LLVM Code Optimisation for Automatic Differentiation

When Forward and Reverse Mode Lead in the Same Direction

Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, Thomas Neumann
{m.schuele,max.springer}@tum.de,{kemper,neumann}@in.tum.de
Technical University of Munich

ABSTRACT

Both forward and reverse mode automatic differentiation derive a model function as used for gradient descent automatically. Reverse mode calculates all derivatives in one run, whereas forward mode requires rerunning the algorithm with respect to every variable for which the derivative is needed. To allow for in-database machine learning, we have integrated automatic differentiation as an SQL operator inside the Umbra database system. To benchmark code-generation to GPU, we implement forward as well as reverse mode automatic differentiation. The inspection of the optimised LLVM code shows that nearly the same machine code is executed after the generated LLVM code has been optimised. Thus, both modes yield similar runtimes but different compilation times.

KEYWORDS

In-Database Machine Learning, Automatic Differentiation, GPU

ACM Reference Format:
Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, Thomas Neumann. 2022. LLVM Code Optimisation for Automatic Differentiation: When Forward and Reverse Mode Lead in the Same Direction. In *Data Management for End-to-End Machine Learning (DEEM'22), June 12, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533028.3533302

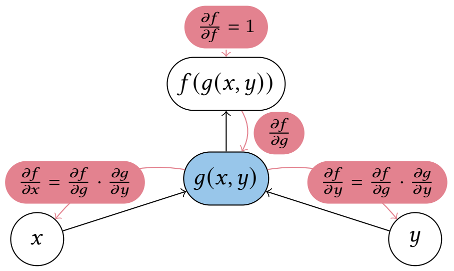

$\frac{\partial f}{\partial f} = 1$
$f(g(x,y))$
$\frac{\partial f}{\partial g}$
$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial y}$
$g(x,y)$
$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial y}$
x
y

Figure 1: Reverse mode automatic differentiation ($\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$).

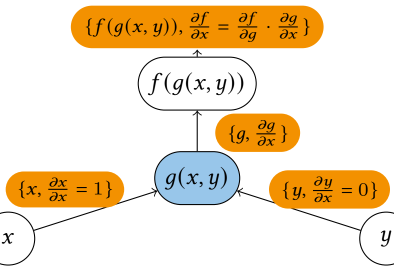

$\{f(g(x,y)), \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}\}$
$f(g(x,y))$
$\{g, \frac{\partial g}{\partial x}\}$
$\{x, \frac{\partial x}{\partial x} = 1\}$
$g(x,y)$
$\{y, \frac{\partial y}{\partial x} = 0\}$
x
y

Figure 2: Forward mode automatic differentiation ($\frac{\partial f}{\partial x}$).

1 INTRODUCTION

In machine learning, optimisation methods such as gradient descent [4, 6] require the derivative of a function to train a model [9, 16]. To derive a function, many machine learning frameworks rely on automatic differentiation by stepwise applying the chain rule. This calculates the derivatives precisely in contrast to numerical differentiation, which only approximates the derivatives as the difference quotient of adjacent points. For automatic differentiation, two modes are possible: Reverse mode (see Figure 1) evaluates the expression first before recursively calculating all derivatives. Whereas forward mode (see Figure 2) calculates a derivative together with the function evaluation in the same pass, but requires one pass per variable for which the derivative is needed. Automatic differentiation is as precise as symbolic differentiation as it computes the derivatives as the product of the partial ones. Laue [12] has proven that caching reduces the "expression swell" of symbolic differentiation, as it calculates the same partial derivatives as forward mode automatic differentiation. In this study, we even show that reverse and forward modes—when optimised—show similar performance due to common subexpressions.

Automatic differentiation has been available in systems developed for machine learning [1–3, 7, 13, 27]. But to train models in SQL [5, 8, 11, 14, 17, 28–31] without the need for manual derivation, we have created a derivation operator in the Umbra database system [10, 15, 18], which generates the derivatives during compile time (Listing 1), and a gradient descent operator, which in addition offloads training to GPU [19–26]. In this study, we benchmark the GPU implementation by comparing forward as well as reverse mode automatic differentiation. This paper first introduces both forward and reverse mode automatic differentiation (Section 2), before it proceeds with the characteristics for generating and optimising LLVM code for GPU (Section 3). We then evaluate the optimised code with regard to compilation and execution time (Section 4).

```
1  create table data (x float, y float); insert into data ...
2  with recursive gd (id, a, b) as (select 1,1::float,1::float UNION ALL
3   select id+1, a-0.05*avg(2*x*(a*x+b-y)), b-0.05*avg(2*(a*x+b-y))
4   from gd, data where id<5 group by id,a,b) select * from gd order by id;
5  with recursive gd (id, a, b) as (select 1,1::float,1::float UNION ALL
6   select id+1, a-0.05*avg(d_a), b-0.05*avg(d_b)
7   from umbra.derivation(TABLE (select id,a,b,x,y from gd,data where id<5),
8   lambda (x) ((x.a * x.x + x.b - x.y)^2))  group by id,a,b)
9  select * from gd order by id;
```

Listing 1: Gradient descent using recursive tables: manually derived (line 3) and using automatic differentiation (l. 6-8).

2 AUTOMATIC DIFFERENTIATION

Both reverse and forward mode automatic differentiation compute the derivative by applying the chain rule $\frac{\partial f(g)}{\partial x} = \frac{\partial f(g)}{\partial g} \cdot \frac{\partial g}{\partial x}$. As both modes calculate the partial derivatives, their values can be reused to compute all needed derivatives. Also, the evaluated subexpressions can be cached, as they are needed as input to derive a function partially. In both algorithms, major characters for variable names represent tokens of the SQL grammar parsed during the semantic analysis for code generation. Minor ones represent their actual scalar value evaluated during runtime.

Reverse mode first evaluates the function, before it calculates each partial derivative recursively in reverse order (Algorithm 1). The uppermost derivation ($\frac{\partial f}{\partial f}$) is 1, which serves as a seed value z' for backpropagation. All the remaining derivatives are the product of the seed value z' and the partial derivative with the original arguments as input. The algorithm returns void but stores the derivatives in a hash table with the variable name as the key.

Algorithm 1 Reverse Mode

```
function DERIVE($Z, z'$)
    if $Z = X + Y$ then DERIVE($X$,$z'$); DERIVE($Y$,$z'$)
    else if $Z = X - Y$ then DERIVE($X$,$z'$); DERIVE($Y$,$-z'$)
    else if $Z = X \cdot Y$ then DERIVE($X$,$z' \cdot y$); DERIVE($Y$,$z' \cdot x$)
    else if $Z = \frac{X}{Y}$ then DERIVE($X, \frac{z'}{y}$); DERIVE($Y, \frac{-z' \cdot x}{y^2}$)
    else if $Z = X^Y$ then
        DERIVE($X$,$z' \cdot y \cdot x^{y-1}$); DERIVE($Y$,$z' \cdot x^y ln(x)$)
    else if $Z = log_Y(X)$ then
        DERIVE($X, \frac{z'}{x \cdot ln(y)}$); DERIVE($Y, \frac{-z' \cdot ln(x)}{y \cdot ln^2(y)}$)
    else if $isVariable(Z)$ then $\frac{\partial}{\partial z} \leftarrow \frac{\partial}{\partial z} + z'$
```

Forward mode calculates the derivative with respect to one variable while evaluating the function in one pass (Algorithm 2). While evaluating the expression, the algorithm returns for every subexpression a pair of the evaluated value and the derivative. Both values serve as input to calculate the partial derivative for each partial function according to the chain rule. The bottommost arguments are the variables, the derivative with respect to a certain variable is one, for all other variables it is zero.

Algorithm 2 Forward Mode

```
function EVAL($Z, V$)
    if $isVariable(Z)$ then
        if $Z = V$ then return $\{z, 1\}$
        elsereturn $\{z, 0\}$
    else$\{x, x'\} \leftarrow$ EVAL($X$,$V$); $\{y, y'\} \leftarrow$ EVAL($Y$,$V$)
        if $Z = X + Y$ then return $\{x + y, x' + y'\}$
        else if $Z = X - Y$ then return $\{x - y, x' - y'\}$
        else if $Z = X \cdot Y$ then return $\{x \cdot y, x' \cdot y + x \cdot y'\}$
        else if $Z = \frac{X}{Y}$ then return $\{\frac{x}{y}, \frac{x' \cdot y - x \cdot y'}{y'^2}\}$
        else if $Z = X^Y$ (w.r.t. $x$) then return $\{x^y, y \cdot x^{y-1} \cdot x'\}$
```

3 USING LLVM WITH PTX

We want to compare the performance of both forward and reverse mode automatic differentiation when generating LLVM code to run as a GPU kernel. While using forward or reverse mode with multiple kernels boils down to getting the derived expression in two different ways, there are more considerations when using them in LLVM.

Both modes show a difference in their code generation, which has an impact on the performance depending on the specific expression. We, therefore, want to compare the behaviour via a simple example. In this simple example, we have one kernel that computes the derivatives of a linear model. To get a meaningful difference concerning *reusing subexpressions* of both methods, we use a linear model with three weights and two inputs: $w_0 \cdot x_0 + x_1 \cdot w_1 + b$.

As a loss function, we use mean squared error, so the complete expression we want to minimise is (y is the label):

$$L = (w_0 \cdot x_0 + x_1 \cdot w_1 + b - y)^2.$$

We now test this expression with our two automatic differentiation methods. For now, we focus on what code both generate, but we will compare the performance in Section 4. Deriving by hand we get the following for two derivatives:

$$\frac{\partial L}{\partial w_0} = 2 \cdot (w_0 \cdot x_0 + x_1 \cdot w_1 + b - y) \cdot x_0,$$
$$\frac{\partial L}{\partial w_1} = 2 \cdot (w_0 \cdot x_0 + x_1 \cdot w_1 + b - y) \cdot x_1.$$

Figure 3 shows the LLVM IR of the kernel calculating the derivatives of L with regard to w_0 and w_1 using forward mode automatic differentiation. We can see that the load instructions—meaning loading data from memory that is necessary to evaluate the derivatives—have to be repeated after the first store instruction. As we have to load the same memory positions again, we have to redo the floating-point instructions even when the term $2 \cdot (w_0 \cdot x_0 + x_1 \cdot w_1 + b - y)$ could be reused for $\frac{\partial L}{\partial w_0}$ and $\frac{\partial L}{\partial w_1}$.

The reason for the repeated load instructions is that the store might influence the result of the loads. At compile time, the optimiser cannot be sure whether the pointers do or do not correspond to separate memory locations, as the kernel can be called with arbitrary pointers. In the C standard, there is the `restrict` keyword for pointer declarations indicating that memory region accesses through `restrict`-annotated pointers do not interfere with each other. LLVM IR comes with a similar feature called `noalias`. Function parameters that have this attribute also allow the various backends to optimise memory accesses assuming that they do not interfere. This has drastic consequences for the generated, optimised LLVM IR.

In Figure 4 we see that the LLVM IR does not have the problems we identified in Figure 3. We see that the `loads` do not have to be repeated. When checking the floating-point instructions we can also see that the result of $2 \cdot (w_0 \cdot x_0 + x_1 \cdot w_1 + b - y)$ is reused. Between the two stores, we only have one necessary multiplication as a floating-point operation.

However, this is only possible because of the order of the operations as translated by automatic differentiation. Floating-point operations are *normally* not associative and the compiler cannot change their order. But in our case the execution order of floating-point operations is set: $\frac{\partial L}{\partial w_0} = x_0 \cdot (2 \cdot ((((w_0 \cdot x_0) + (x_1 \cdot w_1)) + b) - y))$.

To compare the generated LLVM IR from forward mode to reverse mode, Figure 5 shows the generated code of reverse mode

```
%4 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !range !8
%5 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x(), !range !9
%6 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x(), !range !10
%CUDABuiltin_cpp_97_ = mul i32 %6, %5
%CUDABuiltin_cpp_97_0 = add i32 %CUDABuiltin_cpp_97_, %4
%CodeGen_cpp_1539_ = zext i32 %CUDABuiltin_cpp_97_0 to i64
%Autodiff_cpp_700_ = getelementptr double, double* %arg4, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_2 = getelementptr double, double* %arg5, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_4 = getelementptr double, double* %3, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_6 = getelementptr double, double* %2, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_8 = getelementptr double, double* %0, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_10 = getelementptr double, double* %1, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_519_ = load double, double* %Autodiff_cpp_700_, align 8
%Autodiff_cpp_519_11 = load double, double* %Autodiff_cpp_700_4, align 8
%Autodiff_cpp_519_12 = load double, double* %Autodiff_cpp_700_6, align 8
%Autodiff_cpp_501_ = fmul double %Autodiff_cpp_519_11, %Autodiff_cpp_519_12
%Autodiff_cpp_519_13 = load double, double* %Autodiff_cpp_700_10, align 8
%Autodiff_cpp_519_14 = load double, double* %Autodiff_cpp_700_8, align 8
%Autodiff_cpp_501_15 = fmul double %Autodiff_cpp_519_13, %Autodiff_cpp_519_14
%Autodiff_cpp_493_ = fadd double %Autodiff_cpp_501_, %Autodiff_cpp_501_15
%Autodiff_cpp_493_16 = fadd double %Autodiff_cpp_519_, %Autodiff_cpp_493_
%Autodiff_cpp_519_17 = load double, double* %Autodiff_cpp_700_2, align 8
%Autodiff_cpp_497_ = fsub double %Autodiff_cpp_493_16, %Autodiff_cpp_519_17
%Autodiff_cpp_501_18 = fmul double %Autodiff_cpp_497_, 2.000000e+00
%Autodiff_cpp_501_20 = fmul double %Autodiff_cpp_519_13, %Autodiff_cpp_501_18
%7 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1539_
store double %Autodiff_cpp_501_20, double* %7, align 8
%Autodiff_cpp_519_21 = load double, double* %Autodiff_cpp_700_, align 8
%Autodiff_cpp_519_22 = load double, double* %Autodiff_cpp_700_4, align 8
%Autodiff_cpp_519_23 = load double, double* %Autodiff_cpp_700_6, align 8
%Autodiff_cpp_501_24 = fmul double %Autodiff_cpp_519_22, %Autodiff_cpp_519_23
%Autodiff_cpp_519_25 = load double, double* %Autodiff_cpp_700_10, align 8
%Autodiff_cpp_519_26 = load double, double* %Autodiff_cpp_700_8, align 8
%Autodiff_cpp_501_27 = fmul double %Autodiff_cpp_519_25, %Autodiff_cpp_519_26
%Autodiff_cpp_493_28 = fadd double %Autodiff_cpp_501_24, %Autodiff_cpp_501_27
%Autodiff_cpp_493_29 = fadd double %Autodiff_cpp_519_21, %Autodiff_cpp_493_28
%Autodiff_cpp_519_30 = load double, double* %Autodiff_cpp_700_2, align 8
%Autodiff_cpp_497_31 = fsub double %Autodiff_cpp_493_29, %Autodiff_cpp_519_30
%Autodiff_cpp_501_32 = fmul double %Autodiff_cpp_497_31, 2.000000e+00
%Autodiff_cpp_501_34 = fmul double %Autodiff_cpp_519_22, %Autodiff_cpp_501_32
%Autodiff_cpp_704_ = add i32 %CUDABuiltin_cpp_97_0, 32768
%CodeGen_cpp_1599_35 = zext i32 %Autodiff_cpp_704_ to i64
%8 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1599_35
store double %Autodiff_cpp_501_34, double* %8, align 8
ret void
```

Figure 3: LLVM IR: forward mode automatic differentiation. GPU-specific operations to determine memory positions (red); loading operations (blue); storing operations (brown). The remaining lines are floating-point instructions.

```
%3 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !range !8
%4 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x(), !range !9
%5 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x(), !range !10
%CUDABuiltin_cpp_97_ = mul i32 %5, %4
%CUDABuiltin_cpp_97_0 = add i32 %CUDABuiltin_cpp_97_, %3
%CodeGen_cpp_1539_ = zext i32 %CUDABuiltin_cpp_97_0 to i64
%Autodiff_cpp_700_ = getelementptr double, double* %arg4, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_2 = getelementptr double, double* %arg5, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_4 = getelementptr double, double* %2, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_6 = getelementptr double, double* %1, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_8 = getelementptr double, double* %0, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_700_10 = getelementptr double, double* %"01", i64 %CodeGen_cpp_1539_
%Autodiff_cpp_519_ = load double, double* %Autodiff_cpp_700_, align 8
%Autodiff_cpp_519_11 = load double, double* %Autodiff_cpp_700_4, align 8
%Autodiff_cpp_519_12 = load double, double* %Autodiff_cpp_700_6, align 8
%Autodiff_cpp_501_ = fmul double %Autodiff_cpp_519_11, %Autodiff_cpp_519_12
%Autodiff_cpp_519_13 = load double, double* %Autodiff_cpp_700_10, align 8
%Autodiff_cpp_519_14 = load double, double* %Autodiff_cpp_700_8, align 8
%Autodiff_cpp_501_15 = fmul double %Autodiff_cpp_519_13, %Autodiff_cpp_519_14
%Autodiff_cpp_493_ = fadd double %Autodiff_cpp_501_, %Autodiff_cpp_501_15
%Autodiff_cpp_493_16 = fadd double %Autodiff_cpp_519_, %Autodiff_cpp_493_
%Autodiff_cpp_519_17 = load double, double* %Autodiff_cpp_700_2, align 8
%Autodiff_cpp_497_ = fsub double %Autodiff_cpp_493_16, %Autodiff_cpp_519_17
%Autodiff_cpp_501_18 = fmul double %Autodiff_cpp_497_, 2.000000e+00
%Autodiff_cpp_501_20 = fmul double %Autodiff_cpp_519_13, %Autodiff_cpp_501_18
%6 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1539_
store double %Autodiff_cpp_501_20, double* %6, align 8
%Autodiff_cpp_501_34 = fmul double %Autodiff_cpp_519_11, %Autodiff_cpp_501_18
%Autodiff_cpp_704_ = add i32 %CUDABuiltin_cpp_97_0, 32768
%CodeGen_cpp_1599_35 = zext i32 %Autodiff_cpp_704_ to i64
%7 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1599_35
store double %Autodiff_cpp_501_34, double* %7, align 8
ret void
```

Figure 4: Optimised LLVM IR of forward mode automatic differentiation with `noalias` marking pointers.

automatic differentiation, which almost matches the one from Figure 4. Only the last multiply does not happen after the first store.

Previously, we only looked at optimisations that happen on an LLVM IR level. In a second step, we can additionally consider how to optimise the *parallel thread execution* (PTX) code that is generated from our LLVM IR. PTX is the instruction set for CUDA

```
%4 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !range !8
%5 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x(), !range !9
%6 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x(), !range !10
%CUDABuiltin_cpp_97_ = mul i32 %6, %5
%CUDABuiltin_cpp_97_1 = add i32 %CUDABuiltin_cpp_97_, %4
%CodeGen_cpp_1539_ = zext i32 %CUDABuiltin_cpp_97_1 to i64
%Autodiff_cpp_616_ = getelementptr double, double* %arg0, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_3 = getelementptr double, double* %arg00, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_5 = getelementptr double, double* %3, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_7 = getelementptr double, double* %2, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_9 = getelementptr double, double* %0, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_11 = getelementptr double, double* %1, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_519_ = load double, double* %Autodiff_cpp_616_, align 8
%Autodiff_cpp_519_12 = load double, double* %Autodiff_cpp_616_5, align 8
%Autodiff_cpp_519_13 = load double, double* %Autodiff_cpp_616_7, align 8
%Autodiff_cpp_501_ = fmul double %Autodiff_cpp_519_12, %Autodiff_cpp_519_13
%Autodiff_cpp_519_14 = load double, double* %Autodiff_cpp_616_11, align 8
%Autodiff_cpp_519_15 = load double, double* %Autodiff_cpp_616_9, align 8
%Autodiff_cpp_501_16 = fmul double %Autodiff_cpp_519_14, %Autodiff_cpp_519_15
%Autodiff_cpp_493_ = fadd double %Autodiff_cpp_501_, %Autodiff_cpp_501_16
%Autodiff_cpp_493_17 = fadd double %Autodiff_cpp_519_, %Autodiff_cpp_493_
%Autodiff_cpp_519_18 = load double, double* %Autodiff_cpp_616_3, align 8
%Autodiff_cpp_497_ = fsub double %Autodiff_cpp_493_17, %Autodiff_cpp_519_18
%Autodiff_cpp_317_19 = fmul double %Autodiff_cpp_497_, 2.000000e+00
%Autodiff_cpp_302_ = fmul double %Autodiff_cpp_519_14, %Autodiff_cpp_317_19
%Autodiff_cpp_302_23 = fmul double %Autodiff_cpp_519_12, %Autodiff_cpp_317_19
%7 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1539_
store double %Autodiff_cpp_302_, double* %7, align 8
%Autodiff_cpp_623_ = add i32 %CUDABuiltin_cpp_97_1, 32768
%CodeGen_cpp_1599_26 = zext i32 %Autodiff_cpp_623_ to i64
%8 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1599_26
store double %Autodiff_cpp_302_23, double* %8, align 8
ret void
```

Figure 5: LLVM IR: reverse mode automatic differentiation.

programming, which the graphics driver compiles into binary code for GPUs. As explained before, code optimisation *normally* can neither combine or change the order of the floating-point operations. However, we can relax these rules and thus allow the optimiser to reorder and replace floating-point operations.

To allow this relaxation, LLVM IR provides *fast math flags* to specify how the optimiser is allowed to modify floating-point operations. The two flags of interest are `reassoc` and `contract`. `reassoc` allows the optimiser to treat floating-point operations as associative. This is part of LLVM IR already because there the operations are reordered by the optimiser. `contract` allows the combination of multiple floating-point operations. There are no combined floating-point operations in LLVM IR, so setting this flag is just information for the specific backend behind LLVM (in our case NVPTX) to allow for this optimisation.

Figure 6a shows the PTX code generated from the LLVM IR code we showed in Figure 5. Here we do not have any fast math flags enabled. Highlighted in blue are all the floating-point calculations that match their LLVM IR equivalents roughly. For comparison, Figure 6b shows the resulting PTX when the `contract` fast math flag is set. Four floating-point operations, two multiplies (`mul.rn.f64`) and two adds (`add.rn.f64`) have been fused together to two fused multiply-add (`fma.rn.f64`) operations.

4 EVALUATION

System: NVIDIA GeForce RTX 3050 Ti Laptop, Intel Core i7-11800H.

We want to evaluate the impact of optimisation on the generated code of forward and reverse mode automatic differentiation. First, we investigate the impact of setting `noalias` on pointer parameters of the kernel. We compare the execution and compilation time using a linear model of four weights and one bias with mean squared error as the loss function. The batch size refers to the number of tuples.

As `noalias` optimises the generated code, this improves the runtime for evaluating derivations generated by forward mode automatic differentiation (see Figure 7a). Additionally, we can see

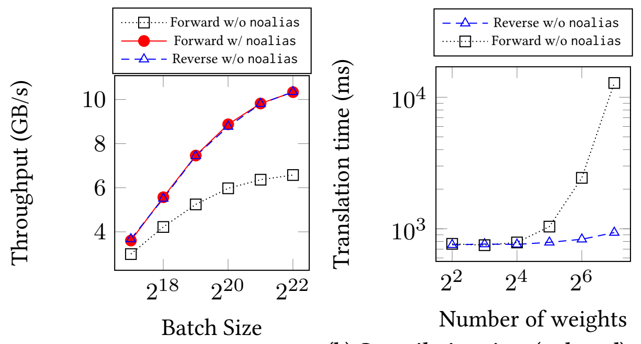
```
ld.param.u64    %rd1, [param_0];
ld.param.u64    %rd2, [param_1];
mov.u32         %r1, %tid.x;
ld.param.u64    %rd3, [param_2];
mov.u32         %r2, %ntid.x;
ld.param.u64    %rd4, [param_3];
mov.u32         %r3, %ctaid.x;
ld.param.u64    %rd5, [param_4];
mad.lo.s32      %r4, %r3, %r2, %r1;
ld.param.u64    %rd6, [param_5];
ld.param.u64    %rd7, [param_6];
mul.wide.u32    %rd8, %r4, 8;
add.s64         %rd9, %rd5, %rd8;
add.s64         %rd10, %rd6, %rd8;
add.s64         %rd11, %rd4, %rd8;
add.s64         %rd12, %rd3, %rd8;
add.s64         %rd13, %rd1, %rd8;
add.s64         %rd14, %rd2, %rd8;
ld.f64          %fd1, [%rd9];
ld.f64          %fd2, [%rd11];
ld.f64          %fd3, [%rd12];
mul.rn.f64      %fd4, %fd2, %fd3;
ld.f64          %fd5, [%rd14];
ld.f64          %fd6, [%rd13];
mul.rn.f64      %fd7, %fd5, %fd6;
add.rn.f64      %fd8, %fd4, %fd7;
add.rn.f64      %fd9, %fd1, %fd8;
ld.f64          %fd10, [%rd10];
sub.rn.f64      %fd11, %fd9, %fd10;
add.rn.f64      %fd12, %fd11, %fd11;
mul.rn.f64      %fd13, %fd5, %fd12;
mul.rn.f64      %fd14, %fd2, %fd12;
add.s64         %rd15, %rd7, %rd8;
st.f64          [%rd15], %fd13;
add.s32         %r5, %r4, 32768;
mul.wide.u32    %rd16, %r5, 8;
add.s64         %rd17, %rd7, %rd16;
st.f64          [%rd17], %fd14;
ret;
```

(a) without fast math flags.

```
ld.param.u64    %rd1, [param_0];
ld.param.u64    %rd2, [param_1];
mov.u32         %r1, %tid.x;
ld.param.u64    %rd3, [param_2];
mov.u32         %r2, %ntid.x;
ld.param.u64    %rd4, [param_3];
mov.u32         %r3, %ctaid.x;
ld.param.u64    %rd5, [param_4];
mad.lo.s32      %r4, %r3, %r2, %r1;
ld.param.u64    %rd6, [param_5];
ld.param.u64    %rd7, [param_6];
mul.wide.u32    %rd8, %r4, 8;
add.s64         %rd9, %rd6, %rd8;
add.s64         %rd10, %rd4, %rd8;
add.s64         %rd11, %rd5, %rd8;
add.s64         %rd12, %rd3, %rd8;
add.s64         %rd13, %rd1, %rd8;
add.s64         %rd14, %rd2, %rd8;
ld.f64          %fd1, [%rd11];
ld.f64          %fd2, [%rd10];
ld.f64          %fd3, [%rd12];
ld.f64          %fd4, [%rd14];
ld.f64          %fd5, [%rd13];
fma.rn.f64      %fd6, %fd4, %fd5, %fd1;
fma.rn.f64      %fd7, %fd2, %fd3, %fd6;
ld.f64          %fd8, [%rd9];
sub.rn.f64      %fd9, %fd7, %fd8;
add.rn.f64      %fd10, %fd9, %fd9;
mul.rn.f64      %fd11, %fd4, %fd10;
mul.rn.f64      %fd12, %fd2, %fd10;
add.s64         %rd15, %rd7, %rd8;
st.f64          [%rd15], %fd11;
add.s32         %r5, %r4, 32768;
mul.wide.u32    %rd16, %r5, 8;
add.s64         %rd17, %rd7, %rd16;
st.f64          [%rd17], %fd12;
ret;
```

(b) with fast math `contract` flag.

Figure 6: Optimised PTX generated by the LLVM IR.



(a) Execution time. **(b) Compilation time (1 thread).**

Figure 7: Comparison of forward mode with and without `noalias` as well as reverse mode without `noalias`.

that forward and reverse mode indeed lead to the same performance if forward mode employs `noalias` in its generated code.

Figure 7b shows the difference in compilation time with forward and reverse mode. The heuristic that for more inputs than outputs, reverse mode performs better than forward mode holds true. And while this could be optimised by executing the automatic differentiation and compilation for each of the weights in parallel in forward mode, the performance difference is still quite drastic.

Regarding the optimised PTX code, there was no difference between enabling or disabling fast math flags, including `contract` and `reassoc`. Especially for the `reassoc` flag, it is highly dependent on the model and its formulation, whether relaxing floating-point operation constraints have an effect.

5 CONCLUSION

This paper has highlighted automatic differentiation as implemented in Umbra to allow for in-database machine learning. In detail, we have discussed forward and reverse mode automatic differentiation for code-generation to GPU. Which mode to use depends on the operator implementation. For example, a flag in the SQL derivation operator interface indicates when to use forward instead of reverse mode, which is set as default. We showed that the *noalias* feature optimises the produced LLVM code in forward mode. The evaluation has proven that the reduced number of expressions in forward mode leads to a similar runtime as for reverse mode. Although we considered code generation to LLVM with PTX as the target, other target architectures are possible as well, as the compiler optimisations also apply to non-virtual assembly languages.

REFERENCES

[1] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.

[2] Matthias Boehm et al. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. www.cidrdb.org.

[3] Patrick Damme et al. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *PVLDB* 13, 11 (2020), 2396–2410.

[4] Ahmed Elgohary et al. 2018. Compressed linear algebra for large-scale machine learning. *VLDB J.* 27, 5 (2018), 719–744.

[5] Edward Gan et al. 2020. CoopStore: Optimizing Precomputed Summaries for Aggregation. *PVLDB* 13, 11 (2020), 2174–2187.

[6] Rainer Gemulla et al. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*. ACM, 69–77.

[7] Ahmed Helal et al. 2021. A Demonstration of KGLac: A Data Discovery and Enrichment Platform for Data Science. *PVLDB* 14, 12 (2021), 2675–2678.

[8] Dimitrije Jankov et al. 2019. Declarative Recursive Computation on an RDBMS. *PVLDB* 12, 7 (2019), 822–835.

[9] Ahmet Kara et al. 2021. Machine learning over static and dynamic relational data. In *DEBS*. ACM, 160–163.

[10] Lukas Karnowski et al. 2021. Umbra as a Time Machine. In *BTW (LNI)*. GI.

[11] Andreas Kunft et al. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.

[12] Sören Laue. 2019. On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation. *CoRR* abs/1904.02990 (2019).

[13] Tae-Jun Lee et al. 2018. Greenhouse: A Zero-Positive Machine Learning System for Time-Series Anomaly Detection. *CoRR* abs/1801.03168 (2018).

[14] Xupeng Li et al. 2017. MLog: Towards Declarative In-Database Machine Learning. *PVLDB* 10, 12 (2017), 1933–1936.

[15] Thomas Neumann et al. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[16] Stefanie Scherzinger et al. 2019. The Best of Both Worlds: Challenges in Linking Provenance and Explainability in Distributed Machine Learning. In *ICDCS*. IEEE.

[17] Maximilian Schleich et al. 2020. LMFAO: An Engine for Batches of Group-By Aggregates. *PVLDB* 13, 12 (2020), 2945–2948.

[18] Maximilian E. Schüle et al. 2017. Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. *PVLDB* 10, 12 (2017), 1921–1924.

[19] Maximilian E. Schüle et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW (LNI)*. GI.

[20] Maximilian E. Schüle et al. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *EDBT*.

[21] Maximilian E. Schüle et al. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD*. ACM, 7:1–7:4.

[22] Maximilian E. Schüle et al. 2019. The Power of SQL Lambda Functions. In *EDBT*.

[23] Maximilian E. Schüle et al. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. ACM, 6:1–6:12.

[24] Maximilian E. Schüle et al. 2021. ArrayQL for Linear Algebra within Umbra. In *SSDBM*. ACM, 193–196.

[25] Maximilian E. Schüle et al. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM*. ACM, 25–36.

[26] Maximilian E. Schüle et al. 2022. ArrayQL Integration into Code-Generating Database Systems. In *EDBT*.

[27] Vraj Shah et al. 2021. Towards Benchmarking Feature Type Inference for AutoML Platforms. In *SIGMOD*. ACM, 1584–1596.

[28] Amir Shaikhha et al. 2021. An Intermediate Representation for Hybrid Database and Machine Learning Workloads. *PVLDB* 14, 12 (2021), 2831–2834.

[29] Ted Shaowang et al. 2021. Declarative Data Serving: The Future of Machine Learning Inference on the Edge. *PVLDB* 14, 11 (2021), 2555–2562.

[30] Jonas Traub et al. 2020. Agora: Bringing Together Datasets, Algorithms, Models and More in a Unified Ecosystem [Vision]. *SIGMOD Rec.* 49, 4 (2020), 6–11.

[31] Hantian Zhang et al. 2021. OmniFair: A Declarative System for Model-Agnostic Group Fairness in Machine Learning. In *SIGMOD*. ACM, 2076–2088.