An Empirical Study of the Performance of the APL370 Compiler

WAI-MEE CHING, RICK NELSON AND NUNGJANE SHI

IBM T. J. Watson Research Center
Yorktown Heights, NY. 10598 USA

ABSTRACT

The performance of a compiler is usually measured in terms of the execution efficiency of compiled code and the speed of compilation. For an APL compiler, we are also concerned about its relative performance with respect to the interpreter: C/I, the ratio of the speed of compiled code over interpretation. We give performance data on 4 groups of examples of moderate size: i) scalar style code where an interpreter does poorly and the C/I ratio is very high, ii) good APL style code where interpreter still does poorly due to inherent iterativeness or recursion, and the C/I ratio is high, and iii) good APL style code where the interpreter is very efficient on large data, and the C/I ratio is low, and iv) some particular primitives. These examples include neural net simulation, machine simulation, network routing, signal processing and mathematical computations. The APL370 compiler not only speeds up applications of iterative nature, but also gives good performance to codes utilizing APL's strength such as boolean selection and boolean data manipulation.

1. INTRODUCTION

A prototype copy of the APL370 compiler has been released on the APLTOOLS disk for IBM internal use. The APL370 compiler compiles an essential subset of APL, which includes recursion, directly into 370 assembly code. The compiled code can run independently of any interpreter or be called from APL2. This version does not yet include the option of directly generating vector instructions for the IBM 3090 vector facility (we shall release the vector version later). Hence, the performance data we discuss here is that of the released sequential version, not the experimental vector version we reported in [1]. Also, the interpreter we compare to is APL2 release 2 (it has no vector option). However, the compiler in the released form needs APL2 release 3 to run because it uses the package feature of release 3 to avoid possible name conflicts.

Since an APL interpreter provides excellent programming environment for APL program development, an APL compiler is not a replacement of the interpreter. The APL370 compiler is designed for execution efficiency, interpreter independent execution and ease of compilation. It is intended to help the following APL users:

- whose APL application program has an inherently iterative component,
- who need to write a module in APL implementing complex algorithms for a non-APL-based system where no APL interpreter is present during execution,
- who have no time, patience or experience to write their APL program in an elegant fashion so as to save interpretation time.

Despite current implementation limitations and remaining bugs in the compiler, we believe our compiler already can be of good help to many APL users to gain the speed and/or convenience of interpreter independent execution. Moreover, it is important to demonstrate that an efficient APL compiler is possible without imposing requirements alien to APL style programming and/or need to rely on other programming language processors for execution performance.

There are two components in the measurement of a compiler performance: First, the execution speed of the compiled code, and the second, the compilation time. We note that even though the second component is less important in comparison with the first, it is by no means negligible.

This is because in real life a successful compilation is usually not the end of the story. People naturally asks if any change can make the compiled code run even faster. Since this question cannot be settled by running on the interpreter, recompilation of a modified program becomes necessary. Hence, slow compilation speed is tolerable but very slow compilation speed is intolerable.

As for a the measurement of compiled code efficiency, a simple way is to measure how much time is required for the execution of each primitive. This can be quickly seen as ineffective for measuring the overall performance. For an interpreter is likely to outperform a compiler on the code for many/most primitives, but still be beaten by a compiler on whole examples. On the other hand, we can always cook up examples of triple loops which occur rarely in real life APL programs to show a speedup of, say 200 times. The real performance of an APL compiler is the speedup it can provide on codes it is likely to encounter in real life applications. Since there is no good way to assemble a sample of APL programs which all agree as representative, we decided to use a collection of varying characteristics to measure the performance of the compiler. Hence we called this an empirical study of the compiler performance.

Seasoned APL programmers typically develop an intuitive criterion as whether an APL program is of ***good APL style***. Usually, such APL programs execute faster (under interpreter) than equivalent but less well thought out programs. Our informal definition is that an APL program is of good APL style if it uses high level primitives and avoids loops whenever possible. We note that such a definition is only possible for a language of very high level where usually exist very different implementations of any particular computation task.

An APL compiler is of limited value if all it can speedup (with respect to the interpreter) are FORTRAN style programs. It must be able to help programs of good APL style in performance so that the efficiency of the compiled code is better than interpretation and is comparable to that produced by an optimizing FORTRAN compiler from corresponding FORTRAN programs. In the case of the APL370 compiler, this is indeed true and the compiled code actually executes faster then FORTRAN if powerful APL features such as bit-manipulations are involved. The compiled code in general executes no slower than the interpreter in stand-alone mode where no copying of data to and from an interpreter environment is involved. In order to prove our point, we collected a group of examples, some are simple and some are from users' real life applications. We group them into four different categories and measured their performance. All the performance data is obtained from running on a ***single processor scalar machine of IBM 3090 model***. The numbers are in ***cpu seconds*** for the table of compilation and in ***cpu milliseconds*** for the tables of execution time of compile code.

One important feature of compiler performance not discussed in this paper is that of space utilization. We only mention that the APL370 compiler currently does not provide a garbage collector at run-time, but it does have features to save space as to minimize the need of a garbage collector.

Finally, we remark that the current version of the APL370 compiler does not have any particular optimization other than in certain isolated but frequently occurring cases like `↑⍴`, `⍴⍴`, `[..⍳..]`, `..⍴0`. It is not that we regard such well known optimization technique like 'drag-along and beating' to combine primitive operations as not desirable; rather, we feel this is less important than the efficient implementation of certain frequently used primitives such as boolean compress when constrained with limited resource. We certainly will include streaming in our newer version of the compiler later to further improve the compiler performance.

2. COMPILATION TIME

We divide the compilation time in that of the front end and that of the back end. The front end time includes parsing, data flow analysis, type shape analysis and data dependency analysis. The back end time includes generation of 370 assembly code from the parse trees produced by the front end and the actual writing of code characters onto the output file. We can see that the back end spends more time than the front end. Part of the back end time is in writing files (about a quarter), and this part can be substantially reduced when we later use the block write feature of APL2. No discernible improvement can be expected of the front end where no output process is involved, unless substantially more efficient new data analysis algorithms become available. On the other hand, since both the compiler is written entirely in APL (VS APL) and essentially confined to the compilable subset, a speedup naturally will follow when we compile the compiler. The back end compilation speed depends very much on the density of the source code defined as the ratio of corresponding assembly code and its APL source code (a scalar style code is much less dense than an equivalent program using many high level primitives, i.e. code of good APL style). Hence, the usual line/sec. measure of compilation speed used in FORTRAN and PASCAL compilers is not really comparable, just as an APL programmer's productivity cannot be measured with the same lines/day accounting used for FORTRAN programmers.

Compilation Time

Example	Fns	Lns	Front	Back	Total	Lns/sec
PRIME4	1	10	0.5	2.11	2.61	3.84
QSORT	2	25	0.99	3.36	4.35	5.75
HEBB	2	63	2.07	9.85	11.92	5.28
PRIME5	1	6	0.37	2.11	2.48	2.42
PRIME3	1	7	0.44	3.23	3.67	1.64
QSOR	1	4	0.29	3.23	3.52	1.14
CHOLESK	2	15	0.96	5.38	6.34	2.37
WAIMEE	4	51	2.06	9.27	11.33	4.5
DAVID	1	25	2.08	12.1	14.18	1.76
COLORK	2	30	2.07	11.78	13.85	2.17
FFOT	1	41	3.17	28.93	32.1	1.28
BFRANK	1	15	0.72	5.38	6.09	2.46
SSOLVE	6	85	5.3	37.5	42.8	1.99
NIH	4	124	5.37	19.12	24.49	5.06

We note that back end time dominates and it roughly corresponds to the amount of 370 assembly code generated. Hence, the lines per second measure confirms our intuition of the denseness of a piece of APL code.

The compilation speed of the APL370 compiler is much faster than other APL compilers we are aware of. For example, it is mentioned in Wiedmann [2] that it takes 34 hours for the STSC's compiler to analyze and speedup an electronics design application. In our case, we recommend users to compile as large a unit as possible because our compilation time is not prohibitively expensive and compiled code is mostly no worse than the interpreter. This saves the effort to pinpoint a function which is an execution bottleneck for compilation.

Even though the issue of compilation speed is less important than that of the execution speed of compiled code, it is not some unimportant issue to be brushed aside. Indeed, for mature compiler technology on scalar languages such as FORTRAN and PASCAL where the difference between the execution speeds of compiled code produced by different compilers is less pronounced, compilation time is a major criterion for compiler performance. While APL programmers use the interpreter to developed their programs, after the code is compiled, they inevitably wonder whether they can do better. And this question can only be answered by recompilation and execution of a changed program. Prohibitive compilation cost will discourage such experimentation.

3. EXECUTION TIME OF COMPILED CODE

We group our examples into four groups: i) sequential style code, ii) APL style code with strong iterative or recursive components, iii) good APL style code already efficient on the interpreter and iv) some primitives. The speedup our compiler can provide are certainly different for each of these groups. An APL compiler can remove the severe performance penalty the interpreter exacts from naive APL users on their scalar style code; but it is more important for an APL compiler to reassure its users that they would not be penalized when writing good APL style code by the compiler. That is, the compiler should not perform worse than the interpreter on good style APL code. We also remark that even though the speedup the compiler provides on group i) is greater than that on other groups, the compiled code actually performs a bit better when an APL program is written in good APL style (compared with its scalarized counter part). So there is no incentive to write scalar style code when using the compiler. On the contrary, soon there should be strong incentive to write good style APL code when we make the vector code version of the compiler available (data on experimental results have already been presented in [1]). And work on progress on automatic multi-tasking of APL programs has already convinced us that to write good APL style code is even more important for the parallel execution on IBM 3090 style multi-processors since the use of high level primitives in APL provides suitable granularities for parallel processing. The execution time excludes output display time (both for interpreter). The compiled code time is measured in stand alone mode which is indistinguishable from the time measured when called from APL2 except the example BFRANK where there is a noticeable difference most likely due to getting input data from its APL2 environment to the compiled module (We are looking into this peculiar problem for very large character inputs). BFRANK is also the only workspace in our collection where for certain inputs the compiled code is running slightly slower than the interpreter. We have evidence that this is not due to the fact we didn't do combining on primitives. We suspect the reason to be that some of our primitives are still not implemented as efficiently as possible.

Only PRIME4 has a hand translated FORTRAN for comparison. This is because PRIME4 is of FORTRAN style (hence its translation is straightforward). It would be interesting to see how compiled APL code compares with FORTRAN on other examples. Unfortunately, to write the FORTRAN counterpart for other examples is not easy task and we do not have access to automatic translation at present.

Group I

PRIME4

N	Interpret.	Compiled	Speedup	FORTRAN
10	10	0	-	0
100	212	4	53	3
500	2069	35	59.1	22
1000	5544	89	62.3	58
2000	14908	233	64	148
5000	55614	847	65.6	522

QSORT

size	Interpreter	Compiled	Speedup
75a	88	0.63	139.9
75b	103	0.72	144.05

HEBB

size	Interpreter	Compiled	Speedup
L 2x6	25	1.22	20.49
R 73			

We note that PRIME4 is not exactly a scalarized version of PRIME5 below because it has boolean short circuit testing built into it. Hence it is more efficient algorithmically, but under interpreter it is slower because its' scalar style. QSORT is the scalar version of quick sort, which is the literal translation of the PASCAL program 2.10 on page 79 of Wirth [3] while QSO is basically an APL one-liner. HEBB is a program in the area of neural network. WAIMEE is a program doing simulated annealing for machine design. We only used 100-th size of real input because the interpreter takes too long time. It is these type of programs a compiler becomes a necessity since interpretation takes too long while rewrite in FORTRAN consumes valuable time of the designer. SSOLVE is the mathematical kernel of a package for interactive chip design available from IBM (this is the same program mentioned in [2] as an application in electronic design). Specifically, SSOLVE solves sparse matrix by the Gauss elimination methods.

NIH is a code from the National Institute of Health. The main function calls a function OPTD in a loop. The function OPTD is very iterative (its' flow graph is shown in the Appendix) and calls other 2 functions, both having only one basic block, inside loops. This is the only code which is not compiled as given. We made three modifications. First, the code was written in APL2 using nested arrays. We wrote an equivalent VS APL version which does not use nested arrays. Second, we rearranged parameters to the main function so that all integer inputs are grouped in the left argument and all floating point inputs are grouped in the right argument. Third, we insert code to initialize several variables a certain size to save heap space. We note that the vectors created during the execution of the code are rather small, so vector facility of IBM 3090 and vector code does not give much help in this example.

Group II

PRIME5

N	Interpreter	Compiled	Speedup
10	7	1	7
100	85	8	10.6
500	549	87	6.3
1000	1250	248	5.04
2000	2903	706	4.11
5000	8944	2906	3.08

PRIME3

N	Interpreter	Compiled	Speedup
1000	14	1.47	9.52

QUICKSOR

size	Interpreter	Compiled	Speedup
100	24	2.14	11.21

CHOLESKY

size	Interpreter	Compiled	Speedup
100	4	0.15	26.67

WAIMEE

size	Interpreter	Compiled	Speedup
L 2x12 R84x84	106179.2	8065.61	13.12

Group III

DAVID

size	Interpreter	Compiled	Speedup
L 12x2 R 62x4	29	2.3	12.6

COLORK

size	Interpreter	Compiled	Speedup
L33/R33x33	117	14.4	8.13
L64/R64x64	358	79.13	4.52
L100/R100x100	993	293.82	3.38

FFOT

size	Interpreter	Compiled	Speedup
8x8	7	0.37	18.91
32x32	14	1.08	12.96
128x128	27	3.67	7.36
1024x1024	98	30.4	3.22

BFRANK

size	Interpreter	Compiled	Speedup
L1041x8/R1x8	5	4.4	1.14
L1041x4/R4x4a	16	8.37	1.91
L1041x4/R4x4b	15	8.43	1.78
L 804x4/R4x4	12	5.61	2.14
L 823x8/R4x8	16	14.4	1.11
L 886x8/R1x8	4	3.7	1.08
L 798x8/R4x8	14	11.96	1.17
L 665x8/R 4x8	12	9.81	1.22
L1139x14/R1x14	7	8.29	0.84
L 701x2/R 2x2	3	1.49	2.01
L 701x4/R 5x4	7	3.27	2.14

SSOLVE

size	Interpreter	Compiled	Speedup
L162xR180 (25 equations)	117.45	20.37	5.76

NIH

Outer iters	Interpreter	Compiled	Speedup
20	384	73.27	5.24
90	792168	153498.8	5.16

We have seen from the above that a compiler which is not very efficient on computation intensive primitives has no advantage over the interpreter in good APL style code when data arrays become quite large. Hence, it is very important

for a compiler to generate very efficient code for computation intensive primitives. We give some sample below:

Group IV

Function		*Z←⍇A*	
size	Interpreter	Compiler	Speedup
10x10	10	2	5
50x50	296	128	2.31
100x100	2307	1019	2.26
Function		*Z←A⍇B*	
size	Interpreter	Compiler	Speedup
10x10	6	2	3
50x50	290	137	2.12
100x100	2331	1090	2.14

Function		*Z←A+.×B*	
size	Interpreter	Compiled	Speedup
10x10	4	1	4
50x50	216	100	2.16
100x100	1770	812	2.18
Function		*Z←A+.÷B*	
size	Interpreter	Compiled	Speedup
10x10	5	2	2.5
50x50	323	191	1.69
100x100	2622	1566	1.67

4. CONCLUSION

We have presented performance data of the APL370 compiler on a variety of examples. We see that the compiler not only speedup scalar type code, but also provides good performance on most good style APL code. Moreover, the compilation time is reasonable and can be further reduced with improved output method.

Acknowledgement We thank Ralph Linsker, David Stein and Dan Milch of IBM T.J. Watson Research Center, Leo Maissel of IBM Poughkeepsie, Dan Chazan of IBM Israel Scientific Center, Bill Frank of IBM Burlington, David Zein of IBM East Fishkill and Richard Simon of the National Institute of Helath for their interest in the APL370 compiler and for providing us interesting real life examples.

REFERENCES

[1] W.-M. Ching and A. Xu, A Vector Code Back End of the APL370 Compiler on IBM 3090 and some Performance Comparisons, Proc. of APL88 Conf., 69-76.

[2] C. Wiedmann, Field Result with the APL Compiler, Proc. of APL86 Conf., 187-196.

[3] N. Wirth, Algorithms+ Data Structures= Programs, Prentice Hall, 1976.

APPENDIX

```
     ∇ P←PRIME5 M;I;J
[1]  ⍝ Print first M primes. ⎕IO=1
[2]    P←M⍴0
[3]    P[I←J←1]←2
[4]   WHILE0:→(I≥M)/0
[5]   W1:→(∨/0=((⌈I*0.5)∘.|P)|J←J+2)/W1
[6]    P[I←I+1]←J
[7]    →WHILE0
     ∇
     ∇ P←PRIME4 M;I;J;IROOT
[1]    P←M⍴0 ⍝ print first M primes
[2]    P[I←J←1]←2 ⍝ ⎕IO=1
[3]   WHILE0:→(I≥M)/0
[4]    IROOT←⌈I*0.5
[5]   LOOPB:K←1
[6]    J←J+2
[7]   LOOPA:→(0=P[K]|J)/LOOPB
[8]    →(IROOT≥K←K+1)/LOOPA
[9]    P[I←I+1]←J
[10]   →WHILE0
     ∇
       ∇ PZ←PRIME3 X;NP;E;S
[1]    P←X⍴1
[2]    E←(S←⌊X*0.5)⍴P[1]←0
[3]   RPT:→(S<NP←P⍳1)/END
[4]    P←P>X⍴(-NP)↑1
[5]    E[NP]←1
[6]    →RPT
[7]   END:PZ←(E/⍳S),P/⍳X
       ∇


     ∇ Z←QSO V;B;W
[1] →(1<⍴V)/S ⍝ ⎕IO←1
[2] Z←V
[3] →0
[4]S:Z←(QSO(~B)/W),V[1],QSO(B←W≥V[1])/W←1↓V
     ∇


     ∇ L←CHOLESK A;N;I;C;J
[1]⍝Cholesky decomp of sym. posi.def.matrix
[2]    N←(¯1+((1+8×⍴A)*0.5))÷2
[3]    L←A[1]*0.5
[4]    I←1
[5]   S2:→(N<I←1+J←I)/0
[6]    C←L LTR1(J↑(⌊((I-1)×I)÷2)↓A)
[7]    L←L,C,(A[⌊(I×I+1)÷2]-C+.×C)*0.5
[8]    →S2
       ∇
```

```
      ∇ L←A0 LTR1 X0;N;I;A;X
[1] ⍝Calc. elements below the main diagonal
[2]  X←,X0
[3]  A←,A0
[4] S1:N←(¯1+((1+8×⍴A)*0.5))÷2
[5]  I←1
[6]  L←,X[1]÷A[1]
[7] S2:→(N<I←I+1)/0
[8]  L←L,(÷A[J+I])×X[I]
     -A[(J←⌊I×J÷2)+⍳J]+.×(J←I-1)↑L
[9]  →S2
      ∇

      ∇ C←ZAP COMADJ A;I;J;N;M
[1]   N←1↑⍴A   ⍝⎕IO←0
[2]   M←¯1↑⍴A
[3]   C←(2⍴N)⍴0
[4]   I←0
[5]  L0:C[J;J]←1+C[J;J←A[;I]/⍳N]
[6]   →(M>I←I+1)/L0
[7]   →(0=ZAP)/0
[8]   C←C×~A
      ∇

∇ R←MEM COLORK A;B;C;I0;J0;NS;W1;W2;W3;Y;Z
[1]  X←C1←1 COMADJ A
[2]  C0←,C1
[3]  C0[(1+B)×⍳B←1↑⍴A]←0
[4]  C←(⍴A)⍴C0
[5]  NS←(1↑⍴C)⍴1
[6] L1:→(0≥B←⌈/,C)/L2
[7]  B←(,C)⍳B
[8]  J0←(⍳0)⍴(1↑⍴C)|B
[9]  I0←(⍳0)⍴⌊B÷1↑⍴C
[10] Y←A[J0;]+2×A[I0;]
[11] W1←(Y=1)/⍳⍴Y
[12] W2←(Y=2)/⍳⍴Y
[13] W3←(Y=3)/⍳⍴Y
[14] Z←(~A[;J0])×+/A[;W1]
[15] A[;I0]←A[I0;]←A[I0;]∨A[J0;]
[16] C[W3;W3]←C[W3;W3]-1
[17] C[W2;W1]←⍉C[W1;W2]←C[W1;W2]+~A[W1;W2]
[18] C[;I0]←C[I0;]←C[I0;]+Z×~A[I0;]
[19] MEM[(MEM=MEM[J0])/⍳⍴MEM]←MEM[I0]
[20] C[I0;W1]←C[W1;I0]←C[J0;]←C[;J0]←A[J0;]
      ←A[;J0]←NS[J0]←0
[21] →L1
[22]L2:R←MEM
     ∇

 ∇ MM←RTMP DAVID M;CROSS;JΔ;LΔ;PΔ;KΔ;MID
[1]  LEV←RTMP[1 2 ;]
[2]  SM← 2 0 ↓RTMP
[3]  MM←M
[4]  MM[;1]←(MM[;1]≠0)×LEV[1;2]
[5]  →((1=1↑⍴LEV)∨∧/LEV[1;2]=LEV[;2])/0
[6]  CUMSM←+\SM[;1]×2-SM[;2]
[7]  FΔ← 2 4 ⍴0
[8]  LΔ←1⌈+/M[;4]∘.≥LEV[;1]
[9]  MM[TΔ/⍳1↑⍴M;1]←LEV[(TΔ←M[;1]≠0)/LΔ;2]
[10] CROSS←(((1↓TΔ),0)∧LΔ≠(1↓LΔ),¯1↑LΔ)/⍳⍴LΔ
[11] JΔ←''⍴⍴CROSS
[12]LOOP:→(JΔ=0)/0
[13] KΔ←CROSS[JΔ]
[14]RP:MID←''⍴LEV[LΔ[1+KΔ]+(MM[KΔ;4]≥
       MM[KΔ+1;4]);1]
[15] PΔ←(|MM[KΔ;4]-MID)÷|MM[KΔ;4]-MM[1+KΔ;4]
[16] LΔ[1+KΔ]←LΔ[1+KΔ]+×LΔ[KΔ]-LΔ[KΔ+1]
[17] FΔ[;1]←LEV[LΔ[1+KΔ];2],0
[18] FΔ[; 2 3 4]←(2 3 ⍴MM[KΔ;1+⍳3])

       (2⍴PΔ)∘.×-⌿MM[1 0 +KΔ;1+⍳3]
[19] MM←MM[⍳KΔ;],[1] FΔ,[1](KΔ,0)↓MM
[20] TΔ1←(CUMSM≥KΔ)⍳1 ⍝ WAS TΔ ****
[21] SM[TΔ1;1]←SM[TΔ1;1]+1+SM[TΔ1;2]=1
[22] →(LΔ[1+KΔ]≠1⌈+/LEV[;1]≤MM[KΔ;4])/RP
[23] JΔ←JΔ-1
[24] →LOOP
      ∇

    ∇ E←A1 BFRANK A2;I;IT;ROW;CT;A2V
[1] ⍝ Bill Frank's version, 9/87
[2] →(~'*'∈A2)/L0 ⍝ Branch if no * in arg
[4]  A2V←,A2
[5]  A2V[(,A2='*')/⍳⍴A2V]←' '
[6]  A2←(⍴A2)⍴A2V
[7] L0:→('_'∈A2)/L1 ⍝ Branch if there are _
[8]  E←,∨/A1∧.=⍉A2 ⍝ do simple inner prod →0
[9]  →0
[10]L1:I←⍳¯1↑⍴A1 ⍝ Indices of data
[11] ROW←1↑⍴A2 ⍝ Loop counter & limit
[12] CT←1
[13] E←(1↑⍴A1)⍴0 ⍝ Initialize results
[14]L2:IT←('_'≠A2[CT;])/I
[15] E←E∨A1[;IT]∧.=A2[CT;IT] ⍝find the match
[16] →(ROW≥CT←CT+1)/L2
    ∇
```

```
      ∇ W←FFOT X0;Z;X1;I;R;U;RO;X
[1]   ⍝ COMPUTES FFT OF A REAL FN WITH 2*N PTS
[2]   ⍝ INVERSE IS FTI.  (⍴W)=(2,1+(2*N)-1)
[5]    →((2⊛⍴X0)≠⌈2⊛⍴X0)/0
[6]    X←((RO←(⍴X0)),1)⍴X0
[7]    Z←X[⍳⌊R÷2;],X[R+⍳⌊(R←⌊(1↑⍴X)÷2)÷2;]
[8]    X←(X[R+⍳R;],X[(3×R)+⍳R←⌊R÷2;])
[9]    Z←Z+.× 2 2 ⍴ 1 1 1 ¯1
[10]   X←X+.× 2 2 ⍴ 1 1 1 ¯1
[11]   W←(R⍴ 1 0)⍀((R,2)↑Z),(R←⌊0.5×R)↑Z[;1]
[12]   W[¯1+2×⍳R;1]←W[¯1+2×⍳R;1]+X[⍳R;1]
[13]   W[¯1+2×⍳R;3]←W[¯1+2×⍳R;3]-X[⍳R;1]
[14]   W[2×⍳R;2]←-X[⍳R;2]
[15]   X←((R←⌊(0.5×R)),0)↓X
[16]   Z←((2×R)⍴ 1 0)⍀((R,0)↓Z),R↓Z[;1]
[17]   Z[¯1+2×⍳R;1]←Z[¯1+2×⍳R;1]+X[⍳R;1]
[18]   Z[¯1+2×⍳R;3]←Z[¯1+2×⍳R;3]-X[⍳R;1]
[19]   Z[2×⍳R;2]←-X[⍳R;2]
[20]   U← 2 1 ∘.○02×(-(¯1+⍳⌊1+RO÷2)÷RO)
[21]   I←3
[22]   →(1=1↑⍴Z)/0
[23]   X←Z[¯1+2×⍳R;]
[24]   Z←Z[2×⍳R;]
[26] LO1:X←X, 0 1 ↓⍉X
[27]   Z←Z,- 0 1 ↓⍉Z
[28]   X1←X×(⍴X)⍴U[1;1+(⌊RO÷2*I)×¯1+⍳⌊1+2*I-1]
[29]   X←X×(⍴X)⍴U[2;1+(⌊RO÷2*I)×¯1+⍳⌊1+2*I-1]
[30] X1←X1-Z×(⍴X)⍴U[2;1+(⌊RO÷2*I)×¯1+⍳⌊1+2*I-1]
[31]   X←X+Z×(⍴X)⍴U[1;1+(⌊RO÷2*I)×¯1+⍳⌊1+2*I-1]
[32]   Z← 0 0 ⍴0
[33]   X1←X1+W[¯1+2×⍳R;], 0 1 ↓⍉W[¯1+2×⍳R;]
[34]   X←X+W[2×⍳R;],- 0 1 ↓⍉W[2×⍳R;]
[35]   W←(2 1 ⌈⍴X1)⍴0
[36]   W[¯1+2×⍳⌊(1↑⍴W)÷2;]←X1[⍳⌊(1↑⍴W)÷2;]
[37]   X1←((⌊0.5×1↑⍴W),0)↓X1
[38]   W[2×⍳⌊(1↑⍴W)÷2;]←X[⍳⌊(1↑⍴W)÷2;]
[39]   Z←((⌊0.5×1↑⍴W),0)↓X
[40]   X←X1
[41]   I←I+1
[42]   R←⌊R÷2
[43]   →(0=1↑⍴X)/0
[44]   →LO1
      ∇
```

Figure 1. Flow Graph of OPTD in NIH