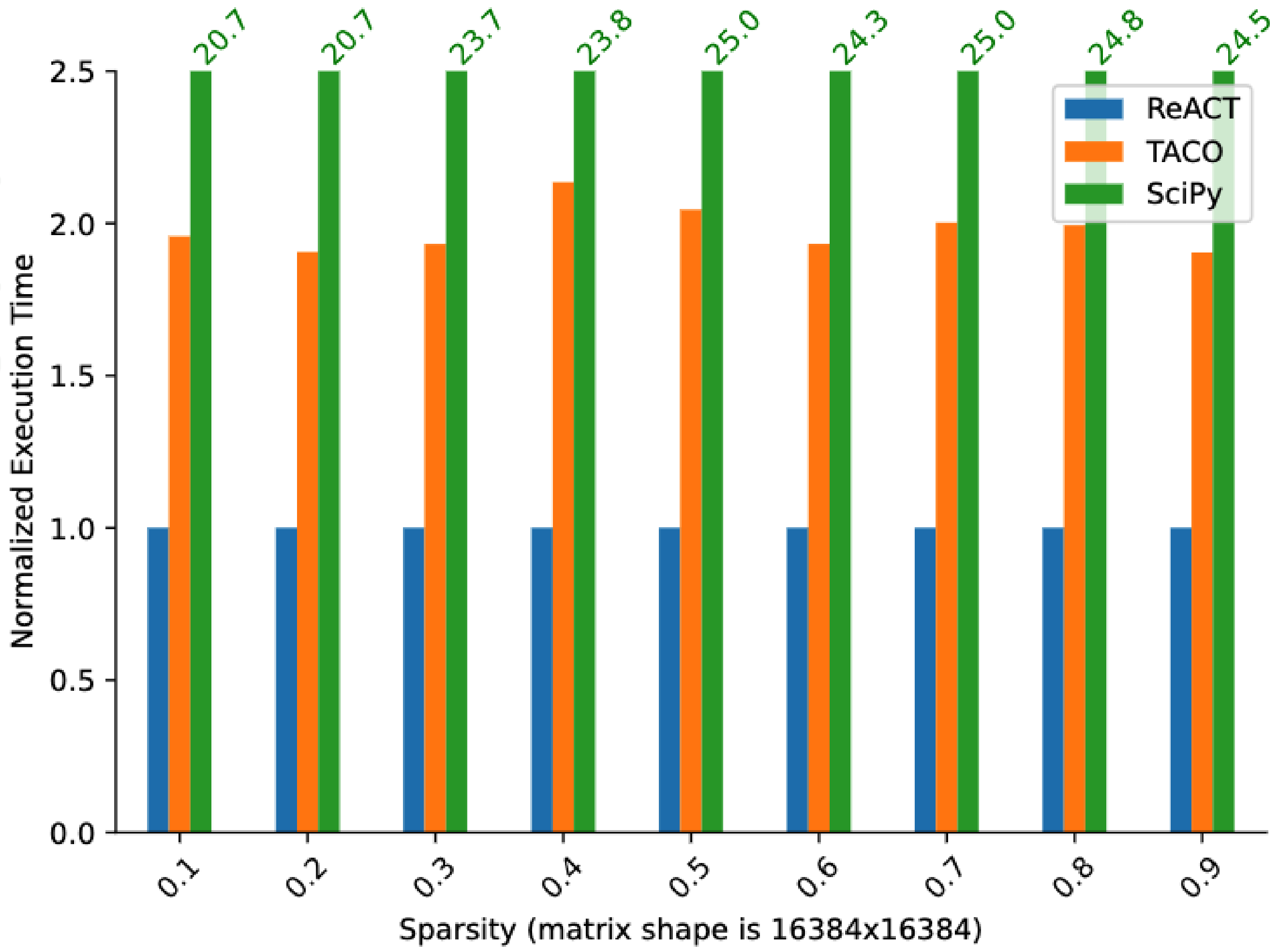
**TACO cannot fuse it into one single kernel while ReACT does, so ReACT has better locality**

**Note: an AMD Ryzen 9 3900X was used for this experiment**



Georgia Tech

# Sparse-softmax results – 23.5x faster than SciPy




| Redundancy types present | TACO output | ReACT output |
|---|---|---|
| Type 1 | No | No |
| Type 2 | No | No |
| Type 3 | Yes | No |
| Type 4 | No | No |

**SciPy's sparse kernels are not parallelized**
**The operations are also not fused**

**Note: an AMD Ryzen 9 3900X was used for this experiment**



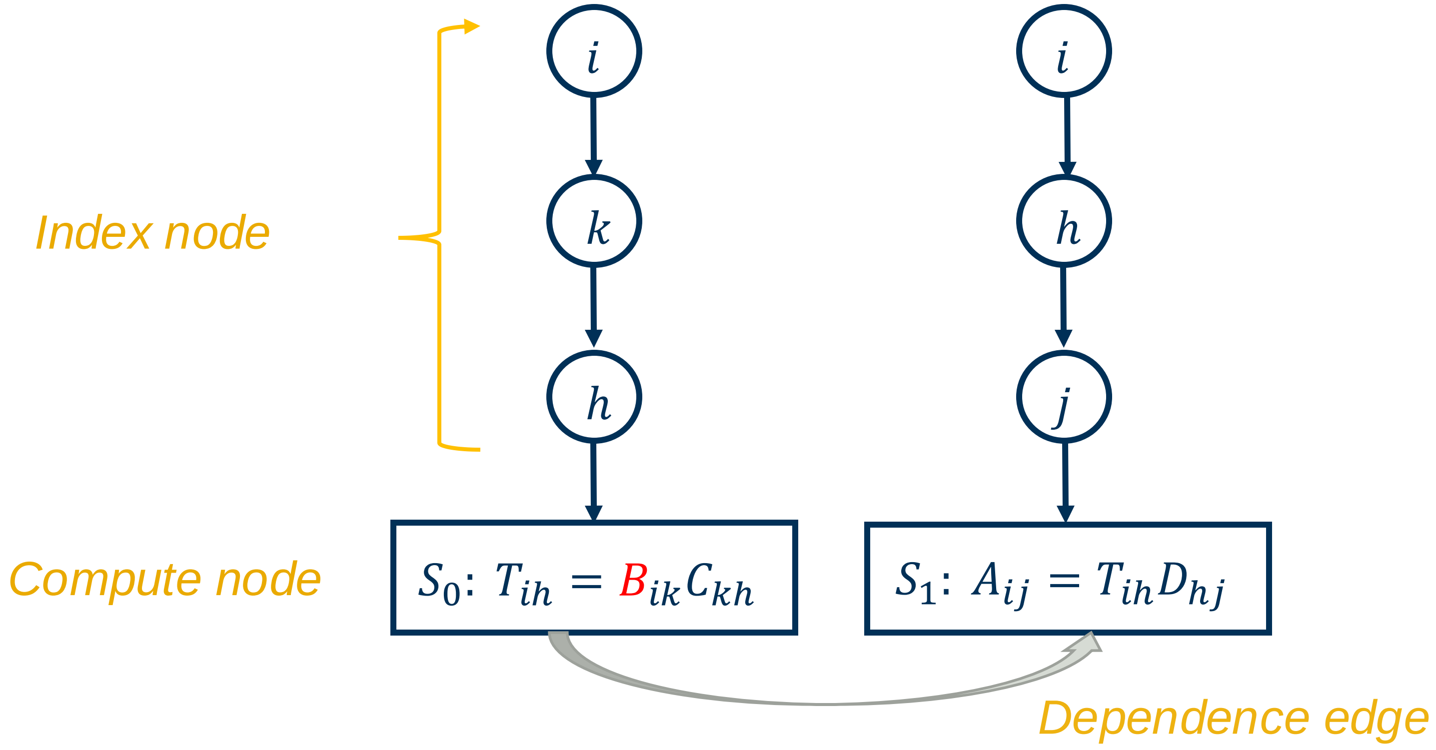
Georgia Tech

# Example: SpMM-MM

- Sparse-dense matmul followed by dense-dense matmul
  - Commonly used in graph neural networks
- Original input expression (sparse matrices are in red, assuming CSR format)
  - Python: $A = B \ @ \ C \ @ \ D$
- Transformations
  - Step 1: convert into *index notation* statements (each statement contains one operator)
    - $S_0$: $T_{ih} = B_{ik} \ @ \ C_{kh}$ (sparse-dense MM)
    - $S_1$: $A_{ij} = T_{ih} \ @ \ D_{hj}$ (dense-dense MM)
    - $T_{ih}$ is compiler-generated temporary variable
  - Step 2: create an *index tree* from the index notation statements
    - Next slide

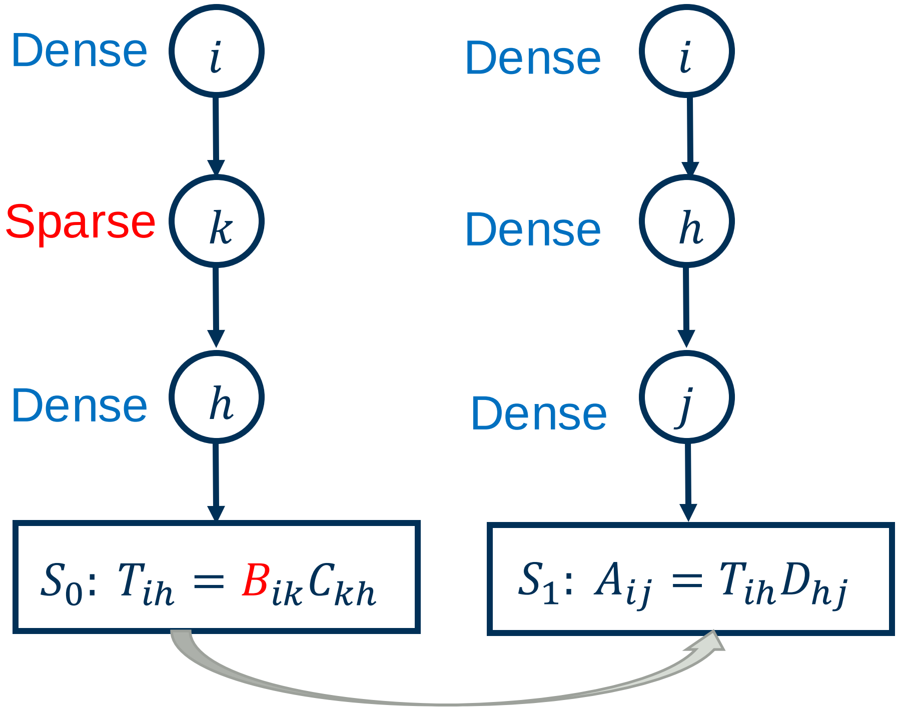
# Index tree of SpMM-MM

- Two operations => create two subtrees
  - $S_0$: $T_{ih} = B_{ik} @ C_{kh}$
  - $S_1$: $A_{ij} = T_{ih} @ D_{hj}$



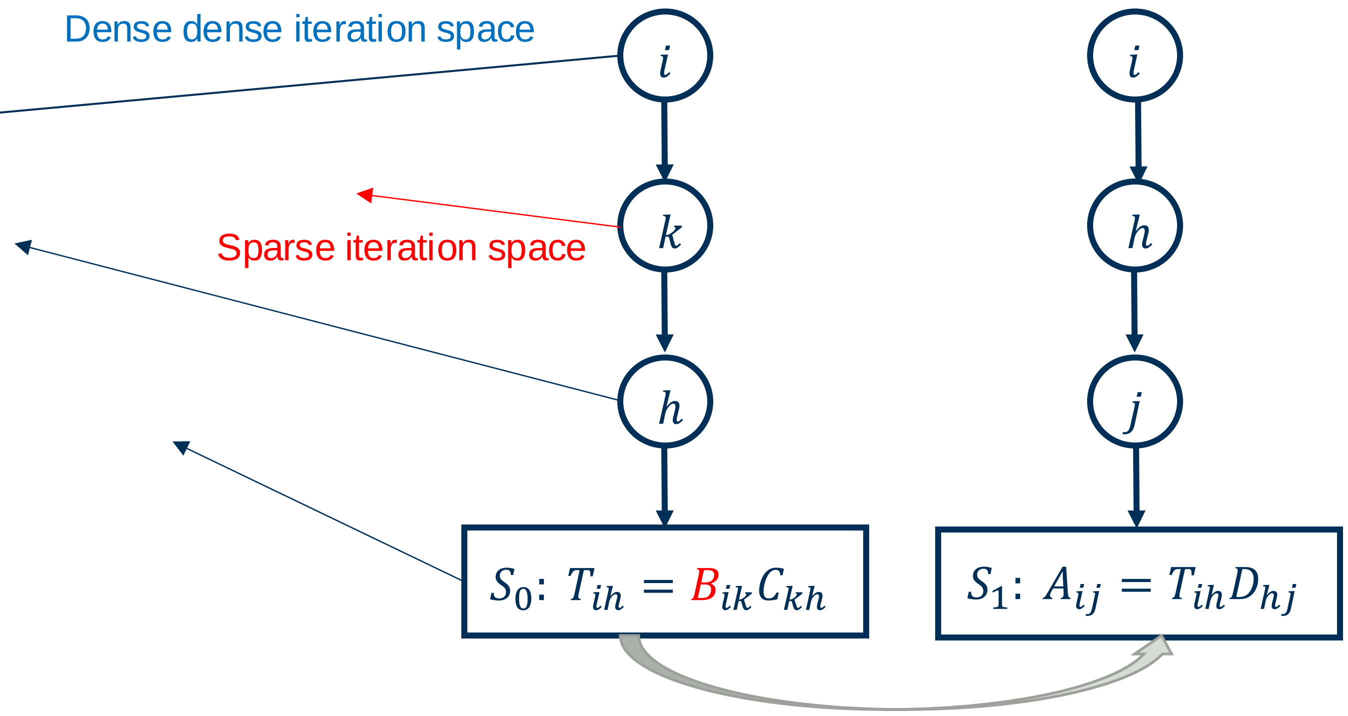
# SpMM-MM index trees

- Annotate each index node as “Dense” or “Sparse”



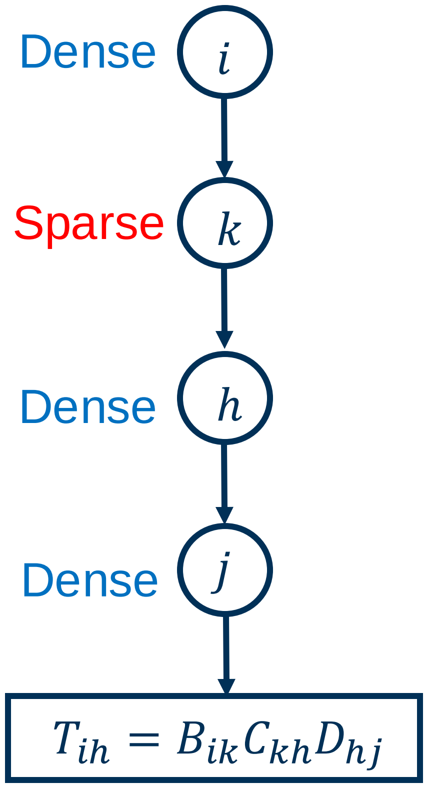
# Index tree corresponding loop structure

```c
for (int i = 0; i < NI; i++) {
    for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
        for (int h = 0; h < NH; h++) {
            ...
            // T[i, h] += B[i, k] * C[k, h]
            T[i, h] += B.vals[k] * C[B.cols[k], h];
            ...
        }
    }
}
```



# SpMM-MM index trees: TACO (maximal fusion )

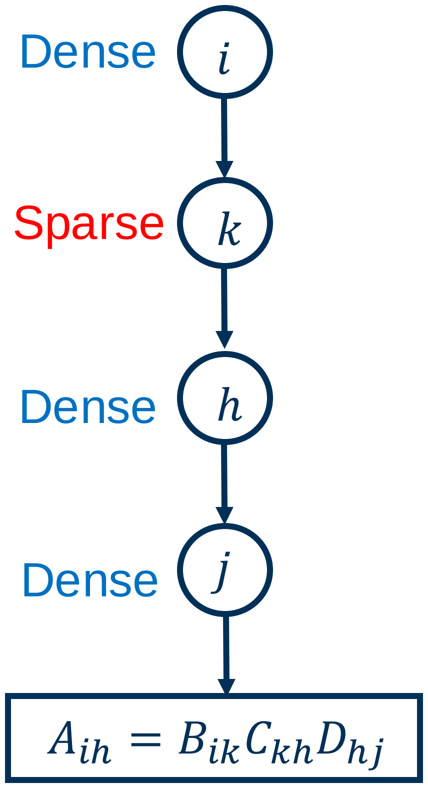
- Time: Bad, $O(NNZ_B * NH * NJ)$
  - Due to type 1 and 2 redundancies
- Intermediate space: Great, $O(1)$
- Locality: Great



# SpMM-MM index trees: TACO (maximal fusion )

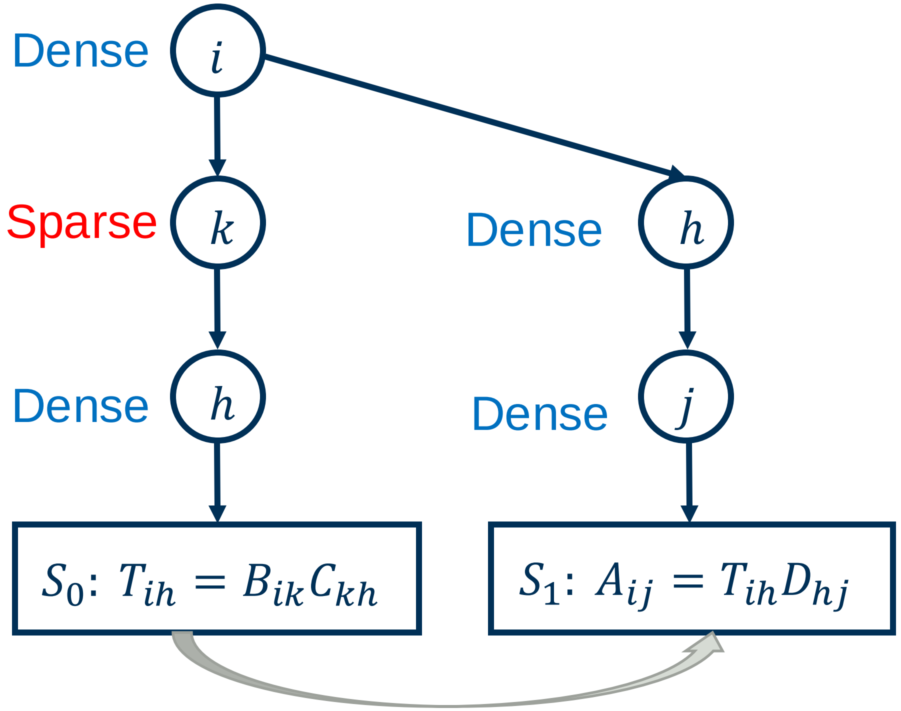
## Generated code

```
for (int i = 0; i < NI; i++) {
    for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
        for (int h = 0; h < NH; h++) {
            for (int j = 0; j < NJ; j++) {
                ...
                // A[i, h] += B[i, k] * C[k, h] * D[h, j]
                A[i, h] += B.vals[k] * C[B.cols[k], h] * D[h, j];
                ...
            }
        }
    }
}
```



# SpMM-MM index trees: ReACT (partial fusion)

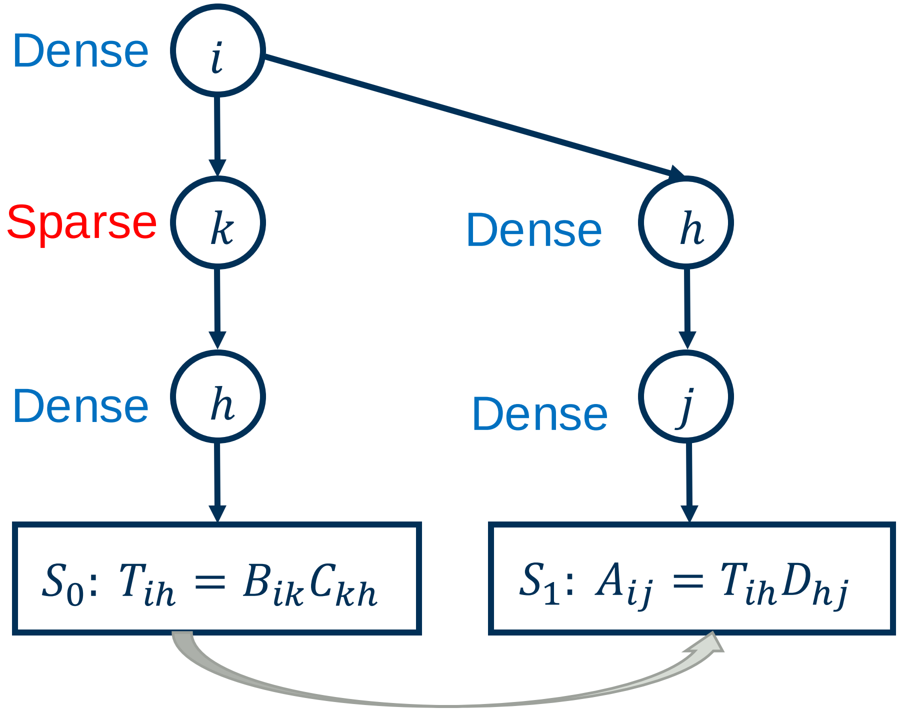
- Time: Good, $O(NNZ_B * NH + NI * NH * NJ)$
  - Typically much smaller than $O(NNZ_B * NH * NJ)$
- Intermediate space: Good, $O(NH)$
  - After memory optimization
- Locality: Good



# SpMM-MM index trees: ReACT (partial fusion)

## Generated code

```
for (int i = 0; i < NI; i++) {
    for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
        for (int h = 0; h < NH; h++) {
            ...
            // T[i, h] += B[i, k] * C[k, h]
            T[h] += B.vals[k] * C[B.cols[k], h];
            ...
        }
    }
    for (int h = 0; h < NH; h++) {
        for (int j = 0; j < NJ; j++) {
            ...
            // A[i, h] += T[i, h] * D[h, j]
            A[i, h] += T[h] * D[h, j];
            ...
        }
        T[h] = 0;
    }
}
```



# ReACT summary

- We identify four common types of redundancies that can occur when generating code for a sequence of dense/sparse tensor operations
- We introduce ReACT, which consists of a set of redundancy-aware code generation techniques and can generate code with reduced redundancies
- Empirical evaluation on real-world applications such as SDDMM, GNN, Sparse-Softmax, and MTTKRP showed that our generated code with redundancy elimination resulted in 1.1× to orders-of-magnitude performance improvements relative to a state-of-the-art tensor algebra compiler (TACO) and library approaches such as scipy.sparse