Using Polly to Optimise Julia Arrays in Loops

Jan Soendermann

***Abstract*—This report describes changes to the Polly loop optimisation framework that make it possible to optimise loop nests in the presence of bounds checks as generated by the programming language Julia for all array accesses. I show substantial and statistically significant speed ups when these optimisations are applied to a gemm kernel.**

***Index Terms*—Polly, Julia, LLVM, Runtime bounds checks, Polyhedral optimisation**

I. Introduction

In this report, I describe modifications to the polyhedral optimisation framework Polly that make it possible to use Polly's optimisation passes to optimise code written in the Julia programming language. I begin with an introduction to Julia, polyhedral optimisation, Polly and other necessary background programs and concepts. I then describe the implementation details of the changes I made to Polly. Finally, I evaluate the performance impact of my modifications and describe possible future work related to my project.

A. Julia

Julia is a programming language designed for scientific computing. It is a dynamically typed language that JIT compiles code before it is executed using LLVM. Julia tries to compete with programming environments and languages that are currently used in scientific research such as MATLAB, R, Stata and Python with packages like NumPy [1]. The developers of Julia claim that it can deliver performance equivalent to that of C while being as expressive as Python[1]. Since its initial release in 2012, it has generated considerable interest in the scientific community and continues to be actively developed.

One noteworthy feature of Julia is that it prefaces all array accesses by bounds checks that make sure no memory addresses are illegally read or written to and thus prevent crashes and memory corruption. A simple array access such as `A[n]` generates code that first compares `n` to the length of the array `A`, and only proceeds if `n` is within bounds. Otherwise it throws an exception. This is, of course, also true for loops that iterate over arrays. These bounds checks are then executed at every iteration.

One other aspect of the Julia language that is relevant for my project are its macros. Specifically, one macro was made use of during the implementation and evaluation of my changes: `@inbounds`. This macro disables bounds check generation for the statements it is applied to. One possible use of `@inbounds` is shown in Listing 1[2].

```
@inbounds for i=1:n, j=1:m
   A[i,j] = i+j
```

Listing 1. Using @inbounds with nested loops

The code shown in Listing 1 is compiled to LLVM IR that is quite close to what two equivalent nested loops in C would be compiled to and includes no bounds checks. It also demonstrates how two nested for loops can be concisely expressed in one line in Julia.

B. Regions and Dominance Trees

One of the fundamental concepts of the LLVM compiler is that of a basic block which is a list of LLVM IR instructions that execute sequentially and end in one of several possible terminator instructions. This means that there is no way to jump into the middle of a basic block—every execution starts at the very first instruction. Equivalently, the only point at which a basic block can be left is its terminator instruction. A function is then a collection of such basic blocks connected by conditional or unconditional branch instructions. This collection forms a graph $G = (V, E)$, commonly called the control flow graph (CFG), which has basic blocks as vertices V and edges E as possible jumps. As an illustration, consider the simple C function shown in Listing 2.

```
void initialise(int *A, int length) {
   for (int i = 0; i < length; i++) {
      if (i > length / 2) {
         A[i] = i;
      } else {
         A[i] = -i;
      }
   }
   return;
}
```

Listing 2. A simple C function

Figure 1 shows the control flow graph that this function forms.

An execution of `initialise` is equivalent to a particular walk of this control flow graph, starting at the first basic block and ending at the basic block that has a `return` statement as its terminator statement.

One concepts that follows naturally from the interpretation of the basic blocks as a graph is that of a (simple) region. A region is defined as a connected subgraph of the CFG that is connected to the remaining part of the graph by only two edges: an incoming and an outgoing one [2]. Figure 1 also shows a region in our control flow graph highlighted in grey. This region happens to form the body of the for loop in our program.

It turns out that in practice, a slightly relaxed definition of regions is more useful, as proper regions are quite rare in real

Jan Soendermann is an MPhil student in the Computer Laboratory at the University of Cambridge

[1] See http://docs.julialang.org/en/release-0.3/manual/introduction/ for a source to this claim

[2] Note that array indices to Julia arrays start at 1

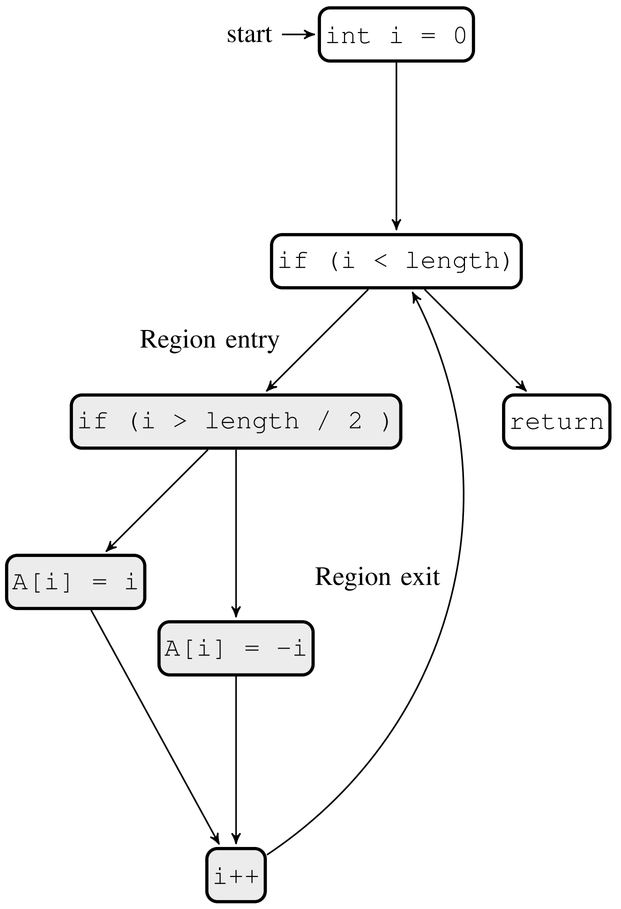

Fig. 1. The CFG with a region marked in grey

code. These “refined regions” [2] are defined as connected subgraphs $G\prime = (V\prime, E\prime)$ of the CFG $G = (V, E)$ with two special vertices, v_{in} and v_{out}, such that $\forall (v, w) \in E, v \in V \setminus V\prime \wedge w \in V\prime \implies w = v_{in}$ and $\forall (v, w) \in E, v \in V\prime \wedge w \in V \setminus V\prime \implies v = v_{out}$. Informally, this means that all incoming edges from the rest of the CFG go to v_{in} and all outgoing edges to the rest of the CFG go from v_{out}. Note that a refined region can be converted to a simple region by inserting two empty blocks into the CFG that bundle the incoming and outgoing edges.

To find the regions in a given CFG, LLVM makes use of the related concept of dominator and post-dominator trees. A dominator tree is a tree in which the basic blocks of the function form the nodes. They are arranged according to the following rule: for every block b_c with a parent block in the tree b_p, every path from the entry node through the control flow graph that passes through b_c first passes through b_p.

The dominator tree of our example program is shown in Figure 2. Note how even though in the CFG, both the `A[i] = i` and the `A[i] = -i` block have outgoing edges to the `i++` block, neither of them dominates the increment block. Instead, it is a direct child of its grandparent in the graph, the second if statement. This is precisely because when reaching the increment, we could have arrived from either of the two array assignments, which means that neither dominates their common child in the graph.

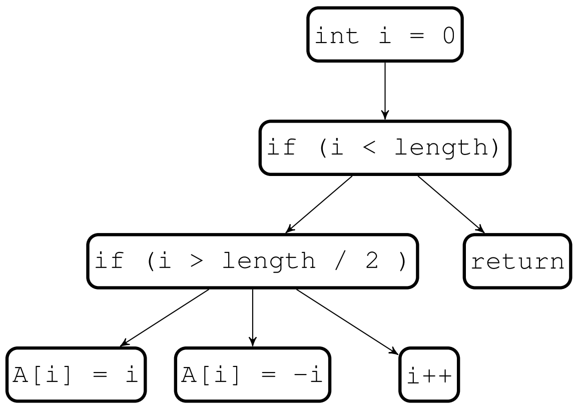

Fig. 2. The dominator tree of our example function

To construct the post-dominator tree of our program, we first need to compute the inverse control flow graph. As the name suggest, the inverse CFG is constructed from the CFG by inverting all the edges in the graph. Additionally, input blocks are turned into exit blocks and vice versa. In the case of multiple exit nodes in the CFG, a new root note is introduced to the post-dominator tree that dominates all the new entry nodes that were exit nodes before. The post-dominator tree is then constructed from the inverted CFG in the same fashion the dominator tree is constructed from the CFG. Although superficially simple concepts, I will return to the details of dominator and post-dominator trees when I describe the implementation details of my project.

The algorithm that builds dominator and post-dominator trees is described in [3]. Both together are used to compute the refined regions of the function. The algorithm that is used for this is not documented publicly.

C. Polly

Polly is a loop optimisation framework for LLVM. It represents loops nests as polyhedra and performs affine transformations on these polyhedra to optimise the code that the polyhedron represents. By doing so, it exposes opportunities to parallelise and vectorise the program and improve data-locality.

I will first describe the theoretical background of Polly before explaining the specifics of Polly's architecture.

1) Polyhedral Representation: To illustrate polyhedral representation and manupulation, we will consider the loop nest shown in Listing 3[3]. This example is taken from [4] and http://llvm.org/devmtg/2010-11/Grosser-Polly.pdf, which also contains more details on polyhedral loop optimisation.

```
for i = 0 to n:
   for j = 0 to i + 2:
      A[i, j] = A[i-1, j] + A[i, j-1]
```

Listing 3. A nested loop example

This is a doubly nested loop and as such will be turned into a 2-dimensional polyhedron. More generally, d nested loops form a d-dimensional polyhedron. Every vertex of the

[3] I left out bounds checks to make the example more readable

polyhedron represents one iteration of the loop and every edge represents a dependence relation. This is illustrated by Figure 3 which shows the polyhedral representation of the nested loops shown in Listing 3 (the colours are only used to make the transformation more clear and do not carry any meaning).

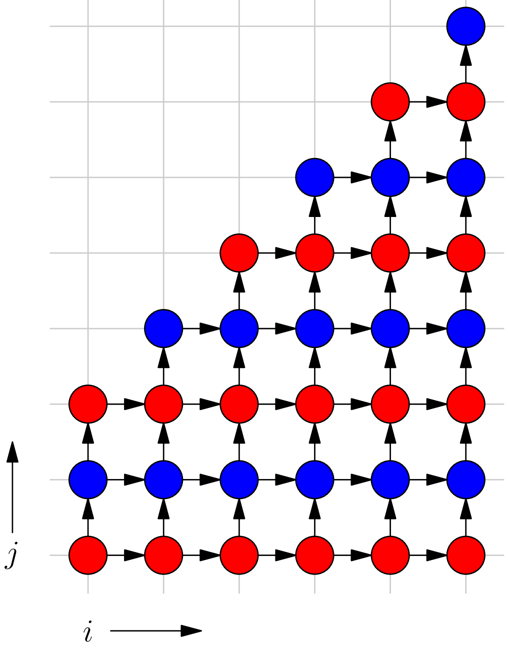

Fig. 3. The polyhedral representation of our nested loops

An incoming arrow to a vertex is to be interpreted as a dependence on the loop iteration from which the arrow originates. This means that the iteration at which the arrow originates has to be executed first. The iteration with $i = 1, j = 1$ is dependent on the two iterations $i = 0, j = 1$ and $i = 1, j = 0$, which can easily be verified by looking at the code in Listing 3.

If we want to use this polyhedral representation to parallelise our loops, we can apply an affine transformation to turn our polyhedron into the form shown in Figure 4. The reader may be familiar with affine transformations from the field of 3D graphics where they are also used to shift, scale, rotate and otherwise manipulate objects in space. In our particular example, a shear transformation is used that shifts all points along the x-axis.

The last step that remains is to translate the modified polyhedron back to code. The result of this translation is shown in Listing 4.

```
for t = 0 to 2*n + 2:
   parallel_for p = max(0, t - n) to
                    min(t, t / 2 + 1):
      A[t-p, p] = A[t-p-1, p] + A[t-p, p-1]
```

Listing 4. A nested loop example

Looking at Figure 4, we see that iterations with $t = v$ exclusively depend on iterations with $t < v$. Therefore, all the iterations with an equal t can happen in parallel, as they do not depend on each other. The index variable t can be though of as the time and p as the different parallel iterations of the loop body.

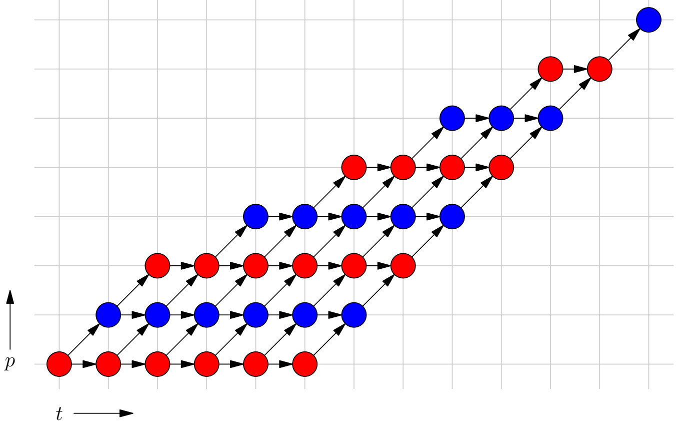

Fig. 4. The transformed polyhedral representation of our nested loops

2) Architecture: Executing Polly on a function involves the following three steps. First, the parts of the function that can be optimised by Polly are detected. They are called Static Control Parts (SCoPs) and are explained in detail below. Second, these SCoPs are translated into a polyhedral representation and dependency analysis is performed on these polyhedra. Finally, the transformed polyhedra are translated back to LLVM IR and optimisation opportunities exposed by the transformations to parallelise or vectorise the code using OpenMP and SIMD instructions are taken. These three steps make up the front, middle and back end of Polly.

SCoPs, that make up an essential part of the Polly architecture, are pieces of our program in which we can statically, i.e. at compile time, determine all control flow and memory accesses [2]. They are regions in the control flow graph that fulfil a number of requirements that go beyond those of a simple or refined region to ensure this static control property.

These additional requirements are as follows[4]:

1) All loop bounds are affine linear functions
2) Only affine linear expressions can be compared in conditions
3) Flow control statements are perfectly nested, i.e. there are no `break`, `continue` or `goto` statements in the code
4) All function calls are side effect free

The basic blocks in a region that forms a SCoP are called SCoP statements. Every such statement has a domain associated with it that describes the iterations, i.e. the configurations of iteration variables for which the statement is executed.

The last important concept related to static control parts is that of assumptions. As the name suggests, these are conditions that will be assumed to hold during the execution of the SCoP. Sometimes, these assumptions will be statically provable. As an example of this, consider the Julia code in Listing 5. Because we know statically that `i` will run from 1 to the size of array A, the bounds check that Julia will generate for the array access can be removed completely during compilation when Polly is run.

[4] See https://llvm.org/svn/llvm-project/polly/trunk/include/polly/ScopDetection.h for a source

```
function init1(A)
   n = size(A)
   for i=1:n
      A[i] = i
   end
end
```

Listing 5. There will never be an out of bounds error in this code

Listing 6 shows a slight modification of Listing 5 for which this is no longer true. Here, the maximum value of the loop index is taken from a parameter to this function. This means that we can not determine at compile time whether `n` will be greater than `size(A)`.

```
function init2(A, n)
   for i=1:n
      A[i] = i
   end
end
```

Listing 6. There might be an out of bounds error when calling this code

In cases like these, Polly generates runtime checks that take the following form:

```
if (assumptions hold)
   execute_optimised_version();
else
   execute_original_code();
```

Listing 7. Check assumptions at runtime

I will show an example of these checks in the Evaluation chapter.

II. Implementation

As described in the introduction, the goal of my project was to move Julia's array bounds checks out of loops. To understand what was necessary to achieve this, it is helpful to first look at the LLVM IR of the basic blocks that are executed when these bounds checks fail. Listing 9 shows this code.

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

Listing 9. Out of bounds error handling in LLVM IR

The second line of this code throws a Julia exception by calling the `jl_throw_with_superfluous_argument` function with the type of the exception and the line number in the Julia code where it occurred. Clearly, the program is not going to return from this call: either the exception is caught somewhere higher up the call stack and execution proceeds there, or the program crashes and the error is displayed to the user. To make this explicit to the optimiser, the `oob` basic block ends in an `unreachable` instruction, an LLVM IR instruction that is used for this specific circumstance: a function that does not return.

Because SCoPs are detected based on regions, the first necessary change was to make the LLVM pass that builds regions from the control flow graph, RegionInfo, work on CFGs that contain `unreachable` instructions. It is not immediately clear how to interpret `unreachables` in the context of CFGs. They could be interpreted as terminator statements but for the region building algorithm to function properly, it requires them to be considered dead ends. They are not exit nodes in the CFG and therefore do not turn into entry nodes in the inverted CFG.

Figure 5 shows the control flow graph and the regions detected in that graph for a function that contains an `unreachable` statement at the end of basic block `_11`[5]. Notice how this block forms a region with the block that branches to it, the one coloured in red.

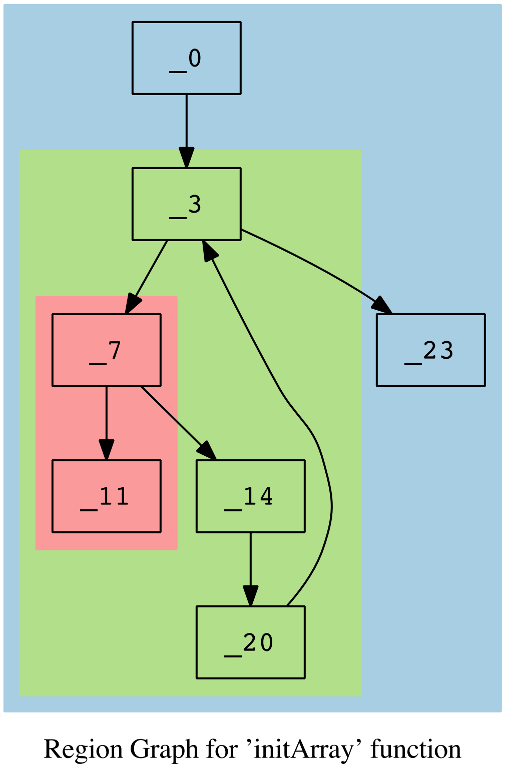

Fig. 5. The regions in a function with an `unreachable` statement

The next modification I made was to the SCoP detection part of Polly. As part of SCoP detection, the function `ScopDetection::allBlocksValid` is called. This functions iterates over the blocks in a potential region and over the instructions in these blocks to check for the existence of instructions that violate the conditions of a SCoP as described in the introduction. We skip these checks, because evetually, these blocks will be moved outside of the loop or removed altogether. Listing 10 shows how they are skipped.

```
for (const BasicBlock *BB : R.blocks()) {
   const TerminatorInst *TI =
                BB->getTerminator();
   if (dyn_cast<UnreachableInst>(TI))
      continue;

   ... check for SCoP conditions ...
}
```

Listing 10. Skipping checks for blocks with `unreachable`

[5] This diagram was generated by running `$ opt -view-regions-only <file.ll>`

```
void ScopStmt::deriveAssumptionsFromUnreachable() {
    auto ExecutionDomain = this->getDomain();
    auto ParameterValuesForWhichStmtIsExecuted = isl_set_params(ExecutionDomain);
    auto ParameterValuesForWhichStmtIsNotExecuted =
                isl_set_complement(ParameterValuesForWhichStmtIsExecuted);
    this->getParent()->addAssumption(ParameterValuesForWhichStmtIsNotExecuted);
}
```

Listing 8. Deriving assumptions from `unreachable` statements

The last main modification that was necessary was to update assumption derivation to take `unreachable` basic blocks into account. To achieve this, the function `ScopStmt::deriveAssumptions` was extended by a call to the function shown in Listing 8 for ScopStmts that represent basic blocks that end in `unreachable`. This function looks at the parameters of the SCoP, which are variables that come from the context that the loop nest is in and do not change throughout the execution of the SCoP. Specifically, it considers those potential values of these parameters for which the SCoP statement is not executed. My new function adds an assumption to its parent SCoP that the parameters will take these values.

This is equivalent to assuming that there will be no out of bounds error. This assumption is then verified before the loop nest at runtime. In some cases, it can even be statically proven that out of bounds error can never occur. In these cases, the checks will be removed completely.

With these changes, there is one problem that remains to be solved before Julia can automatically optimise its loops using Polly entirely automatically. For a program of the form shown in Listing 11, Julia generates bounds checks of the form `0 <= f1(i) * m + f2(i) < n*m`. This, however, violates requirement 2 of a SCoP as described in the introduction. Polly can only handle bounds checks that compare each dimension separately. In our example, they would have to look like this: `0 <= f(i) < n && 0 <= f2(i) < m`.

```
A = zeros(Int32, n, m)

for i := 1:x
  A[f1(i), f2(i)] = i
end
```

Listing 11. Multidimensional arrays in a Julia loop

To solve this problem, Julia will have to be edited to generate checks that Polly can understand. I considered this to be outside of the scope of my project and discuss this further in the Future Work section.

III. Evaluation

I evaluate the results of my project in two stages. First I show that my changes worked and that Polly can now handle bounds checks that end in `unreachable`s. Second, I show the speed ups that these changes will bring once they are incorporated into Polly and Julia is adapted to solve the problem mentioned above.

To test my changes, I wrote a short C program that computes a gemm kernel and performs the operation $C = AB+C$ where A, B and C are matrices. Such computations are common in scientific code and therefore make for a good test case. The main part of this program is shown in Listing 12 and includes array bounds checks in the loop nest that conform to the format Polly expects. Like the `oob` block shown in Listing 9, the `oob()` function in my code ends in an unreachable instruction, generated by the `__builtin_unreachable()` call.

```
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 12. C reconstruction of proposed Julia output

Note how the parameters to `gemm` include the array dimensions twice: once in parameters `n`, `m` and `o` and once in `n2`, `m2` and `o2`. The first set is used to inform the compiler of the dimensions of the multidimensional arrays that contain the matrices. The second set is used as maximum values for our loop iteration variables. This is done to make our test case more interesting because it means that Polly can not simply statically infer that no out of bounds error ever occurs.

Using `opt` to output a high level representation generated by Polly when run on this C code gives the result shown in

Listing 13, again slightly edited for readability. As the loop variables start at 0 and only get increased, Polly can statically remove the checks that ensure that they do not have values below zero (`if (i < 0) oob();` etc.). For the remaining bounds checks that protect against the loop variables running past the end of the arrays, one run time check outside of the loop nest was added. Recall that `n` is one of the array dimensions that the compiler knows, due to the initialisation of the arrays in the parameter list and that `n2` is the maximum value of loop variable `i` that is used to index the arrays. The check therefore ensures that this upper bound is always lower than the actual size of the array.

```
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);
else
    {  /* original code */ }
```

Listing 13. Polly moves bounds checks outside the loop nest

To evaluate the performance gain that can be expected from my changes, I reimplemented a gemm kernel in Julia, adding the `@inbounds` macro to disable runtime checks. To ensure that the optimisations ran correctly, I initialised matrices A and B with random values using a fixed random seed. Matrix C was initialised with zeroes. After running, the program dumped matrix C into a file, producing the same result in all runs. I used the Julia `time` function to measure the time the `gemm` function took to run.

Running this code ten times, five each for the unoptimised and the Polly-optimised version, produced the result shown in Figure 6. The optimisation produce a highly statistically significant ($p < 0.001$) speed up of roughly 8x. For this test, I used Julia commit d137a9f4a0, commit c41acffe22 of LLVM and commit 8fcf499090 of Polly[6].

IV. Conclusion

My changes enable Polly to successfully optimise loop nests in the presence of bounds checks. Although there is still some minor work left to make Julia and Polly work together seamlessly, it can already be seen that this has the potential of bringing substantial speed ups of programs that spend much time in nested loops. As Julia is a scientific programming language and such loop nests are common in scientific code, I expect the performance gains in real world code to be considerable.

A. Future Work

The next step is to modify Julia to generate bounds checks for multidimensional arrays that Polly can understand. Once this has been done, the changes could be merged upstream. On the julia-dev mailing list, Tim Holy has suggested adding a `@polly` macro that could be used to switch on Polly optimisations[7]. If there is a reason to retain the current form of the bounds checks, this macro could also be used to switch on generation of Polly-friendly bounds checks and default to the current form otherwise.

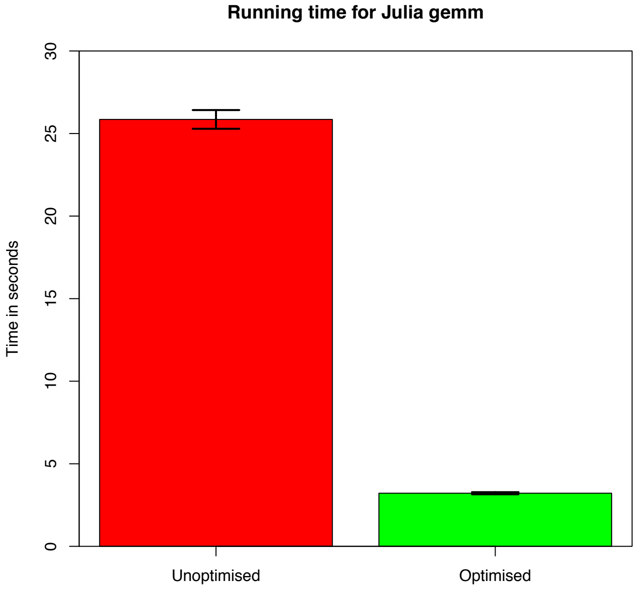

Fig. 6. Running time for the unoptimised and optimised versions

Building on that, new heuristics could be added to Polly specifically targeting common Julia patterns.

V. Acknowledgements

I am very grateful to Tobias Großer for guiding me through this project and for always being patient and helpful when I was stuck and had questions.

I would also like to thank David Chisnall for teaching me about state-of-the-art compiler design and for getting me interested in polyhedral optimisations.

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *CoRR*, vol. abs/1411.1607, 2014. [Online]. Available: http://arxiv.org/abs/1411.1607

[2] T. Grosser, "Enabling polyhedral optimizations in llvm," Ph.D. dissertation, Universität Passau, 2011.

[3] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, Jan. 1979. [Online]. Available: http://doi.acm.org/10.1145/357062.357071

[4] M. Griebl and C. Lengauer, "The loop parallelizer loopo," in *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jülich*. Forschungszentrum, 1996, pp. 311–320.

[6] These all refer to the program's git repositories

[7] https://groups.google.com/d/msg/julia-dev/lNPeqD6uZrQ/qmSB4I9Lgs4J