

# Performance Engineering of Software Systems

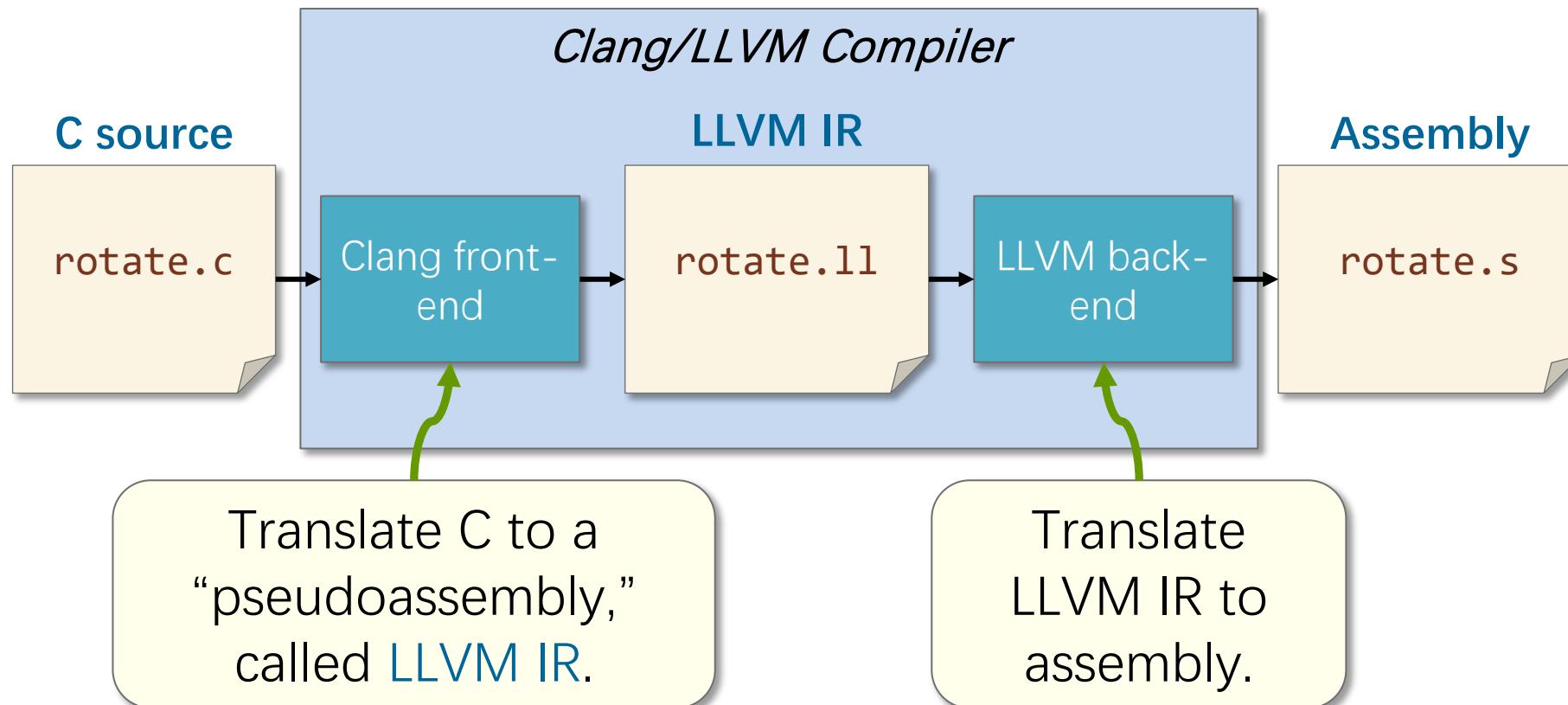
## LECTURE 20 What Compilers Can and Cannot Do

Tao B. Schardl



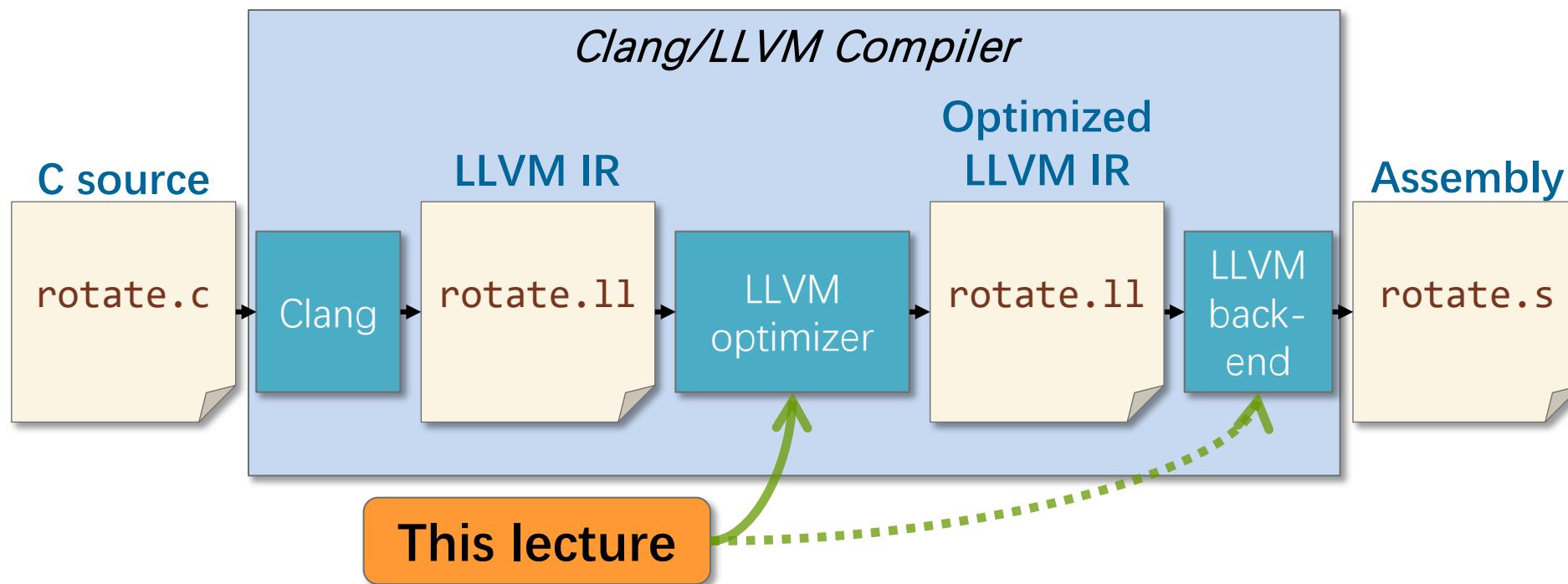
# Clang/LLVM Compiler Pipeline

Recall Lecture 5: The basic compiler pipeline.

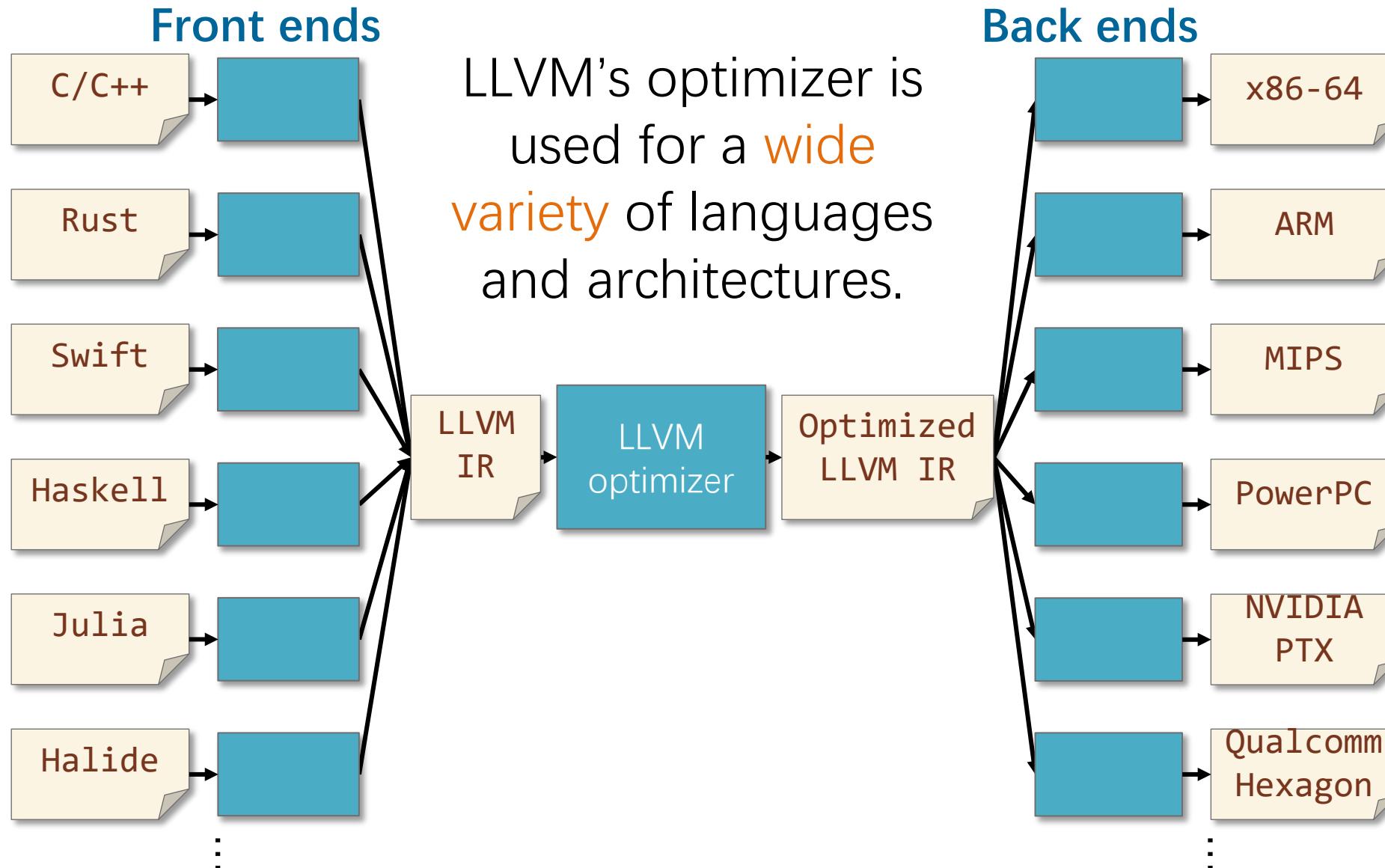


# Clang/LLVM Compiler Pipeline Expanded

LLVM's **optimizer** transforms LLVM IR to make the resulting code run **faster**.



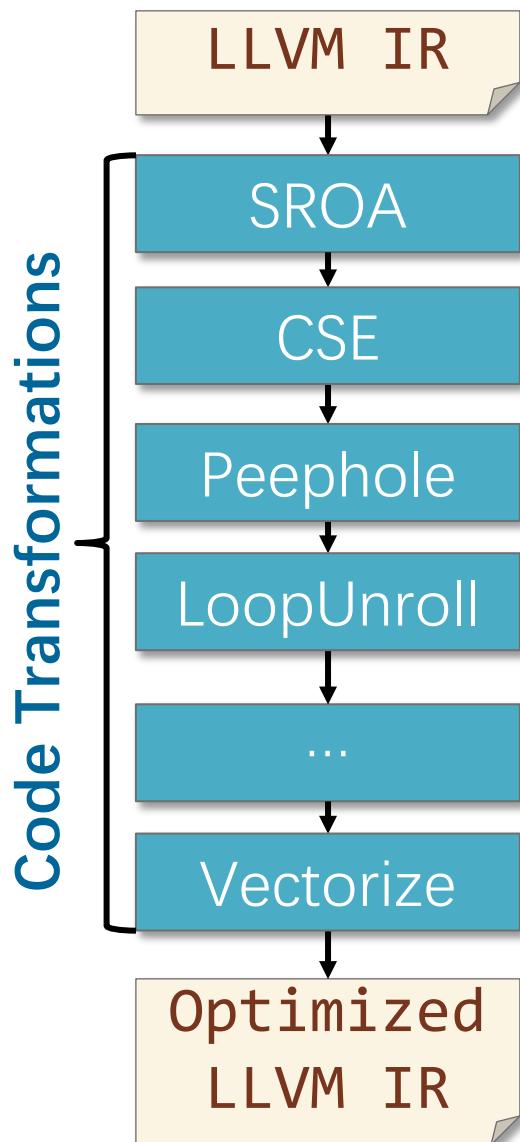
# Larger Context of the Compiler



# Why Study Compiler Optimizations?

- Compilers have a **big impact** on software performance.
- Compilers help ensure that simple, readable, and maintainable code is **fast**.
- You can understand the **differences** between the source code and the IR or assembly.
- Compilers can make **mistakes**.
- Compilers are conservative, thus notoriously **finicky**.
- Understanding compilers can help you use them more **effectively**.
- Even if they are **temperamental**, it's much easier to use a compiler than do the optimizations and transformations yourself!
- Compilers can **save** you performance-engineering work.

# Simple Model of the Compiler



An optimizing compiler performs a sequence of **transformation passes** on the code.

- Each transformation pass analyzes and edits the code, typically to try to optimize the code's performance.
- A transformation pass might run multiple times.
- Passes run in a predetermined order that the compiler writers found works well most of the time.

# Compiler Reports

Clang/LLVM can produce **reports** for many of its transformation passes, not just vectorization:

- **Rpass=<string>**: Produce reports of which optimizations matching <string> were successful.
- **Rpass-missed=<string>**: Produce reports of which optimizations matching <string> were not successful.
- **Rpass-analysis=<string>**: Produce reports of the analyses performed by optimizations matching <string>.

The argument <string> is a **regular expression**.  
To see the whole report, use “.\*” as the string.

# An Example Compiler Report

```
$ clang -O3 -std=gnu11 -c CollisionWorld.c -Rpass=.* -Rpass-analysis=.*
...
CollisionWorld.c:92:39: remark: hoisting load [-Rpass=licm]
    for (int i = 0; i < collisionWorld->numOfLines; i++) {
                                         ^
CollisionWorld.c:91:6: remark: load of type i32 eliminated [-Rpass=gvn]
void CollisionWorld_lineWallCollision(CollisionWorld* collisionWorld) {
    ^
CollisionWorld.c:79:3: remark: CollisionWorld_lineWallCollision can be
inlined into CollisionWorld_updateLines with cost=245 (threshold=250)
[-Rpass-analysis=inline]
    CollisionWorld_lineWallCollision(collisionWorld);
    ^
CollisionWorld.c:79:3: remark: CollisionWorld_lineWallCollision inlined
into CollisionWorld_updateLines [-Rpass=inline]
CollisionWorld.c:135:5: remark: loop not vectorized: loop control flow
is not understood by vectorizer [-Rpass-analysis=loop-vectorize]
    for (int j = i + 1; j < collisionWorld->numOfLines; j++) {
    ^
...
...
```

# Compiler Reports: Good and Bad

**GOOD NEWS:** The compiler can tell you a lot about what it's doing.

- Many transformation passes in LLVM can report places where they successfully transform code.
- Many can also report the conclusions of their analysis.

**BAD NEWS:** Reports can be hard to understand.

- The reports can be **long** and use LLVM **jargon**.
- Not all transformation passes generate reports.
- Reports can be **misleading** and don't always tell the whole story.

We want **context** for understanding these reports.

# Outline

- Overview of compiler optimizations
- Examples of compiler optimizations
  - Scalar optimizations
  - Optimizing structure usage
  - Function inlining
- Case studies

# OVERVIEW OF COMPILER OPTIMIZATIONS



# ~~New Bentley Rules~~

## Data structures

- Register allocation
- Replace memory with registers
- Replace aggregates with scalars
- Data alignment

## Loops

- Hoisting
- ~~Sentinels~~
- Loop unrolling
- Loop fusion \*
- Eliminating wasted iterations

\*

---

\* Restrictions may apply.

## Compiler Optimizations

### Logic

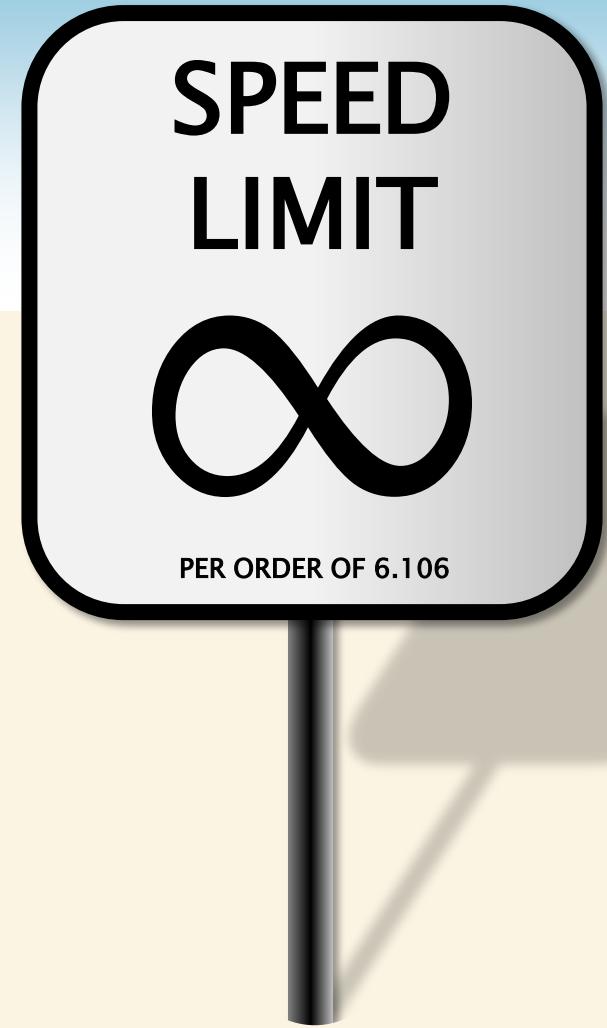
- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- ~~Creating a fast path~~
- Short-circuiting
- Ordering tests \*
- Combining tests \*

### Functions

- Inlining
- Tail-recursion elimination

**Moving target:** Compiler developers implement new optimizations over time.

# EXAMPLES OF COMPILER OPTIMIZATIONS



# Example: N-Body Simulation

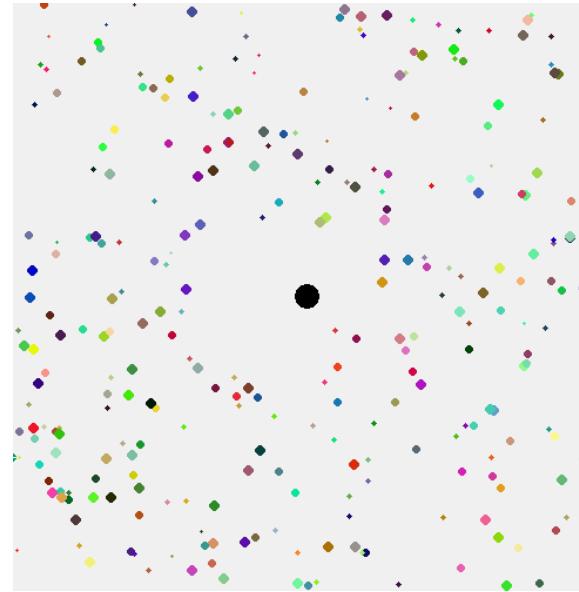
PROBLEM: Simulate the behavior of  $n$  massive bodies in 2D space under the influence of gravity.

LAW OF GRAVITATION:

$$\mathbf{F}_{21} = (G m_1 m_2 / |\mathbf{r}_{12}|^2) \text{ unit}(\mathbf{r}_{21})$$

KEY STEP: For each body, use the force acting on that body to update its position and velocity.

Let's look at a few different implementations to see how compiler optimizations **affect** how we can write this code.



# Abstract Implementation

```
void update_positions(int nbodies, body_t *bodies,  
                      double time_quantum) {
```

```
    for /* Each body */ {  
        // Compute the new vel
```

Number of  
bodies

```
        // Update the position of ith body based on  
        // the average of its old and new velocity.
```

```
        // Set the new velocity of ith body.
```

```
    }  
}
```

# Abstract Implementation

```
void update_positions(int nbodies, body_t *bodies,  
                      double time_quantum) {
```

```
    for /* Each body */ {  
        // Compute the new veloc
```

Array storing force,  
velocity, position, and  
mass of each body.

```
        // Update the position of ith body based on  
        // the average of its old and new velocity.
```

```
        // Set the new velocity of ith body.
```

```
    }  
}
```

# Abstract Implementation

```
void update_positions(int nbodies, body_t *bodies,  
                      double time_quantum) {
```

```
    for /* Each body */ {  
        // Compute the new veloc
```

Length of time that  
modeled by one step of  
the simulation.

```
        // Update the position of ith body based on  
        // the average of its old and new velocity.
```

```
        // Set the new velocity of ith body.
```

```
    }  
}
```

# Abstract Implementation

```
void update_positions(int nbodies, body_t *bodies,  
                      double time_quantum) {  
  
    for /* Each body */ {  
        // Compute the new velocity of ith body.  
  
        // Update the position of ith body based on  
        // the average of its old and new velocity.  
  
        // Set the new velocity of ith body.  
    }  
}
```

Let's look at a few different implementations of this function.

Starting point: A **fast** baseline that uses abstractions **minimally**.

# Implementation 1: Explicit

```
void update_positions(int nbodies, double *bodies,  
                      double time_quantum) {
```

```
    for (
```

// Compute the new velocity of ith body.

// Update the position of ith body

// the average of its old and new velocity.

```
}
```

```
}
```

Simple representation:  
Every 7 consecutive  
**double**s stores the  
position, velocity, force,  
and mass of 1 body.

Body 1  
position in x

Body 1  
velocity in x

Body 1  
force in x

Body 1  
mass

// Set the new velocity of ith body.

Array **bodies**:

Body 1  
position in y

Body 1  
velocity in y

Body 1  
force in y

# Implementation 1: Explicit

```
void update_positions(int nbodies, double *bodies,
                      double time_quantum) {
    for (int i = 0; i < 7 * nbodies; i += 7) {
        // Compute the new velocity of ith body.

        // Update the position of ith body based on
        // the average of its old and new velocity.

        // Set the new velocity of ith body.

    }
}
```

Loop over  
the bodies.

# Implementation 1: Explicit

```
void update_positions(int nbodies, double *bodies,
                      double time_quantum) {
    Force in x    i = 0; i < 7 * nbodies; i += 7) {
        // Compute the new velocity of ith body.
        double new_velocity_x = bodies[i + 4] *
            (time_quantum / bodies[i + 6]);
        double new_velocity_y = bodies[i + 5] *
            (time_quantum / bodies[i + 6]);
        // Update the position of ith body based on
        // average of its old and new velocity.
        // Set the new velocity of ith body.
    }
}
```

Compute new velocity as force times time quantum divided by mass.

Mass

# Implementation 1: Explicit

```
void update_positions(int nbodies, double *bodies,
                      double time_quantum) {
    for (int i = 0; i < 7 * nbodies; i += 7) {
        // Compute the new velocity of ith body.
        double new_velocity_x = bodies[i + 4] *
            (time_quantum / bodies[i + 6]);
        Position in x
        double new_velocity_y = bodies[i + 5] *
            (time_quantum / bodies[i + 6]);
        // Update the position of ith body based on
        // the average of its old and new velocity.
```

```
bodies[i] += (bodies[i + 2] + new_velocity_x) *
    (time_quantum / 2.0);
```

```
bodies[i + 1] += (bodies[i + 3] + new_velocity_y) *
    (time_quantum / 2.0);
    // Set the new velocity of ith body.
```

Position in y

```
}
```

```
}
```

Update position  
using average  
velocity and time  
quantum.

Previous  
velocity in x

Previous  
velocity in y

# Implementation 1: Explicit

```
void update_positions(int nbodies, double *bodies,
                      double time_quantum) {
    for (int i = 0; i < 7 * nbodies; i += 7) {
        // Compute the new velocity of ith body.
        double new_velocity_x = bodies[i + 4] *
            (time_quantum / bodies[i + 6]);
        double new_velocity_y = bodies[i + 5] *
            (time_quantum / bodies[i + 6]);
        // Update the position of ith body based on
        // the average of its old and new velocity.
        bodies[i] += (bodies[i + 2] + new_velocity_x) *
            (time_quantum / 2.0);
        bodies[i + 1] += (bodies[i + 3] + new_velocity_y) *
            (time_quantum / 2.0);
        // Set the new velocity of ith body.
        bodies[i + 2] = new_velocity_x;
        bodies[i + 3] = new_velocity_y;
    }
}
```

Store new  
velocities

# SCALAR OPTIMIZATIONS



# Implementation 1: Explicit

Let's see how the compiler optimizes this code (aside from vectorization).

```
void update_positions(int nbodies, double *bodies,
                      double time_quantum) {
    for (int i = 0; i < 7 * nbodies; i += 7) {
        // Compute the new velocity of ith body.
        double new_velocity_x = bodies[i + 4] *
            (time_quantum / bodies[i + 6]);
        double new_velocity_y = bodies[i + 5] *
            (time_quantum / bodies[i + 6]);
        // Update the position of ith body based on
        // the average of its old and new velocity.
        bodies[i] += (bodies[i + 2] + new_velocity_x) *
            (time_quantum / 2.0);
        bodies[i + 1] += (bodies[i + 3] + new_velocity_y) *
            (time_quantum / 2.0);
        // Set the new velocity of ith body.
        bodies[i + 2] = new_velocity_x;
        bodies[i + 3] = new_velocity_y;
    }
}
```

First focus

Second focus

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /
             bodies[i + 6]);
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

Note: This LLVM IR has been partially optimized for brevity.

```
%12 = add nuw nsw i64 %11, 4          LLVM IR
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    double new_velocity_x =
        bodies[i + 4] *
        (time_quantum /
         bodies[i + 6]);
    double new_velocity_y =
        bodies[i + 5] *
        (time_quantum /
         bodies[i + 6]);
    // ...
}
}
```

Register %11  
stores i.

```
%12 = add nuw nsw i64 %11, 4          LLVM IR
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /
             bodies[i + 6]);
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

```
%12 = add nuw nsw i64 %11, 4    LLVM IR
%13 = getelementptr inbounds double,
      double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
      double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
      double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
      double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /
             bodies[i + 6]),
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

```
%12 = add nuw nsw i64 %11, 4    LLVM IR
%13 = getelementptr inbounds double,
      double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
      double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
      double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
      double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /    LLVM IR
             bodies[i + 6]);
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    double new_velocity_x =
        bodies[i + 4] *
        (time_quantum /
         bodies[i + 6]);
    double new_velocity_y =
        bodies[i + 5] *
        (time_quantum /
         bodies[i + 6]);
    // ...
}
}
```

LLVM IR

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    double new_velocity_x =
        bodies[i + 4] *
        (time_quantum /
         bodies[i + 6]);
    double new_velocity_y =
        bodies[i + 5] *          ←
        (time_quantum /           ←
         bodies[i + 6]);
    // ...
}
}
```

```
%12 = add nuw nsw i64 %11, 4    LLVM IR
%13 = getelementptr inbounds double,
      double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
      double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
      double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
      double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /
             bodies[i + 6]);
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

LLVM IR

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    double new_velocity_x =
        bodies[i + 4] *
        (time_quantum /
         bodies[i + 6]);
    double new_velocity_y =
        bodies[i + 5] *
        (time_quantum /    LLVM IR
         bodies[i + 6]);
    // ...
}
}
```

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# LLVM IR for Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    double new_velocity_x =
        bodies[i + 4] *
        (time_quantum /
         bodies[i + 6]);
    double new_velocity_y =
        bodies[i + 5] *    LLVM IR
        (time_quantum /
         bodies[i + 6]);
    // ...
}
}
```

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26
```

# Optimizing Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /
             bodies[i + 6]),
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

Eliminate the  
common  
subexpression!

```
LLVM IR
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = add nuw nsw i64 %11, 6
%24 = getelementptr inbounds double,
       double* %1, i64 %23
%25 = load double, double* %24, align 8
%26 = fdiv double %2, %25
%27 = fmul double %22, %26 %17
```

# Optimizing Implementation 1

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /  
             bodies[i + 6]);
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /  
             bodies[i + 6]);
        // ...
    }
}
```

Eliminate the  
common  
subexpression!

LLVM IR

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = fdiv double %2, %17
%24 = fmul double %22, %23
               %18
```

# Optimizing Implementation 1: Result

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i + 4] *
            (time_quantum /
             bodies[i + 6]);
        double new_velocity_y =
            bodies[i + 5] *
            (time_quantum /
             bodies[i + 6]);
        // ...
    }
}
```

LLVM IR

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double,
       double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double,
       double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double,
       double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = fmul double %22, %18
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
            (time_quantum/2.0);
        // ...
    }
}
```

Register %11  
stores i.

```
5: ; preds = %3 LLVM IR
...
    br label %10

10: ; preds = %5, %10
...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
    double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
    double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %36, %35
...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    // ...
    bodies[i] +=
        (bodies[i + 2] +
         new_velocity_x) *
        (time_quantum/2.0);
    bodies[i + 1] +=
        (bodies[i + 3] +
         new_velocity_y) *
        (time_quantum/2.0);
    // ...
}
}
```

5: ; preds = %3 LLVM IR

...  
br label %10

10: ; preds = %5, %10

...

```
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
       double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
       double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %36, %35
...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
            (time_quantum/2.0);
        // ...
    }
}
```

LLVM IR

```
5: ... ; preds = %3
      br label %10
10: ... ; preds = %5, %10
      ...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
       double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
      ...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
       double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %36, %35
      ...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
(time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
(time_quantum/2.0);
        // ...
    }
}
```

LLVM IR

```
5: ... ; preds = %3
      br label %10
10: ... ; preds = %5, %10
      ...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
       double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
      ...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
       double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %36, %35
      ...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
  for (int i = /* ... */ ) {
    // ...
    bodies[i] +=
      (bodies[i + 2] +
       new_velocity_x) *
(time_quantum/2.0);
    bodies[i + 1] +=
      (bodies[i + 3] +
       new_velocity_y) *
(time_quantum/2.0);
    // ...
  }
}
```

```
5: ; preds = %3 LLVM IR
...
  br label %10

10: ; preds = %5, %10
...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
  double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
  double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %36, %35
...
  br i1 %42, label %10, label %9, !llvm.loop !9
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    // ...
    bodies[i] +=
        (bodies[i + 2] +
         new_velocity_x) *
        (time_quantum/2.0);
    bodies[i + 1] +=
        (bodies[i + 3] +
         new_velocity_y) *
        (time_quantum/2.0);
    // ...
}
}
```

5: ; preds = %3 LLVM IR

...  
br label %10

10: ; preds = %5, %10

...  
%24 = add nuw nsw i64 %11, 2  
%25 = getelementptr inbounds double,  
 double\* %1, i64 %24  
%26 = load double, double\* %25, align 8  
%27 = fadd double %26, %19  
%28 = fdiv double %2, 2.000000e+00  
%29 = fmul double %28, %27

...  
%32 = add nuw nsw i64 %11, 3  
%33 = getelementptr inbounds double,  
 double\* %1, i64 %32  
%34 = load double, double\* %33, align 8  
%35 = fadd double %23, %34  
%36 = fdiv double %2, 2.000000e+00  
%37 = fmul double %36, %35

...  
br i1 %42, label %10, label %9, !llvm.loop !9

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
            (time_quantum/2.0);
        // ...
    }
}
```

LLVM IR

```
5: ... ; preds = %3
      br label %10
10: ... ; preds = %5, %10
      ...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
       double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
      ...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
       double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %36, %35
      ...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
(time_quantum/2.0);
        // ...
    }
}
```

5: ; preds = %3 LLVM IR

...  
br label %10

10: ; preds = %5, %10  
...  
%24 = add nuw nsw i64 %11, 2  
%25 = getelementptr inbounds double,  
 double\* %1, i64 %24  
%26 = load double, double\* %25, align 8  
%27 = fadd double %26, %19  
%28 = fdiv double %2, 2.000000e+00  
%29 = fmul double %28, %27  
...  
%32 = add nuw nsw i64 %11, 3  
%33 = getelementptr inbounds double,  
 double\* %1, i64 %32  
%34 = load double, double\* %33, align 8  
%35 = fadd double %23, %34  
%36 = fdiv double %2, 2.000000e+00  
%37 = fmul double %36, %35  
...  
br i1 %42, label %10, label %9, !llvm.loop !9

# LLVM IR for Implementation 1, Cont'd

```
void update_positions(    C
    int nbodies,
    double *bodies,
    double time_quantum) {
for (int i = /* ... */) {
    // ...
    bodies[i] +=
        (bodies[i + 2] +
         new_velocity_x) *
        (time_quantum/2.0);
    bodies[i + 1] +=
        (bodies[i + 3] +
         new_velocity_y) *
        (time_quantum/2.0);
    // ...
}
}
```

5: ; preds = %3 LLVM IR

...  
br label %10

10: ; preds = %5, %10

...  
%24 = add nuw nsw i64 %11, 2  
%25 = getelementptr inbounds double,  
double\* %1, i64 %24  
%26 = load double, double\* %25, align 8  
%27 = fadd double %26, %19  
%28 = fdiv double %2, 2.000000e+00  
%29 = fmul double %28, %27  
...  
%32 = add nuw nsw i64 %11, 3  
%33 = getelementptr inbounds double,  
double\* %1, i64 %32  
%34 = load double, double\* %33, align 8  
%35 = fadd double %23, %34  
%36 = fdiv double %2, 2.000000e+00  
%37 = fmul double %36, %35

...  
br i1 %42, label %10, label %9, !llvm.loop !9

# Optimizing Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
(time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
(time_quantum/2.0);
        // ...
    }
}
```

Eliminate the common subexpression!

LLVM IR

```
5: ; preds = %3
...
br label %10

10: ; preds = %5, %10
...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
       double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
       double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fdiv double %2, 2.000000e+00
%37 = fmul double %28, %35
...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# Optimizing Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
            (time_quantum/2.0);
        // ...
    }
}
```

Replace with a  
cheaper  
operation!

```
5: ; preds = %3 LLVM IR
...
    br label %10

10: ; preds = %5, %10
...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
    double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fdiv double %2, 2.000000e+00
%29 = fmul double %28, %27
%28 = fmul double %2, 5.000000e-01
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
    double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34

%36 = fmul double %28, %35
...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# Optimizing Implementation 1, Cont'd

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */ ) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
            (time_quantum/2.0);
        // ...
    }
}
```

Move loop-invariant code out of loop!

```
5: ; preds = %3 LLVM IR
...
    br label %10

10: ; preds = %5, %10
...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
    double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fmul double %2, 5.000000e-01
%29 = fmul double %28, %27
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
    double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
...
%36 = fmul double %28, %35
...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# Optimizing Implementation 1, Cont'd: Result

```
void update_positions(C
    int nbodies,
    double *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        // ...
        bodies[i] +=
            (bodies[i + 2] +
             new_velocity_x) *
            (time_quantum/2.0);
        bodies[i + 1] +=
            (bodies[i + 3] +
             new_velocity_y) *
            (time_quantum/2.0);
        // ...
    }
}
```

```
5: ; preds = %3 LLVM IR
...
%28 = fmul double %2, 5.000000e-01
br label %10

10: ; preds = %5, %10
...
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double,
    double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19

%29 = fmul double %28, %27
...
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double,
    double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34

%36 = fmul double %28, %35
...
br i1 %42, label %10, label %9, !llvm.loop !9
```

# Optimized Implementation 1

## LLVM IR for loop body, -O1 optimization

```
%12 = add nuw nsw i64 %11, 4
%13 = getelementptr inbounds double, double* %1, i64 %12
%14 = load double, double* %13, align 8
%15 = add nuw nsw i64 %11, 6
%16 = getelementptr inbounds double, double* %1, i64 %15
%17 = load double, double* %16, align 8
%18 = fdiv double %2, %17
%19 = fmul double %14, %18
%20 = add nuw nsw i64 %11, 5
%21 = getelementptr inbounds double, double* %1, i64 %20
%22 = load double, double* %21, align 8
%23 = fmul double %22, %18
%24 = add nuw nsw i64 %11, 2
%25 = getelementptr inbounds double, double* %1, i64 %24
%26 = load double, double* %25, align 8
%27 = fadd double %26, %19
%28 = fmul double %7, %27
%29 = getelementptr inbounds double, double* %1, i64 %11
%30 = load double, double* %29, align 8
%31 = fadd double %30, %28
store double %31, double* %29, align 8
%32 = add nuw nsw i64 %11, 3
%33 = getelementptr inbounds double, double* %1, i64 %32
%34 = load double, double* %33, align 8
%35 = fadd double %23, %34
%36 = fmul double %7, %35
%37 = add nuw nsw i64 %11, 1
%38 = getelementptr inbounds double, double* %1, i64 %37
%39 = load double, double* %38, align 8
%40 = fadd double %39, %36
store double %40, double* %38, align 8
store double %19, double* %25, align 8
store double %23, double* %33, align 8
```

### Summary:

- 7 **loads**
- 1 **fdiv**
- 4 **fmul**
- 4 **fadd**
- 4 **stores**

# OPTIMIZING STRUCTURE USAGE



# N-Body Simulation Structures

Suppose we try  
cleaning up the code  
by introducing some  
higher-level  
structures.

## 2D vector

```
typedef struct vec_t {  
    double x, y;  
} vec_t;
```

## Data structure for a body

```
typedef struct body_t {  
    // Position vector  
    vec_t position;  
    // Velocity vector  
    vec_t velocity;  
    // Force vector  
    vec_t force;  
    // Mass  
    double mass;  
} body_t;
```

# Implementation 2: Using Structures

```
void update_positions(int nbodies, body_t *bodies,
                      double time_quantum) {
    for (int i = 0; i < nbodies; ++i) {
        // Compute the new velocity of ith body.
        double new_velocity_x = bodies[i].force.x *
            (time_quantum / bodies[i].mass);
        double new_velocity_y = bodies[i].force.y *
            (time_quantum / bodies[i].mass);
        // Update the position of ith body based on
        // the average of its old and new velocity.
        bodies[i].position.x +=
            (bodies[i].velocity.x + new_velocity_x) *
            (time_quantum / 2.0);
        bodies[i].position.y +=
            (bodies[i].velocity.y + new_velocity_y) *
            (time_quantum / 2.0);
        // Set the new velocity of ith body.
        bodies[i].velocity.x = new_velocity_x;
        bodies[i].velocity.y = new_velocity_y;
    }
}
```

Our focus

Array indices have been replaced with accesses to structure members.

QUESTION: What does this abstraction **cost** in performance?

# The `getelementptr` Instruction

The `getelementptr` instruction computes a memory address from a `pointer` and a `list of indices`.

C code

```
vec_t* A;  
A[i].y;
```

Example: Compute the address

`(%struct.vec_t* %0 + %1) + 1` using `pointer arithmetic`.

```
%3 = getelementptr inbounds %struct.vec_t,  
      %struct.vec_t* %0, i64 %1, i32 1
```

Pointer to the  
memory for `A`

Index `i`

Field number of  
`y` in `vec_t`.

See <https://llvm.org/docs/GetElementPtr.html>

# LLVM IR for Implementation 2

```
void update_positions(C
    int nbodies,
    body_t *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
bodies[i].force.x *
(time_quantum /
bodies[i].mass);
// ...
    }
}
```

LLVM IR

```
%18 = getelementptr inbounds %struct.body_t,
       %struct.body_t* %15, i64 %17
%19 = getelementptr inbounds %struct.body_t,
       %struct.body_t* %18, i32 0, i32 2
%20 = getelementptr inbounds %struct.vec_t,
       %struct.vec_t* %19, i32 0, i32 0
%21 = load double, double* %20, align 8
```

# LLVM IR for Implementation 2

```
void update_positions(C
    int nbodies,
    body_t *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i].force.x *
            (time_quantum /
             bodies[i].mass);
    } // ...
}
```

Register **%15** stores **bodies** and register **%17** stores **i**.

```
LLVM IR
%18 = getelementptr inbounds %struct.body_t,
        %struct.body_t* %15, i64 %17
%19 = getelementptr inbounds %struct.body_t,
        %struct.body_t* %18, i32 0, i32 2
%20 = getelementptr inbounds %struct.vec_t,
        %struct.body_t* %19, i32 0
        , align 8
```

Compute the address of the **body\_t** at  
**%15 + %17\*sizeof(body\_t)**.

# LLVM IR for Implementation 2

```
void update_positions(C
    int nbodies,
    body_t *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
            bodies[i].force.x *
            (time_quantum /
             bodies[i].mass);
    } // ...
}
```

```
typedef struct body_t {C
    // Position vector
    vec_t position;
    // Velocity vector
    vec_t velocity;
    // Force vector
    vec_t force;
    // Mass
    double mass;
} body_t;
```

```
%18 = getelementptr inbounds %struct.body_t,
       %struct.body_t* %15, i64 %17
%19 = getelementptr inbounds %struct.body_t,
       %struct.body_t* %18, i32 0, i32 2
%20 = getelementptr inbounds %struct.vec_t,
```

Effect: Compute the address of field 2 in `body_t` at `%18`:  
`%18 + 0*sizeof(body_t) + 2*sizeof(vec_t)`.

# LLVM IR for Implementation 2

```
void update_positions(  
    int nbodies,  
    body_t *bodies,  
    double time_quantum) {  
    for (int i = /* ... */ ) {  
        double new_velocity_x =  
            bodies[i].force.x *  
            (time_quantum /  
             bodies[i].mass);  
        // ...  
    }  
}
```

```
typedef struct vec_t {  
    double x, y;  
} vec_t;
```

Effect: Compute the address of field `0` in `vec_t` at `%19`:  
`%19 + 0*sizeof(vec_t) + 0*sizeof(double)`.

LLVM IR

```
%18 = getelementptr inbounds %struct.body_t,  
      %struct.body_t* %15, i64 %17  
%19 = getelementptr inbounds %struct.body_t,  
      %struct.body_t* %18, i32 0, i32 2  
%20 = getelementptr inbounds %struct.vec_t,  
      %struct.vec_t* %19, i32 0, i32 0  
%21 = load double, double* %20, align 8
```

# LLVM IR for Implementation 2

```
void update_positions(C
    int nbodies,
    body_t *bodies,
    double time_quantum) {
    for (int i = /* ... */) {
        double new_velocity_x =
bodies[i].force.x *
            (time_quantum /
             bodies[i].mass);
    // ...
    }
}
```

LLVM IR

```
%18 = getelementptr inbounds %struct.body_t,
        %struct.body_t* %15, i64 %17
%19 = getelementptr inbounds %struct.body_t,
        %struct.body_t* %18, i32 0, i32 2
%20 = getelementptr inbounds %struct.vec_t,
        %struct.vec_t* %19, i32 0, i32 0
%21 = load double, double* %20, align 8
```

# Optimizing Implementation 2

```
void update_positions(  
    int nbodies,  
    body_t *bodies,  
    double time_quantum) {  
    for (int i = /* ... */ ) {  
        double new_velocity_x  
        bodies[i].force.x *  
        (time_quantum /  
         bodies[i].mass);  
        // ...  
    }  
}
```

C

Simplify the arithmetic of the address calculation!

Effect: Compute the address at  
 $\%15 + \%17 * \text{sizeof}(\text{body\_t}) +$   
 $2 * \text{sizeof}(\text{vec\_t}) +$   
 $0 * \text{sizeof}(\text{double})$ .

~~%18 = getelementptr inbounds %struct.body\_t,  
 %struct.body\_t\* %15, i64 %17~~  
~~%19 = getelementptr inbounds %struct.body\_t,  
 %struct.body\_t\* %18, i32 0, i32 2~~  
~~%20 = getelementptr inbounds %struct.vec\_t,  
 %struct.vec\_t\* %19, i32 0, i32 0~~

LLVM IR

%20 = getelementptr inbounds %struct.body\_t,  
 %struct.body\_t\* %15, i64 %17, i32 2, i32 0

8

# Optimized Implementation 2

## LLVM IR for loop body, -O1 optimization

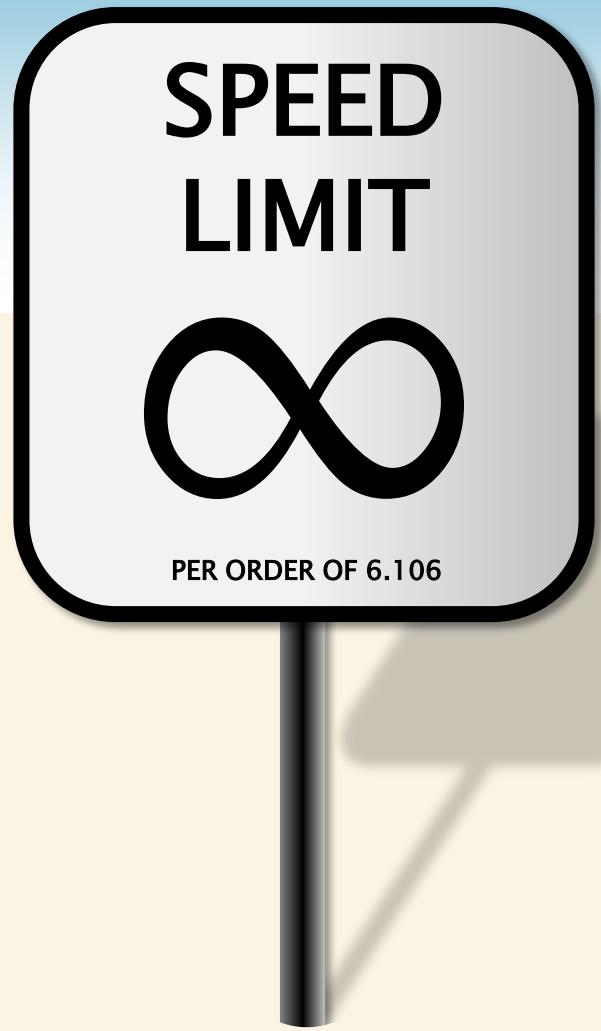
```
%11 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 2, i32 0
%12 = load double, double* %11, align 8
%13 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 3
%14 = load double, double* %13, align 8
%15 = fdiv double %2, %14
%16 = fmul double %12, %15
%17 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 1
%18 = load double, double* %17, align 8
%19 = fmul double %18, %15
%20 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 1
%21 = load double, double* %20, align 8
%22 = fadd double %21, %16
%23 = fmul double %6, %22
%24 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 1
%25 = load double, double* %24, align 8
%26 = fadd double %25, %23
store double %26, double* %24, align 8
%27 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 1, i32 1
%28 = load double, double* %27, align 8
%29 = fadd double %19, %28
%30 = fmul double %6, %29
%31 = getelementptr inbounds %struct.body_t, %struct.body_t* %1,
%32 = load double, double* %31, align 8
%33 = fadd double %32, %30
store double %33, double* %31, align 8
store double %16, double* %20, align 8
store double %19, double* %27, align 8
```

### Summary:

- 7 loads
- 1 fdiv
- 4 fmul
- 4 fadd
- 4 stores

Same as  
implementation 1!

# FUNCTION INLINING



# Basic Routines for 2D Vectors

Let's introduce  
a few simple  
**routines** to  
operate on 2D  
vectors.

```
typedef struct vec_t {  
    double x, y;  
} vec_t;  
  
static vec_t vec_add(vec_t a, vec_t b) {  
    vec_t sum = { a.x + b.x, a.y + b.y };  
    return sum;  
}  
  
static vec_t vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };  
    return scaled;  
}
```

# Implementation 3: Function Calls

```
void update_positions(int nbodies, body_t *bodies,
                      double time_quantum) {
    for (int i = 0; i < nbodies; ++i) {
        // Compute the new velocity of
        vec_t new_velocity =
            vec_scale(bodies[i].force,
                      time_quantum / bodies[i].mass);
        // Update the position of ith body
        // the average of its old and new velocity.
        bodies[i].position =
            vec_add(bodies[i].position,
                    vec_scale(vec_add(bodies[i].velocity,
                                      new_velocity),
                              time_quantum / 2.0));
        // Set the new velocity of ith body
        bodies[i].velocity = new_velocity;
    }
}
```

Our focus

This code expresses operations in terms of vectors instead of individual components.

QUESTION: What does this abstraction cost in performance?

# Example: Updating Positions

Let's see how the compiler optimizes function calls  
(after other optimizations).

C code from update\_positions

```
vec_add(bodies[i].position,  
        vec_scale(vec_add(bodies[i].velocity,  
                           new_velocity),  
                  time_quantum / 2.0));
```

Extract values  
from a **struct**-  
type LLVM IR  
register.

LLVM IR

```
%24 = extractvalue { double, double } %23, 0  
%25 = extractvalue { double, double } %23, 1  
%26 = fmul double %2, 5.000000e-01  
%27 = call { double, double } @vec_scale(double %24,  
                                         double %25, double %26)
```

Call  
**vec\_scale**.

# Function Inlining

LLVM IR snippet from update\_positions

LLVM IR for vec\_scale

```
define internal { double, double }  
@vec_scale(double %0, double %1, doubl  
    %4 = fmul double %0, %2  
    %5 = fmul double %1, %2  
  
    %24 = extractvalue { double, double } %23, 0  
    %25 = extractvalue { double, double } %23, 1  
    %26 = fmul double %2, 5.000000e-01  
    %27 = call { double, double } @vec_scale(double %24,  
        double %25, double %26)  
}
```

IDEA: The code for vec\_scale is small, so copy-paste it into the call site.

# Function Inlining

LLVM IR snippet from update\_positions

```
%24 = extractvalue { double, double } %23, 0
%25 = extractvalue { double, double } %23, 1
%26 = fmul double %2, 5.000000e-01
%27 = call { double, double }
    @vec_scale(double %24, double %25, double %26)
%4.in = fmul double %24, %26
%5.in = fmul double %25, %26
%28 = insertvalue { double, double } undef, double %4.in, 0
%29 = insertvalue { double, double } %28, double %5.in, 1
ret { double, double } %29
```

Step 1: Copy  
code from  
`vec_scale`.

Step 2: Remove call  
and return.

# Further Optimization

```
%24 = extractvalue { double, double } %23, 0
%25 = extractvalue { double, double } %23, 1
%26 = fmul double %2, 5.000000e-01

%4.in = fmul double %24, %26
%5.in = fmul double %25, %26
%28 = insertvalue { double, double } undef, double %4.in, 0
%29 = insertvalue { double, double } %28, double %5.in, 1

%20 = fmul double %2, 5.000000e-01
%4.in = fmul double %24, %26
%5.in = fmul double %25, %26
```

immediately unpack them.

```
%28 = insertvalue { double, double } undef, double %4.in, 0
%29 = insertvalue { double, double } %28, double %5.in, 1
%30 = extractvalue { double, double } %4.in, 0
%31 = extractvalue { double, double } %5.in, 1
```

IDEA: Remove these useless operations.

# Sequences of Function Calls

C code

```
vec_add(bodies[i].position,  
        vec_scale(vec_add(bodies[i].velocity,  
                           new_velocity),  
                  time_quantum / 2.0));
```

LLVM IR

```
%23 = call { double, double } @vec_add(double %20,  
                                         double %22, double %17, double %18)
```

```
%24 = extractvalue { double, double } %23, 0
```

```
%25 = extractvalue { double, double } %23, 1
```

```
%26 = fmul double %2, 5.000000e-01
```

```
%4.in = fmul double %24, %26
```

```
%5.in = fmul double %25, %26
```

```
...
```

```
%34 = call { double, double } @vec_
```

```
double %33, double %4.in, dou
```

IDEA: Inline  
vec\_add as well.

...and then remove  
useless instructions...

# Sequences of Function Calls

C code

```
vec_add(bodies[i].position,
         vec_scale(vec_add(bodies[i].velocity,
                           new_velocity),
                   time_quantum / 2.0));
```

Optimized LLVM IR

```
%22 = fadd double %19, %16
%23 = fadd double %21, %17
%26 = fmul double %2, 5.000000e-01
%4.in = fmul double %22, %26
%5.in = fmul double %23, %26
...
%31 = fadd double %28, %4.in
%32 = fadd double %30, %5.in
```

SUMMARY: Function inlining and additional transformations can eliminate the cost of the function abstraction.

# LLVM IR for Implementation 3

## Optimized LLVM IR for loop body (without vectorization)

```
%11 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 3
%12 = load double, double* %11, align 8
%13 = fdiv double %2, %12
%14 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32
%15 = load double, double* %14, align 8
%16 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32
%17 = load double, double* %16, align 8
%18 = fmul double %15, %13
%19 = fmul double %13, %17
%20 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32
%21 = load double, double* %20, align 8
%22 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32
%23 = load double, double* %22, align 8
%24 = fadd double %18, %21
%25 = fadd double %19, %23
%26 = fmul double %6, %24
%27 = fmul double %6, %25
%28 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %10, i32 0, i32 0
%29 = load double, double* %28, align 8
%30 = getelementptr inbounds %struct.body_t, %struct.body_t* %1, i64 %
%31 = load double, double* %30, align 8
%32 = fadd double %29, %26
%33 = fadd double %31, %27
store double %32, double* %28, align 8
store double %33, double* %30, align 8
store double %18, double* %20, align 8
store double %19, double* %22, align 8
```

### Summary:

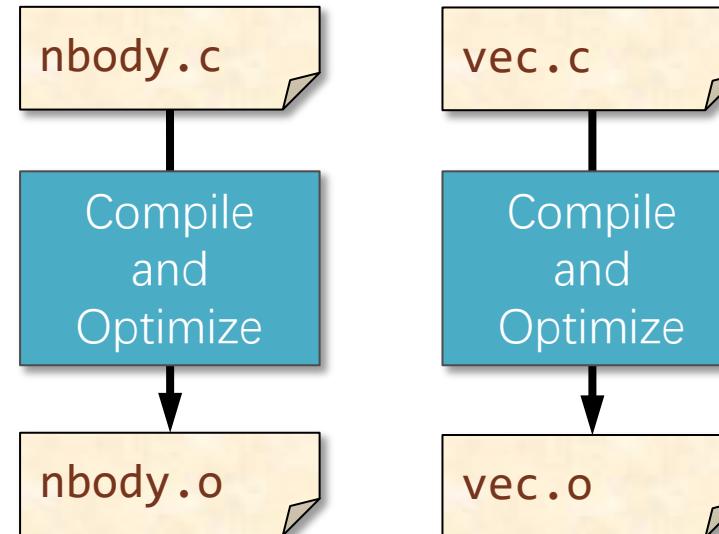
- 7 loads
- 1 fdiv
- 4 fmul
- 4 fadd
- 4 stores

Same as  
implementations 1  
and 2!

# Problems with Function Inlining

Why doesn't the compiler inline **all** function calls?

- Some function calls, such as recursive calls, **cannot be inlined** except in special cases, e.g., “recursive tail calls.”
- The compiler cannot inline a function defined in another **compilation unit** unless one uses **whole-program optimization**.
- Function inlining can **increase code size**, which can hurt performance.



# Controlling Function Inlining

**QUESTION:** How does the compiler **know** whether or not inlining a function will hurt performance?

**ANSWER:** It doesn't know. It makes a **best guess** based on **heuristics**, such as the function's size.

Tips for controlling function inlining:

- Mark functions that should **always** be inlined with `__attribute__((always_inline))`.
- Mark functions that should **never** be inlined with `__attribute__((noinline))`.
- Use **link-time optimization (LTO)** to enable whole-program optimization.

# ASSEMBLY-LEVEL OPTIMIZATIONS



# Arithmetic Opt's: C vs. LLVM IR

Many compiler optimizations happen on the compiler's **intermediate representation (IR)**, although not all of them.

EXAMPLES: Let **n** be a **uint32\_t**.

## C code

```
uint32_t x = n * 8;
```

```
uint32_t y = n * 15;
```

```
uint32_t z = n / 71;
```

Register **%0** holds  
the value of **n**.

## LLVM IR

```
%2 = shl nsw i32 %0, 3
```

```
%3 = mul nsw i32 %0, 15
```

```
%4 = udiv i32 %0, 71
```

# Arithmetic Opt's: C vs. Assembly

Most compiler optimizations happen on the compiler's **intermediate representation (IR)**, although not all of them.

EXAMPLES: Let **n** be a **uint32\_t**.

## C code

```
uint32_t x = n * 8;
```

```
uint32_t y = n * 15;
```

```
uint32_t z = n / 71;
```

**Magic number** equal to  $2^{38}/71+1$ .

## Assembly

```
leal    (,%rdi,8), %eax
```

```
leal    (%rdi,%rdi,4), %eax  
leal    (%rax,%rax,2), %eax
```

```
movl    %edi, %eax  
movl    $3871519817, %ecx  
imulq   %rax, %rcx  
shrq    $38, %rcx
```

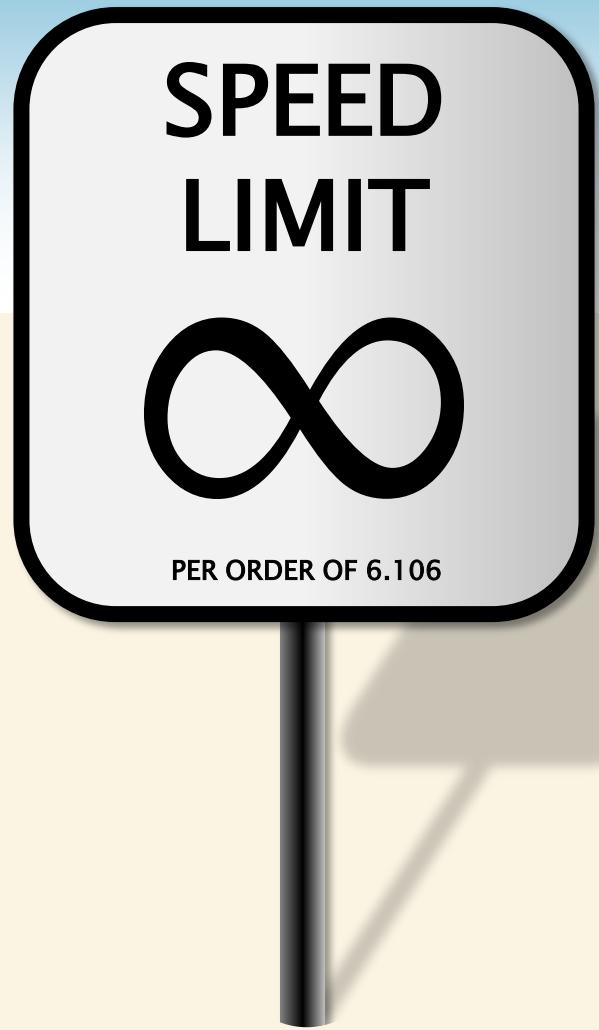
Register **%rdi** holds the value of **n**.

# Other Assembly-Level Opt's

Compilers perform a **variety** of optimizations at the assembly level, including the following:

- **Register allocation:** Choose which architecture registers will store different values at different points in time.
- **Instruction reordering:** Order assembly instructions to effectively use the specific processor pipeline.
- Standard compiler optimizations — such as loop-invariant-code motion or common-subexpression elimination — on instructions that emerge only at the assembly level.

# CASE STUDY 1



# Implementation 4: System Pointer

Suppose that we modify the n-body simulation code to encapsulate all of the arguments to **update\_positions** into a **single data structure**.

**Data structure  
for the system**

```
typedef struct system_t {  
    // Number of bodies  
    int nbodies;  
    // Array of bodies  
    body_t *bodies;  
    // Time quantum  
    double time_quantum;  
} system_t;
```

# Implementation 4: System Pointer

```
void update_positions(system_t *sys) {
    for (int i = 0; i < sys->nbodies; +-
        // Compute the new velocity of ith body
        vec_t new_velocity =
            vec_scale(sys->bodies[i].force,
                      sys->time_quantum / sys->bodies[i].mass);
        // Update the position of ith body based on
        // the average of its old and new velocity.
        sys->bodies[i].position =
            vec_add(sys->bodies[i].position,
                    vec_scale(vec_add(sys->bodies[i].velocity,
                                      new_velocity),
                              sys->time_quantum / 2.0));
        // Set the new velocity of ith body.
        sys->bodies[i].velocity = new_
    }
}
```

Accesses to **nbodies**, **time\_quantum**, and **bodies** go through the **sys** pointer.

QUESTION: What does this abstraction **cost** in performance?

# LLVM IR for Implementation 4

## LLVM IR for loop body, -O1 optimization

```
%11 = load %struct.body_t*, %struct.body_t** %6, align 8
%12 = load double, double* %7, align 8
%13 = getelementptr inbounds %struct.body_t, %struct.body_t* %11, i64 %10, i32 2
%
%11 = load %struct.body_t*, %struct.body_t** %6, align 8
%12 = load double, double* %7, align 8
%
%18 = load double, double* %16, align 8
%19 = load double, double* %18, align 8
%20 = fmul double %17, %15
%21 = fmul double %15, %19
%22 = getelementptr inbounds %struct.body_t, %struct.body_t*, i64 %10, i32 1, i32 0
%23 = load double, double* %22, align 8
%
%28 = fmul double %12, 5.000000e-01
%
%27 = fadd double %21, %25
%28 = fmul double %12, 5.000000e-01
```

Reload bodies and  
time\_quantum.

Recompute  
time\_quantum/2.0.

```
%37 = load %struct.body_t*, %struct.body_t** %6, align 8
%38 = getelementptr inbounds %struct.body_t, %struct.body_t* %37,
       i64 %10, i32 1, i32 0
store double %20, double* %38, align 8
%39 = getelementptr inbounds %struct.body_t, %struct.body_t* %37,
       i64 %10, i32 1, i32 1
store double %21, double* %39, align 8
%41 = load i32, i32* %2, align 8
%42 = sext i32 %41 to i64
%
%42 = sext i32 %41 to i64
```

Reload bodies  
and nbodies.

Summary:

- 11 loads
- 1 fdiv
- 5 fmul
- 4 fadd
- 4 stores

# What Went Wrong

PROBLEM: The stores in the function cause the compiler to act **conservatively**, because it is unsure about potential **memory aliasing**.

```
void update_positions(system_t *sys) {  
    for (int i = 0; i < sys->nbodies; ++i) {  
        // ...  
        sys->bodies[i].position = ...  
        // ...  
        sys->bodies[i].velocity = ...  
    }  
}
```

Compiler **fails** to deduce that these stores do not affect other parts of the system\_t structure.

# What Went Wrong, Cont'd

```
void update_positions(system_t *sys) {  
    for (int i = 0; i < sys->nbodies; ++i) {  
        // Compute the new velocity of body.  
        vec_t new_velocity =  
            vec_scale(sys->bodies[i].force,  
                      sys->time_quantum / sys->bodies[i].mass);  
        // Set the new velocity of body  
        sys->bodies[i].velocity = new_velocity;  
    }  
}
```

Compiler isn't sure that previous stores won't modify these values!

old and new positions of body  
n = i].position

Compiler can't prove that this value is loop-invariant.

```
vec_scale(vec_add(sys->bodies[i].velocity,  
                  new_velocity),  
          sys->time_quantum / 2.0));
```

// Set the new velocity of body.

```
sys->bodies[i].velocity = new_velocity;
```

# Fundamental Problem: Alias Analysis

**ALIAS-ANALYSIS PROBLEM:** Given **two pointers** in a program, determine **statically** — by studying the code itself — if they can **alias** — point to the **same** memory location.

- **BAD NEWS:** This problem is **undecidable**, meaning no compiler can solve it in general.
- In practice, compilers try to solve **common, special instances** of this problem, by analyzing the program's control flow, data types, programmer annotations — e.g., **const, restrict** — and more.
- Improvements to alias analysis make compiler optimizations a **moving target**.

# Implementation 5: Globals

```
int nbodies;
body_t *bodies;
double time_quantum;
void update_positions() {
    for (int i = 0; i < nbodies; ++i) {
        // Compute the new velocity of body.
        vec_t new_velocity =
            vec_scale(bodies[i].force,
                      time_quantum / bodies[i].mass);
        // Update the position of body based on
        // the average of its old and new velocity.
        bodies[i].position =
            vec_add(bodies[i].position,
                    vec_scale(vec_add(bodies[i].velocity,
                                      new_velocity),
                              time_quantum / 2.0));
        // Set the new velocity of body.
        bodies[i].velocity = new_velocity
    }
}
```

The `nbodies`, `time_quantum`, and `bodies` variables reside in global memory.

QUESTION: What does this change **cost** in performance?

# LLVM IR for Implementation 5

## LLVM IR for loop body, -O1 optimization

```
%8 = load %struct.body_t*, %struct.body_t** @bodies, align 8
%9 = load double, double* @time_quantum, align 8
%10 = getelementptr inbounds %struct.body_t, %struct.body_t* %8, i64 %7, i32 1, i32 0
%11 = load %struct.body_t*, %struct.body_t** @bodies, align 8
%12 = load double, double* @time_quantum, align 8
%13 = load double, double* @time_quantum, align 8
%14 = fmul double %12, %12
%15 = fmul double %12, %16
%16 = load double, double* @time_quantum, align 8
%17 = fmul double %14, %12
%18 = fmul double %12, %16
%19 = getelementptr inbounds %struct.body_t, %struct.body_t* %8, i64 %7, i32 1, i32 0
%20 = load double, double* %19, align 8
%21 = fadd double %18, %17
%22 = fmul double %9, 5.000000e-01
%23 = fadd double %21, %22
%24 = fmul double %9, 5.000000e-01
%25 = fmul double %9, 5.000000e-01
%26 = fadd double %24, %25
%27 = fmul double %9, 5.000000e-01
%28 = fadd double %26, %27
%29 = fmul double %9, 5.000000e-01
%30 = fadd double %28, %29
%31 = fmul double %9, 5.000000e-01
%32 = fadd double %30, %31
%33 = fmul double %9, 5.000000e-01
%34 = load %struct.body_t*, %struct.body_t** @bodies, align 8
%35 = getelementptr inbounds %struct.body_t, %struct.body_t* %34, i64 %7, i32 1, i32 0
store double %17, double* %35, align 8
%36 = getelementptr inbounds %struct.body_t, %struct.body_t* %34, i64 %7, i32 1, i32 1
store double %18, double* %36, align 8
store double %19, double* %36, align 8
%37 = getelementptr inbounds %struct.body_t, %struct.body_t* %34, i64 %7, i32 1, i32 0
store double %18, double* %37, align 8
```

Reload **bodies** and **time\_quantum**.

Recompute **time\_quantum/2.0**.

Reload **bodies**.

Similar problems to system-pointer version!

Summary:

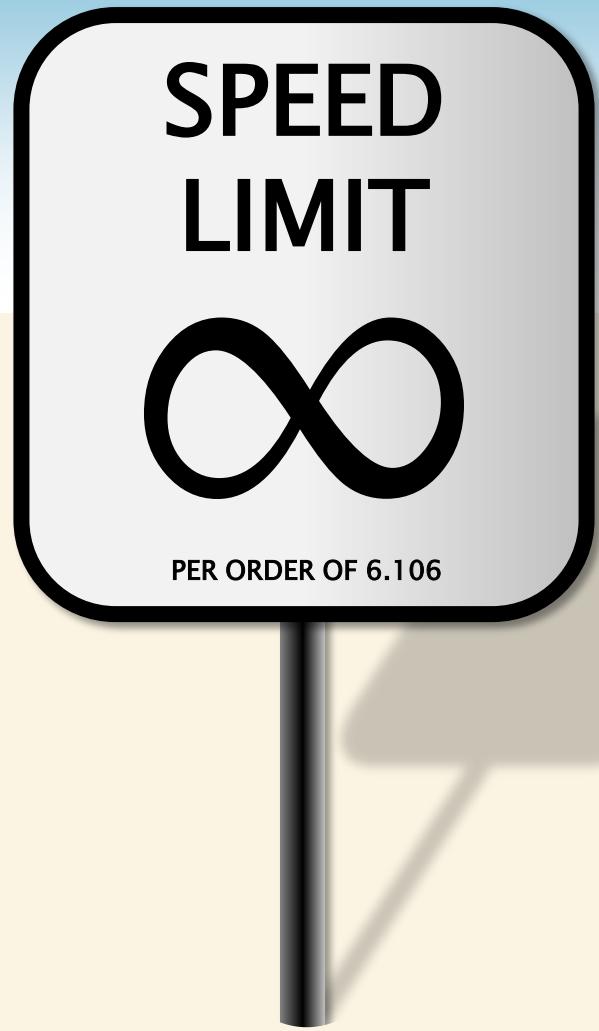
- 10 loads
- 1 fdiv
- 5 fmul
- 4 fadd
- 4 stores

# Take-Aways

The compiler must optimize code that uses **multiple pointers** conservatively, in case those pointers **alias**.

- Although compilers may **improve** their alias analysis over time, there will always be **deficiencies**.
- **Global data** can also trigger alias-analysis problems for the compiler.
- In some cases, the compiler will optimize code by generating **runtime checks** for memory aliasing.  
(See homework 3.)
- **TIP:** Use **qualifiers** (e.g., **const**, **restrict**) and **local variables** to help the compiler out.

## CASE STUDY 2



# Example: Simple Accesses

Compilers effectively optimize memory accesses based on simple integer offsets.

QUESTION: Does the compiler vectorize this code?

```
void foo(int *A, int n) {  
    for (int i = 0; i < n; ++i)  
        A[i]++;  
}
```

C



```
...  
%19 = getelementptr inbounds i32, i32* %0, i64 %17  
%20 = bitcast i32* %19 to <4 x i32>*  
%21 = load <4 x i32>, <4 x i32>* %20, align 4  
%22 = getelementptr inbounds i32, i32* %19, i64 4  
%23 = bitcast i32* %22 to <4 x i32>*  
%24 = load <4 x i32>, <4 x i32>* %23, align 4  
%25 = add nsw <4 x i32> %21, <i32 1, i32 1, i32 1, i32 1>  
%26 = add nsw <4 x i32> %24, <i32 1, i32 1, i32 1, i32 1>  
store <4 x i32> %25, <4 x i32>* %20, align 4  
store <4 x i32> %26, <4 x i32>* %23, align 4  
...
```

LLVM IR

Yes.

# Example: Indirect Accesses

QUESTION: Does the compiler vectorize this code?

```
void foo(int *restrict A,  
         const int *restrict B, int n) {  
    for (int i = 0; i < n; ++i)  
        A[B[i]]++;  
}
```

...

```
%31 = getelementptr inbounds i32, i32* %1, i64 %29  
%32 = load i32, i32* %31, align 4  
%33 = sext i32 %32 to i64  
%34 = getelementptr inbounds i32, i32* %0, i64 %33  
%35 = load i32, i32* %34, align 4  
%36 = add nsw i32 %35, 1  
store i32 %36, i32* %34, align 4  
...
```

LLVM IR

No.

# Fundamental Problem: Indirect Accesses

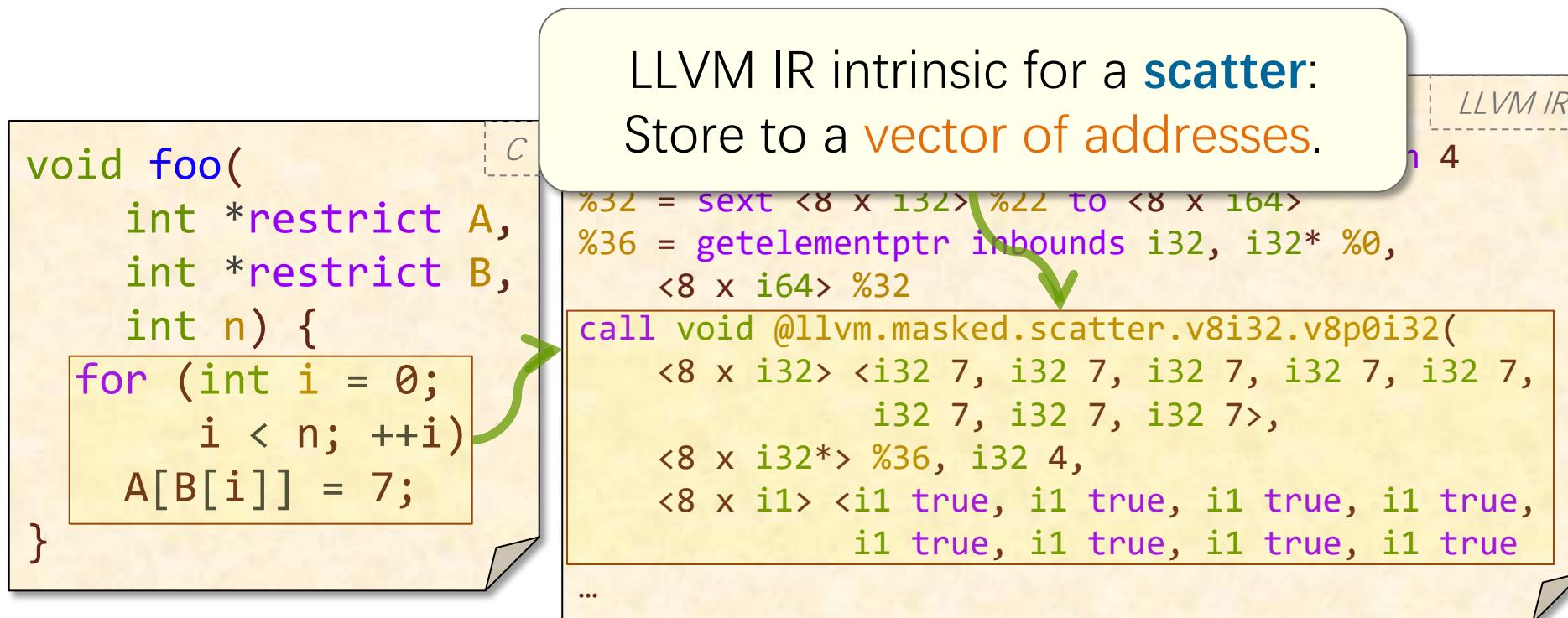
PROBLEM: Compilers have a hard time optimizing **indirect memory accesses**, where the memory location accessed is itself read from memory.

The accesses into A  
might not be known until  
runtime.

```
void foo(int *restrict A,  
         const int *restrict B,  
         int n) {  
    for (int i = 0; i < n; ++i)  
        A[B[i]]++;  
}
```

# Vector Scatter and Gather

Sometimes, compilers can use **gather** and **scatter** instructions to vectorize indirect accesses.



Using these instructions depends on the available hardware, the code, and the costs of different operations.

# Optimizing Local Variables

A compiler can optimize indirect memory accesses when it can **statically analyze** the indirection.

```
void foo(int *A) {  
    int B[4] = {2, 0, 1, 3};  
    for (int i = 0; i < 4; ++i)  
        A[B[i]]++;  
}
```

The small loop is **fully unrolled**, enabling further optimizations.

```
define void @foo(i32* nocapture noundef %0) {  
    %2 = bitcast i32* %0 to <4 x i32>*  
    %3 = load <4 x i32>, <4 x i32>* %2, align 4  
    %4 = add nsw <4 x i32> %3, <i32 1, i32 1, i32 1, i32 1>  
    store <4 x i32> %4, <4 x i32>* %2, align 4  
    ret void  
}
```

LLVM IR

# Optimizing Local `malloc`'d Variables

Compilers can perform such optimizations even when local variables are allocated using `malloc`.

```
void foo(int *A) {  
    int *B = (int *)malloc(sizeof(int) * 4);  
    B[0] = 2; B[1] = 0; B[2] = 1; B[3] = 3;  
    for (int i = 0; i < 4; ++i)  
        A[B[i]]++;  
    free(B);  
}
```

C

The compiler treats  
`malloc` specially for  
alias analysis.

```
define void @foo(i32* nocapture noundef %0) {  
    %2 = bitcast i32* %0 to <4 x i32>*  
    %3 = load <4 x i32>, <4 x i32>* %2, align 4  
    %4 = add nsw <4 x i32> %3, <i32 1, i32 1, i32 1, i32 1>  
    store <4 x i32> %4, <4 x i32>* %2, align 4  
    ret void  
}
```

LLVM IR

# Optimization Failure With Locals

But compilers can still **fail** to optimize indirect accesses even when they have perfect information.

```
void foo(int *A, int n) {  
    int *B = (int *)malloc(sizeof(int) * n);  
    for (int i = 0; i < n; ++i)  
        B[i] = i;  
    for (int i = 0; i < n; ++i)  
        A[B[i]]++;  
    free(B);  
}  
...  
%32 = load i32, i32* %31, align 4  
%33 = sext i32 %32 to i64  
%34 = getelementptr inbounds i32, i32* %0, i64 %33  
%35 = load i32, i32* %34, align 4  
%36 = add nsw i32 %35, 1  
store i32 %36, i32* %34, align 4  
...
```

The compiler **fails** to recognize that **B[i]** is the identity function.

LLVM IR

# Take-Aways

The compiler may struggle to optimize **indirect memory accesses**.

- Such indirect memory accesses may be **input dependent**, which substantially limits what the compiler can do.
- Existing compiler **limitations** may prevent optimization of indirect accesses.
- Compilers are **better** able to optimize **local** variables than nonlocal variables.
- **Check** the LLVM IR or assembly to verify whether the compiler did the optimizations you want.

# SUMMARY



# Summary: What the Compiler Does

Compilers transform code through a sequence of **transformation passes**.

- The compiler knows a set of transformations, many of which resemble **Bentley-rule** work optimizations.
- The compiler repeatedly goes over the code and mechanically applies these transformations.
- One transformation can **enable** other transformations.

Compilers perform many more transformations than those shown in this lecture.

# Summary: What Stops The Compiler?

Compiler generated code must be **correct** in **all cases**, including:

- For any given input, and
- For any variation of the architectures.

Compilers are **conservative**, often use **heuristics**, and must assume the worst of:

- Memory aliasing
- Indirect accesses
- Cost model

Use **compiler flags**, **annotations** and simple code **rewrites** to trigger the optimizations.

# Want to Improve the Compiler?

You can also try [modifying](#) the compiler itself to overcome some of its current limitations!

[SHAMELESS PLUG:](#) If you're interested in UROP or MEng projects related to compilers, come talk to me!