Hoisting Julia Array Bounds Checks out of Loop Nests using Polly

Jan Soendermann[1], *Tobias Grosser*[2], *David Chisnall*[1]

[1]*University of Cambridge,* `jjes2@cam.ac.uk, David.Chisnall@cl.cam.ac.uk`; [2]*ETH Zürich,* `tobias@grosser.es`;

UNIVERSITY OF CAMBRIDGE

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

- Julia is a high performance computing language aiming to replace R, Matlab and similar languages and frameworks
- Much of the scientific programs written in Julia would profit greatly from the vectorisation and parallelisation optimisations implemented in polly
- Julia adds bounds checks to all array accesses. When they occur in loop nests, these bounds checks currently make polly optimisations impossible
- We show changes to polly that make it possible to statically derive assumptions that ensure that the bounds checks always succeed
- This makes it possible to hoist the bounds checks outside of the loop nest and execute a fully optimised version of the loop nest when the assumptions hold

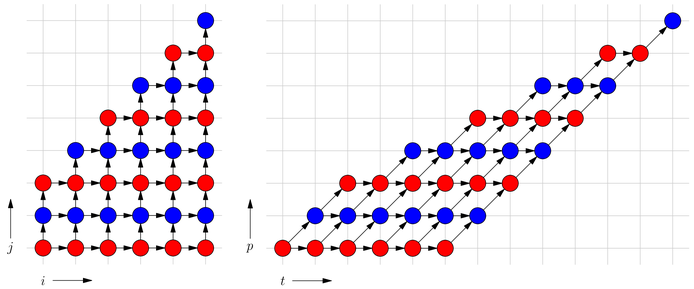

Figure 1: *Example of a polyhedral optimisation of two nested loops. Every node represents one iteration of the loop body, arrows represent dependencies. After transforming the original (left) to the optimised version (right), the inner loop can be parallelised because inner loop operations in one iteration of the outer loop do not depend on each other anymore.*

Approach: We use Polly to hoist array bounds checks outside of the loops in which they occur.

Method

Listing 1 shows LLVM IR code of the block that is executed when out of bounds array accesses happen. The `unreachable` instruction allows us to derive the assumptions described above by inverting the sets of values for parameters coming into the loop nest from its context for which this instruction gets executed.

```
oob:
%e = load %jl_value_t** @jl_bounds_exception
call void @jl_throw_with_superfluous_argument(
                    %jl_value_t* %e, i32 5)
unreachable
```

Listing 1: Out of bounds error handling in LLVM IR

Results

As our changes to Polly have not yet to be integrated into Julia, we tested them by implementing a gemm kernel in C, imitating the code that Julia generates. This code is shown in Listing 2.

```
void oob() __attribute__((noreturn));

void gemm(long n, long m, long o, float A[n][o],
                  float B[m][o], float C[n][m], long n2,
                  long m2, long o2) {
    for (long i = 0; i < n2; i++) {
        for (long j = 0; j < m2; j++) {
            for (long k = 0; k < o2; k++) {
                if (i < 0)
                    oob();
                if (i >= n)
                    oob();
                if (j < 0)
                    oob();
                if (j >= m)
                    oob();
                if (k < 0)
                    oob();
                if (k >= o)
                    oob();

                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void oob() {
    printf("Error\n");
    exit(-1);

    __builtin_unreachable();
}
```

Listing 2: C reconstruction of Julia output

The result of running our modified version of Polly on the code above are shown in Listing 3. The three `if` statements that check for `< 0` have been statically removed. The other three `if` statements have been moved outside of the loop nest.

```
if (n >= n2 && m >= m2 && o >= o2 ? 1 : 0)
    for (int c0 = 0; c0 < n2; c0 += 1)
      for (int c1 = 0; c1 < m2; c1 += 1)
        for (int c2 = 0; c2 < o2; c2 += 1)
          Stmt_if_end22(c0, c1, c2); // Matrix update
else {  /* original code */ }
```

Listing 3: Polly moves bounds checks outside the loop nest

To run a realistic benchmark, we implemented a gemm kernel in Julia. The results of running this code in unoptimised and optimised versions are shown in Figure 2

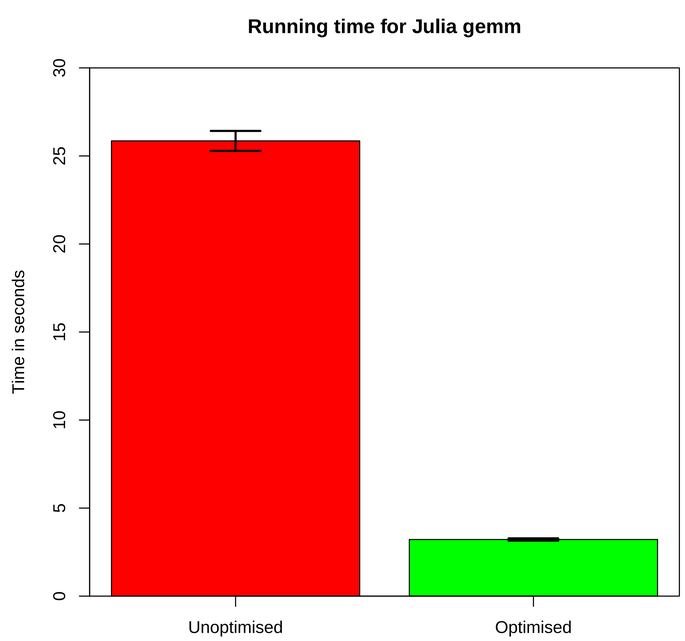

Figure 2: *Timing execution of a gemm kernel implemented in Julia. The left version is without any optimisations, the left version uses the* `@inbounds` *macro to completely remove bounds checks and run polyhedral optimisations, similar to what our modified version of Polly will achieve, once it is adapted into Julia.*

Conclusions

- Big and significant speed ups of Julia programs can be achieved with our changes. Especially for linear algebra code that is common in scientific computing, we expect substantial improvements in performance.
- Further work on the Julia implementation is necessary to integrate our changes. One possible way of doing so is to add a `@polly` macro that leaves the choice of whether to use Polly's optimisations to the user.