Accelerating Information Experts through Compiler Design

Aaron W. Hsu
Indiana University, USA
awhsu@indiana.edu

Abstract

Dyalog APL is a tool of thought for information experts, enabling rapid development of domain-centric software without the costly software engineering feedback loop often required. The Dyalog APL interpreter introduces performance constraints that hinder the analysis of large data sets, especially on highly-parallel computing architectures. The Co-dfns compiler project aims to reduce the overheads involved in creating high-performance code in APL. It focuses on integrating with the APL environment and compiles a familiar subset of the language, delivering significant performance and platform independence to information experts without requiring code rewrites and conversion into other languages.

The design of the Co-dfns compiler, itself an APL program, possesses a unique architecture that permits implementation without branching, recursion, or other complex forms of control flow. By integrating specific optimizations, the generated code competes with hand-written C code in the domain of financial simulations, exceeding it when integrated into the environment. Preliminary results demonstrate platform independent performance across CPUs and GPUs without modification of the source. Work continues to improve performance both of the architecture and the generated code. Eventually, the project hopes to convincingly demonstrate a wider range of techniques that extend the suitable domain for effective array programming.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – code generation, compilers, memory management (garbage collection), optimization, run-time environments.

General Terms Algorithms, Performance, Design, Languages

Keywords APL, array programming, compiler architecture

1. Introduction

Dyalog APL [Dyalog 2015] is a modern interpreted language that follows traditional APL systems in providing a "tool of thought" [Iverson 2007] to information experts who develop complex software with minimal external software engineering resources. APL's concise, mathematical nature encourages array-oriented data exploration. APL programs often exceed performance expectations, outperforming more traditional systems by a large margin [Grosvenor 2013]. Nonetheless, the Dyalog interpreter imposes certain performance limitations.

The Co-dfns compiler project [Hsu 2014] provides a compiler integrated into the Dyalog environment to support turnkey performance for information experts without sacrificing the advantages of APL. The compiler leverages the high-level nature of APL to deliver platform independent performance of data parallel, and in the future, task parallel applications, scaling APL programs to larger data sets and a wider selection of hardware.

The Co-dfns project hopes to push the boundaries of "suitable" array programming domains. The compiler itself represents such a domain, since its implementation is a pure, data-parallel program in the dfns APL dialect. The simplicity and directness of implementation, lacking any complex encoding scheme, demonstrates the approach's effectiveness. Important benefits include increased parallelism and analysis opportunities arising from the intentionally restricted and simplified language subset used to construct the compiler.

The compiler accepts code in the dfns Dyalog APL syntax, which is lexically scoped and spiritually functional, rather than the traditional dynamically scoped format. Generated code integrates directly with the interpreter through the DWA [Dyalog 2014] infrastructure. Performance exceeds that of traditional hand-written C integrated using the foreign function interface alone, while also possessing platform independence, compiling to GPU and CPU targets unmodified, with significant performance improvements in both cases.

We make the following contributions:

- A demonstration of the feasibility of writing a purely data-parallel compiler without recursion, branching, or complex control flow
- Performance analysis of the costs of integrating foreign code into the Dyalog APL environment by comparing the performance of hand-written C code against the Co-dfns compiler, which integrates more completely with the host environment
- A compiler that delivers device independent performance gains in the range of 50 - 10,000% over the base interpreter by compiling unmodified APL to the CPU and GPU. These gains are possible in part from the high-level nature of the APL language
- An enumeration of the optimizations used to deliver increased performance on scalar heavy computational code when written in APL

2. Compiler Architecture

The Co-dfns compiler has a unique architecture. In line with project goals, "eating our own dog food" motivates the design. In other words, demonstrate the practicality of a data-parallel compiler written in a restricted subset of dfns APL. This motivates

ARRAY'15, June 13, 2015, Portland, OR, USA
ACM. 978-1-4503-3584-3/15/06
http://dx.doi.org/10.1145/2774959.2774968

```
r←0.02 ⋄ v←0.03 ⋄ coeff←0.31938153 ¯0.356563782 1.781477937 ¯1.821255978 1.33027442

CNDP2←{(1 ¯1)[⌊B]×((0 ¯1)[⌊B←ω≥0])+R←(÷(○2)*0.5)×(*(L×L)÷¯2)×{coeff+.×ω*1+⍳5}¨÷1+0.2316419×L←|
ω}

Run←{S←0⌷ω ⋄ X←1⌷ω ⋄ T←2⌷ω ⋄ vsqrtT←v×T*0.5
  D1←((⊛S÷X)+(r+(v*2)÷2)×T)÷vsqrtT ⋄ D2←D1-vsqrtT
  CD1←CNDP2 D1 ⋄ CD2←CNDP2 D2 ⋄ e←*(-r)×T
  ((S×CD1)-X×e×CD2),[0.5](X×e×1-CD2)-S×1-CD1
}
```

Figure 1. Black Scholes Benchmark

a different perspective on compiler construction. The core passes use function composition over a set of general purpose array operators and vectorized array primitives. There are no explicit branching statements, such as switches, if-statements, or visitor-style design patterns. The compiler uses no recursive or other means of complex control flow. Instead, all control flow is direct and simple function composition. All the basic operators are data-parallel or vectorized and have parallel semantics. (Note that the Co-dfns language includes recursion, branching, and traditional control flow; the compiler simply refrains from their use.) Only the parser utilizes more complex patterns. The parser uses recursive parser combinators, but evidence suggests that a suitable data-parallel approach to parsing APL exists [Bunda 1984].

The compiler does not yet self-host, so the interpreter hosts instead. The interpreter executes the primitives either sequentially or using vectorization on standard CPUs. The compiler generates more efficient vectorized and parallel code in C or CUDA for the same primitives as an alternative, particularly on parallel architectures like the Xeon Phi and the NVIDIA GPUs. The back-end relies on an existing C or CUDA compiler to generate an executable.

The resulting code exhibits a high degree of concision and uniformity. The core compiler passes are less than 150 lines including whitespace and comments in pure dfns. The concision gained from this dense programming practice arguably improves the hackability since the entire compiler can be easily laid out with a few printed pages of paper.

The compiler represents the AST as a matrix where each row is a node in the AST ordered by depth-first pre-order traversal. Columns represent attributes for each node. A depth attribute indicates the depth of the node in the AST and linearizes the parent child information normally stored as pointers. In order to maximize locality when processing nodes, a second attribute called a "node coordinate" or reference encodes the information stored in the depth attributes in such a way that given any two arbitrary nodes, the parent child relationship of the two is obvious without requiring any traversal of the matrix. This permits arbitrary computation over sub-trees selected by their parent-child relationships without requiring complex control flow.

The memory management design of the compiler greatly affects its performance in real applications. In line with the explicit goal to integrate with Dyalog APL and, particularly, to reduce the overheads involved for the information expert, the DWA (Direct Workspace Access) system of Dyalog APL links the memory manager of the interpreter to the compiler's generated code. This allows generated code to operate directly on native interpreter data, without an import/export phase. Initial versions of the compiler used a separate representation for arrays and handled memory management outside of the DWA infrastructure. Doing so liberated the compiler but imposed a performance penalty. The performance of earlier versions derived primarily from this choice of memory management, as discussed below.

3. Optimizations

The current compiler optimizations improve the performance of financial simulation software, epitomized in the Black Scholes benchmark. This code contains mostly numerical calculations over large arrays, and in particular, scalar computations, instead of heavy manipulations of the shapes of arrays. Four major optimizations significantly improve the performance of this benchmark.

3.1 Scalar Loop Fusion

In APL, primitive scalar operations, such as addition or subtraction, operate point-wise over every scalar element in an array argument. In other words, they implicity map or fork over each element in an array (possibly nested) to compute a sum or difference between individual scalar elements. When composed together, as in the following example, the interpreter must iterate over the entire array multiple times:

```
(C+A×B)÷D
```

If the arrays are large, then these multiple iterations can eliminate the benefits of cache. This results in poorer than expected performance, despite each individual primitive using optimized vector operations.

The compiler recognizes these loops and fuses them together. This is a well studied and difficult problem [Darte 2000] for languages like Fortran, which use things like polyhedral models to fuse arbitrary loops. The compiler uses a more naive fusion, simply by recognizing adjacent, data dependent scalar expressions such as above, and creating a single loop in the generated code rather than a distinct loop for each scalar expression as the interpreter must.

This optimization improves cache behavior and allows better vectorization when using lower level C and CUDA compilers. The generated loops ensure that they are easily vectorized and, if the user desires, automatically parallelized. Generally, parallelization of this sorts works well on the GPU, but tests show better performance with vectorization alone on normal CPUs like the Intel i7.

3.2 Function Inlining and Allocation

The compiler inlines all functions when possible. This is especially true for the operators which take functions as operands. By inlining these operations, it is often possible to reduce significant overheads involed in repeated iterations. Furthermore, because Dyalog APL does not permit functions to return functions as values, though they can be passed as operands to operators, there is no need for closures, as in most functional programming

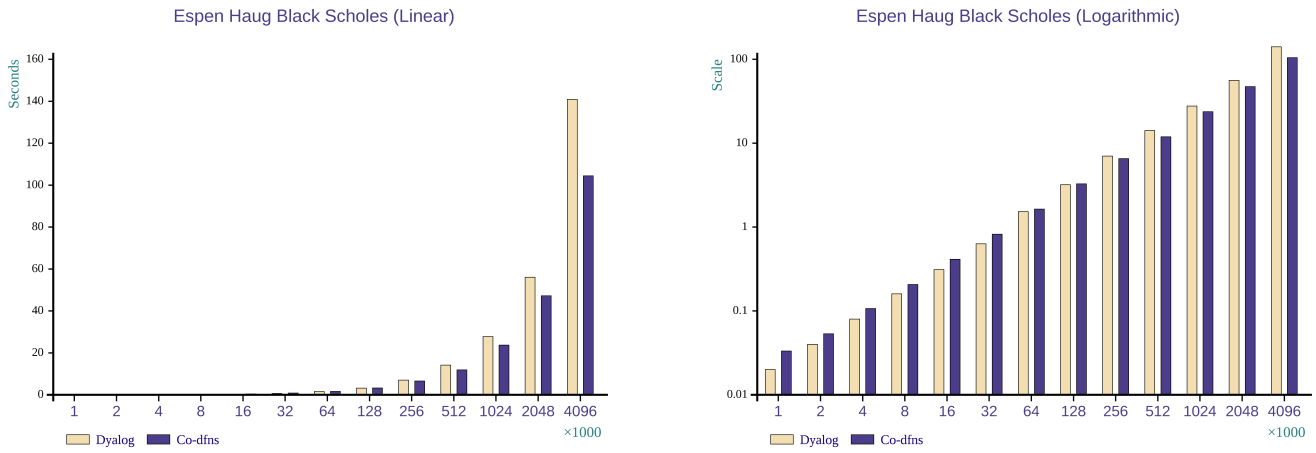

Figure 2. Performance of un-optimized Co-dfns compiler code in the Black Scholes benchmark on the CPU.

languages. The current version still creates some environments to hold some free variables, but the next version of the compiler will mostly eliminate additional environment overheads and almost completely eliminate function call overheads.

3.3 DWA as an Optimization

While the above optimizations are fairly classical, the most surprising and significant optimization for the current targeted benchmark turned out to be the integration of DWA (Direct Workspace Access) [Dyalog 2014] into the compiler. In this case, the compiler operates directly over structures inside of the interpreter, completely eliminating copying and conversion overheads associated with calling foreign code from within the interpreter. The benchmark assumes that information experts will conduct most of their development through the interpereter, and use the compiler as a later optimization stage or to deploy. By eliminating the overhead for calling into foreign functions, performance increased dramatically above and beyond other optimizations.

4. Performance

The following benchmark results come from an Intel Core i7-3610QM @ 2.30GHz with 8 Hyperthreaded cores. The machine contains 16GB of RAM and an NVIDIA GTX 675M graphics card. Results were obtained with the 64-bit Linux version of Dyalog APL.

4.1 Without Optimizations

The previous version of Co-dfns [Hsu 2014a] used its own internal representation for arrays that differed slightly in structure from that used by the Dyalog interpreter. It also did not include explicit inlining of primitives. It did use a reasonably efficient representation of functions, however. A subsequent version included the ability to target CUDA-capable hardware [Hsu 2014b].

The CPU based code executed with speedups of roughly 20 - 30% as seen in Figure 2. On the GPU, without some specific special implementations of certain parts of the code, the performance actually slightly degraded from the interpreter. However, with the appropriate primitives in place, the performance was the same as that of the CPU, around 20 - 30%.

In fact, through exploration of various implementation techniques and analyzing the code, the most an user could expect from the compiler on these sorts of ideal scalar computations with the then-current architecture was about a 30% improvement over the interpreter. Even if the computation happened almost instantly, the cost of transferring data in and out of the interpreter overwhelmed every other runtime cost. Data copy overhead was simply too great.

Unfortunately, there is little way in which to support generalized array operations in and out of the interpreter without incurring these overheads except for a drastic change in array representation and management.

4.2 With Optimizations

In the next version of the compiler, a new design used Dyalog's DWA (Direct Workspace Access) infrastructure, which allowed foreign code to operate over native interpreter structures. While somewhat more complicated then the normal foreign function interface, in this case, it provides significant benefits. By designing the compiler to operate over native Dyalog arrays instead of requiring a translation/conversion phase, the compiler completely eliminates the overheads involved in transferring data in and out of the interpreter. When combined with the current optimizations that are in the compiler, including scalar function loop fusion and primitive inlining, performance improves in the range of 50-90% on the CPU. Figure 3 includes the alternative of using a handwritten C version of Black Scholes with the standard FFI interface in the Dyalog interpreter to compare the performance of hand-writing the code instead of using the Co-dfns compiler.

4.3 Preliminary GPU Results

While no complete results yet exist for the GPU, initial tests demonstrate 100 - 200× improvements in runtime performance compared to the interpreter. This comes close to the expected ideal performance of the test graphics card with these problems; we are confident that ongoing work to include additional optimizations and the new DWA-based architecture will permit information experts operating on large scalar domains to maximize their GPU's capabilities through Dyalog APL and Co-dfns without requiring any changes in their program code.

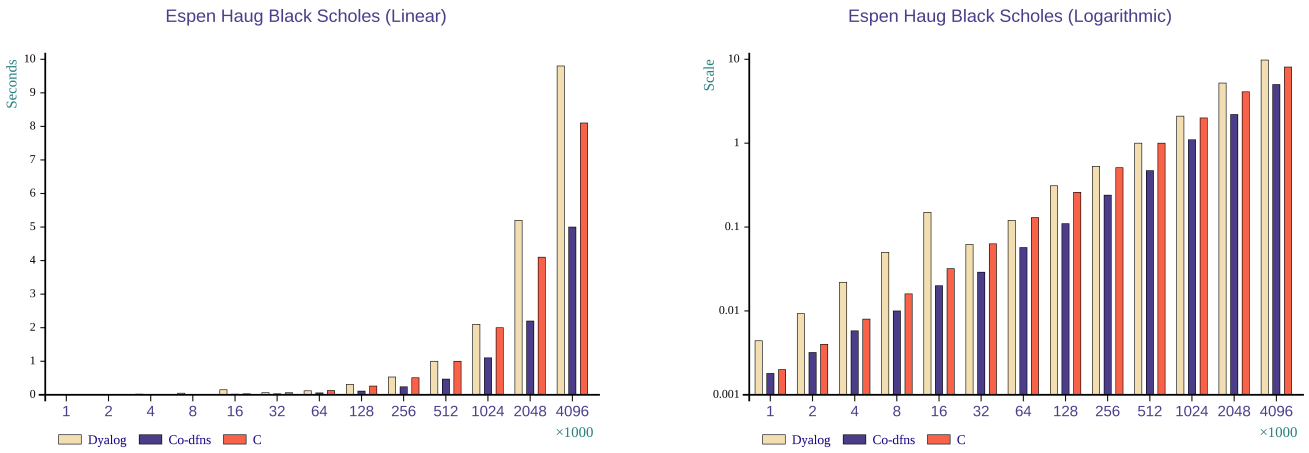

Figure 3. Performance of optimized Co-dfns code in the Black Scholes benchmark on the CPU.

4.4 Comparing to C

In Figure 3 the performance of the interpreter and the compiler appear in relation to hand written C code. The hand written C code simulates the process an information expert might use to optimize a particular piece of code that is going too slowly. That is, they would call out to a specific C library call to handle the inner loops, or they might write their own version in C or CUDA and then call into that code from the rest of the interpreter. The graphs show the dangers of doing so. While the handwritten C code by itself performs better than the interpreter, it is difficult to use such code inside of an inner loop or just in a single "hot spot" because the overheads of getting data in and out of the foreign functions is too high. Thus, while the performance of the compiled code and the C code are not that far apart, the C code performs much worse because it does not integrate with the interpreter's data structures.

An information expert could manually integrate with the interpreter using the same DWA API that Dyalog provides, but this greatly increases the complexity of the code and the overall interfaces, and makes it more difficult to use "off the shelf" tuned libraries. By using the compiler instead, this difficult work of hand-writing optimized code or linking into existing high-performance tuned libraries disappears and the compiler can be relied upon to produce code of similar performance, even using these tuned libraries underneath without any user intervention.

5. Ongoing Work

The Co-dfns compiler continues to evolve at a rapid pace, with significant components rewritten often as new techniques and solutions present themselves. The following projects give an idea of the roadmap for Co-dfns moving forward and especially for the near term improvements planned for the compiler.

We are working closely with industry partners through Dyalog, Ltd. to develop benchmarks and implementations of end-user code directly from the code bases of current Dyalog APL customers. This will allow us to not only examine Co-dfns from the perspective of traditional micro-benchmarks, but also give us a host of ready-to-use real life applications, which often diverge from micro-benchmarks in interesting ways.

We actively examine new programming methods and approaches for constructing the compiler. Specifically, the design of the current compiler closely aligns with a model of programming espoused by Moseley and Marks [2006] called Functional Relational Programming, though it is not strictly the same. We are investigating the benefits of more directly following this model to gain some "for free" optimizations in using the approach. Whether these benefits outweigh the increased and perhaps arbitrary abstraction we do not yet know.

Work by Lenore Mullin [1988] provides a firm base from which to develop a hybrid verification/type system on top of APL. A number of type systems have been created in industry [Shack-Nielsen 2014] and academia [Slepak 2014] to type APL code of various forms. Our audience for such a type system is the information experts themselves, coupled with the tuning expert, which presents interesting challenges in its design. We have preliminary models of such a type system, but no working implementation as yet.

The construction of programs in the branchless, recursion-less style of the compiler exposes a side-benefit that will be integrated into the compiler at a later date. Namely, with the simple control flow and well understood primitives, it is possible to automatically derive a worst-case performance model from the code without complex analysis. This allows us to obtain worst-case performance metrics about the runtime complexity of the compiler as well as memory usage automatically. We plan to integrate this feature into the compiler itself, so that the information expert can receive an automatic analysis of the runtime complexity of any program run through the compiler without requiring hand-working the solution.

We are in the process of integrating constant folding and constant propagation together with an optimization that lifts and minimizes runtime type checks to the entry of a function. These optimizations combine to greatly reduce the code size and type dispatch tables in the generated code. They have positive benefits for the automatic vectorization of loops by reducing the number of memory references and free variables inside of a loop.

The comparison against C does not take into account the performance of C against the performance of the compiled code without the transfer and management overheads. We intend to conduct more thorough benchmarking and performance analysis that takes these differences into account and provides a more clear picture into the performance of the compiler generated code.

6. Related Work

The APEX compiler [Bernecky 1997] developed vectorized approaches to handling certain analyses to compile traditional

APL, including a SIMD tokenizer [Bernecky 2003]. It uses a SSA representation, and converts the dynamic scope of traditional APL functions into a static form early on. It also uses a matrix format to represent the AST. Traditional APL did not have nested function definitions, however, and thus the APEX compiler does not have any specific approaches to dealing with function lifting. Bernecky suggests an approach for introducing parallelism into the compiler itself. While APEX is not self-hosting, its efforts to introduce parallel compiler passes should inform Co-dfns pass design. APEX does suffer from strong restrictions, though some could evaporate given enough effort. Both APEX and Co-dfns share some restrictions, such as not allowing the dynamic fixing of new functions at runtime. However, Co-dfns strives to ensure greater compatibility with dfns programs in Dyalog than analogous programs and APEX. Additionally, the design of APEX and its architecture differ from Co-dfns significantly.

Bernecky further identified methods of reducing or optimizing the computational complexity or cost of certain array operations, allowing improved performance of easy to understand array expressions [1999].

Timothy Budd implemented a compiler [1984; 1988] for APL which targeted vector processors as well as one for C code. They used a method of lazy evaluation to avoid intermediate data copying. Budd provided thoughts and some ideas on how the compiler might be implemented in parallel as well. Budd's compiler delayed computation until the point at which the program requested the value. As in most traditional APL compilers, the accepted language does not coincide with normal APL programs of the time. Budd's compiler also managed allocation without requiring a garbage collector.

Walter Schwarz implemented an APL to C compiler for the ACORN system targeting the CM-2 machine, demonstrating performance potential for APL as a massively parallel language [1991].

W. Ching and D. Ju have spent significant work on the ELI language and other APL-class language implementations, especially on parallelized code and optimization. [Ching 1990; 2000; Ching, et al. 1993; Ching and Katz 1994; Hendriks and Ching 1990; Ju and Ching 1991; Ju, et al. 1991]

J. D. Bunda and J. A. Gerth presented a method for doing table driven parsing of APL which suggested a parallel optimization for parsing, but did not elucidate the algorithm [1984].

Single-assignment C [Grelck and Scholz 2006] attempts to deliver a high-level, C-like language that uses arrays as first-class data types. It focuses heavily on a functional paradigm and automatically parallelizes code. The same issues of memory copying and array management occur in SAC as in Co-dfns, but SAC is a purely functional language, whereas Co-dfns admits array mutation and variable assignment within a single scope.

Dyalog has provided a good deal of input into Co-dfns, and so it makes sense that the current release of their interpreter begins to implement some of the ideas in Co-dfns. This includes a facility for doing coarse-grained parallelism with futures, but does not include any ability to do more refined concurrent operations since the interpreter lacks single assignment arrays for synchronization; the interpreter itself makes no guarantees when effects occur in parallel. [Kromberg and Foad 2013]

The McLAB project [Casey, et al. 2010] implements a compiler and supporting systems for MATLAB code. Like Co-dfns, it provides an explicit intermediate language for work, but goes to a lower level with its own JIT. It also produces Fortran and C code, which are not explicit targets of Co-dfns. The MATLAB language itself differs from Co-dfns, which naturally results in differences of approach with McLAB and Co-dfns. Co-dfns adapts the APL language with explicit parallelism constructs, rather than emphasizing automatic parallelization of the runtime primitives.

Languages like X10 [Charles, et al. 2005], Fortress [Steele 2006], and Chapel [Chamberlain, et al. 2007] also strive to scale programming to large distributed systems. They inherit much of their linguistic history from Fortran, Java, and C++, rather than APL. They also emphasize a more object-oriented approach than the array-centric, functional approach of Co-dfns. They expose much more explicit syntactic constructs for controlling data layout and synchronization, whereas Co-dfns tries to minimize explicit syntax as much as possible.

A number of systems such as ZPL [Lin and Snyder 1994] and Accelerate [Chakravarty, et al. 2011] are able to provide interesting implementation strategies for array programming, each emphasizing different elements. They all take the overall approach of altering the language design in favor of making certain features prevalent. Accelerate, for instance, lifts rank to the type level, meaning that shapes are no longer first class entities. ZPL uses a more traditional language but enables predictable data layout for distributed computing. Accelerate takes advantage of the language and implementation to make heavy use of fusion.

Eric Holk's Harlan system [Holk 2011] takes a unique approach. Targeting the GPU explicitly, it tries to introduce traditional programming concepts as native concepts on the GPU, so that traditional programs can run reasonably on the GPU. This approach, instead of lifting array programming to the general-purpose sphere, pulls general-purpose language constructs such as ADTs into the GPU and array world through the use of region inference and a number of other transformations. It also uses the nanopass style of compiler design and includes a macro system on top of it to reduce the number of core forms the compiler must consider.

7. Conclusion

The Co-dfns compiler enables information experts to convert their interpreted APL programs into efficient C and CUDA code that integrates into the Dyalog APL interpreter. This allows them to optimize hotspots in their code and to realize increased performance without rewriting key elements of their software. It also increases the range of hardware platforms available to the Dyalog APL programmer, as they are now able to execute their code on highly parallel processors like GPUs and the Xeon Phi. To achieve this performance on scalar-centric algorithms like Black Scholes, integration into the interpreter makes the largest single performance difference, eliminating interpreter overheads getting data in and out of the system. After this, scalar function loop fusion becomes critical to make effective use of the cache hierarchies on modern architectures.

In addition to producing fast code, the compiler is written as a simple, data-flow, data-parallel program without any explicit recursion, branching, or other forms of complex control-flow. The directness and concision afforded by this method demonstrates that the number of domains for which array-oriented programming is suitable is larger than commonly granted.

References

Robert Bernecky. 1997. *APEX: The APL parallel executor.* (1997).

Robert Bernecky. 1999. Reducing computational complexity with array predicates. *ACM SIGAPL APL Quote Quad* 29, no. 3 (1999): 39-43.

Robert Bernecky. 2003. An SPMD/SIMD parallel tokenizer for APL. In *Proceedings of the 2003 conference on APL: stretching the mind*, pp. 21-32. ACM, 2003.

Timothy A. Budd. 1984. An APL Compiler for a Vector Processor. *ACM Trans. Program. Lang. Syst.* 6, 3 (July 1984), 297-313. DOI=10.1145/579.357248 http://doi.acm.org/10.1145/579.357248

Timothy A. Budd. 1988. *An APL compiler*. NY: Springer-Verlag, 1988.

J. D. Bunda and J. A. Gerth. 1984. APL two by two-syntax analysis by pairwise reduction. *SIGAPL APL Quote Quad* 14, 4 (June 1984), 85-94. DOI=10.1145/384283.801081 http://doi.acm.org/10.1145/384283.801081

Andrew Casey, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, Rahul Garg, Soroush Radpour, Olivier Savary Belanger, Laurie Hendren, and Clark Verbrugge. 2010. McLab: An extensible compiler toolkit for MATLAB and related languages. In *C 3 S 2 E-10*. Montreal.

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP '11)*. ACM, New York, NY, USA, 3-14. DOI: http://doi.acm.org/10.1145/1926354.1926358

B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* vol. 21 no. 3. (August 2007). 291-312.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. ACM, New York, NY, USA, 519-538. DOI=10.1145/1094811.1094852 http://doi.acm.org/10.1145/1094811.1094852

Wai-Mee Ching. 1990. Automatic parallelization of APL-style programs. In *Conference Proceedings on APL 90: For the Future* (Copenhagen, Denmark, August 13 - 17, 1990). P. Gjerløv, Ed. APL '90. ACM, New York, NY, 76-80. DOI= http://doi.acm.org/10.1145/97808.97826

Wai-Mee Ching. 2000. The design and implementation of an APL dialect, ELI. In *Proceedings of the international Conference on Apl-Berlin-2000 Conference* (Berlin, Germany, July 24 - 27, 2000). APL '00. ACM, New York, NY, 69-76. DOI= http://doi.acm.org/10.1145/570475.570485

Wai-Mee Ching, Paul Carini, and Dz-Ching Ju. 1993. A primitive-based strategy for producing efficient code for very high level programs. *Comput. Lang*. 19, 1 (Jan. 1993), 41-50. DOI= http://dx.doi.org/10.1016/0096-0551(93)90038-3

Wai-Mee Ching and Alex Katz. 1994. An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (Washington, D.C., November 14 - 18, 1994). IEEE Computer Society Press, Los Alamitos, CA, 59-68.

Alain Darte. 2000. On the complexity of loop fusion. *Parallel Computing*, 26(9), 1175-1193. DOI=10.1016/S0167-8191(00)00034-X http://dx.doi.org/10.1016/S0167-8191(00)00034-X

Dyalog, Ltd. 2014. SA4: Introduction to Direct Workspace Access (DWA). (September 2014). In *Dyalog '14*. (September 2014). Retrieved March 23, 2015 from http://www.dyalog.com/user-meetings/dyalog14.htm

Dyalog, Ltd. 2015. *Dyalog: A Tool of Thought for Software Solutions*. (March 2015). Retrieved March 23, 2015 from http://www.dyalog.com

Clemens Grelck and Sven-Bodo Scholz. 2006. SAC — A Functional Array Language for Efficient Multi-threaded Execution. In *International Journal of Parallel Programming*, Vol. 34, No. 4, (August 2006). DOI: 0.1007/s10766-006-0018-x

Paul Grosvenor. 2013. COSMOS Performance Improvements. In *Dyalog '13*. (October 2013). Retrieved March 23, 2015 from http://video.dyalog.com/Dyalog13/?v=oK_XGobiFmM

Ferdinand Hendriks and Wai-Mee Ching. 1990. Sparse matrix technology tools in APL. In *Conference Proceedings on APL 90: For the Future* (Copenhagen, Denmark, August 13 - 17, 1990). P. Gjerløv, Ed. APL '90. ACM, New York, NY, 186-191. DOI= http://doi.acm.org/10.1145/97808.97844

Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willock, Arun Chauhan, Andrew Lumsdaine. 2011. Declarative Programming for GPUs. In *International Conference on Parallel Computing (ParCo 2011)*. (September 2011).

Aaron W. Hsu. 2014a. Co-dfns: Ancient Language, Modern Compiler. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, , Pages 62 , 6 pages. DOI=10.1145/2627373.2627384 http://doi.acm.org/10.1145/2627373.2627384

Aaron W. Hsu. 2014b. Co-dfns Report: Performance and Reliability Prototyping. In *Dyalog '14*. (September 2014). Retrieved March 23, 2015 from http://video.dyalog.com/Dyalog14/?v=8VPQmaJquB0

Kenneth E. Iverson. 2007. Notation as a tool of thought. In *ACM Turing award lectures*. ACM, New York, NY, USA. DOI=10.1145/1283920.1283935 http://dx.doi.org/10.1145/1283920.1283935

Dz-Ching Ju. and Wai-Mee Ching. 1991. Exploitation of APL data parallelism on a shared-memory MIMD machine. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Williamsburg, Virginia, USA, April 21 - 24, 1991). PPOPP '91. ACM, New York, NY, 61-72. DOI= http://doi.acm.org/10.1145/109625.109633

Dz-Ching Ju, Wai-Mee Ching, and Chuan-lin Wu. 1991. On performance and space usage improvements for parallelized compiled APL code. In *Proceedings of the international Conference on APL '91* (Palo Alto, California, USA, August 04 - 08, 1991). APL '91. ACM, New York, NY, 234-243. DOI= http://doi.acm.org/10.1145/114054.114080

Morten Kromberg and Jay Foad. 2013. Parallel Language Features in Version 14.0. Video. In *Dyalog '13*. (October 2013). Retrieved on March 23, 2015 from http://video.dyalog.com/Dyalog13/?v=Bmx_yUKxVv0

Calvin Lin and Lawrence Snyder. 1994. ZPL: An array sub-language. *Languages and Compilers for Parallel Computing Lecture Notes in Computer Science* Volume 768. 96-114.

Ben Moseley and Peter Marks. 2006. Out of the tar pit. In *SOFTWARE PRACTICE ADVANCEMENT (SPA)*.

Lenore Mullin. 1988. *A mathematics of array*s. Ph.D. Dissertation. School of Computer and Information Science, Syracuse University.

Walter Schwarz. 1991. Acorn Run-Time System for the CM-2. In *Arrays, Functional Languages, and Parallel Systems*, pp. 35-57. Springer US, 1991.

Anders Shack-Nielsen. 2014. *Parsing APL for Static Analysis*. (September 2014). Retrieved March 23, 2015 from http://video.dyalog.com/Dyalog14/?v=7Z0wzY9Oip4

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *Programming Languages and Systems* (pp. 27-46). Springer Berlin Heidelberg. DOI=10.1007/978-3-642-54833-8_3 http://dx.doi.org/10.1007/978-3-642-54833-8_3

Guy L. Steele Jr. 2006. Parallel Programming and Parallel Abstractions in Fortress. In *FLOPS*.