ABSTRACT
In this position paper, we argue the need for more rigorous specification of kernels in the PolyBench/C benchmark suite. Currently, the benchmarks are mostly specified by their implementation as C code, with a one sentence description of what the code is supposed to do. While this is sufficient in the context of automated loop transformation, the lack of precise specification may have let some questionable behaviors as benchmark kernels remain unnoticed.

As an extreme example, two kernels in PolyBench/C 3.2 exhibit parametric speed up with respect to the problem size when its questionable properties are used. Abusing such properties can provide arbitrary speedup, which can be some factor of millions, potentially threatening the credibility of any experimental evaluation using PolyBench.

1. INTRODUCTION
Optimizing compiler research and empirical evaluation are inseparable in most cases. It is often impossible to directly show how much benefit a transformation can bring in an analytical manner, and researchers rely on empirical evidence. When many groups of researchers work in the same area, it is beneficial for the whole community to share a common set of benchmarks for evaluating their work. Benchmark suites enhance reproducibility of the experiments and results from different papers can be compared with greater consistency.

PolyBench [5] is one of such benchmark suites developed for the polyhedral community. PolyBench is now being used by many members of our community, greatly contributing in making a common ground for empirical validations.

In this paper, we bring the reader’s attention to questionable properties found in a small subset of kernels in PolyBench. In two extreme cases, exploiting these properties enables transformations that can have parametric, in other words, arbitrary, speedup over the original code, even with sequential execution.

Having such questionable properties risks the full experimental validation using PolyBench to be questioned. In this paper, we explain these properties and argue the need for a more rigorous specifications of PolyBench kernels.

2. DYNAMIC PROGRAMMING
In this section, we inspect the dynprog kernel in PolyBench that implements dynamic programming. Figure 1 shows the original kernel.

2.1 Memory Optimization
The first optimization that can be performed on dynprog is memory contraction. The variable that uses the most memory in this kernel is sum_c. It is easy to see with exact dependence analysis [1] that the lifetime of values written to this array is one iteration, in the original execution order.

In fact, this array is really used as an accumulation variable to compute the sum. The obvious optimization is to replace the array by a scalar, saving \(O(n^3)\) memory. Although such a contraction hinders parallelism, a 3D array is clearly an overkill, as a 2D array is enough for most parallel schedules. Basic reuse analysis can easily find better memory allocations than the original single assignment allocation [4, 6, 8].

2.2 Reducing Work
Another optimization is related to the outermost \texttt{iter} loop. Note that the \texttt{c} array is initialized at every iteration of the \texttt{iter} loop at the first statement. The only value-based dependence that is carried by the \texttt{iter} loop is that of the variable \texttt{out}_l written and read only at the last statement.
```c
#pragma scop
for (i=0; i<LENGTH; i++)
    for (j=0; j<LENGTH; j++)
        c[i][j] = 0;
for (i=0; i<=LENGTH - 2; i++) {
    for (j=i+1; j<=LENGTH - 1; j++) {
        sum = 0;
        for (k=i+1; k<=j-1; k++)
            sum += c[i][k] + c[k][j];
        c[i][j] = sum + W[i][j];
    }
} out_l += (TSTEPS)*c[0][LENGTH - 1];
#pragma endscop
```

Figure 2: “Optimized” dynprog kernel. 3D array is reduced to a scalar, and one loop is completely eliminated.

Since the value of c[0][LENGTH-1] accumulated every iteration of the iter loop does not change, it is possible to remove the outermost loop by replacing the last statement with the following:

```c
out_l += (TSTEPS)*c[0][LENGTH - 1];
```

Figure 2 shows the dynprog kernel after the optimizations.

Automating the above transformation requires much more sophisticated analysis compared to memory re-allocation. However, it is definitely possible with a combination of existing techniques for detecting reductions [7, 10] and for transforming programs using the semantics of reductions [2, 3].

In fact, the outermost loop seems to be an artificially added loop to increase the amount of computation, and thus it is probably not intended to be optimized. However, it is part of the kernel function, and it is even within the region marked by the pragmas as scop. Many tools currently do optimize the entire kernel.

### 2.3 Impact on Performance

The three versions of the kernel were executed on a machine with a Xeon X3450, 2.66 GHz quad-core, and 8GB of memory. The programs were compiled with GCC 4.7.2 using O3 optimizations. Reported speedups are the execution times of the original program divided by the execution times of the optimized version.

Figure 3 shows the speedup when only the memory optimization was applied. The trip count of the outermost loop (TSTEPS) was set to one and only the program parameter related to memory (LENGTH) was changed. With LENGTH = 1200, the original program ran in 4.247 seconds, while the optimized version ran in 0.387 seconds.

Figure 4 shows the speedup with both optimizations combined. The parameter LENGTH was set to 1000 and TSTEPS was changed for each data point. The original program took 459.25 seconds where the optimized version only took 0.22 seconds when TSTEPS was set to 1000.

Note that the speedup can theoretically be any number you wish, since it increases as you increase the parameter. The memory optimizations are not exploiting the outermost loop, but can be seen as a more critical optimization since the program cannot even execute without the optimization for large problem sizes.

Even without completely removing the outer loop, the result obtained from using the current dynprog program must be presented with great care. For example, transformations

Figure 3: Speedup only from memory re-allocation. With LENGTH = 1300, the original program cannot allocate memory. The speedup increases in a stepwise manner (first and last points), as the memory footprint of the original program exceeds the capacity of different memory subsystems.

Figure 4: Speedup after both optimizations (Figure 2). The amount of work in the original program increases linearly as TSTEPS increases, where it is independent in the optimized version, and thus resulting in a linear increase in speedup.
# pragma scop
for (t=0; t<NITER; t++) {
    for (j=0; j<MAXGRID; j++)
        for (i=0; i<MAXGRID; i++)
            for (cnt=0; cnt<LENGTH; cnt++)
                S1: diff[j][i][cnt] = sum_tang[j][i];
                for (j=0; j<MAXGRID; j++)
                    for (i=0; i<MAXGRID; i++)
                        S2: sum_diff[j][i][0] = diff[j][i][0];
                        for (cnt=1; cnt<LENGTH; cnt++)
                            S3: sum_diff[j][i][cnt] = sum_diff[j][i][cnt-1] +
                               diff[j][i][cnt];
                            mean[j][i] = sum_diff[j][i][LENGTH-1];
                        }
            for (i=0; i<MAXGRID; i++)
                path[0][i] = mean[0][i];
            for (j=1; j<MAXGRID; j++)
                for (i=0; i<MAXGRID; i++)
                    path[j][i] = path[j-1][i-1] + mean[j][i];
        }
    # pragma endscop

Figure 5: reg_detect kernel from PolyBench/C 3.2

such as tiling would take advantage of data locality that
would not be present in a true dynamic programming in-
stance. Thus, presenting the result using tiled dynprog as
tiled dynamic programming would be problematic.

3. REGULARITY DETECTION

In this section, we will look at the reg_detect kernel from
PolyBench. Figure 5 shows the original kernel.

3.1 Inlining

The first optimization is a simple inlining. In S1, the diff
array is initialized by values of sum_tang. Note that a 3D
array is initialized by a 2D array. However, this array is
never written again until the next iteration of the t loop
that re-initializes the array. Thus, the 3D array is useless,
and can completely be removed by inlining statement S1 to
S2 and S3.

3.2 Memory Optimization

The next optimization is memory contraction, similar to
the one shown in Section 2.1. The array sum_diff is initial-
ized at S2 and then used in S3 as an accumulator variable
for the summation. Since the result is immediately copied
to mean, sum_diff can be reduced to a scalar.

Figure 6 shows the kernel after the two optimizations
described so far.

3.3 Reducing Work

There are multiple places where work can be reduced in
this kernel. One is the outermost loop that repeats the same
computation. With exact value-based dependence analysis,
it can be found that no dependence is carried by the out-
nermost loop. Thus, the outermost loop can completely be
eliminated.

Furthermore, the summation using sum is actually adding
the same value LENGTH times. Thus, computing an element
of mean can be simplified as:

# pragma scop
for (t=0; t<NITER; t++) {
    for (j=0; j<MAXGRID; j++)
        for (i=0; i<MAXGRID; i++)
            sum = sum_tang[j][i];
            for (cnt=1; cnt<LENGTH; cnt++)
                sum += sum_tang[j][i];
            mean[j][i] = sum;
        }
    # pragma endscop

Figure 6: reg_detect kernel after inlining S1 and contract-
ing sum_diff array to a scalar.

3.4 Impact on Performance

The three versions of the kernel were executed with the
same environment as described in Section 2.3.

Figure 8 shows the speedup using the first two optimiza-
tions presented in this section (code in Figure 6). The pa-
rameter NITER was set to one, and only the two program
parameters LENGTH and MAXGRID related to memory were
changed. The same values were used for these two param-
eters. With LENGTH = MAXGRID = 1200, the original code
took 26.9 seconds while the optimized version took 0.0017
seconds.

Figure 9 illustrates the speedup with all optimizations
combined. The parameters LENGTH and MAXGRID were set
to 1000 and NITER was changed for each data point. With
NITER = 1000, the original code took 1296 seconds whereas
the optimized version took 0.0006 seconds.

It is important to note that unusual speedup (i.e., a factor
of 15,000) can be achieved without exploiting the redundant
work coming from the outermost artificial loop.

It is unclear what is being computed in this kernel, but the
implementation in PolyBench does match the original imple-
mentation cited within the source code comments. However,
the original implementation does not have the outermost
loop repeating the same computation.
4. OTHER ISSUES

The issues found in the two kernels discussed above are (in our opinion) the most critical ones. We have thoroughly inspected all the kernels in PolyBench/C 3.2, and have found questionable behaviors in several of them. These findings are summarized below.

- Some linear algebra benchmarks may produce `nan` with default inputs.
- `adi` has a bug that causes forward and backward substitution to be independent of each other [9]. This makes the program to be tileable in all dimensions, where a correct implementation of Alternate Direction Implicit method is not.
- `symm` does not correspond to any of the configurations in BLAS.
- `trmm` is not correct; the values written to `B` are sometimes read for multiplication.
- `doitgen` uses 3D arrays for something that only requires 1D.
- `fdtd-apml` seems like it is missing the time dimension.
- Description of the `2mm` kernel on the website is different from the code. The website mentions $E = C.D; D = A.B$ where the code implements $E = \alpha A.B.C + \beta D$.
- `gramschmidt` is actually QR decomposition, and it appears that it is not using the Gram-Schmidt method to perform the decomposition.
- `lu` is actually LU decomposition, and `ludcmp` does LUD and forward/backward substitution.
- `cholesky` and `trisolv` should be in the same category as `lu` (solvers). `cholesky` and `lu` both perform a decomposition of a square matrix into upper and lower triangular matrices. `trisolv` is performing forward substitution. Thus, if `lu` and `ludcmp` belong to solvers category, these two kernels should as well.
- `mvt` may not be a good name since there is a BLAS routine named `gemvt` that perform a similar but different computation.
- `gauss-filter` is on the website, but is not in the distribution.

5. DISCUSSION

Some of the issues found are bugs without question. The issue in `trmm` is one such example: values multiplied in matrix multiplication should be inputs, not some computed value. This bug comes from incorrect array accesses, which seems to be due to mis-translation from a Fortran implementation. However, many others do not have a clear cut “fix”.

Let us revisit the `dynprog` example. If the first loop nest that initializes the `c` array is removed, different values would be computed at each iteration of the outer loop, making the “optimization” illustrated previously unapplicable. However, such a change would make the kernel no longer correspond to (repeated) dynamic programming, and it becomes unclear what it is supposed to be computing.
The situation is even less clear for memory allocations in the program. For many of the polyhedral tools that retain the original memory allocation, having a compact memory allocation may leave no room for parallelism. Similarly, those that feature memory contraction cannot expect much gains for already tightly allocated programs.

On the other hand, starting from a single assignment implementation, and showing great performance improvement is unlikely to be convincing. What is a “reasonable” memory allocation is a difficult question, and it may be dependent on the intent of the experiments.

Many of the issues can be addressed, at least partially, by having detailed specifications of the kernels.

6. SPECIFYING THE COMPUTATION

We propose to enhance PolyBench by providing a detailed specification of the computation. PolyBench/C would become a reference implementation of these specifications in C, and should come with detailed description of the algorithm being used, as well as other implementation choices, such as memory allocation.

6.1 Example: Cholesky Decomposition

We illustrate the specification we seek using cholesky, which implements Cholesky decomposition, as an example.

Cholesky decomposition is a special case of LU decomposition for positive-definite matrices.

It takes the following as input,

\[ A: N \times N \text{ positive-definite matrix} \]

and gives the following as output:

\[ L: N \times N \text{ lower triangular matrix such that } A = LL^T \]

The Cholesky-Banachiewicz algorithm is used by the C reference implementation. The algorithm computes the following, where the computation starts from the upper-left corner of \( L \) and proceeds row by row.

\[
L(i,j) = \begin{cases} 
  i = j & : & k(i,j) - \sum_{k=0}^{j-1} L(j,k)^2 \cr 
  L(i,j) & - & \sum_{k=0}^{j-1} L(i,k)L(k,j) \cr 
  i > j & : & L(j,j) \end{cases}
\]

In the reference implementation, memory allocation is almost in-place. A separate vector is used to store the diagonal elements, but other values are stored in-place.

6.2 Benefits of Specification

One of the immediate benefits is that the expected input is clearly specified. Some linear-algebra kernels expect matrices with certain properties. Incorrect inputs may cause numerically unstable operations, such as division by zero, which is one of the reasons \texttt{nan} is produced.

Questionable properties would be explained in the specification, leaving less room to be criticized. Since polyhedral analysis has the potential to automate algorithmic simplifications of redundant work, kernels with redundant loops may serve as a good “challenge”.

Having specifications as the central point of reference helps maintaining consistency among various versions of PolyBench implementations. For example, it would be useful to have hand optimized versions of the kernels, but such an effort should not be constrained by the implementation details in the C version. Similarly, implementations for other platforms (e.g., PolyBench/GPU) would benefit from specifications of the computation.

7. CONCLUSION

In this paper, we have discussed questionable properties found in PolyBench/C 3.2 kernels. In extreme cases, these properties can be exploited to produce arbitrary speedups, threatening the credibility of experimental evaluation.

We are exposing ourselves to numerous criticisms by using these kernels with questionable properties. We believe that part of the problem is the lack of detailed specification of the computation. We propose to enrich PolyBench by providing a detailed specification of the computation, clearly separated from implementation details, to improve the understanding of the kernels.¹

8. REFERENCES


¹Preliminary version for most of the kernels in PolyBench/C 3.2 is available at: \url{http://polyweb.irisa.fr/polybench-report.pdf}