Efficient Kernel Generation via Specialization

Tong Zhou[1], Animesh Jain[2], Vivek Sarkar[1]

[1] Georgia Institute of Technology, [2] Meta

Introduction

Input: A sequence of dense or sparse (batched) matrix multiplications, point-wise and reduction operations.

Output: Efficient (fused) CPU or GPU kernel(s).

Challenge: It's difficult for general code generation approaches to generate cuBLAS- or Intel MKL- competitive kernels, while calling library functions introduces redundant memory accesses across operators, and requires manual effort to optimize.

Key Insights

Decoupling code generation into a **high-level** phase, which does fusion and tiling and a **low-level** phase, which pattern matches and replaces sub-kernels with hand-optimized primitives, such as MKL or cuBLAS GEMM.

A Step-By-Step Transformation Example

SDDMM-SpMM: A Common Pattern in GNN and Transformers

0 $A = B.multiply(C @ D) @ E$ ➡ Input

```
for i in range(NI):
    for j in sparse_range(indices_of_row(i)):
        for k in range(NK):
            T[i,j] += C[i,k] * D[k,j]
        T1[i,j] = T[i,j] * B[i,j]

for i in range(NI):
    for h in range(NH):
        for j in sparse_range(indices_of_row(i)):
            A[i,h] = T1[i,j] * E[j,h]
```

1

```
for i in range(NI):
    for j in sparse_range(indices_of_row(i)):
        for k in range(NK):
            T[i,j] += C[i,k] * D[k,j]
        T1[i,j] = T[i,j] * B[i,j]

    for h in range(NH):
        for j in sparse_range(indices_of_row(i)):
            A[i,h] = T1[i,j] * E[j,h]
```

2

```
for i in range(0, NI, BI):
    for j in sparse_range(indices_of_row(i)):
        for k in range(0, NK, BK):
            for ii in range(i, i+BI):
                for jj in range(j, j+BJ):
                    for kk in range(k, k+BK):
                        T[ii,jj] += C[ii,kk] * D[kk,jj]

        for ii in range(i, i+BI):
            for jj in range(j, j+BJ):
                T1[ii,jj] = T[ii,jj] * B[ii,jj]

    for h in range(0, NH, BH):
        for j in sparse_range(indices_of_row(i)):
            for ii in range(i, i+BI):
                for hh in range(h, h+BH):
                    for jj in range(j, j+BJ):
                        A[ii,hh] = T1[ii,jj] * E[jj,hh]
```

3

```
for i in range(0, NI, BI):
    k = 0
    h = 0
    for j in sparse_range(indices_of_row(i)):
        for ii in range(i, i+BI):
            for jj in range(j, j+BJ):
                for kk in range(k, k+BK):
                    T[ii,jj] += C[ii,kk] * D[kk,jj]

        for ii in range(i, i+BI):
            for jj in range(j, j+BJ):
                T1[ii,jj] = T[ii,jj] * B[ii,jj]

        for ii in range(i, i+BI):
            for hh in range(h, h+BH):
                for jj in range(j, j+BJ):
                    A[ii,hh] = T1[ii,jj] * E[jj,hh]
```

4

```
for i in range(0, NI, BI):   # in parallel
    k = 0
    h = 0
    for j in sparse_range(indices_of_row(i)):
        MATMUL(T[i:i+BI, j:j+BJ], C[i:i+BI, k:k+BK], D[k:k+BK, j:j+BJ])
        PW_MUL(T1[i:i+BI, j:j+BJ], T[i:i+BI, j:j+BJ], B[i:i+BI, j:j+BJ])
        MATMUL(A[i:i+BI, h:h+BH], T1[i:i+BI, j:j+BJ], E[j:j+BJ, h:h+BH])
```

5

SpMM-MM Performance Results

Input	Shape	F+S+T	F+S	Fusion	SciPy
cora	2708x2708x1433x7	1.0	1.4222	2.9221	1.1259
citeseer	3312x3312x3703x6	1.0	1.4426	3.5333	3.0085
citeseer_full	4230x4230x602x6	1.0	1.2997	2.8927	1.0954
flickr	7575x7575x12047x9	1.0	1.0643	1.6490	1.4883
amazon_photo	7650x7650x745x8	1.0	1.0883	1.5452	1.0405
amazon_cs	13752x13752x767x10	1.0	1.1310	1.5928	1.7458
dblp	17716x17716x1639x4	1.0	1.2193	2.2634	2.8473
coauthor_cs	18333x18333x6805x15	1.0	1.3497	2.3276	1.5753
pubmed	19717x19717x500x3	1.0	1.1444	2.0229	2.6001
cora_full	19793x19793x8710x70	1.0	3.9991	6.8913	1.8226

Table 1: Normalized execution time on an 8-core AMD Ryzen 7 processor. "F+S+T" stands for fusion + specialization + tiling.

Conclusion

By combing high-level fusion/tiling and low-level specialization, we can generate high-performance kernels that match hand-crafted fused kernels and achieve performance improvement over pure library-based approaches.

References

1. **ReACT: Redundancy-Aware Code Generation for Tensor Expressions.** Tong Zhou, et al.
2. **GPU Kernels for Block-Sparse Weights.**
3. Scott Gray, et al