Science of Computer Programming 111 (2015) 23–50

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Extensible sparse functional arrays with circuit parallelism

John T. O'Donnell

School of Computing Science, University of Glasgow, Glasgow G12 8QQ, United Kingdom

ARTICLE INFO

Article history:
Received 11 December 2013
Received in revised form 23 November 2014
Accepted 18 December 2014
Available online 5 January 2015

Keywords:
Functional array
Sparse array
Extensible array
Functional programming
Circuit parallelism

ABSTRACT

A longstanding open question in algorithms and data structures is the time and space complexity of pure functional arrays. Imperative arrays provide update and lookup operations that require constant time in the Random Access Machine (RAM) theoretical model, but it is conjectured that there does not exist a RAM algorithm that achieves the same complexity for functional arrays, unless restrictions are placed on the operations. The main result of this paper is an algorithm that does achieve optimal unit time and space complexity for update and lookup on functional arrays. This algorithm does not run on a RAM, but instead it exploits the massive parallelism inherent in digital circuits. The algorithm also provides unit time operations that support storage management, as well as sparse and extensible arrays. The main idea behind the algorithm is to replace a RAM memory by a tree circuit that is more powerful than the RAM yet has the same asymptotic complexity in time (gate delays) and size (number of components). The algorithm uses an array representation that allows elements to be shared between many arrays with only a small constant factor penalty in space and time. This system exemplifies circuit parallelism, which exploits large numbers of transistors per chip in order to speed up key algorithms. Extensible Sparse Functional Arrays (ESFA) can be used with both functional and imperative programming languages. The system comprises a set of algorithms and a circuit specification, and it has been implemented on a GPGPU.

1. Introduction

A longstanding problem in algorithms and data structures is the complexity of operations on pure functional arrays. This question has both theoretical and practical significance, because arrays are fundamental to much software and the complexity of their operations affects the complexity of many algorithms.

An imperative array supports two operations: fetching an array element $a[i]$ and modifying an array element $a[i] := v$. Both operations require $O(1)$ time, according to common cost models, and they do not require any space beyond the memory originally allocated for the array. After an element of an imperative array is modified the previous content of that element is destroyed.

A pure functional program defines new values but does not perform side effects, such as modifying an existing value. Thus a pure functional array allows new arrays to be constructed but does not allow old ones to be changed. When a functional array is updated, the result is a new array that differs from the old array at one index, but the old array is still accessible. This paper generalizes functional arrays to handle sparse and extensible operations as well, and the data

E-mail address: john.odonnell@glasgow.ac.uk.

http://dx.doi.org/10.1016/j.scico.2014.12.005
0167-6423/

structure is called extensible sparse functional arrays (ESFA). Such an array maps indices to values, and is related, though not identical, to finite maps, hash tables and hash maps.

Imperative arrays are trivial to implement, as they rely on basic machine instructions and addressing modes. In contrast, straightforward implementations of functional arrays are inefficient in space or time, and the most efficient implementations are complex and still asymptotically slower than imperative arrays. Recopying an array being updated gives $O(1)$ access time but causes each update to take $O(n)$ space and time, where n is the array size. Algorithms that maintain balanced binary trees require $O(\log n)$ time for the operations.

Since unrestricted access to functional arrays is inefficient, there has been relatively little exploration of algorithms that rely on them. Nevertheless, functional arrays remain interesting in their own right, and they do have practical applications.

Functional arrays are not simply arrays used in a functional language. Imperative and functional arrays are distinct data structures that support different operations. Both data structures can be used in both imperative and functional languages (imperative array operations can be expressed in a pure functional language using monads or unique types). The choice between imperative and functional arrays should be based on the needs of the algorithm using them, not on the programming language used to express the algorithm.

This paper discusses the relationship between imperative and functional arrays, and conjectures that it is impossible to implement functional arrays with $O(1)$ access time using a Random Access Machine (a theoretical model of computation). However, the main result of the paper is a system that does indeed implement functional arrays with the same time and space complexity as imperative arrays, as long as the complexities are compared fairly using the same cost models. This result appears to contradict the conjecture—but the new algorithm runs on a different model of computation which we call *circuit parallelism*.

The essential idea is that the Random Access Machine model—as well as conventional computers—makes inefficient use of the digital logic components that make up the memory. The time (or space) complexity of a system depends on the time (or space) complexity of the underlying machine model as well as of the algorithm. A memory contains an address decoder that takes $O(\log N)$ time for a memory of size N, but the decoder performs no other useful computation. It is customary to calculate the time complexity of an algorithm by counting the number of Random Access Machine steps that it performs, while assuming that each RAM step takes unit time. The system presented here transfers some of the computation from the algorithm to the hardware, where it takes place alongside the address decoder *without increasing the time complexity (gate delay) or the size complexity (gate count) of the machine*. By redesigning the hardware as well as the software (called "hardware/software co-design") we can sometimes beat lower bounds on complexity of RAM algorithms.

Parts of the ESFA system were presented in 1993 [1]. This is an expanded version of a paper that appeared in PPDP 2013 [2], which extended the 1993 work by discussing the operations for sparse and extensible access, analyzing the algorithm and hardware complexity, and giving correctness proofs of key parts of the system. This paper defines both a pure functional interface and a stateful interface, introduces invariants on the representation, defines the implementation using parallel combinators, proves its correctness using equational reasoning, and gives extended examples.

Every one of the ESFA operations takes a small fixed number of clock cycles. No iteration is used in any of the operations; the execution time is constant, and does not depend on the past history of updates or deletions that led to the current state of the machine. No restrictions are placed on the operations that can be performed in order to achieve these tight time bounds. This constant time performance does not come at the cost of increased hardware complexity: the clock speed of the hardware (as a function of the gate delay) has the same time complexity as the clock for an ordinary addressable memory, and the hardware complexity in terms of number of logic gates and flip flops is also the same.

The algorithms presented here use *circuit parallelism*. This approach originated in associative processors [3] and active data structures [4]; other examples include priority queues [5], systems with chunks of memory organized as trees [6], smart memories for multicore processors [7], and associative searching [8]. Circuit parallelism is also the target platform for compilation of a declarative committed-choice rule language [9]. The idea is to bring the parallelism inherent in digital circuits to bear directly on the computations required by an algorithm, rather than organizing the circuit into conventional processors. In circuit parallelism, the computation is melded into the memory at the level of individual words, allowing algorithms that perform a computation in parallel on every word in the memory; it differs from data parallelism, where an operation performs a computation on every word of a data structure (rather than every word in the machine). Current hardware trends will make this approach increasingly productive, as the number of transistors per chip continues to increase.

The algorithms presented here are fine-grain and massively parallel, and they require suitable hardware in order to be usable. They do not run efficiently on a sequential computer that lacks a hardware accelerator. The fastest platform for ESFA is a direct VLSI implementation of the underlying parallel circuit, but an FPGA or GPU chip could also be used (see Section 8).

The ESFA system has been implemented and tested in several ways. First, this paper contains an executable specification written in Haskell. Second, it is implemented using a digital circuit that is specified and simulated using the Hydra hardware description language [10]. The design has not been fabricated as a physical chip, but the Hydra specification is precise down to the level of flip flops and logic gates (it is "synthesizable"), and the simulation is accurate in clock cycles and also in gate delays within a cycle. Third, the system is implemented as a program, written in C and CUDA [11], that runs on a general purpose GPU [12]. The GPU implementation gives reasonable performance, and there is extremely low variation in execution