Active Data Structures on GPGPUs

John T. O'Donnell, Cordelia Hall, and Stuart Monro

School of Computing Science, University of Glasgow,
Glasgow, United Kingdom
http://www.dcs.gla.ac.uk/~jtod/

Abstract. Active data structures support operations that may affect a large number of elements of an aggregate data structure. They are well suited for extremely fine grain parallel systems, including circuit parallelism. General purpose GPUs were designed to support regular graphics algorithms, but their intermediate level of granularity makes them potentially viable also for active data structures. We consider the characteristics of active data structures and discuss the feasibility of implementing them on GPGPUs. We describe the GPU implementations of two such data structures, extensible sparse functional (ESF) arrays and index intervals. We measure their performance and discuss the potential of active data structures as an unconventional programming model that can exploit the capabilities of emerging fine grain architectures such as GPUs.

Keywords: active data structure, data parallel, circuit parallel, GPU, GPGPU, persistent kernel, index interval, functional array, selection.

1 Introduction

GPUs originated as parallel graphics processing units, but they have evolved into general purpose parallel processors, sometimes called GPGPUs. A GPU contains a moderately large number ($\sim 10^3$) of processing elements that execute multiple threads concurrently. They have been applied successfully to a wide range of scientific applications, which consist largely of data parallel iterations over arrays.

It is important to discover how widely applicable GPUs are because they fit well with current trends in computer architecture, and they provide a good level of granularity. The density of transistors per chip continues to grow, but individual processors are no longer becoming faster. Therefore computer architects are seeking effective ways to use additional transistors to support parallel computing.

A current trend is to reduce the granularity of computation. The granularity of a parallel system relates the size and number of processing elements. Multicore systems have on the order of 10 to 100 processors and memories. GPGPUs have a finer granularity, with a larger number of processing elements (thousands) that are smaller and less powerful than CPUs [5]. Very fine grain SIMD parallel systems, such as the Connection Machine [8], have large numbers of very small

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 875–884, 2014.

processors. FPGAs (field programmable gate arrays) and CGRA (coarse grain reconfigurable architectures) allow even finer grain parallelism, but they are less portable and harder to program than GPUs. In his Turing Award lecture, Backus argues that finer grain parallel programming models can reduce the bottleneck between a processor and a memory [2].

Applications for GPUs are often organised as a collection of tasks coordinated by the CPU. A task that runs on the GPU is called a kernel, consisting of many concurrent threads organised into several blocks that run on parallel processing elements in the GPU. When all threads have completed, the kernel terminates. A full application will typically perform many kernel launches and terminations. This programming model supports regular parallelism across large and regular data structures.

There is increasing interest in investigating the applicability of GPUs to a broader range of applications. A general discussion of the limitations of GPUs and how to overcome them is given in [7]. One approach is to provide MIMD computations on a GPU, which is essentially SIMD, using interpretation [4]. A cost model for GPU computation is presented in [17].

This paper investigates the application of GPUs to *active data structures* [1], which cause a single operation to affect many elements of a data structure. Examples of active data structures include associative memories and content addressable parallel processors [6]. Active data structures are well suited for "circuit parallelism", such as FPGA or VLSI implementation. These platforms are hard to use and nonportable, so it would be valuable to provide GPU implementations of active data structures. However, there are many challenges in doing so.

The contribution of this paper is an experimental investigation of the suitability of GPUs for active data structures that exploit fine grain parallel computation. We discuss the characteristics of active data structures and the technical challenges in implementing them on a GPU. We then describe the implementation and performance of two experimental case studies: index interval selection and extensible sparse functional arrays. The ESF array algorithm provides operations that have extremely low variance in execution time, making them especially useful for real time applications. To help make the programming techniques more widely available, and to enable others to replicate our experiments, the source code and further documentation are available on the web [13].

Section 2 introduces active data structures, and Section 3 discusses the architectural capabilities needed to implement them. Section 4 describes several active data structures and their experimental implementation on a GPU, and gives performance results. Section 5 concludes.

2 Active Data Structures

An active data structure provides a set of high level operations, where each operation may cause changes to many elements within the structure. This could be simply a software abstraction layer, but it is also possible to use parallelism to implement the operations.

The earliest active data structures were associative memories [6]. A conventional memory identifies each memory element by an address. The fetch and store operations supply an address, and the memory hardware uses a tree of multiplexers or demultiplexers to access the data. Associative memory (also called content-addressable memory) stores words that contain several fields: for example each word might contain (x, y) where the components x and y are bit fields. A memory access instruction provides the value of one of the bit fields (say $x = 147$), and the hardware returns the corresponding value of the other field.

Associative memory can be implemented efficiently with *circuit parallelism*. The memory contains a set of registers, each holding all the bit fields of an element. A dedicated comparison circuit is attached to every register in the memory. When an access instruction provides a value of the x field, this is broadcast ("fanned out") to all the comparison circuits, which determine whether the corresponding element matches. These comparisons are performed in parallel; the time to search n items is typically reduced by a factor of at least $O(\log n)$. Associative memory can reduce communication costs for data intensive applications, with organisation of the data into suitable chunks [10].

The basic idea behind associative memory—to add some logic to each word in a memory, and also to add some logic to the address decoder tree—can be extended further, to powerful "smart memories" that mix memory and computation at a very fine grain. Many applications take this approach, including hardware cache and translation lookaside buffers, database searches, and signal processing. The most efficient platform for such a system is a specialised VLSI design, but the design cost is high and the result is inflexible. It would be valuable to adapt GPUs to active data structures, because GPUs are more cost effective, more portable, and more flexible than custom chips.

3 GPU Capabilities Required for Active Data Structures

GPUs were originally intended to accelerate graphics rendering, a specialised class of algorithm. Because of their high performance and relatively low power consumption, they have been applied to an increasingly wide range of applications, especially in scientific programming.

This does not mean, however, that GPUs are well suited for all kinds of programs. They have several properties that can potentially limit their use for some applications. GPUs offer better support for data parallelism than task parallelism. They have a complex memory model, with limitations in sharing of data and widely varying latency. Synchronisation among threads requires careful programming and can introduce significant overheads.

Many GPU applications are organised as a set of tasks that are coordinated by the CPU. Each task is a function (called a kernel) that is called by the CPU and that runs on the GPU. When all threads in the kernel finish, the kernel terminates and the CPU resumes its computation. Communication between the CPU and the GPU takes place through global memory, which is shared by both systems.

The computation in a GPU takes place in a set of blocks; each block can run a set of threads concurrently. Blocks use different processing elements, so threads in different blocks run in parallel, but threads in the same block may be implemented by receiving very short time slices (one clock cycle) on a single processing element. This hides the latency of memory accesses: if a thread has requested a memory fetch, it is more efficient to allow other threads to execute for a clock cycle rather than stalling the entire system until the data has arrived from the memory. Thus threads can provide parallelism *in effect* even though they may not be truly parallel at the level of the processor clock.

GPUs have a complex memory model, with four or five different kinds of memory. Global memory is accessible to all threads in all blocks, and also to the CPU, but it has high latency. Shared memory is accessible to all the threads within the same block, but not to other blocks or the CPU. Global memory is more flexible but its latency is two orders of magnitude higher than shared memory. Other types of GPU memory, such as constant and texture memory, are not used in our algorithms.

Communication across blocks. The overhead of communication between threads in different blocks is high, for two reasons: (1) threads across blocks must communicate via the slow global memory rather than shared memory; and (2) it is more costly to implement a barrier synchronisation across blocks than across threads within a block. To achieve fast execution, it is necessary to minimise the use of costly communications across blocks, and to do this efficiently where it is necessary.

Persistent shared memory. For efficient execution, it is important to keep active data structures in shared memory. There are two problems with this: (1) threads in different blocks cannot access each other's shared memory, so they can communicate only using global memory, and (2) when the kernel (the function running on the GPU) returns, the contents of shared memory is lost. When inter-block communication is needed, the threads can write data out to global memory, requiring synchronisation across blocks. To achieve persistence, it is necessary to organise the kernel as a long running loop that performs many active data structure operations. In effect, the kernel runs indefinitely. This prevents use of the common program organisation of a sequence of kernel calls. Instead, it is necessary to treat the GPU as a server, and to introduce a concurrent dialogue between the CPU and the GPU (see below).

Synchronisation across blocks. The GPU system provides a primitive barrier synchronisation for threads within a block, but not for threads in different blocks. Active data structures often require multiple blocks, either to gain access to more memory (there is a strict limit on fast shared memory for all the threads in a block) or to increase parallelism.

A common approach for applications that need multiple blocks is to organise the algorithm into sections; each section is initiated by the CPU with a kernel call and runs independently in the blocks. When all the threads have terminated,

the kernel terminates, returning control to the CPU. This effectively provides a cross-block barrier synchronisation, and also allows the CPU to assist with communication and any sequential computations that are required.

That common approach is not workable for many active data structure algorithms: it prevents the use of persistent shared memory, it increases overhead, and it introduces a sequential bottleneck. An alternative, which we use in the ESFA system described below, is to use a barrier that operates across blocks [16].

Dialogue between CPU and GPU. Many GPU applications are organised as a sequence of calls from the CPU to a kernel running on the GPU. Data is communicated between the CPU and GPU using global memory and functional arguments and results. If persistent shared memory is required, however, then the GPU needs to run a long-term server: the kernel is effectively an infinite loop that terminates only when the server is shut down. We implement the interaction by running the CPU and the kernel in parallel, with a concurrent protocol using locks to coordinate the CPU and the GPU. First the CPU starts the kernel, which initialises its state (kept in persistent shared memory). The CPU uses a lock to signal that it has a request; the GPU waits on this lock, copies the request, and signals that it has the data. A similar protocol is used by the GPU to tell the CPU that it has a result.

4 Case Studies

We have implemented two active data structure systems on a GPGPU, in order to assess the capabilities of the system and the performance, which are discussed in the following subsections. The system used is running CUDA version 4 on an NVidia GeForce GTX 590, with 512 CUDA cores (16 multiprocessors with 32 cores per mp), and a 1.22 GHz clock speed, with CUDA capability 2.0.

4.1 Index Intervals

Data structures based on *index intervals* [11] support a family of operations on ordered data structures, including selection and sorting algorithms. The algorithms on index intervals are related to quicksort. The idea is to represent explicitly what is known about the location of a value x within the sorted array, rather than using the physical memory address of the value to represent its ordering. Associative searches, rather than indexed addresses, are used to retrieve the data.

Consider an unsorted array $x_0, x_1, \ldots, x_{n-1}$. For any data value x_i, we represent the information that is known about its position in the (unknown) sorted array as a pair (lo, hi), such that $lo \leq j \leq hi$, where j is the position of x_i in the fully sorted array. The (lo, hi) pairs state explicitly the partial information that is known at any time about the data values. Initially nothing is known, so every value has an index interval of $(0, n-1)$, but if the array is fully sorted, then the element x_j will have index interval (j, j).

An operation *improve* takes an index interval, which defines a set Y of elements that have this interval, and refines the interval for every member of Y. This is done in parallel and takes a small constant number of steps, where each step operates in parallel on every value. First an element of Y, called the splitter, is selected. This is broadcast (in parallel) to every element. The elements are then compared (in parallel) with the splitter, and a local flag is set indicating the result. The numbers of elements that are less than (and also equal to) the splitter are counted (using a parallel fold), and this enables each cell to refine its index interval (using a parallel local computation). The entire *improve* operation consists of two parallel folds and two parallel maps.

As shown in [11], index interval sorting has a slightly lower computational complexity than conventional quicksort. Its larger significance, however, is the ability to use the *improve* step just where needed.

4.2 GPU Implementation of Index Intervals

Each array element is stored in one "cell", which is in effect a record of fields operated on by one thread. Parallel fold is used to locate an imprecise index interval, and a parallel map causes every cell to refine its index intervals. Table 1 shows the times for the local computations required by the *improve* algorithm. This includes the comparisons and interval adjustments in the cells, but not the parallel folds.

Table 1. Performance of *improve* step for refining index intervals. N is the number of cells, T is the mean time in microseconds over 1000 measurements, and stdev is the standard deviation.

N	T	stdev
1024	27.5	0.932189
2048	42.2	1.187385
4096	66.3	3.829351
8192	107.2	1.995101
10240	125.9	4.736220

4.3 ESF Arrays

Extensible sparse functional arrays (ESFA) [12] [14] are a complex data structure with demanding computational requirements. It is possible to use imperative arrays in a functional language [9] [3], and there are many algorithms for purely functional data structures [15]. Imperative arrays provide access time of O(1), and it is conjectured that this is impossible for functional arrays on a von Neumann architecture. Nevertheless, ESFA achieves O(1) access time for functional arrays (and several generalisations, including sparse and extensible arrays) using circuit parallelism. Thus the ESFA algorithm is inherently massively parallel.

The algorithm is implemented as a layer of software that runs on a digital circuit that implements a "smart memory". A program can build a collection

of ESFA using the *update* operation, which takes an array, index, and value, and creates a new array which is identical to the old one except that it has the specified value at the index. The essential point is that the update is purely functional: update creates a new array but the old one is still available.

It is straightforward to implement functional arrays by building tree structures in the heap, but this means that all operations require a number of steps which is proportional to the tree height. In the best case this gives logarithmic time, but many practical cases have array lookup time that is linear in the size of the array. It is possible to rebalance the trees periodically, but that adds additional significant overheads.

The ESFA system implements every operation in a small constant number of steps. Each step runs on a digital circuit (which can be real or virtual) that has the same topology as a conventional random access memory (RAM). The circuit provides a set of registers for each memory cell, and a tree of logic gates providing access to the cells from a controller. In a conventional RAM chip, the tree circuit is a word multiplexer, while in the ESFA system the tree nodes perform parallel fold calculations. Furthermore, each leaf cell in the ESFA circuit contains a small amount of logic circuitry, enabling every cell to perform a simple calculation on every clock cycle. Typical leaf cell calculations are to compare two natural numbers and set a flag with the result, and to increment a natural number if a flag is True.

ESFA is a powerful application of active data structures. It makes a challenging test case for GPU implementation because each ESFA operation performs a small calculation (a comparison and increment) in every cell in the machine, not just every cell in the data structure referenced by the operation.

4.4 GPU Implementation of ESF Arrays

The GPU implementation requires all of the techniques discussed above in Section 3. We used the lock-free algorithm of Xiao and Feng [16] for synchronising across blocks. We used our own parallel fold algorithm, which is more general than many published algorithms and also makes efficient use of the shared and global memories. We also developed a concurrent algorithm for controlling communication and synchronisation between the CPU and the GPU. The kernel needs to be organised carefully, especially to avoid race conditions and to minimise the scope of conditionals. The program is available on the web [13].

Table 2 shows the performance of the algorithm, as a function of the number of ESFA cells. A number of observations can be made from this data. The total execution time (the T column) grows slowly as a function of the log of the number of cells. The fastest execution occurs with the smallest ESFA memory size, but at the largest memory size (larger by a factor of 512) the execution time is only three times higher. The execution is most efficient at the largest size. Furthermore, the measurements are highly repeatable with low variance.

Table 3 compares the performance of the parallel GPU implementation of ESFA with a sequential simulator. For small data, the overhead of parallelisation makes the GPU slower than the sequential CPU. However, the sequential time

Table 2. ESFA performance on GPU. Each line shows the configuration (k = tree depth, $b \times t$ = blocks × threads, N = number of cells) and the total time T in milliseconds for a run consisting of 50,000 operations (including a mix of updates, lookups, and deletes). For each configuration, the execution was repeated 25 times, and the result of every operation was compared with the result calculated by an executable specification. Thus $10 \times 25 \times 50,000 = 12,500,000$ major operations were performed without error. stdv = standard deviation of T; d = max-min of T. The t column gives the time per ESFA operation in microseconds, and the t_c column gives the time in microseconds per operation per cell.

k	$b \times t$	N	T(ms)	stdv	d	$t(\mu s/op)$	$t_c(\mu s/op/c)$
4	4×4	16	5,071.1	32.3	104.0	101.4	6.34
5	4×8	32	5,374.3	30.9	99.9	107.5	3.36
6	4×16	64	5,606.1	25.9	117.1	112.1	1.75
7	8×16	128	6,447.8	26.0	123.7	128.9	1.01
8	16×16	256	7,946.3	32.7	122.8	158.9	0.62
9	16×32	512	7,994.0	14.0	65.7	159.9	0.31
10	16×64	1,024	8,426.2	25.1	101.0	168.5	0.16
11	32×64	2,048	12,144.0	35.9	125.0	242.9	0.12
12	32×128	4,096	13,424.4	33.5	106.5	268.5	0.06
13	32×256	8,192	15,037.2	44.7	183.1	300.7	0.03

Table 3. Parallel speedup, comparing execution time for performing 50,000 operations on the sequential simulator and on the parallel GPU implementation of ESFA. The CPU times were measured one time, the GPU times (taken from Table 2) are the mean of 25 runs. The speedup is $T_{\text{cpu}}/T_{\text{gpu}}$; this is the speedup as a function of the number of ESFA cells, *not* as a function of parallel processors in the GPU.

k	T_{cpu}	T_{gpu}	Speedup
4	1,232 ms	5,071.1 ms	0.243
5	2,067 ms	5,374.3 ms	0.385
6	4,046 ms	5,606.1 ms	0.722
7	9,426 ms	6,447.8 ms	1.462
8	24,765 ms	7,946.3 ms	3.117
9	65,193 ms	7,994.0 ms	8.155
10	176,428 ms	8,426.2 ms	20.938
11	534,878 ms	12,144.0 ms	44.045
12	1,866,045 ms	13,424.4 ms	139.004
13	7,228,694 ms	15,037.2 ms	480.721

grows linearly with the size of the data structure because each operation acts on every element, while the GPU time grows slowly.

These results demonstrate how effective GPUs can be for active data structures. Consider an ESFA machine with 8K cells. An efficient sequential algorithm (*not* a simulator) using a tree structure may require an average of 13 RAM accesses to perform an ESFA operation, with additional overhead for loop control.

With fast memory hardware, this would amount to around 0.1 to 1 microsecond. However, if an array is long—say 5000 elements—and the tree is not balanced, the access would require 50 to 500 microseconds. For comparison, ESFA running on the GPU requires 300 microseconds. It is, of course, possible to rebalance the trees periodically, but that introduces further overhead, requires more storage, and increases the variance in execution time. The ESFA algorithm *always* requires a time of 300 microseconds, without any extra overheads, making it valuable for real time applications where each operation must be completed within a deadline.

5 Conclusion

Active data structure algorithms scale well to very large numbers of processors, but they have several characteristics that make them challenging for implementation on GPUs. Key issues include communication and synchronisation across blocks, persistent shared memory, and interaction protocols between the CPU and GPU. We have identified algorithms that solve those key issues. One is a lock-free block synchronisation algorithm by Xiao and Feng [16], and we have developed others (a CPU/GPU dialogue, and efficient and general fold and scan algorithms).

Using these algorithms, we have evaluated the effectiveness of a GPU for implementing active data structures, using two case studies: a relatively straightforward index interval selection/sorting algorithm, and a very complex extensible sparse functional array algorithm. The speedup results compared with sequential versions of the active data structures are excellent, with a speedup factor of 480 on the ESFA algorithm.

There may be sequential algorithms using conventional data structures that are faster on a sequential machine. The key question is: can GPUs provide good enough performance for active data structures to compete well against the best sequential algorithm? The results are encouraging. ESFA on a GPU is sometimes slower than a sequential tree implementation of functional arrays, but in some realistic cases it is significantly faster, unless extra time is taken to rebalance the trees sequentially. Furthermore, the GPU ESFA algorithm has time complexity of $O(1)$ in the worst case as well as the average case. This makes it well suited for real time applications, where predictability, low variance, and a good worst-case time are more important than raw speed. Future research includes investigating active data structures on other platforms with massive fine grain parallelism, including coarse grain reconfigurable architectures (CGRA) and FPGAs.

References

1. Andrews, G.R., Dobkin, D.P.: Active data structures. In: ICSE 1981: Proc. 5th Int. Conf. on Software Engineering, pp. 354–362. IEEE Press (1981)
2. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8), 613–641 (1978), doi:10.1145/359576.359579

3. Chakravarty, M.T., Keller, G., Lee, S., McDonnell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Declarative Aspects of Multicore Programming. ACM (January 2011)
4. Dietz, H.G., Dalton Young, B.: MIMD Interpretation on a GPU. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 65–79. Springer, Heidelberg (2010)
5. Owens, J.D., et al.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
6. Foster, C.: Content Addressable Parallel Processors. Cengage Learning EMEA (1976) ISBN-13: 978-0442224332
7. Gaster, B.R., Howes, L.: Can GPU programming be liberated from the data-parallel bottleneck? Computer 45(8), 42–52 (2012), doi:10.1109/MC.2012.257
8. Daniel Hillis, W.: The Connection Machine. MIT Press (1985) ISBN 978-0262081573
9. Peyton Jones, S.L., Wadler, P.L.: Imperative functional programming. In: Proc. 20th ACM Symposium on Principles of Programming Languages (POPL), pp. 71–84 (1993)
10. Nath, P., Urgaonkar, B., Sivasubramaniam, A.: Evaluating the usefulness of content-addressable storage for high-performance data-intensive applications. In: Proceedings of the ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC 2008), pp. 35–44. ACM (2008)
11. O'Donnell, J.T.: Functional microprogramming for a data parallel architecture. In: Proceedings of the 1988 Glasgow Workshop on Functional Programming, pp. 124–145. Computing Science Department, University of Glasgow (1988)
12. O'Donnell, J.T.: Data parallel implementation of Extensible Sparse Functional Arrays. In: Reeve, M., Bode, A., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694, pp. 68–79. Springer, Heidelberg (1993)
13. O'Donnell, J.T.: GPU Programming. School of Computing Science, University of Glasgow (2012), `http://www.dcs.gla.ac.uk/~jtod/research/gpu/`
14. O'Donnell, J.T.: Extensible sparse functional arrays with circuit parallelism. In: 15th International Symposium on Principles and Practice of Declarative Programming. ACM (September 2013)
15. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
16. Xiao, S., Feng, W.C.: Inter-block GPU communication via fast barrier synchronization. In: IPDPS, pp. 1–12. IEEE (2010)
17. Zhang, Y., Owens, J.D.: A quantitative performance analysis model for GPU architectures. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 382–393 (2011)