June 29, 2017

LLVM FUSING KERNEL COMPILER DESIGN

Christopher Daniel Leary

Robert Coleman Springer IV

LLVM FUSING KERNEL COMPILER DESIGN

BACKGROUND

A computational graph defines sequences of operations by the types of operations, the data that is input into and output from each operation, and computational dependencies. A compiler can translate a computational graph to produce compiled code that is executable on devices.

SUMMARY

A compiler can include a transformation component that translates computational graph operations into device-specific operations. The transformation component can translate a sequence of computational graph operations into a fused sequence of device-specific operations that optimize the performance of the operations on the device.

DESCRIPTION OF DRAWINGS

FIG. 1 illustrates a computational graph that contains multiple operations to add two vectors.

FIG. 2 illustrates an example computational graph that includes complex and simple operations, where an operation may be fused into the input location of a complex operation.

FIG. 3 illustrates an example computational graph that includes simple and complex operations, where an operation may be fused into the output location of a complex operation.

DETAILED DESCRIPTION

A compiler can include a transformation component that translates computational graph operations into optimized device-specific operations. The transformation component can map computational graph operations into LLVM intermediate representation (IR) operations and chain the LLVM IR operations together to produce a sequence of operations that have faster

execution times on the devices than operations that are not chained.

A computational graph expresses computations, e.g. of a machine learning model, with nodes representing operations and directed edges representing dependencies between operations. An incoming edge to a node represents a flow of an input into the node, i.e., an input argument to the operation represented by the node. If all arguments required for an operation are available to the operation node, the node is enabled and can be executed.

An outgoing edge from a node represents a flow of an output of the operation represented by the node to be used as an input to an operation represented by another node. Thus, a directed edge connecting a first node in the graph to a second node in the graph indicates that an output generated by the operation represented by the first node is used as an input to the operation represented by the second node.

The operations represented in the computational graph can be linear algebraic operations, e.g., matrix multiply, neural network operations, or operations for a different kind of machine learning model. For example, FIG. 1 illustrates a computational graph that contains multiple operations to add four vectors.

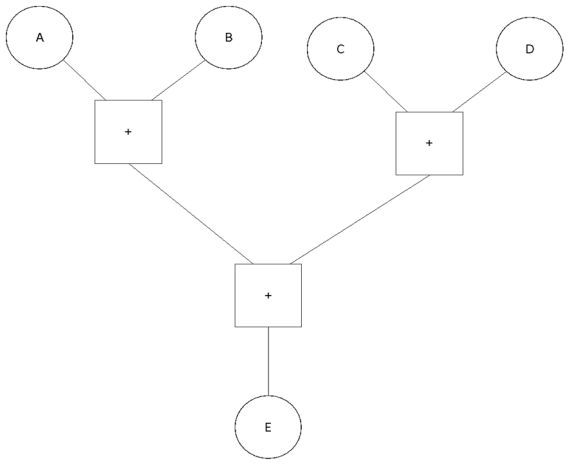

FIG. 1

Example operations to create the computational graph of FIG. 1 is illustrated in Table 1.

```
%1 = PARAM A
%2 = PARAM B
%3 = ADD %1, %2
%4 = PARAM C
%5 = PARAM D
%6 = ADD %3, %5
```

Table 1

Typically, combining multiple operations requires additional implicit output locations. For example, as illustrated in Table 2, the compiler code to create the computational graph as compiler operations contains multiple temporary output locations prior to the final explicit output operation.

```
%1 = [Sequence of IR to load @threadIdx.x; omitted.]
%2 = getelementptr float* %a, i64 %1
%3 = load float* %2; Load a[index]
%4 = getelementptr float* %b, i64 %1
%5 = load float* %4; Load b[index]
%6 = fadd %3, %5
%7 = getelementptr float %temp_1, %1
store float %6, float *%7 ; Store to temp buffer %1
%8 = getelementptr float* %c, i64 %1
%9 = load float* %8; Load c[index]
%10 = getelementptr float* %d, i64 %1
%11 = load float* %10 ; Load d[index]
%12 = fadd %9, %11
%13 = getelementptr float %temp_2, %1
store float %12, float* %13 ; store to temp buffer #2
%14 = getelementptr float %temp_1, %1
%15 = load float* %14; Load temp_buffer_1[index]
%16 = getelementptr float %temp_2, %1
%17 = load float* %16, %2; Load temp_buffer_2[index]
%18 = fadd %15, %17
%19 = getelementptr float* %c, i64 %1
%20 = store float %18, float* %19; store to e[index]
```

Table 2

As shown in the assignment of %7, the result of the addition of *a* and *b* is stored in a temporary buffer, *temp_1*. The result of the addition of *c* and *d* is stored in temporary buffer,

temp_2. The values in *temp_1* and *temp_2* are added together and stored in *e*. The computational graph is evaluated in a left-to-right postorder fashion.

Complex operations, e.g. a basic linear algebra subprogram (BLAS) routine, are usually too complex for a user to implement directly

with high performance. Instead, they are invoked as device-specific function calls. For example, a computational graph can contain complex operations, GEMM functions, which are GEneral Matrix Multiply operations. These complex operations, e.g. GEMM functions, must be callable while on a specific device.

In order to translate computational graph operations into LLVM IR operations, a compiler needs to create a parse tree from source code. At each node, the compiler converts the computational graph operations into LLVM IR by generating LLVM IR operations, identifying and tracking implicit outputs, and producing block synchronization instructions if necessary to implement the operations of the computational graph.

A compiler parses source code to extract operations and operands. The compiler creates a computational graph by generating nodes representing operations and edges representing dependencies between operations. Each type of operation requires at least one input on which it operates, except for input parameters, which implicitly require no input. The output of the last instruction is sent to an output parameter which only has one input and no output edges. The compiler then traverses the computational graph by starting at the output node and recursively evaluates predecessor nodes left-to-right and then evaluates the operation described at the current node. Evaluation begins with identifying the exact operation to be executed and ensuring that the input operands are compatible for the operation.

For simple operations that can be expressed as LLVM IR operations, the compiler can

generate the operations directly in-line. The compiler collects arguments for the operation, performs the operation, and stores the results of the operation in an output location.

The compiler can emit complex operations by using pre-supplied device functions, e.g. library functions. These operations read their input locations and write their output locations via hidden internal logic. The compiler has to identify the implicit output location for each complex operation.

In order to optimize operations compiled on a device, a compiler can perform *operation fusion* when creating LLVM IR operations. Operation fusion is the process of combining successive operations into a single function, omitting both the function call and the data movement between operations. Operation fusion reduces the need for temporary buffers since only the final result needs to be copied to an output buffer. Operation fusion also reduces memory calls by removing reads from and writes to temporary buffers.

Defining pairs of operations which a compiler can fuse is rather difficult. Operations of addition, subtraction, or negation between two buffers containing data of equal shape can be fused. The compiler can also fuse addition, subtraction, multiplication, or division operations between a scalar and any buffer and map operations over any buffer.

As disclosed, FIG. 1 illustrates a computational graph that contains multiple operations to add two vectors. Table 2 illustrates LLVM IR operations that represent the operations of the computational graph without operation fusion. Without operation fusion, the compiler code requires multiple temporary storage buffers. The transformation component of an example compiler performs operation fusion alleviating the need for these temporary storage buffers. For example, example fused compiler code to create the computational graph of FIG. 1 is illustrated in Table 3.

```
%1 = [Sequence of IR to load @threadIdx.x; omitted.]
%2 = getelementptr float* %a, i64 %1
%3 = load float* %2; Load a[index]
%4 = getelementptr float* %b, i64 %1
%5 = load float* %4; Load b[index]
%6 = fadd %3, %5
%7 = getelementptr float* %c, i64 %1
%8 = load float* %8; Load c[index]
%9 = getelementptr float* %d, i64 %1
%10 = load float* %9; Load d[index]
%11 = fadd %8, %10
%12 = fadd %6, %17
%13 = getelementptr float* %c, i64 %5
store float %12, float* %13; store to e[index]
```

Table 3

In this fusion example, the compiler code of Table 3 does not need the stores of *temp_1* and *temp_2* that the compiler code of Table 2 required since the addition operation results are fused in-line. Additionally, the compiler no longer needs to execute the loads from *temp_1* and *temp_2* that represent the input loading for the final addition. For simple operations, fusion is simply a matter of eliminating the temporary storage buffers between operations and using intermediate results from one operation as input to the next.

Because pre-supplied device functions, such as GEMM functions, can have complex input access patterns, generating fused operations before pre-supplied device function invocations does not provide desired optimization. Instead, to optimize compiler code, the proposed compiler captures a sequence of operations as an LLVM function to be evaluated when an input matrix is read inside a pre-supplied device function and only executes these operations when a unit of input is read.

Since a computational graph created from computational operations is traversed bottom-to-top, each node passes, to its predecessor nodes, the LLVM function indicating where its instructions should be generated when creating fusable regions of operations.

FIG 2 illustrates an example computational graph that includes complex (GEMM) and simple (addition) operations.

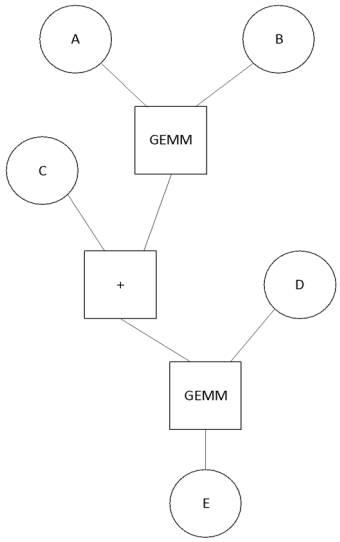

FIG. 2

In this example, a GEMM function is the last operation and produces the kernel's output, E. When traversing its

predecessor nodes, the GEMM function provides each node with the LLVM function to capture the sequence of input operations in a "portable" function. Each operation's intermediate representation instructions should be put into the LLVM function as long as the operations are fusable. If a non-fusable node is emitted, the series of fusable instructions is terminated. Table 4 illustrates the operations to produce the computational graph of FIG. 2. Table 5 illustrates the fused compiler code produced by fusing the inputs to the lower GEMM operation in FIG. 2.

```
%1 = PARAM A
%2 = PARAM B
%3 = MUL %1, %2
%4 = PARAM C
%5 = ADD %4, %3
%6 = PARAM D
%7 = MUL %5, %6
```

Table 4

```
define float @gemm_2_lhs_input (float* %buf, int64, %index) {
%1 = getelementptr float* %buf, i64 %index
%2 = load float* %1; Load output of first GEMM
%3 = getelementptr float* %c, i64 index
%4 = load float* %2; load data in "c"
%5 = fadd %2, %4
ret float %5
}
```

Table 5

In this example, the node representing the "lower" GEMM function is visited and passes an LLVM function to both its predecessor nodes. The "Add" node is visited, and then the C parameter is visited. The C parameter node puts an operation of a load to C into the LLVM function. The compiler then traverses the upper GEMM function node. Since this node is not fusable, the input LLVM function is complete.

FIG. 3 illustrates an example computational graph that includes simple and complex operations.

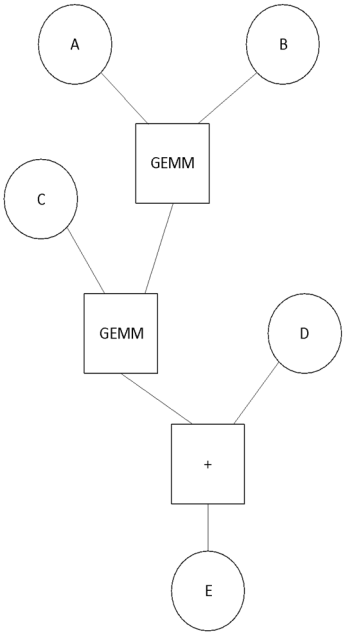

FIG. 3

A complication arises when a final sequence of computational graph operations is fusable as illustrated in FIG. 3. There may be a benefit to executing those last operations when the pre-supplied device function, e.g. the GEMM function, is writing to its output buffer. The last operations must again be captured in an LLVM function. If, in traversing the tree, a non-fusable operation is encountered, then the LLVM function will be generated at that operation's write time. If all operations are fusable, then the last operations should be invoked at kernel top-level. Table 6 illustrates the source code to produce the computational graph of FIG. 3. Table 7 illustrates the compiler code produced by fusing into the output LLVM function when evaluating the computational graph of FIG. 3.

```
%1 = PARAM A
%2 = PARAM B
%3 = MUL %1, %2
%4 = PARAM C
%5 = MUL %4, %3
%6 = PARAM D
%7 = ADD %5, %6
```

Table 6

```
define float &gemm_2_output(float* %buf, int64 %index, float %value)
{
%1 = getelementptr float* %d, i64 %index
%2 = load float* %1
%3 = fadd %value, %2
%4 = get elementptr float* %buf, i64 %index
store float %3, float* %4
}
```

Table 7

ABSTRACT OF THE DISCLOSURE

An example translation component is provided for performing operation fusion optimization. The example component can create a sequence of device-specific compiler operations that performs the operations described by a sequence of computational graph operations in the least amount of time.