Compilers 101

Optimizations – Part 2

Previously…

- Preprocessing
- Lexical analysis
- Syntax analysis
- Semantic analysis
- IR Generation
- IR Optimization
- Optimization
- Code generation

- Fronten
d
- Middle-end
- Backend

Previously… (2)

- LLVM IR
- LLVM IR passes
- LLVM IR optimizations

Today

- opt tool showcase
- Loop optimizations

Running optimizations

- Generate LLVM IR
- Use opt tool
  - Guide: https://llvm.org/docs/CommandGuide/opt.html

opt tool

- LLVM optimizer
  - Input: LLVM IR
  - Output: LLVM IR
- Passes list: https://llvm.org/docs/Passes.html
- Options: https://llvm.org/docs/CommandGuide/opt.html
- Command line examples:

```
opt -print-passes
opt -O2 input.ll -o optimized.bc
opt -passes=’dce’ input.bc
opt -dce input.bc
opt -load MyPass.so -my-custom-pass input.ll -o
output.ll
```

Compiler Explorer

Is an interactive compiler exploration website. Edit code in C, C++, C#, F#, Rust, Go, D, Haskell, Swift, Pascal, ispc, Python, Java, or any of the other 30+ supported languages components, and see how that code looks after being compiled in real time.

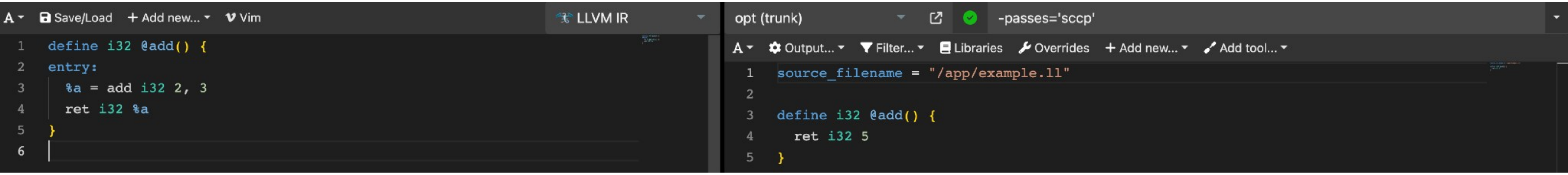
Official website: https://godbolt.org/

GitHub repository: https://github.com/compiler-explorer/compiler-explorer

Running LLVM IR optimizations

Learn by example (1):

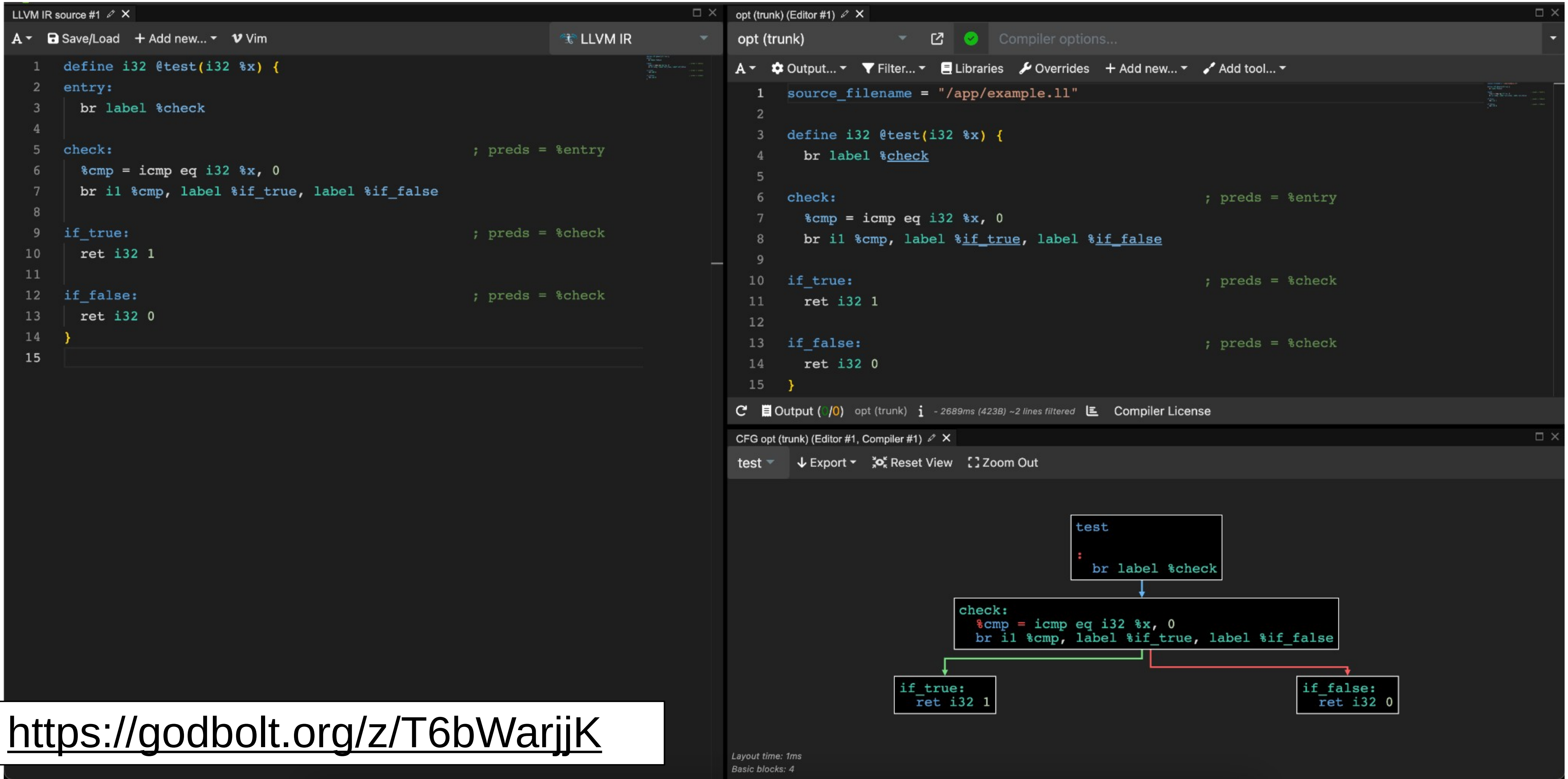
- SCCP:



https://godbolt.org/z/vYzMohM3e

CFG visualization

```
opt -dot-cfg cfg_example.ll
dot -Tpng cfg_example.dot
```



https://godbolt.org/z/T6bWarjjK

Running LLVM IR optimizations

Learn by example (2):

- CFG simplification:

LLVM IR source #1 — LLVM IR

```
define i32 @test(i32 %x) {
entry:
  br label %check

check:                                            ; preds = %entry
  %cmp = icmp eq i32 %x, 0
  br i1 %cmp, label %if_true, label %if_false

if_true:                                          ; preds = %check
  ret i32 1

if_false:                                         ; preds = %check
  ret i32 0
```

opt (trunk) (Editor #1) — opt (trunk) -passes='simplifycfg'

```
source_filename = "/app/example.ll"

define i32 @test(i32 %x) {
  %cmp = icmp eq i32 %x, 0
  %common.ret.op = select i1 %cmp, i32 1, i32 0
  ret i32 %common.ret.op
}
```

https://godbolt.org/z/hK4ndPbcq

Loop optimizations

Why are they important?

- Pareto principle
  - 20 percent of effort gives 80 percent of result
- Most computationally intensive algorithms involve loops

Terminology

- Trip count is a minimum number of times a loop executes
- Induction variable is a variable that gets increased or decreased on each loop iteration
- Loop invariant is the part of a computation inside a loop that remains unchanged (invariant) in each iteration

Detecting induction variables

- i is a basic induction variable in a loop L if the only definitions of i within L are of the form i := i +/- c where c is loop invariant
- k is a derived induction variable in loop L if
  - There's only 1 definition of k within L in the form of k := j */+ c where j is an induction variable
  - If j is an induction variable in the family of i then
    - The only definition of j that reaches k is the one in the loop
    - There's no definition of i on any path between the definition of j and the definition of k

Structured control flow

Structured control flow is a programming paradigm characterized by the clear, hierarchical organization of control paths through a program.

- It relies exclusively on structured control constructs such as conditionals (if, else), loops (for, while), and function calls, which lead to well-defined entry and exit points.
- The key attribute of structured CF is that it avoids the use of arbitrary jumps or goto statements, making the control flow predictable and easier to follow.

Structured control flow

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a

https://dl.acm.org/doi/pdf/10.1145/362929.362947

Structured control flow

Is this program structured?

```c
void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        if (a[i] == 42)
          break;
    }
}
```

Loop-invariant code motion

```
void baz();
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        bar(i);
        baz();
        a[0] = 42;
    }
}
```

```
void baz();
void bar(int i);

void foo(char *a, int N) {
    a[0] = 42;
    for (int i = 0; i < N; i++) {
        bar(i);
        baz();
    }
}
```

Conditions for safe hoisting

- hoisting - moving to outer scope
- sinking - moving to inner scope

- An invariant assignment d: x := y op z is safe to hoist if
  - d dominates all loop exits at which x is live and
  - there is only one definition of x in the loop, and
  - x is not live at the entry point for the loop

Loop unroll

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        bar(i);
    }
}
```

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N / 4; i += 4) {
        bar(i + 0);
        bar(i + 1);
        bar(i + 2);
        bar(i + 3);
    }
    for (int i = N - N % 2; i < N; i++) {
        bar(i);
    }
}
```

When it works and when it fails?

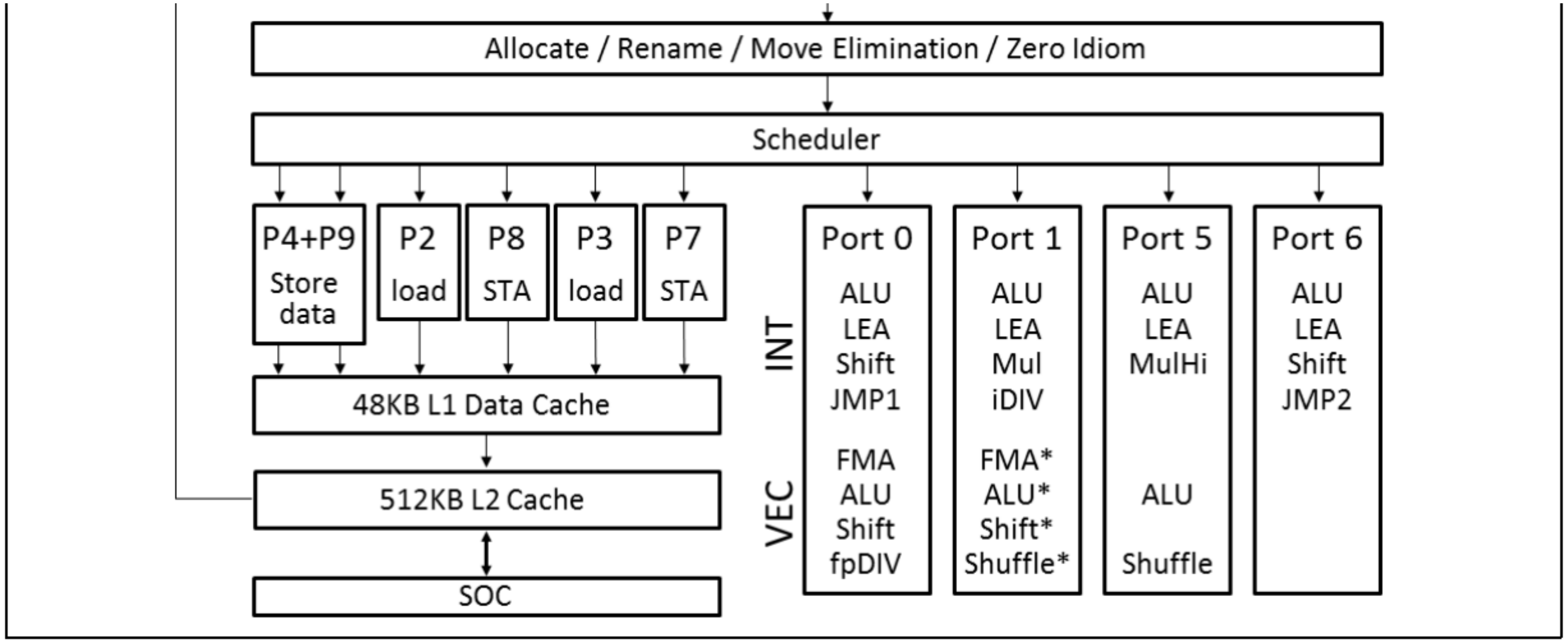


Figure 2-1. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Loop fusion

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = 42;
    }
    for (int i = N; i >= 0; i--) {
        bar(i);
    }
}
```

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = 42;
        bar(N - i);
    }
}
```

Loop peeling

```c
void foo(int *a, int *b, int N) {
    int p = 10;
    for (int i = 0; i < N; i++) {
        a[i] = b[i] + b[p];
        p = i;
    }
}
```

```c
void foo(int *a, int *b, int N) {
    a[0] = b[0] + b[10];
    for (int i = 1; i < N; i++) {
        a[i] = b[i] + b[i - 1];
    }
}
```

Loop unswitch

```
void foo(int *a, int *b, bool c, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = i;
        if (c) {
            b[i] = 42;
        }
    }
}
```

```
void foo(int *a, int *b, bool c, int N) {
    if (c) {
        for (int i = 0; i < N; i++) {
            a[i] = i;
            b[i] = 42;
        }
    } else {
        for (int i = 0; i < N; i++) {
            a[i] = i;
        }
    }
}
```

Control flow sink

```c
void bazz();
void bazz(int i);
void bar(int i);

void foo(char *a, int N) {
    a[0] = 42;
    for (int i = 0; i < N; i++) {
        bar(i);
        bazz();
        bazz(a[0]);
    }
}
```

```c
void bazz();
void bazz(int i);
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        bar(i);
        bazz();
        a[0] = 42;
        bazz(a[0]);
    }
}
```

Loop reroll

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N / 4; i += 4) {
        bar(i + 0);
        bar(i + 1);
        bar(i + 2);
        bar(i + 3);
    }
    for (int i = N - N % 2; i < N; i++) {
        bar(i);
    }
}
```

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        bar(i);
    }
}
```

Loop fission

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = 42;
        bar(N - i);
    }
}
```

```c
void bar(int i);

void foo(char *a, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = 42;
    }
    for (int i = N; i >= 0; i--) {
        bar(i);
    }
}
```

Loop tiling

```
void foo(int N) {
    for (int i = 0; i < N; i++) {
        // ...
    }
}
```

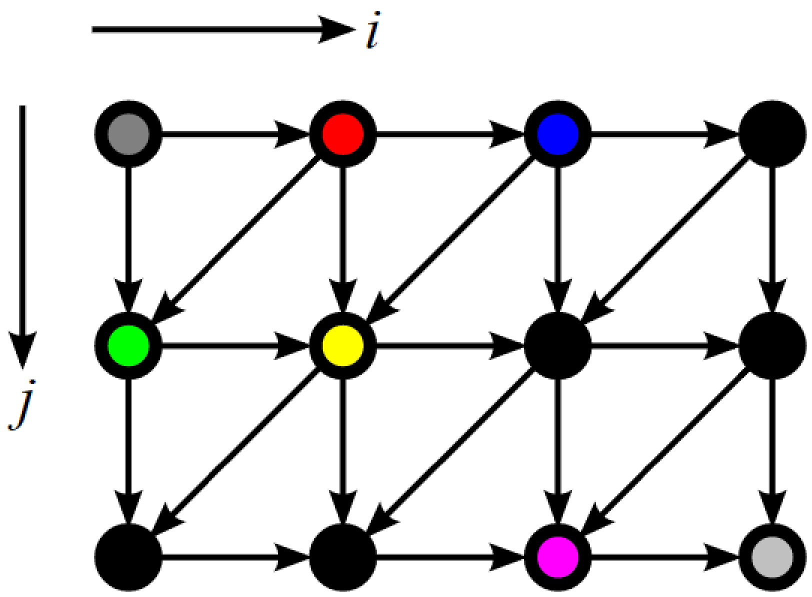
```
#include <algorithm>

void foo(int N) {
    int B = 10;
    for (int i = 0; i < N; i += B) {
        for (int j = i; i < std::min(N, j + B); j++) {
            // ...
        }
    }
```

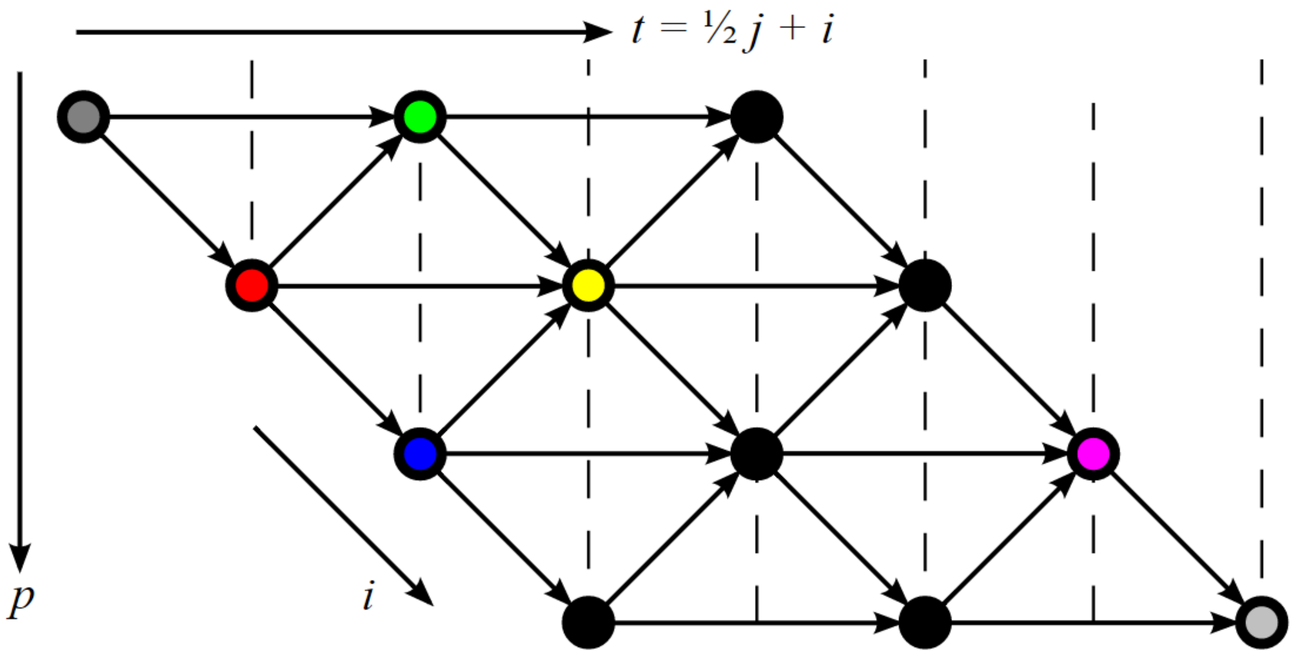
Loop rotation

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

```
void foo(int *a, int *b, int n) {
  int i = 0;
  if (n > 0) {
    do {
      a[i] = a[i] + b[i];
      i++;
    } while (i < n);
  }
}
```

Polytope model





https://en.wikipedia.org/wiki/Polytope_model

Affine transformations

```c
#define ERR(x, y) (dst[x][y] - src[x][y])

void dither(unsigned char** src, unsigned char** dst, int w, int h)
{
    int i, j;
    for (j = 0; j < h; ++j) {
        for (i = 0; i < w; ++i) {
            int v = src[i][j];
            if (i > 0)
                v -= ERR(i - 1, j) / 2;
            if (j > 0) {
                v -= ERR(i, j - 1) / 4;
                if (i < w - 1)
                    v -= ERR(i + 1, j - 1) / 4;
            }
            dst[i][j] = (v < 128) ? 0 : 255;
            src[i][j] = (v < 0) ? 0 : (v < 255) ? v : 255;
        }
    }
}
```

```c
void dither_skewed(unsigned char **src, unsigned char **dst, int w, int h)
{
    int t, p;
    for (t = 0; t < (w + (2 * h)); ++t) {
        int pmin = max(t % 2, t - (2 * h) + 2);
        int pmax = min(t, w - 1);
        for (p = pmin; p <= pmax; p += 2) {
            int i = p;
            int j = (t - p) / 2;
            int v = src[i][j];
            if (i > 0)
              v -= ERR(i - 1, j) / 2;
            if (j > 0)
              v -= ERR(i, j - 1) / 4;
            if (j > 0 && i < w - 1)
              v -= ERR(i + 1, j - 1) / 4;
            dst[i][j] = (v < 128) ? 0 : 255;
            src[i][j] = (v < 0) ? 0 : (v < 255) ? v : 255;
        }
    }
}
```

Polyhedral transformations in LLVM

Polly is a high-performance loop and data-locality optimizer embedded framework within the LLVM. It utilizes mathematical models to analyze and optimize memory access patterns, significantly enhancing code efficiency, particularly in loops and matrix multiplications

Docs & info: https://polly.llvm.org/

Next time…

- Vectorization

Lab assignment #2

- Write a plugin with simple LLVM pass
- Task list in Google Docs
- Deadline: April, 9
- Where to seek help
  - https://www.llvm.org/docs/WritingAnLLVMNewPMPass.html
  - https://github.com/llvm/llvm-project/tree/main/llvm/examples/IRTransforms
  - contact teachers

Test

https://forms.gle/ywfMNSijV2XqDiVF8

Submission time: **10 minutes**

Почему loop unroll оптимизация применима и эффективна не всегда?

Your answer

Приведите примеры оптимизационных пассов для циклов. Объясните, зачем каждая может быть нужна.
Note: точное название пасса и флага не требуется

Your answer

Backup: me@gooddoog.ru

Extra materials

- Loop Optimizations in LLVM: The Good, The Bad, and The Ugly - https://llvm.org/devmtg/2018-10/slides/Kruse-LoopTransforms.pdf
- EuroLLVM Developers' Meeting: J. Doerfert & T. Grosser "Analyzing and Optimizing ... Polly“ - https://www.youtube.com/watch?v=mXve_W4XU2g
- LLVM Developers’ Meeting: T. Grosser “Polly - Polyhedral optimizations in LLVM” - https://www.youtube.com/watch?v=WwfZkQEuwEE