An ELI-to-C Compiler: Production and Performance

Hanfeng Chen
School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
E-mail: chenh@cs.queensu.ca

Wai-Mee Ching
201 Kensington Way,
Mount Kisco, NY, USA 10549
E-mail: waimeeching@gmail.com

Abstract. ELI is a succinct array-based interactive programming language derived from APL. An ELI compiler which does not require additional variable declarations can offer programmers the productivity of a script language and the execution performance of a compiled language in a single language platform. We first implemented such a compiler in ELI, which translates flat array ELI programs into C. After generating the translated C version of the compiler we cut the C source file into a large group of small pieces and compiled it to produce object code for its distribution. We present the compiler performance on three common examples in comparison with native C code and the speedup gained over ELI interpreter.

Keywords: array programming language, compiler, bootstrapping, performance

1. Introduction. There are widely used interpreter based array languages such as MATLAB and Python, which provide programming productivity, but suffer in execution performance in comparison with compiled languages such as C or FORTRAN. In situations where execution performance does matter, one conventional approach is to use some profiling tool to find out where a program spends most of its time and rewrite the computation intensive portion, presumably much smaller than the whole program, in a compiled language and then replace that part of the program with a call to the compiled module. A better approach would be to provide a compiler for the scripting array language directly to the extent possible, and then either execute the compiled version of a whole program or replace the portion of a program which can be compiled with a call to the compiled module. This indeed has been done for the array language APL, APL/370 compiler in [6] and COMPC (APL-to-C translator) [8] at IBM T.J. Watson Research Center. While that compiler did not end in a product, it was tested internally and by a few select customers [2]; its structure, deployed technologies and performance were published in scientific literature [6,8]. ELI is an interactive array programming language based on APL, [4] i.e. its core part covers APL1, i.e. ISO APL[11], but it uses ASCII characters instead of the special APL characters. In addition to the features specified in [11], ELI [4,7] has complex numbers, symbols, date/time as raw data types, lists for non-homogeneous data, dictionaries, tables and basic query capability. ELI is available on Windows, Mac OS and Linux at http://fastarray.appspot.com/.

The initial ELI compiler *ecc.esf* is written in ELI, and it translates an ELI program using only core ELI features, i.e. the portion corresponds to APL1, into C. This is similar to COMPC, and indeed it follows the same technical approach and requirement: No variable declarations are required, but types and ranks of input parameters need to be supplied when invoking *ecc*. In order to widen the distribution of this compiler while maintaining its integriy, we decide to compile the source code of compiler by first converting it into ecc.c and then using gcc to turn it into an executable file. Due to the size of the compiler, both steps incurred considerable difficulties, and the main purpose of this paper is to report our strategy and experience in dealing with these challenges. We also hope this will demonstrate that ELI as a very high-level array-oriented language can provide programming productivity not only in computational tasks in scientific, engineering and financial area but also in enabling a small team to implement systems of considerable complexity.

We present performance data of the compiler on three common examples from the Princeton suite [3] showing that while the compiled ELI code is not as fast as hand-coded C it is good enough for people to seriously consider a fair trade off between convenience in using a succinct array language and that of coding in a verbose language merely for execution performance. Finally, we'll discuss future work on this compiler. Obviously, we would like to expand

the capability of the compiler to accommodate more language features of ELI beyond flat arrays of APL1 to make it more useful for a variety of programming jobs, e.g. supporting list, complex number and datetime. In future work, we will investigate the research and development of an automatic parallelizing compiler for multi-core machines following the recent work in APL by Ching and Zheng [9]. The next section will briefly introduce the ELI language, the compiler restrictions and its main technical features. Section 3 will be a detailed discussion on self-compilation; section 4 on performance measures and section 5 on future work. First, we give a brief introduction of ELI.

ELI has four basic data types: ***numeric*** (*boolean, integer, real and complex numbers*), ***character***, ***symbolic*** and ***temporal*** (*date, month, time, second, minute, datetime*). A single data item is called a ***scalar***; a homogeneous and rectangular collection of data is called an ***array***. ELI provides a large number of ***primitive*** functions (each denoted by one or two characters) which operate on scalars as well as on arrays as a whole; a primitive function is either monadic (i.e. has a right argument) or dyadic (i.e. has a left and a right arguments). There are two kinds of primitive functions: ***scalar*** and ***mixed***. A scalar primitive function (which includes *arithmetic*, *logical* and *relational* functions) has the property that it operates on arrays as an extension of its operation on elements of the arrays. Each array has a *shape* (dimensions) and the shape of all scalars is an *empty vector* (vector of length 0). A mixed function may query the shape of a data item or *reshape* it. In general, a mixed function either transforms an array or takes part of it, including indexing. In addition, there are four ***operators***: ***reduce***, ***scan***, ***inner*** and ***outer products*** which apply to one or two scalar primitive functions to produce a *derived function*. For example, reduce (*/*) applied to addition + is a *summation* (+/); Inner product (:) applied to + and * is the matrix multiplication function (+:*) for two compatible matrices. A line of code in ELI operates from right to left as a chain of operations in a *dataflow* style, i.e. the output of one operation feeds as the input of next operation; and all functions are of equal precedence, so 3*2+1 is 9 not 7. For example,

```
+\1+2*!10
```

results in the partial sums of odd integers from 3 to 21. A user ***defined*** *function* can be monadic or dyadic, or *niladic*(i.e. takes no argument), it can yield results or no result; defined functions can have local variables and the *scoping rule* for *shadowing* variables is ***dynamic***. ELI provides branching as in APL1; but ELI also provides C-like control structures. ELI compiler covers these features except data types new to APL1 (complex numbers, symbols, temporal data and control structures).

The general data structure in ELI is ***list*** which is a linear collection of data items enclosed by parenthesis (..) and separated by semicolon ';', each of which can be a scalar, an array or another list. There is an ***each*** operator " which for a function f, f", applies f to each item in its argument list. A ***dictionary*** in ELI is a special list which maps a set of *keys*, i.e. the *domain*, to a set of *lists*, called the *range*. A dictionary whose range is a group of lists of equal length is called a column dictionary whose transpose is called a ***table***. ELI provides***esql*** which contains a set of statements is similar to standard SQL to query and capable to process data in tables of a database. Yet, these features of ELI are not covered by the current compiler.

2. Writing the ELI compiler in ELI. APL is an array-oriented programming language invented by Ken Iverson to teach mathematics, for which he received the Turing Award in 1979 [10]. In comparison with other array languages developed later such as MATLAB and Python, APL has two distinct features:

1) a pair of *monadic* (has only right operand) and *dyadic* (has left and right operands) primitive (*array*) functions, is represented by a single character symbol.
2) a line of APL code executes from right to left in a *dataflow style*, i.e. the output of one operation feeds the next operation as input.

This results in a succinct and very productive language which has been used profitably in a wide range of areas such as finance, actuary work, logistics and computer aided design. ELI [4], at its core, is just an ASCII version of APL1: it replaces each APL character representing primitive functions with one or two ASCII characters, but still maintain

the *one-character one-symbol* feel of APL thus encourages a *dataflow style* of programming. However, ELI has control structures, complex numbers, temporal data which are not prescribed in ISO APL. Moreover, unlike IBM APL2 which introduces nested arrays into APL, ELI provides lists for non-homogeneous data and data nesting. ELI is available on Windows, Mac OS and Linux at http://fastarray.appspot.com/.

ELI, like APL and other interactive array languages, suffers on execution performance in comparison with compiled languages while excels in programming productivity. Many people used APL as a prototyping language, i.e. one quickly writes an initial version of a program in APL, and once tested, rewrites in a production language such as C or FORTRAN. Similarly, professional Python programmers advice writing a program in Python first, then using a profiling tool to find a segment where the program spends most of time and then rewrite that segment in a compiled language. The final program is a mixture of Python calling a compiled sub-module. Both approaches while improve execution performance decrease the convenience and productivity of the original script language. A better approach is of course to provide a compiler for the script language involved. This brings up to the earlier work of APL/370 compiler [6] and COMPC, an APL-to-C translator [8]. As COMPC evolved from the APL/370 compiler, *ecc.esf* evolved from COMPC; COMPC is written in IBM VS APL system covers APL1 while *ecc.esf* is written in ELI (all ELI script files are of file type *esf* but ordinary text file of *. txt type also works as a source file for ELI as long as it confines to ELI syntax, see ELI Primer [7]).

Just as COMPC only covers most features in APL1, *ecc* can only compile ELI programs where only flat arrays are involved (arrays in ELI are by definition of homogeneous type, or flat). ELI also restricts *function* names not to be used as *variable* names which are allowed in APL1. An ELI *compilation unit* consists of a main function and all functions in the workspace called directly or indirectly by that main function which takes two parameters; the parameters can be lists, but then the first line in the main function must assign the list(s) to a group of arrays or scalars. No place in any function in the compilation unit can invoke the primitive function *execute* (!.) which dynamically interprets its argument (character) string. For the time being, we also exclude the use of complex numbers, symbols and temporal data in a compilation unit (these data types are not in APL1), though it is fairly easy to extend the compiler to cover these newer data types. Other than taking (two) input parameters, functions in a compilation unit cannot access any global variables in a workspace.

The compiler *ecc* consists of a front end and a backend. The front end takes the text (matrix) of the main function and each defined function encountered in a depth first search fashion, turn each of them into a list of parse trees. We note here that the ecc parser does not utilize the popular *yacc* but uses a two-symbol look-ahead algorithm which scans a line left-to-right to build a parse tree. It then combines lines into basic blocks and builds a flow-graph for each function. A major technical feature of the front end is its use of Tarjan's *fast interval finding* algorithm [16] to implement the *reach-definition* calculation of Allen and Cocke [1] for *u-d chaining*. Since the types and shape, at least rank (i.e. dimension) of input parameters have been indicated when invoking the compiler, that information is propagated through all functions and u-d chaining it forms the basis of type-shape analysis in deciding the types and shapes of all variables and parsing nodes of all functions in a compilation unit.

The backend consists of a main function, *treewalk*, and a group of functions each implementing a (group of) primitive function(s) in ELI such as arithmetic functions, logical functions, comparisons, rotate and membership. ecc first declares a set of global variables in C for the compiled code to use, and it also declared a set of variables for each function in the compilation unit as their appropriate data types and dimensions have been found in the front end. It then takes the annotated parse trees produced by the front end and applies *treewalk* to each parse tree visiting each node from the lower right corner up to the root. There are four kinds of nodes in a parse tree: constant node, variable node, primitive function node and defined function node. When *treewalk* encounters a variable node, it accesses the variable table to translate its name. When it encounters primitive function nodes, it calls the function implementing that primitive function. When it encounters a defined function node, it generates a function call in C to that function.

The *ecc.esf* compiler has been tested on over 100 ELI programs with the largest one having 1000 lines of code. The majority of the programs in this test suite come from APL educational samples and applications in a variety of fields.

3. Self-compilation. We remark here that while ELI provides workspace for organizing programming task as in APL, it is more convenient in ELI to use script files *(*.esf*) for saving/loading large programs. *ecc.esf* has 15,000 lines of code which, when translated into C results in 333,778 lines of *ecc.c*. Even though ecc.esf is fully functional and for moderate size ELI programs the compilation time is not an issue, we would like to convert *ecc.esf* into *ecc.o*, i.e. compiled C object code. This is because we want to maintain the integrity of *ecc* by offering the public only access to the executable of the compiler.

Let us call the initial version of the ELI compiler, written in ELI *ecce* instead of ecc.esf for convenience. The main function of *ecce* is COMPILE whose left argument LPARM is a character string and whose right argument RPARM is a numeric vector. LPARM's initial part is the name of the main function of the unit to be compiled, followed by a blank and two characters indicating the types of the left and right arguments ('C' for character, 'I' for integer and 'E' for floating point), and RPARM is an integer vector starting with `[]IO` and followed by shape elements of the left and right arguments (for a scalar, it is 0, for an array it is the rank followed by dimensions with _1 indicating an unknown length). Hence, after loading *ecc.esf*, the compiler *ecce* is invoked by

```
'mainfn ltrt' COMPILE qdio shpl shpr
```

(in case there is no left argument, then that part is empty). To compile *ecce* itself, it would be

```
'COMPILE CI' COMPILE 0 1 _1 1 _1
```

The source code of the functions in a compilation unit is an implicit input to the compiler. This is supplied in ELI by the system function `[]CR`:

```
mtxt<-[]CR'fnam'
```

fetches the code of a function named `fnam` in a character matrix.

Initially, *ecce* just uses system functions in ELI in an ELI environment, but do not generate their C equivalent. One example is the system function `[]CR` for which we introduce a new function called `PFQDCR` in *ecce* to generate corresponding code in C by calling the equivalent code in *elimacros.cpp*. Another example is the system function `[]FM` which is used to save the value of a variable into a file; its corresponding C code is the function *writecode*, declared in *elimacros.cpp*. We also add capability to ecce so that it can take esf files to enable us to pass in LPARM and RPARM as files.

One of the most important restrictions in *ecc* is that the type and the rank of a variable cannot be changed during compilation. This forces us in several circumstances to rename variables manually for different use of a variable name. We note that the annoying restriction can be eliminated had we implemented SSA (static single-assignment) process in basic blocks, but we decide to delay this work for quick delivery of ecc. The complete compilation procedure has to go through front end and back end. In front end, the compiler initializes tables, parses the source code of each line in each defined function encountered and checks variables.

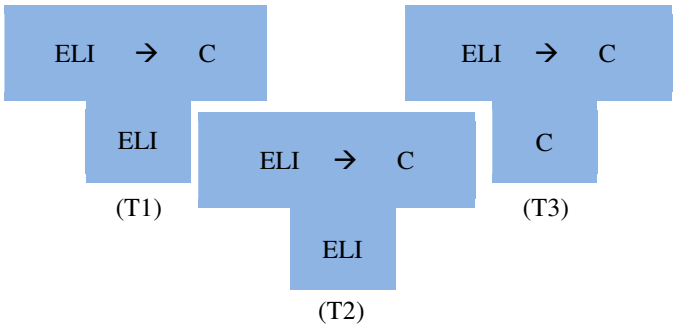

Figure 1: T-Diagram of ELI self-compilation

Tombstone diagram (T-Diagram) in figure 1 is used to describe bootstrapping in compiling compiler itself.

- In stage T1, ELI compiler, written in ELI, runs in ELI interpreter and it can translate ELI source code to C code.
- In stage T2, the ELI interpreter based compiler compiles itself in the ELI interpreter environment. The procedure of bootstrapping generates about 330K lines of C code.
- In stage T3, we use the available GNU C/C++ compiler to build an executable file. After we implement I/O facilities and internal data structures, it can directly translate ELI code into C code without ELI interpreter environment.

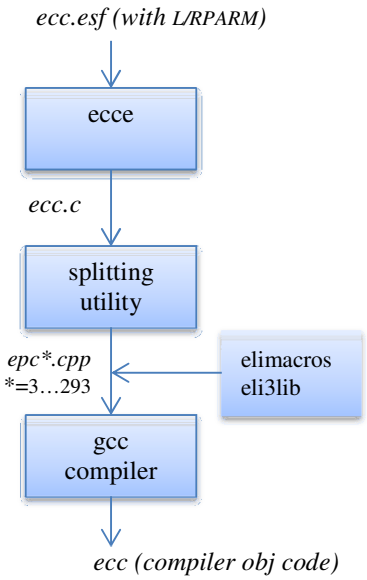

Figure 2: The workflow of ELI compiler self-compilation

Because *ecc* has to read data from *esf* files, it needs a data structures to store these information, struct _eline is used to store the information on lines in *esf* file. The declaration of _eline is as following:

```
typedef struct _eline{
    char* val;
    eline *nxt; int elen;
```

```
    eline(){nxt=NULL; val=NULL; elen=-1;}
    eline(char* s){val=(char*)malloc((elen=strlen(s))+1); strcpy(val,s);}
}*L,EL;
```

We are not getting into details of structs for function node and variable node here but just note that variable *ftot* is the index to the array variable *ftable* and it indicates how many functions we have processed so far.

```
extern FN ftable[MAXF];
extern int ftot;
```

After *ecce* has translated itself into *ecc.c*, a substantial problem comes out: the size of the C file *ecc.c* far exceeds those generated in our test suite. There are 291 functions in *ecc.c* with more than 330k lines of code which clearly is too huge for a C compiler to deal with. Hence, we must partition it into pieces for separate compilation. We decide that one function per file is a simple way to get that done, and we write a program in C to split that single large file into 291 files, utilizing the fact that each generated function starts with an obvious delimiter:

```
char delim[] = "/*              CODE SEGMENT FOR"; // function flag
```

During the process of compiling functions separately, we create other four files for the whole project:

```
Makefile    store important configuration
efunc.h     the declaration of all functions
ehead.h     use "extend" to globalize constant variables
emain.cpp   main function
ecp*.cpp    * ranges from 3 to 293 for the original functions from ecc.o, like
            epc03.cpp
```

In `Makefile`, we attach eli3lib.o as library functions and *elimacros.o* as I/O interface. Although we choose C++, most features are pure C style. In every ecp*.cpp file, the two lines for including heads are necessary:

```
#include "ehead.h"
#include "efunc.h"
```

There are several debugging macros defined in *elimacro.h* to quickly detect unexpected errors: The system macro __FILE__ and __LINE__ help identify which file and line trigger the debugging macro.

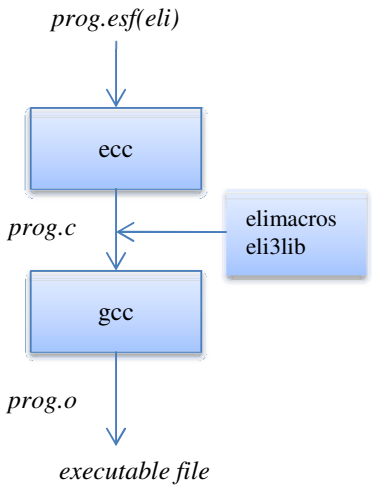

Figure 3: A demo about how to compile & run an application program

4. Performance of *ecc*. Overall, *ecc* achieves a speedup for compilable ELI programs comparable or better than that of COMPC does for corresponding APL programs [8]. The reason that sometimes the speedup is better reflects the fact that the ELI interpreter is less optimized compared with mature product interpreters. It also shows our deliberate strategy of relying on compiler for performance rather than doing case by case tune-up of the interpreter code. As we noted in [5], MATLAB offers a compiler, but it is mainly for producing standalone executable than to significantly speedup an application. However, speedup of compiled code over its interpreted version for a scripting language is not as important as a comparison with an equivalent program written in a compiled language such as C. While we do not expect a C program generated from an ELI program by the compiler to execute as fast as a hand-coded C program, getting data on this helps people to weigh the pros and cons in using a scripting language for a particular programming task. For this purpose, we picked two C programs, *Black-Scholes* is from the Princeton suite [3], *K-means* and *Hotspot* are from the Rodinia Benchmark [14,15]. We quickly wrote their ELI version (available at http://fastarray.appspot.com/compile.html) and feed them through the compiler. The machine we use to measure performance is an ASUS laptop with Intel 4-core i7-3610QM CPU, clocked at 2.3GHz and with 8 GB DDR3-1333 memory. It runs *Ubuntu 12.04*, 64-bit version. The C++ compiler we used to generate machine code is g++ *v4.4* for Linux. The compiler flag for C programs is set to *–O3*. We ran each program at least 5 times to get the average timing. The ELI version is *v0.2* for Linux. Here is a brief description of each example followed by performance charts.

1. **Black-Scholes**: The program is used to compute the price of options using Black-Scholes formula and the cumulative normal distribution function. They are implemented in *BlkSchlsEqEuroNoDiv* and *CNDF* functions respectively. The ELI version is in array form without loops. It uses arithmetic primitives in a dataflow style, i.e. output of one operation feeds as input to the next from right to left. It also uses the feature of a function's multi-arguments input.

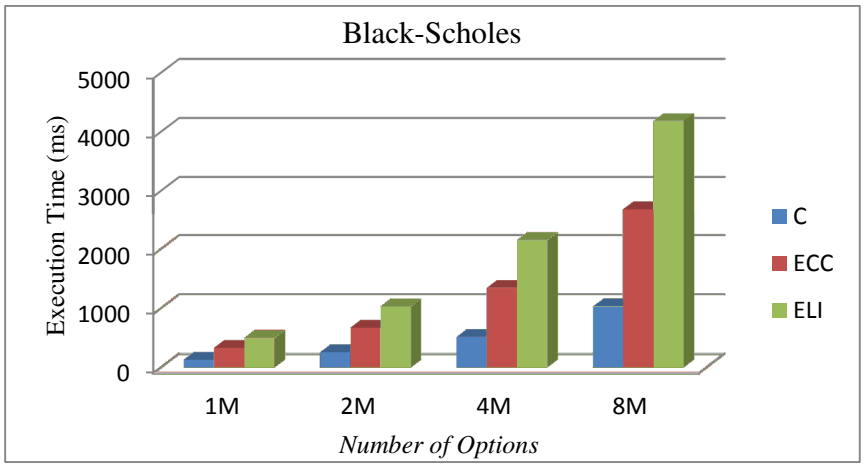

Figure 4: Given different number of options in standard Black-Scholes programs from [3].

Table 1 The result of Black-Scholes

Black-Scholes (ms)	1M	ratio	2M	ratio	4M	ratio	8M	ratio
C	129.6	1	255.7	1	513.9	1	1025.3	1
ECC	330	2.55	670	2.62	1340	2.61	2685	2.62
ELI	499	3.85	1032	4.04	2150	4.18	4183	4.08
Average speedup between ECC and ELI							2.78x	

The result of Black-Scholes shows that C always gives the best performance but ECC at least help ELI gain good speedup in approximate **2.78x** on average. Because the program is quite straightforward: the main function

BlkSchlsEqEuroNoDiv only calls *CNDF* function two times and there is no loop inside programs, it only gains modest speedup.

2. **K-means**: It implements a clustering algorithm which is used extensively in data-mining. In K-means, a data object is comprised of several values, called ***features*** [2]. It is an iterative program, particularly suited for C; K = 3 means there are 3 *clusters* on various numbers of data points. For each data point, there are 30 *features* in our input data.

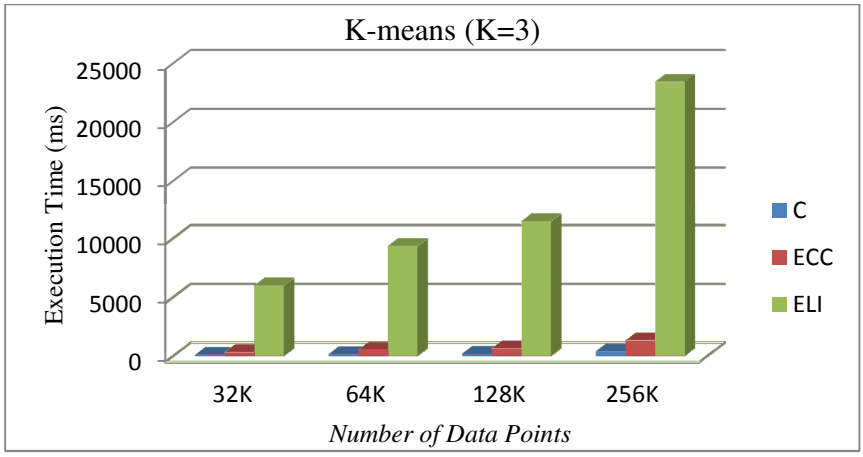

Figure 4: Given K=3 in standard K-means programs from [14,15]

Table 2 The result of K-means (K=3)

(K = 3)-means (ms)	32K	ratio	64K	ratio	128K	ratio	256K	ratio
C	104.4	1	162.4	1	200.5	1	397.7	1
ECC	350	3.53	540	3.33	670	3.34	1340	3.37
ELI	6030	57.8	9370	57.7	11490	57.3	23440	58.9
Average speedup between ECC and ELI							17.08x	

Because of loops inside K-means to find the best fitted cluster groups, it significantly decreases the performance of ELI interpreter. With the help of ECC compiler, the compiled code gains speedup in about 17x on average compared with ELI code. Even though there is a performance gap between C and ECC, the ELI code is compact with only 13 lines of code. It is clear that ECC significantly improves the performance compared with original ELI code.

3. **HotSpot**: A thermal simulation program to iteratively solve a series of differential equations on a rectangle which is used for chip-design. Each output cell in the computational grid represents the average value of temperatures of the corresponding area of the chip [14,15]. The number of iterations is set to 20 and the size of data is increased from 256x256 to 2048x2048.

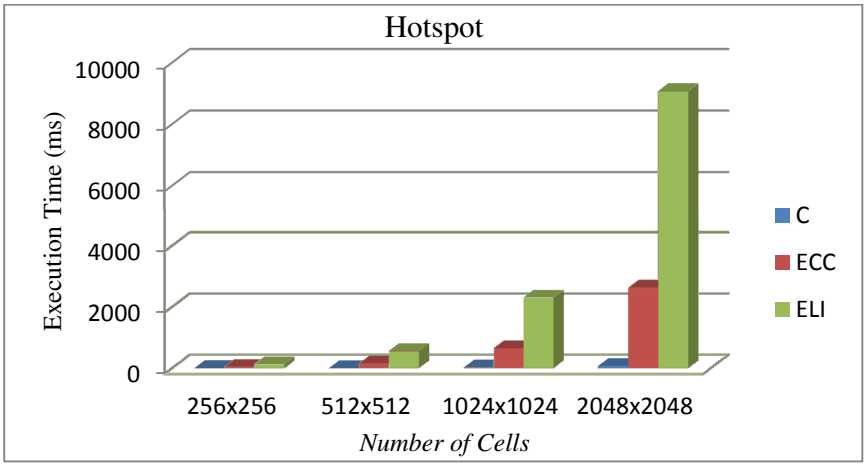

Figure 5: Given matrix size from 256x256 to 2048x2048

Table 3 The result of Hotspot

Hotspot (ms)	256x256	ratio	512x512	ratio	1024x1024	ratio	2048x2048	ratio
C	1	1	4.5	1	19.5	1	77.5	1
ECC	40	40	150	33.3	650	33.3	2630	33.9
ELI	126	126	560	124.4	2308	118.4	9079	117.1
Average speedup between ECC and ELI							3.47x	

The result shows that ECC has more than 3x speedup on average. Since there is a main loop in the program, *ecc* speedup is better than in BlackScholes. C is still faster than ELI since ELI code is written in matrix form while C version is in scalar m(i,j). If we incorporate the techniques related to array operations employed in [5] into *ecc* we expect *ecc* to perform much better.

We see that the hand-coded C code executes around 2.5 and 3.5 faster than the compiled ELI code in Black-Scholes and K-means but much faster in HotSpot. In all, the compiled ELI code improves substantially in speed over interpreted ELI. This is quite satisfactory in light of the difference in the code sizes of the original C version and the ELI version, which indicates a measure of programming productivity, and the ease of compiling. In the next section, we'll outline our planned work aiming at closing this performance gap with C for array-oriented programs, i.e. programs whose main parts are dominated array primitives such as Black-Scholes and HotSpot.

5. Future Work. We plan to do further work on the *ecc* compiler in three directions: expand compiler coverage, implement automatic parallelization and injective compilation. For the compiler to be useful for all ELI users, the most urgent thing is to remove restrictions on *ecc* as much as possible. Some restrictions such as allowing new data types of complex numbers, symbols and temporal data can be done easily. Some restriction such as the exclusion of the execution function is inherently difficult to overcome. A more important restriction to remove is that of flat array only rule, i.e. the exclusion of lists since lists are crucial in handling non-regular data. To remove this restriction in general would involve introducing pointers at each list item, and hence complicated and time-consuming. We may be able to extend *ecc* to handle homogeneous list for non-rectangular data first as well as to implement the *each* operator on these lists. In short, we'll explore the extent we can accommodate lists of various properties.

The second direction of future compiler work is in a sense the main motivation for undertaking the ELI project, that is to implement automatic parallelization for compiled ELI programs to run on multi-core machines similar to what

has been done in APL [9]. We believe that for programs which can be expressed in data parallel fashion, i.e. can be written in an array-oriented fashion, the resulting compiled code would be competitive with hand-coded C programs. Of course, a C program can also be parallelized, but it is so much more work to hand parallelize a C program, data parallel or not. The real challenge in this front is to implement task parallelism as outlined, but has not been implemented in a real compiler.

The compiled code of an ELI program operates in a self-contained unit; it has its own stack and heap. The third direction of future compiler work is to compile a small piece of program to be callable from a large existing environment. We call this ***injective compiling***. For example, the library file *standard.esf* which is distributed with the ELI interpreter contains many frequently used functions written in ELI. It would be nice for an ELI program to call such a library function in compiled C instead of an ELI function which needs to be further interpreted. The difficulty of course is that library function operates in the environment of the interpreter. This also will provide a convenient way for an ELI program which on the whole is not compilable because it uses lists or the execute function to call a compiled sub-portion.

6. Conclusion. We briefly introduced the array programming language ELI, and a restricted compiler *ecc.esf* written in ELI, which translates programs written in the core part of ELI into C. We describe how we compiled *ecc.esf* into *ecc.c* and later into *ecc*. We presented performance data on three commonly used examples and on the usage of the compiler. Finally, we discussed future work on the compiler to improve the performance of compiled code.

Appendix

Table 4 ELI notations used in the three programs. Source code and compiled code at http://fastarray.appspot.com

Name	Symbol	Function
assign	`x<-value`	the value is assigned to x
power	`L*.R`	the base is L and the exponent is R
multiply	`L*R`	L multiple R
divide	`L%R`	L is divided by R
log	`L%.R`	logarithmic function
reshape	`L#R`	R is reshaped to multi-dimensions by L
not equal	`L~=R`	return Boolean, check whether L is not equal R
take	`L^.R`	take the first(last) L items of array R, if L>0(<0)
drop	`L!.R`	remove the first(last) L items of array R, if L>0 (<0)
reduction	`+/R`	reduction based on the last axis of R
and	`L^R`	and operation between L and R
negative	`_1`	negative 1

References

1. Allen, F., Cocke, J.: A program data analysis procedure, Comm. ACM 19(3), 137-147, 1976.
2. Bates, J.: Some observation on using Ching's APL-to-C translator, APL Quote Quad 25(3), 47-48, 1995.
3. Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications, PACT'08, Proc. 17th Int'l Conf. on Proc., Arch. and Compilation Tech. 2008.
4. Chen, H., Ching, W.-M., ELI: a simple system for array programming, Vector J. Br. APL Assoc. **26,**2013.
5. Chen, H., Ching, W.-M., Zheng D.: A comparison study on execution performance of MATLAB and APL, posted at http://fastarray.appspot.com/.
6. Ching, W.-M.: Program analysis and code generation in an APL/370 compiler, IBM J. Res. Dev. **30**, 594-605, 1986.
7. Ching, W.-M.:ELI Primer, with assistance of Chen,H., at http://fastarray.appspot.com/.

8. Ching, W.-M, Ju, D.: An APL-to-C compiler for the IBM RS/6000: compilation, performance and limitations, APL Quote Quad 23(3), 15-21, 1993.
9. Ching, W.-M., Zheng D.: Automatic parallelization of array-oriented programs for a multi-core machine, Int. J. Parallel Prog. 2012.
10. Iverson, K.:Turing award lecture: notation as a tool of thought, Comm. ACM 23(8), 444-465, 1980.
11. Int'l Organization for Standardization: ISO Draft Standard APL, APL Quote Quad 14(2),7-272, 1983.
12. The Math Works: MATLAB Compiler, 6th ed., 2002.
13. *ecc*:the ELI-to-C Compiler User's Guide, 2013, at http://fastarray.appspot.com/.
14. Che,S., Boyer,M., Meng,J., Tarjan,D., Sheaffer,J. W., Lee,S.-H., and Skadron,K.. Rodinia: A Benchmark Suite for Heterogeneous Computing. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct. 2009.
15. Che,S., Sheaffer,J. W., Boyer,M., Szafaryn,L. G., Wang,L., and Skadron,K.: A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In Proceedings of the IEEE International Symposium on Workload Characterization, Dec. 2010.
16. Tarjan, R. Testing Flow-Graph Reducibility, J. Comput. Syst. Sci. 9, 355-365, 1974.