Enzyme: Efficient Cross-Platform AD by Synthesizing LLVM

William S. Moses Tim Kaler

{wmoses, tfk}@mit.edu

EuroAD

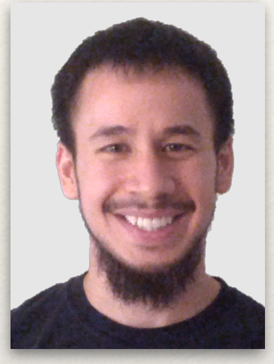
July 2, 2019

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Part of exploration on AD by MIT Supertech Research Groups

https://supertech.mit.edu

Charles E. Leiserson



Tao B. Schardl

Daniele Vettorel

Funding provided by DOE CSGF fellowship and IBM

“Holy Grail” of Automatic Differentiation

- ***General:*** we should be able to run AD on arbitrary programs
- ***Easy to Use:*** the amount of code one needs to modify to use AD should be small
- ***Fast:*** executing AD shouldn’t take too long
- ***Correct:*** AD should produce the right answer

State of AD

	Usable	Fast	Examples
Library / DSL Specific	✓	✓	
High Level Lang. Specific	✓	?	Zygote
Low Level Lang. Specific	✗	?	Adept
Language Independent			None* 😢

More General

* There are some "language independent" ones but they require rewriting for said framework in a way that makes it rather unusable

State of AD

	Usable	Fast	Examples
Library / DSL Specific	✓	✓	
High Level Lang. Specific	✓	?	Zygote
Low Level Lang. Specific	✗	?	Adept
Language Independent			Enzyme (this work)

More General

* There are some "language independent" ones but they require rewriting for said framework in a way that makes it rather unusable

Why Generality Matters

- Taking derivatives of arbitrary programs gives programmers composability — they only need to care about the tool they're building rather than any code they're differentiating
    - e.g. 'I want to build ML tool for predicting the result of this simulator'
- Most programs aren't written in the same language / framework as your tool and thus won't work with your AD

Idea: Generality by Bootstrapping

- A sufficiently general AD system for a particular language (or framework) works not only with code in that language, but any code for higher level languages written in the lower level language.
  - i.e a good C differentiator should be able to also differentiate Python code
- If we create a general AD for a low level language we get the higher languages (mostly) for free

Presenting Enzyme (work in progress)

- Reverse-mode automatic differentiation tool built in LLVM to handle a variety of languages and frameworks
- Performs differentiation by *synthesizing* a new function
- Clean interface that doesn't require rewriting existing programs to use

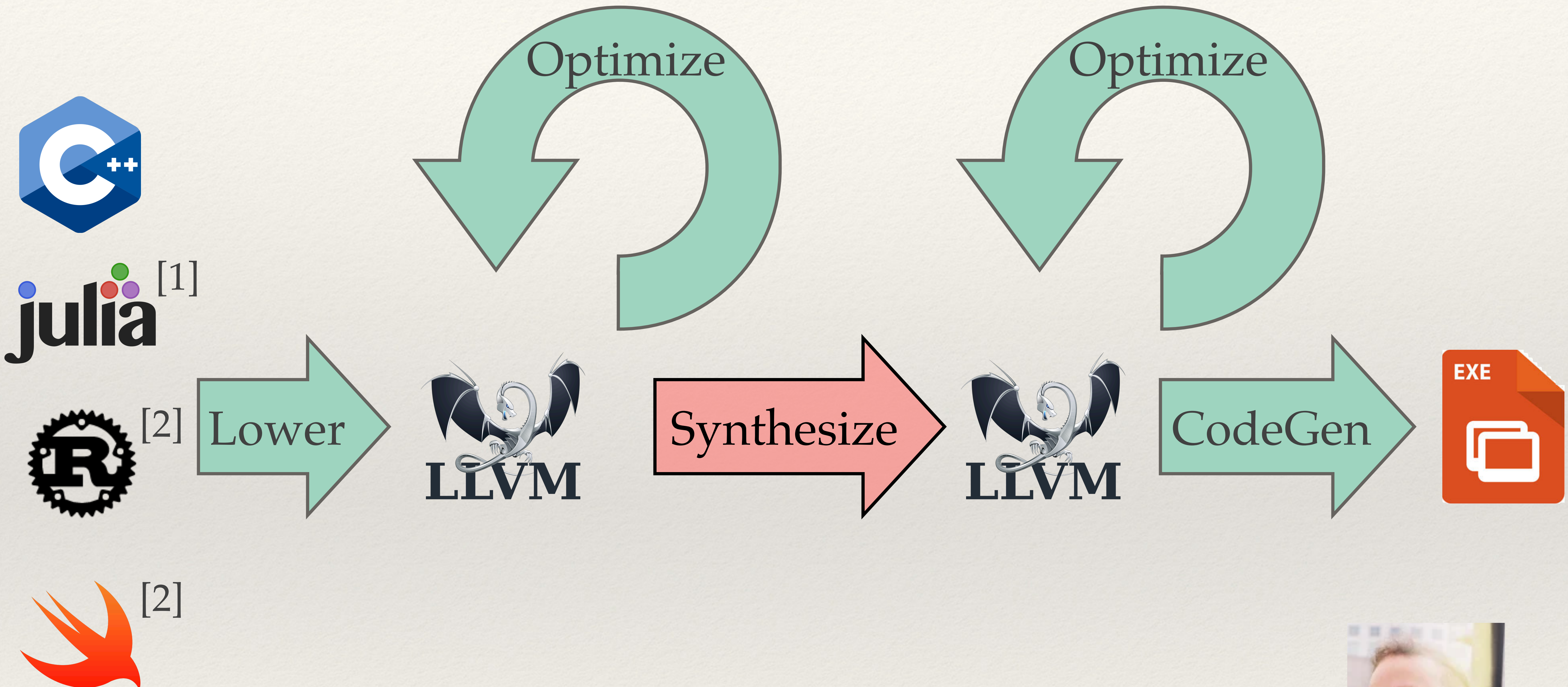
Presenting Enzyme (work in progress)

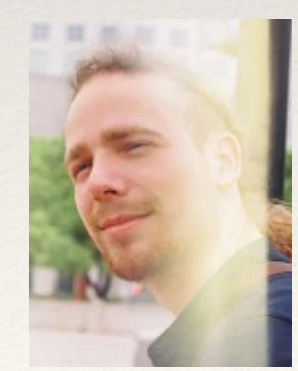
- Reverse-mode automatic differentiation tool built in LLVM to handle a variety of languages and frameworks
- Performs differentiation by *synthesizing* a new function
- Clean interface that doesn't require rewriting existing programs to use

Our beta* can match the performance of less general AD on a variety of benchmarks!

[*] Beta is in progress and not yet feature-complete.
Planned open sourcing once published and ABI-stable.

Enzyme Pipeline





[1] Frontend for Julia joint with Valentin Churavy

[2] Lowering pass needs to be implemented for each language. C/C++ and Julia implemented presently.

Valentin Churavy

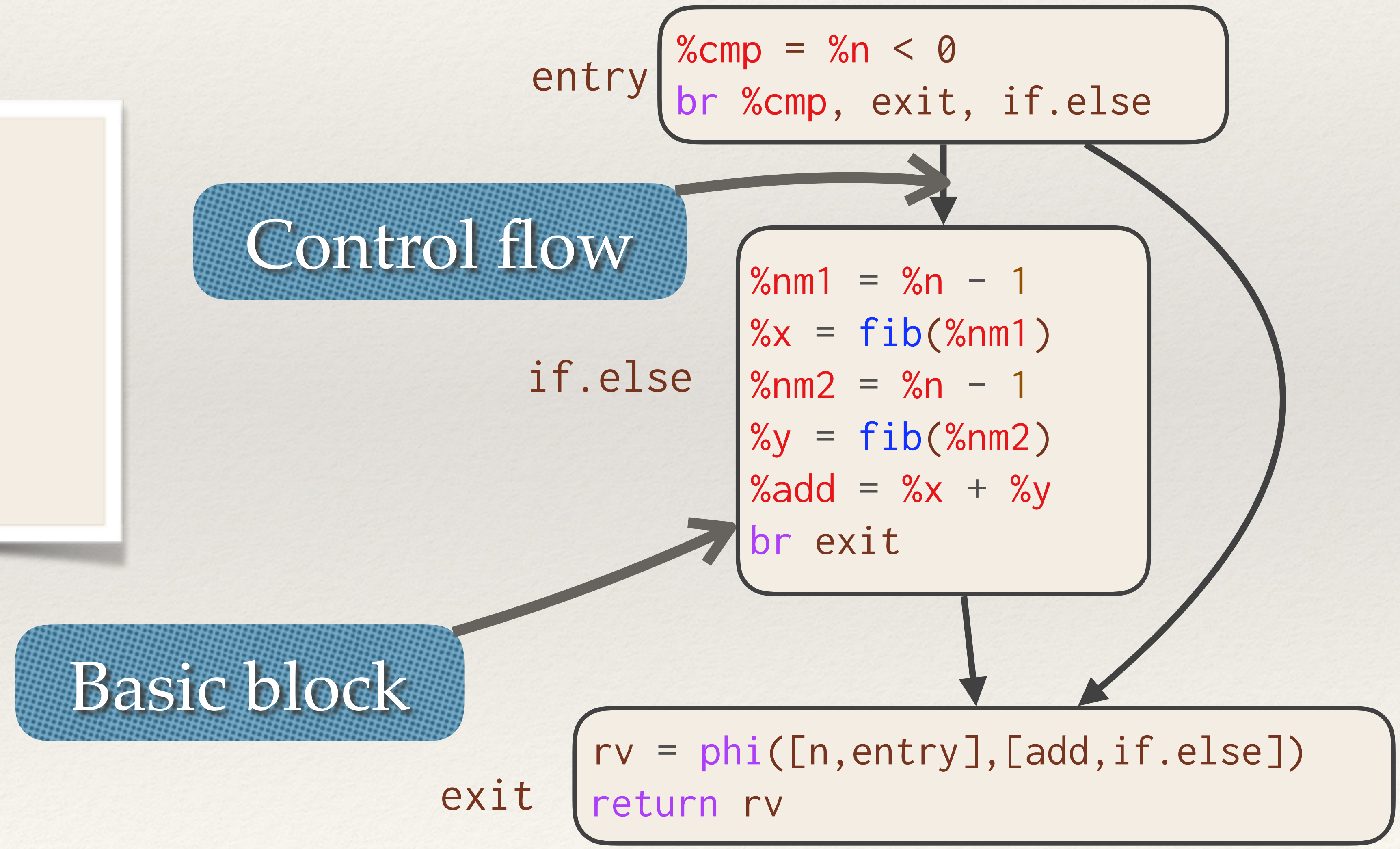
What is LLVM

- Generic low-level compiler infrastructure
  - “Cross platform assembly”
- Goal is compiling arbitrary code as efficiently as possible
  - Well-defined semantics and high-level constructs
  - Large collection of optimization and analysis passes for handling

LLVM IR

LLVM represents each function as a **control-flow graph (CFG)** of **BasicBlocks**, containing lists of **Instructions**.

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = fib(n - 1);
  y = fib(n - 2);
  return x + y;
}
```



What is Synthesis?

- Instrumentation-based approaches
  - Store the operations and values of the forward pass in a tape that is later “interpreted” by the reverse pass
  - Can store data by *overloading* a language’s types/functions or *rewriting/transforming* the source code to include it (such as in compiler instrumentation)
- Synthesis-based approaches
  - Statically analyze the function to produce a new function with the relevant operations

Why Synthesis?

- Synthesis is often faster
- Overloading all of LLVM's instructions and fixing its ~4 million lines of code is both impractical and unsustainable
- Since we must do program rewriting / transformation anyways (and LLVM has tools for making it easier), might as well do synthesis rather than instrumentation for faster results

Core Algorithm

- Iterate through all instructions in the original function to detect whether they are active (could modify derivatives) or not.
- For active value in the original function, allocate and zero memory to store the derivative of all of its occurrences.
- For each block in the original function, compute the adjoint of its active instructions in reverse order, caching and reloading any necessary values from the forward pass

Optimizing away the "Tape"

- To compute adjoints, it may be necessary to use values computed in the forward pass
  - Traditionally stored in a stack-based tape mechanism
- Idea: carefully cache all values individually in a form LLVM understands (to simplify implementation)
  - Existing optimization passes can optimize the memory away
  - Without optimization may use more memory than traditional tape, after optimization uses far less

Optimizing Communication

- Compute the forward pass and backward passes together
  - Let LLVM optimize how values are shared / reused from forward to backward pass
  - Dead code-elimination can get rid of the forward pass if not needed!
- After optimizations, forward pass and backward pass can be split* [useful for recursive calls]

[*] Splitting is in progress.

Case Study: ReLU-f

```
double relu f(double x) {
  double result;
  if (x > 0)
    result = f(x);
  else
    result = 0;
  return result;
}
```

```
double diffe_reluf(double x) {
  return __builtin_autodiff(reluf, x);
}
```

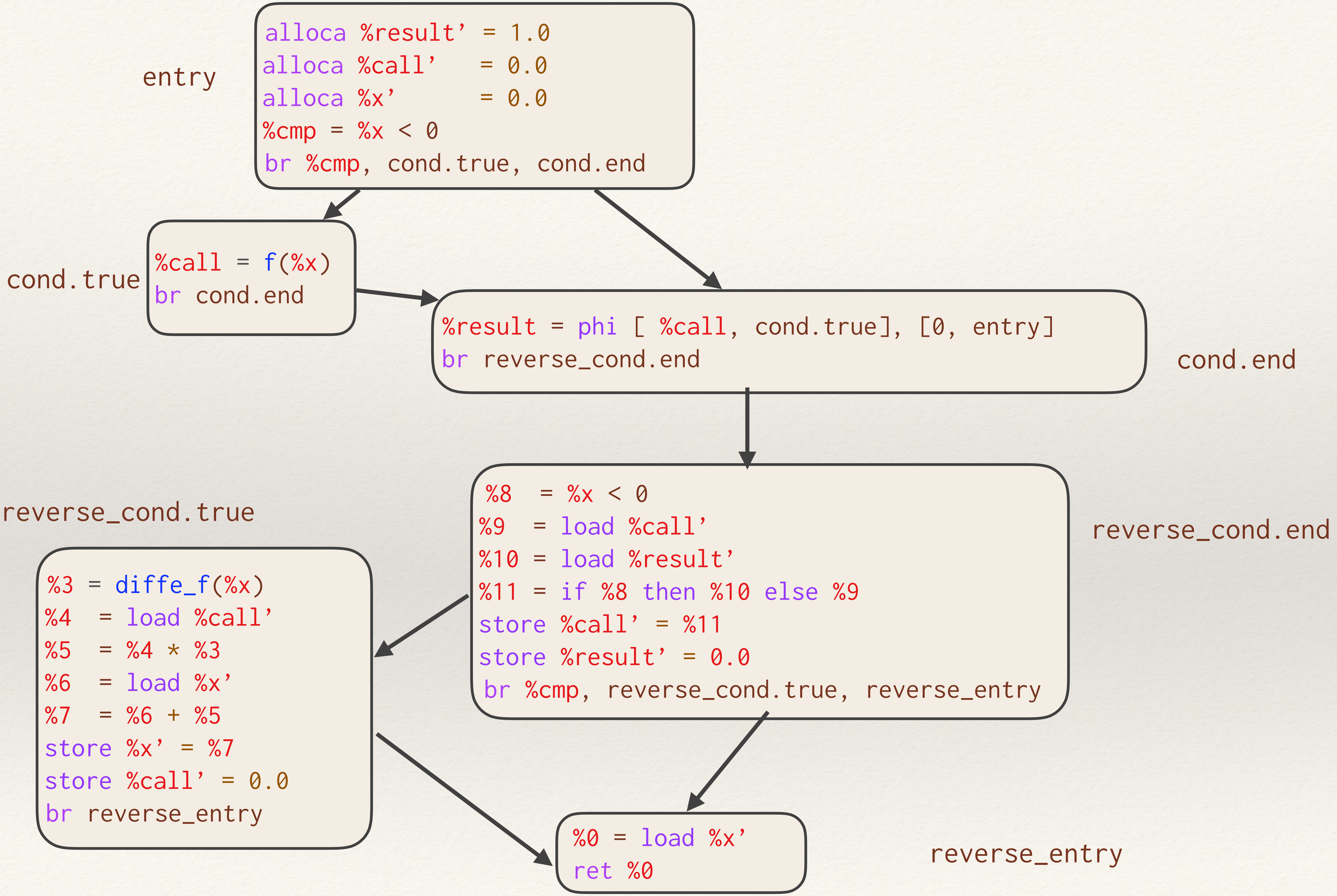
```
define double @reluf(double %x)
```

```
entry:
  %cmp = %x < 0
  br %cmp, cond.true, cond.end

cond.true:
  %call = f(%x)
  br cond.end

cond.end:
  %result = phi [ %call, cond.true], [0, entry]
  ret %result
```

```
define double @diffe_reluf(double %x)
```



Run LLVM memory optimization

```
define double @diffe_reluf(double %x)
```

entry
```
%cmp = %x < 0
br %cmp, cond.true, cond.end
```

cond.true
```
%call = f(%x)
br cond.end
```

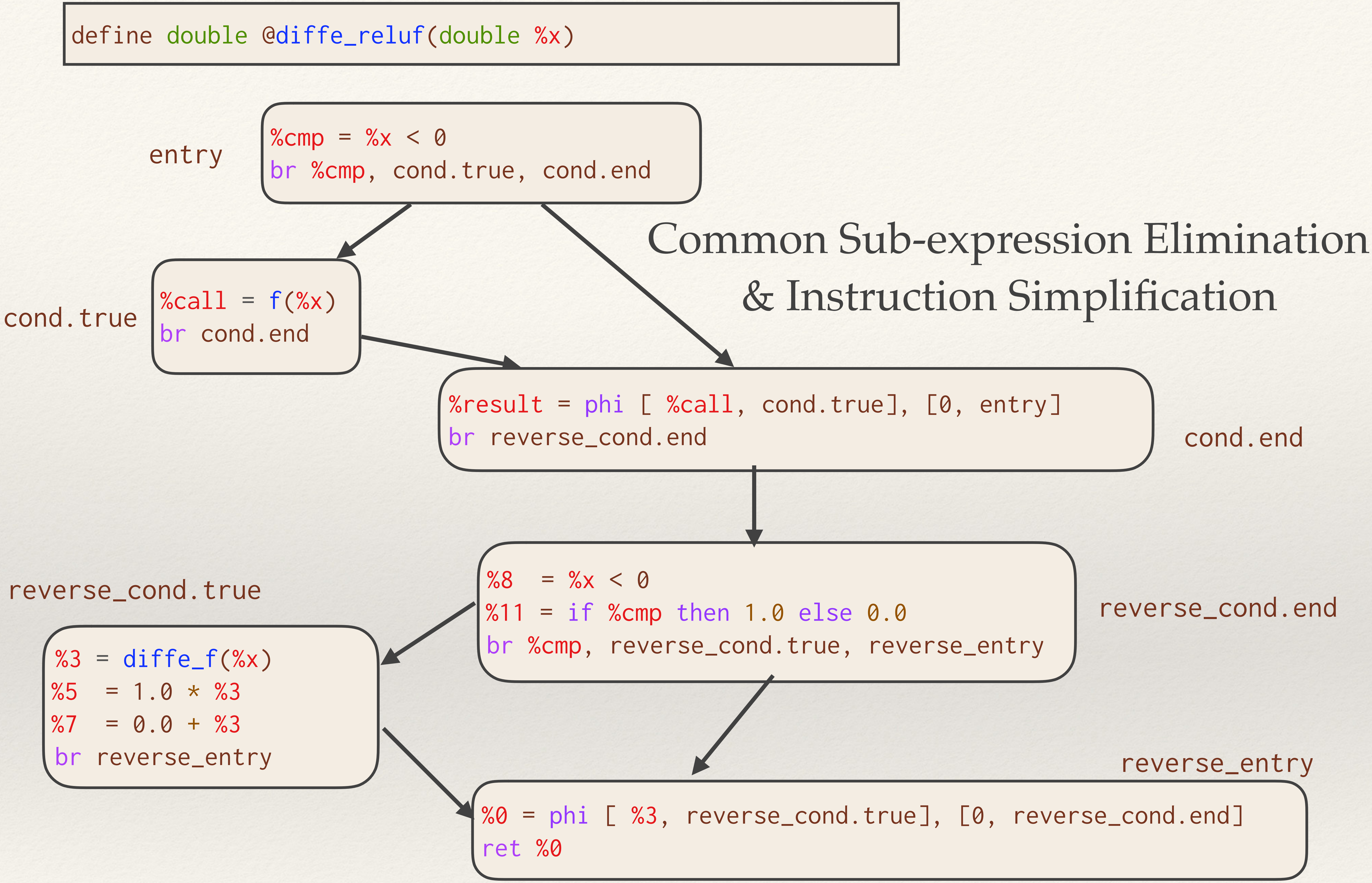
cond.end
```
%result = phi [ %call, cond.true], [0, entry]
br reverse_cond.end
```

reverse_cond.end
```
%8  = %x < 0
%11 = if %8 then 1.0 else 0.0
br %cmp, reverse_cond.true, reverse_entry
```

reverse_cond.true
```
%3 = diffe_f(%x)
%5  = %11 * %3
%7  = 0.0 + %5
br reverse_entry
```

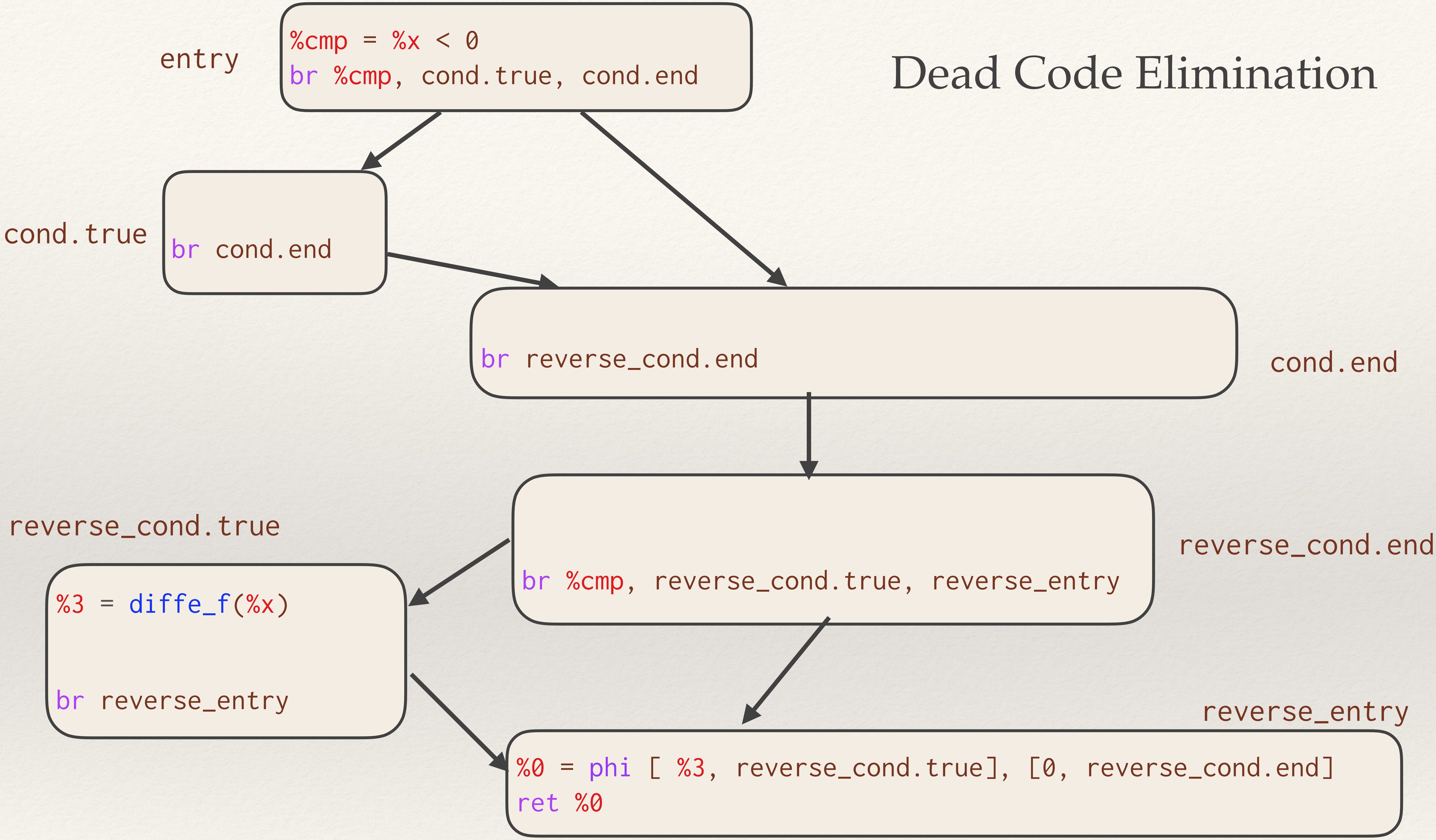
reverse_entry
```
%0 = phi [ %7, reverse_cond.true], [0, reverse_cond.end]
ret %0
```

Common Sub-expression Elimination & Instruction Simplification



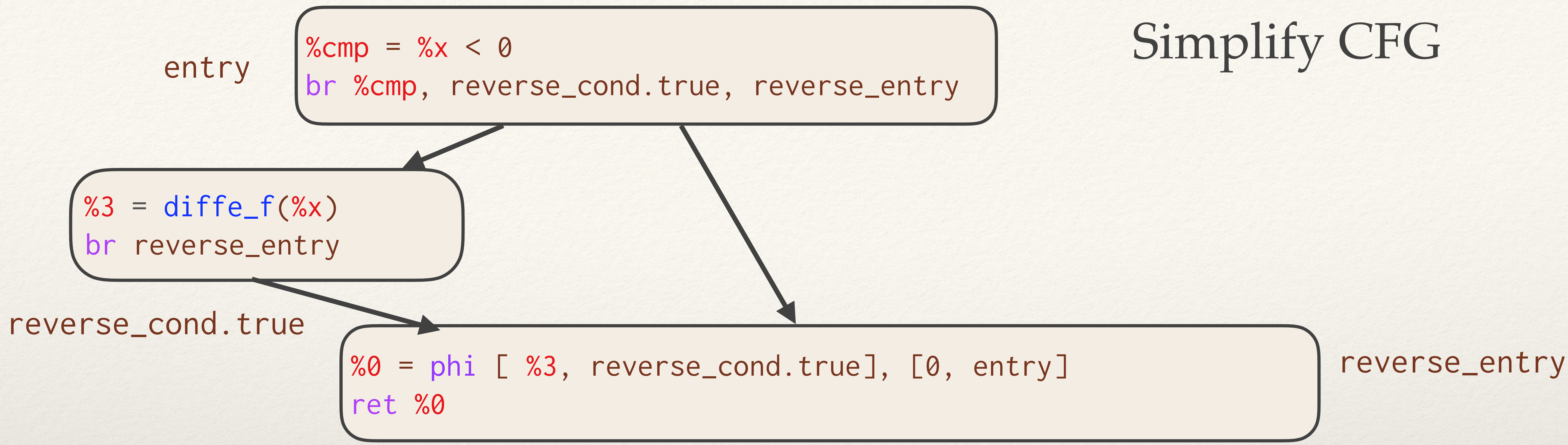
Dead Code Elimination

```
define double @diffe_reluf(double %x)
```



```
define double @diffe_reluf(double %x)
```

Simplify CFG



Essentially the optimal hand-compiled program!

```c
double diffe_reluf(double x) {
  double result;
  if (x > 0)
    result = diffe_f(x);
  else
    result = 0;
  return result;
}
```

More Advanced Details

Loops

- Loops require special handling since an SSA Value can have multiple distinct realizations per iteration of the loop
- Idea: Statically allocate an array of sufficient size to store all loop allocations in outermost loop preheater
  - With correct attributes, LLVM is able to understand this allocation and similarly optimize
- If loop bounds cannot be calculated statically, dynamically reallocate array
  - Requires modification to LLVM memory analyses to understand semantics of `realloc`.

Active Variable Detection*

- All function arguments are denoted as either *inactive, active* (with reasonable defaults for the user)
- Non-pointer value is inactive if it is created by using only inactive values or never used in creation of an active value
- Pointer values require examining stores to uses / users
- Algorithm as heuristic to avoid creating unnecessary computation / synthesis and avoid asking for ill-defined derivatives (i.e a function prints an active variable — what is the derivative of the print function)

[*] Work in progress — suggestions appreciated

Complex Data Types

- Calling a derivative function with complex data types (e.g arrays) requires passing a second data structure to store derivative outputs
- Structs with multiple elements may contain both active variables and constants
  - e.g. an array storing its size — size is constant
  - Variable marked as active
  - Rely on active variable detection to identify if a particular element of struct derivatives

Local Data Structures

- Local data structures with active variable need to be duplicated to store derivative information
  - Leverage all data structures are created by specific memory instructions (malloc/free/new/delete/etc)
- Allocations are copied in forward pass to create differential structures
- Frees are delayed until the reversed version of the block that allocated in case values are used in the reverse pass

Case Study: Read Sum

```
double sum(double* x) {
  double total = 0;

  for(int i=0; i<10; i++)
    total += read() * x[i];

  return total;
}
```

```
void diffe_sum(double* x,
                double* xp) {
  return
   __builtin_autodiff(sum, x, xp);
}
```

```
define double @sum(double* %x)
```

entry

```
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```

for.cleanup

```
%result = phi [ %call, cond.true], [0, entry]
ret %result
```

```
define void @diffe_sum(double* %x, double* %xp)
```

After lowering and some optimizations

entry

```
%cache = @malloc(8 x double)
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
store %cache[%i] = %call
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, reversefor.body, for.body
```

reversefor.body

```
%i' = phi [ 9, for.body ], [ %i’.next, reversefor.body ]
%i’.next = %i' - 1
%cached_read = load %cache[%i’]
store %xp[%i’] = %cached_read + %xp[%i’]
%exit2 = %i = 0
br %exitcond, %exit2, reversefor.body
```

exit

```
@free(%cache)
ret
```

Case Study: Read Sum

```
define void @diffe_sum(double* %x, double* %xp)
```

After more optimizations

entry

```
%call0 = @read()
store %xp[0] = %call0
%call1 = @read()
store %xp[1] = %call1
%call2 = @read()
store %xp[2] = %call2
%call3 = @read()
store %xp[3] = %call3
%call4 = @read()
store %xp[4] = %call4
%call5 = @read()
store %xp[5] = %call5
%call6 = @read()
store %xp[6] = %call6
%call7 = @read()
store %xp[7] = %call7
%call8 = @read()
store %xp[8] = %call8
%call9 = @read()
store %xp[9] = %call9
ret
```

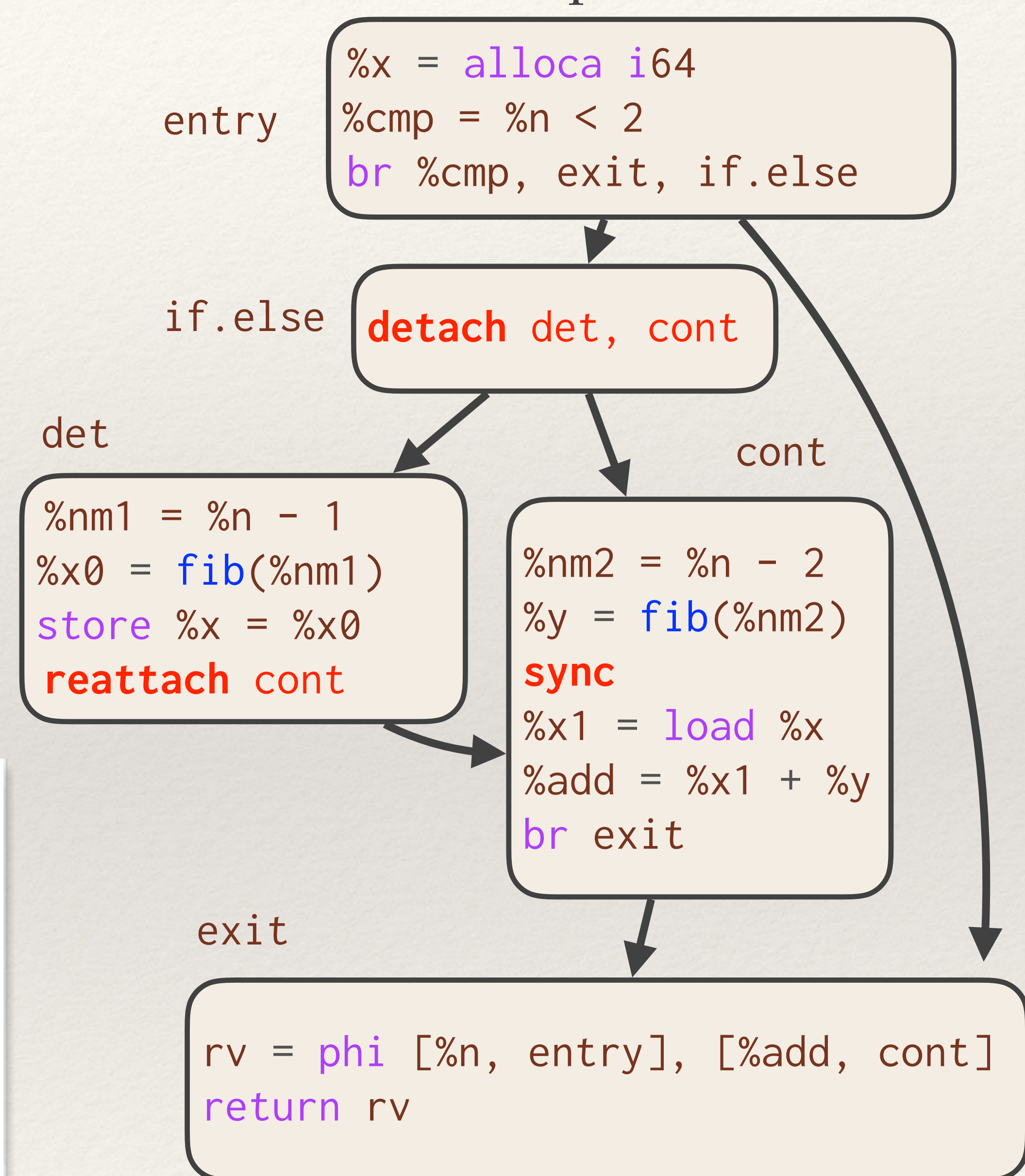
```
void diffe_sum(double* x, double* xp) {
   xp[0] = read();
   xp[1] = read();
   xp[2] = read();
   xp[3] = read();
   xp[4] = read();
   xp[5] = read();
   xp[6] = read();
   xp[7] = read();
   xp[8] = read();
   xp[9] = read();
}
```

Parallelism*

- Build off prior work [1] representing parallelism (OpenMP, Cilk, etc) in compiler
- Reverse pass can remain in parallel, with dependencies reversed
- Updates to adjoints in parallel tasks done with reducer or atomic add to prevent races

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = spawn fib(n - 1);
  y = fib(n - 2);
  sync;
  return x + y;
}
```

Tapir CFG



[1] Tapir; Tao. B Schardl, **William S Moses**, Charles E. Leiserson; PPoPP 2017
[*] Work in progress — suggestions appreciated

Custom Derivatives*

- Enzyme can compute derivatives of any function in current compilation module
- Functions compiled in a unit (i.e. libraries, linked objects) can be handled by compiling library with Enzyme, creating library with derivatives included
- Functions can be marked with a custom derivative function via metadata

[*] Work in progress — suggestions appreciated

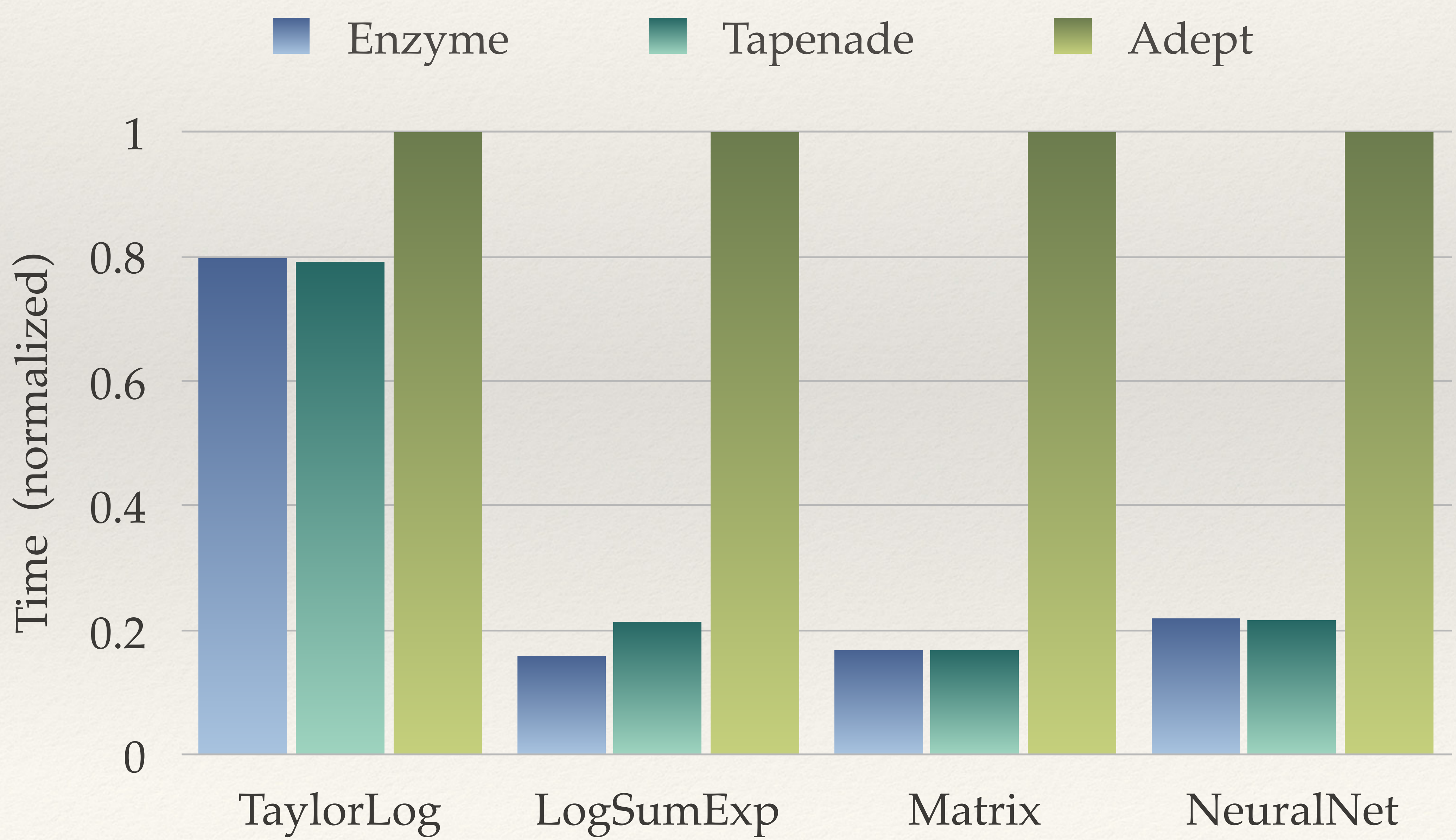
Preliminary Tests of Beta Implementation

All programs run serially

Intel E5520 @ 2.27GHz, 64GB 1066MHz DDR3, Ubuntu 16.04

Time Graph

Between 19% and 534% speedup over Adept
Comparable with Tapenade



Taylor Expand Log

Use a Taylor series to compute the log function, evaluated at x=0.5

$$f(x) = \sum_{i=1}^{N} \frac{x^i}{i} \approx -\log(1-x)$$

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{1-x}$$

```
#define ITERS 10000000
double taylor_log(double x) {
  double sum = 0;

  for(int i=1; i<=ITERS; i++)
    sum += pow(x, i) / i;

  return sum;
}
```

```
double derivative(double x) {
  return __builtin_autodiff(taylor_log, x);
}
```

Taylor Expand Log

10000000 iterations

	Enzyme	Adept	Tapenade
Normal	3.71	3.74	3.70
Forward	3.70	4.50	3.71
Forward +Reverse	3.72	4.67	3.70

LogSumExp

Smooth approximation to maximum function, often used in machine learning.

```
#define N 10000000
double logsumexp(double* x, size_t n) {
  double A = 0;

  for(int i=1; i < n; i++) {
    A = max(A, x[i]);
  }

  double sema = 0;

  for(int i=0; i < n; i++) {
    sema += max(x[i] - A);
  }

  return max(sema) + A;
}
```

```
double derivative(double* input, double* inputp, size_t n) {
  return __builtin_autodiff(logsumexp, input, inputp, n);
}
```

LogSumExp

10000000 elements

	Enzyme	Adept	Tapenade
Normal	0.364	0.364	0.364
Forward	0.364	2.994	0.364
Forward +Reverse	0.605	3.836	0.817

Find Matrix by Gradient Descent

Find a matrix that produces a vector close to zero when multiplied by vec

```
#define N 2000
#define M 2000
double matvec(double* mat, double* vec) {
  double* out = malloc(sizeof(double)*N);

  double A = 0;
  for(int i=1; i < N; i++) {
    out[i] = 0;
    for(int j=1; j < M; j++) {
     out[i] += mat[i*M+j] * vec[j];
    }
  }
  double sum = 0;
  for(int i=0; i < N; i++) {
    sum += out[i] * out[i];
  }
  free(out);
  return sum;
}
```

```
#define ITERS 1000
#define RATE 0.00000001
double descent(double* mat, double* dmat,
               double* vec){

  for(int iter=1; iter < ITERS; iter++) {

    memset(dmat, 0, sizeof(double)*N*M);
    __builtin_autodiff(matvec, mat, dmat,
                       diffe_const, vec);

    out[i] = 0;
    for(int i=1; i < N*M; i++) {
      mat[i] -= dmat[i] * RATE;
    }
  }
  double sum = 0;
  for(int i=0; i < N; i++) {
    sum += out[i] * out[i];
  }
}
```

Find Matrix by Gradient Descent

	Enzyme	Adept	Tapenade
Forward	4.698	25.356	4.704
Gradient Descent	22.039	130.957	21.828

Training Simple Neural Network

Enzyme	Adept	Tapenade	Handwritten
73.663	338.097	73.008	72.076

Picked first C MNIST Code on Github:

https://github.com/AndrewCarterUK/mnist-neural-network-plain-c

- 1-layer fully connected layer => softmax => cross-entropy loss
- Batch size 100
- 1000 iterations
- Learning rate 0.5

Conclusions

- Need four things in AD: *generality*, *usability*, *speed*, and *correctness*
- Created a prototype tool: Enzyme
  - Provides first "true" cross platform AD (to our knowledge)
    - Compatible with any tool lowering to LLVM (Tensorflow, Rust, C/C++, Julia, etc)
  - Matches state of art performance by building off on compiler optimizations
  - Demonstrates possibility of a general AD that is efficient and easy-to-use
- Future Work:
  - Feature completion and more frontends
  - Heuristics (e.g. recompute vs cache)
  - ABI stability and open source release / publication

Backup Slides

Matrix Vector: Single Iteration

```
#define N 20000
#define M 20000
#define ITERS 1
```

	Enzyme	Adept
Normal	1.119	0.0006
Forward	1.119	11.016
Forward +Reverse	1.210	13.445

Taylor Expand Log

```
static adouble logger(adouble x) {
  adouble sum = 0;
  for(int i=1; i<=ITERS; i++) {
    sum += pow(x, i) / i;
  }
  return sum;
}

static double logger_and_gradient(double xin, double& xgrad) {
    adept::Stack stack;
    adouble x = xin;
    stack.new_recording();
    adouble y = logger(x);
    y.set_gradient(1.0);
    stack.compute_adjoint();
    xgrad = x.get_gradient();
    return y.value();
}
```

Taylor Expand Log (Julia)

$$f(x) = \sum_{i=1}^{N} \frac{x^i}{i} \approx -\log(1-x)$$

```
#define ITERS 10000000
double logger(double x) {
  double sum = 0;

  for(int i=1; i<=ITERS; i++)
    sum += pow(x, i) / i;

  return sum;
}
```

```
function jl_f1(f::Float64)
  sum = 0 * f;
  for i = 1:10000000
    sum += f^i / i;
  end
  return sum;
end
```

```
; Enzyme derivative code
@show autodiff(fl_f1, 0.5)
@time autodiff(fl_f1, 0.5)
```

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{1-x}$$

$$\frac{\partial}{\partial x} f(x=0.5) \approx 2$$

```
using Zygote
@show jl_f1‘(0.5)
@time jl_f1‘(0.5)
```

Taylor Expand Log

10000000 iterations

	Enzyme	Adept	Enzyme-Julia	Zygote-Julia	AutoGrad-Julia
Normal	3.74	3.72	3.82	3.82	3.82
Forward	3.74	4.56	3.82	3.82	3.82
Forward +Reverse	3.90	4.65	3.95	44.694	896.30

LogSumExp

```c
#define N 10000000
double logsumexp(double* x, size_t n) {
  double A = 0;
  for(int i=1; i < n; i++) {
    A = max(A, x[i]);
  }
  double sema = 0;
  for(int i=0; i < n; i++) {
    sema += max(x[i] - A);
  }
  return max(sema) + A;
}
```

```julia
function logsumexp(x::Array{Float64,1})
  A = maximum(x)
  ema = exp.(x .- A)
  sema = sum(ema)
  return log(sema) + A
end
```

Taylor Expand Log

10000000 iterations

	Enzyme	Adept
Normal	3.74	3.72
Forward	3.74	4.56
Forward +Reverse	3.90	4.65

LogSumExp

10000000 elements

	Enzyme	Adept
Normal	0.364	0.364
Forward	0.364	2.994
Forward +Reverse	0.605	3.836

Find Matrix by Gradient Descent

	Enzyme	Adept
Forward	4.731	25.606
Gradient Descent	22.672	133.354

Training Simple Neural Network

Enzyme	Adept	Handwritten
73.718	338.097	72.178

Picked first C MNIST Code on Github:

https://github.com/AndrewCarterUK/mnist-neural-network-plain-c

- 1-layer fully connected layer => softmax => cross-entropy loss
- Batch size 100
- 1000 iterations
- Learning rate 0.5