

Theoretical Model of Computation and Algorithms for FPGA-based Hardware Accelerators[★]

Martin Hora¹, Václav Končický², and Jakub Tětek²

¹ Computer Science Institute, Charles University,
Malostranské nám. 25, CZ – 11800 Prague, Czech Republic
`mhora@iuuk.mff.cuni.cz`

² Department of Applied Mathematics, Charles University,
Malostranské nám. 25, CZ – 11800 Prague, Czech Republic
`{koncicky,jtetek}@kam.mff.cuni.cz`

Abstract. While FPGAs have been used extensively as hardware accelerators in industrial computation [20], no theoretical model of computation has been devised for the study of FPGA-based accelerators. In this paper, we present a theoretical model of computation on a system with conventional CPU and an FPGA, based on word-RAM. We show several algorithms in this model which are asymptotically faster than their word-RAM counterparts. Specifically, we show an algorithm for sorting, evaluation of associative operation and general techniques for speeding up some recursive algorithms and some dynamic programs. We also derive lower bounds on the running times needed to solve some problems.

1 Introduction

While Moore’s law has dictated the increase in processing power of processors for several decades, the current technology is hitting its limits and the single core performance is largely staying the same. New paradigm is needed to speed up computation. Many alternative architectures have been made, such as multicore, manycore (specifically GPU) and FPGA, which is the one we consider in this paper. FPGA is a special hardware for efficient execution of boolean circuits. While there exists a theoretical model which allows theoretical treatment of algorithms in the multicore and manycore model [17], no such model has been devised for computation with FPGA-based accelerators. In this paper, we consider heterogeneous systems which have a conventional CPU and an FPGA. This is especially relevant since Intel has recently introduced combined CPU with an FPGA [13], which should lower the latency between CPU and FPGA. We define a theoretical model for such systems and show several asymptotically fast algorithms on this model solving some fundamental problems.

1.1 Previous work

While there is no theory of FPGA algorithms, there are experimental results which show that FPGA can be used to speed up the computation by one to two orders of magnitude and decrease the power consumption by up to three orders of magnitude. Chrysos et al. [6] and Mahram [18] show how FPGA can be used to speed up computations in bioinformatics. FPGAs have also been used to speed up genetic algorithms, as shown by Alam [2]. In the following paper, the authors discuss what causes the performance gains of using an FPGA [10]. Che et al. [4] and Grozea et al. [9] showed a comparison with GPU and multicore CPU. Several algorithms have been implemented on FPGA and benchmarked, including matrix inversion [16], AES [5] and k-means [14].

[★] This work was carried out while the authors were participants in the DIMATIA-DIMACS REU exchange program at Rutgers University.

The work was supported by the grant SVV-2017-260452.

1.2 Our contributions

The main contribution of this paper is the definition of a model of computation which captures computation with FPGA-based accelerators (Section 3). In this model, we speed up classical algorithms for some fundamental problems, specifically sorting (Section 4.2), evaluation of associative operation (Section 4.3), speed up some dynamic programming problems including longest common subsequence and edit distance (Section 4.4) and speed up some recursive algorithms including Strassen algorithm, Karatsuba algorithm as well as many state-of-the-art algorithms for solving NP-hard problems (Section 4.5). In Section 5 we show some lower bounds which follow from a relation to cache-aware algorithms.

2 Preliminaries

2.1 Boolean Circuits

Definition 1 (Boolean circuit). *A boolean circuit with n inputs and m outputs is a labelled directed acyclic graph. It has n vertices $x_1, \dots, x_n$ with no incoming edges, which we call the input nodes, and m vertices $y_1, \dots, y_m$ with no outgoing edges, which we call output nodes. We call the vertices $x_1, \dots, x_n, y_1, \dots, y_m$ the I/O nodes. Each non-input vertex in this graph corresponds to a boolean function from the set $\{\vee, \wedge, \neg, \text{id}\}$ and is called gate. All gate vertices have indegree equal to the arity of their boolean function.*

The circuit computes a function $\{0, 1\}^n \rightarrow \{0, 1\}^m$ by evaluating this graph. At the beginning of computation, an input node x_i is set to the i -th bit of input. In each step of computation, gates which received their input calculate the function and send their output to the connected vertices. When all output nodes receive their value, the circuit outputs the result with the value of y_i being the i -th bit of output.

We consider two complexity measures of circuits – *circuit size*, defined as the number of gates, and *depth*, defined as the length of the longest path from an input node to an output node.

We denote the size and depth of a circuit C by $\text{size}(C)$ and $\text{del}(C)$, respectively. Size complexity and depth of a family of circuits $\mathcal{C}$ is denoted by $\text{size}_{\mathcal{C}}(n)$ and $\text{del}_{\mathcal{C}}(n)$, respectively, where n is the number of inputs, also called the order of the circuit (in section 4.5, the number of inputs and the order of a circuit can differ. In this case, n is the order of the circuit).

Definition 2 (Synchronous circuit). *A circuit is synchronous if for every gate g , all paths from input nodes to g have the same length and all output nodes are in the same depth.*

Note that our notion of synchronous circuits is stronger than the one introduced by Harper [12]. The original definition does not require the output gates to be in the same depth.

For more detailed treatment of this topic, refer to [25].

2.2 Field Programmable Gate Arrays

Field Programmable Gate Array, abbreviated as FPGA, is a hardware which can efficiently execute boolean circuits. It has programmable gates, which can be programmed to perform a specific function. The gates can then be connected via interconnects into the desired circuit. Usual FPGA consists of a grid of gates with interconnects between them. The architecture of FPGAs only allows for the execution of synchronous circuits. There are techniques for delaying signals which can be used for execution of asynchronous circuits. However, only limited number of signals can be delayed. For this reason, we limit the model to synchronous circuits.

Let G denote the number of gates of the FPGA. Then we assume that any set of circuits with a total of at most G gates can be realised by the interconnects. This assumption tends to be mostly correct in practice, at least asymptotically. The FPGA communicates with other hardware using I/O pins, which are usually placed at the edges of the FPGA. Let I denote the number of I/O pins. It usually holds that $I \approx \sqrt{G}$ since the I/O pins are placed at the edges of the chip.

In the theoretical model, we count execution of a circuit on FPGA as a single operation, not accounting for the memory transfers. This is to capture the increased throughput of FPGA compared to a conventional CPU. Moreover, the speed of RAM is on the same order of magnitude as the speed with which the data can be read by the FPGA. For example, Intel claims that their new FPGA embedded on CPU has I/O throughput of 160Mbs [13], while modern RAM memories have both reading and writing speed at least 160Mbs (for dual-channel) and up to 480Mbs (for quad-channel) [21].

The computation of the FPGA takes place in discrete time-steps and can be pipelined. Each time-step, output signals of layer i of the circuit go to the layer $i + 1$, while layer i receives and processes the output signals of layer $i - 1$. This means that in each time-step, we can give the FPGA an input and we get the output delayed by a number of steps proportional to the depth of the circuit. If we run the FPGA at least as many times as is the depth of the circuit, the amortized time spent per input is, therefore, $\mathcal{O}(1)$.

2.3 I/O model

We use the I/O model of computation to give several lower bounds in our model. The computer in I/O model is a word-RAM [11] with two levels of memory – cache and main memory. Both the main memory and the cache are partitioned into memory blocks. A block consists of B aligned consecutive words. The processor can only work with data stored in the cache. When a word is accessed in main memory, its whole block is loaded into the cache. When a word which is already in the cache is accessed, the computer does not access the main memory. Only M words fit into the cache. When the cache is full and a block is to be loaded into it, a block which is stored in the cache has to be evicted from it. This is done according to a specified eviction strategy, usually to evict the least recently used item (abbreviated as LRU).

The *I/O complexity* is defined to be the function f such that $f(n)$ is equal to the maximum number of memory blocks transferred from the main memory to the cache over all inputs of length n .

A *cache-aware algorithm* is an algorithm in the word-RAM model with the knowledge of B and M . The measure of I/O complexity is usual for cache-aware algorithms.

3 Model of Computation

Throughout this paper, we use the following model of computation. We call the model Pipelining Circuit RAM, abbreviated as PCRAM. The computer consists of word-RAM of word-size w together with circuit execution module with G circuit gates and I circuit inputs/outputs. A program in the PCRAM model consists of two parts. First is a program in the word-RAM running in time polynomial in G and I , outputting circuits with a total of at most G gates and I I/O nodes. We call the set of circuits generated in the first phase $\mathcal{C}$. Second is the main program, working in a modified word-RAM:

At the beginning of execution of the main program, the values G and I are stored in memory. We always assume that $G \geq I$. Additionally, one might assume that $G \in \Theta(I^2)$ or, more generally, $G \in \mathcal{O}(I^c)$ for some c because it is usually the case in practice as explained in Section 2.2. This assumption can greatly simplify sharing of resources, as is described below.

The word-RAM has an additional instruction $\text{RUNCIRCUIT}(C, s, t)$ that starts computation of circuit C on specified input. The instruction has three parameters. Parameter $C \in \mathcal{C}$ identifies which circuit should be run. Parameter s is the source address of the input data. If the circuit has i inputs then the contiguous block of i bits starting from address s is used as input for the circuit. Similarly, t is the address where the output of the circuit is stored after the circuit computes its output. The output is also stored as a contiguous block of bits. Instruction RUNCIRCUIT lasts one time-step and it only starts execution of the circuit and does not wait for the results. The computation time of a circuit is proportional to its depth. If we are evaluating instruction $\text{RUNCIRCUIT}(C, s, t)$ then we get the output starting at address t exactly $\text{del}(C)$ steps after we inserted the input into the circuit. The model allows for use of the RUNCIRCUIT instruction without waiting for the results of the previous RUNCIRCUIT instruction. This is called pipelining and it is a critical feature of the PCRAM model which makes it possible to speed up many algorithms.

We also consider a randomized version of PCRAM with an extra operation which assigns to a memory cell a value chosen uniformly at random independently from other calls of the operation.

We express the complexity of an algorithm in terms of the input size n , the word size w , the number of available circuit inputs and outputs I and the number of available gates G . The complexity of a query on a data structure is measured in time, defined as the number of steps executed on the RAM, and delay, defined as the number of steps when the RAM is waiting for an output of a circuit. The reason for devising two complexity measures is that when performing multiple queries, the processor time is sequential and cannot be pipelined, whereas the execution of the circuit can potentially be pipelined, not necessarily resulting in the delay adding up.

Note that in general, it is not possible to use a constant number of circuits and assume that they all have G gates and I I/O nodes available. However, if all circuits use at most polynomial number of gates in size of their input and output, the speedup caused by the circuits is polynomial in I . Shrinking the circuits by a constant factor will, therefore, incur at most constant slowdown. In this paper, whenever we use multiple circuits, they all use a polynomial number of gates and we do not further discuss the necessity to share resources between circuits. In some cases, it might also be useful to assume that G is polynomial in I to simplify the matters with sharing resources, as already mentioned.

We assume that we can copy $O(I)$ bits in memory in time $O(1)$. For the case when G is polynomial in I , this follows from the following theorem.

Theorem 1. *In the PCRAM model, a contiguous block of $O(I)$ bits can be copied in time $O(1)$.*

Proof. Let CP_k be a copy circuit of size k that is formed by k identity gates. CP_k copies a contiguous block of k bits in time $O(1)$ and it uses $2k$ I/O pins. Next, let us denote K the greatest power of two such that $4K \leq I$. We put $\mathcal{C} = \{CP_1, CP_2, CP_4, \dots, CP_K\}$. In total the circuits in $\mathcal{C}$ use $2K - 1 < G$ gates and $4K - 2 < I$ I/O pins.

For every contiguous block of $O(I)$ bits there exists $s \in \{2^0, 2^1, 2^2, \dots, K\}$ such that the block can be covered by $O(1)$ contiguous intervals of size s , not necessarily pairwise disjoint. The block can then be copied by $O(1)$ calls of $\text{RUNCIRCUIT}(CP_s)$ instruction in time $O(1)$. $\square$

The running time of the algorithms will often depend on “number of elements that can be processed on the circuit at once”. We usually call this number k and define it when needed.

4 Algorithmic Results

In this section, we show several algorithms that are asymptotically faster than the best known algorithms in the word-RAM.

It would also be possible to use in PCRAM a circuit to speed up data structures. However, since efficient data structures often depend on doing random access in memory, it will be difficult to improve the running time of the data structure by more than a factor of $\Theta(\log(I))$. It is likely that this improvement will not be enough to justify practical use of an FPGA. In Section 5, we give lower bounds in the PCRAM model, including lower bound on search trees.

4.1 Aggregation

We show an efficient algorithm performing aggregation operation on an array. While this operation cannot be done in constant time on word-RAM, pipelining can be used to get constant time per operation when amortized over great enough number of instances in the PCRAM model. In Section 4.2, we use this operation for sorting numbers.

Problem 1 (Aggregation).

Input: array A of n numbers, n -bit mask M

Output: t – the number of ones in M and array B – a permutation of A such that if $M[i] = 1$ and $M[j] = 0$, then $A[i]$ precedes $A[j]$ in array B for all i, j .

In this section, we use $k \in \Theta(\min \{ \frac{I}{w}, \frac{G}{w} / \log^2 \frac{G}{w} \})$.

Theorem 2. *Aggregation of n numbers can be computed on PCRAM in time $\mathcal{O}(\frac{n}{k} + \log^2 k)$.*

Proof. Without loss of generality, we suppose that n is divisible by k . Otherwise, we use a circuit to extend the input to the smallest greater size divisible by k .

We construct a circuit that can aggregate k numbers using a sorting network of order k [3]. We call this the aggregator circuit. We use a sorting network to sort M in descending order. The sorting network uses modified comparators that do not only compare and potentially swap values in M , but also swap the corresponding items in array A when swapping values in M . We use synchronous sorting network called bitonic sorter [3]. Bitonic sorter of order k uses $\mathcal{O}(k \log^2 k)$ comparators and has depth $\mathcal{O}(\log^2 k)$. The modified comparator that also moves items in array A can be implemented using $\mathcal{O}(w)$ gates in depth $\mathcal{O}(1)$, since we are only comparing one-bit values. Therefore, our modification of the sorting network uses $\mathcal{O}(kw \log^2 k)$ gates, its depth is $\mathcal{O}(\log^2 k)$ and uses $\mathcal{O}(kw)$ I/O nodes.

Number t has $\lceil \log k \rceil$ bits and each of them can be derived independently on others from the position of the last occurrence of bit 1 in sorted array M . First, the circuit determines in parallel for every position of sorted M whether it is the last occurrence of 1. This can be done in constant depth with $\mathcal{O}(k)$ gates. Then, for every bit of t , the circuit checks if the last 1 is at one of the positions that imply that the bit of t is equal to 1. The value of one bit of t can be computed in depth $\mathcal{O}(\log k)$ with $\mathcal{O}(k)$ gates using a binary tree of OR gates, therefore we need $\mathcal{O}(k \log k)$ gates to compute the value of t from sorted M in depth $\mathcal{O}(\log k)$.

In total the circuit uses $\mathcal{O}(kw \log^2 k)$ gates, it takes $k(w + 1)$ input bits, produces $kw + \lceil \log k \rceil$ output bits and its depth is $\mathcal{O}(\log^2 k)$.

The algorithm runs in two phases:

In the *first phase*, split the input into chunks $(A_1, M_1), (A_2, M_2), \dots, (A_{n/k}, M_{n/k})$ of size k and use the aggregator circuit to separately aggregate elements in each block. The circuit produces outputs $t_1, t_2, \dots, t_{n/k}$ and $B_1, B_2, \dots, B_{n/k}$. It may be necessary to wait for the delay after this phase.

In the *second phase*, for every $i \in 1, 2, \dots, n/k$ split array B_i into two arrays P_i and S_i , where P_i is the prefix of B_i of length t_i and S_i is the rest of B_i . Concatenate the arrays $P_1, P_2, \dots, P_{n/k}, S_1, S_2, \dots, S_{n/k}$.

In total, the algorithm runs in time $\mathcal{O}(\frac{n}{k} + \log^2 k)$. $\square$

The construction can likely be asymptotically improved by using a sorting network with $\mathcal{O}(n \log n)$ comparators and depth $\mathcal{O}(\log n)$ [1]. However, such sorting networks are complicated and impractical due to large multiplicative constants.

Theorem 3. *The aggregation of m arrays with a total length n can be done in time $\mathcal{O}(\frac{n}{k} + m + \log^2 k)$.*

Proof. First, we run the first phase of the algorithm from proof of Theorem 2 for all the arrays and then run the second phases. This way, we only wait for the delay before the second phase at most once. $\square$

4.2 Sorting

We show an asymptotically faster modification of randomized Quicksort with expected running time $\mathcal{O}(\frac{n}{k} \log n + \text{polylog}(n, k, w))$. We achieve this by improving the time complexity of the partition subroutine. Throughout this section, we use $k \in \Theta(\min \{ \frac{I}{w}, \frac{G}{w \log w} / \log^2 \frac{G}{w \log w} \})$.

Problem 2 (Pivot Partition).

Input: array A of n numbers, pivot p

Output: arrays A_1 and A_2 , such that $A_1 \cup A_2 = A$ and A_1 consists exactly of elements $a \in A$, s.t. $a \leq p$, where $\cup$ is the disjoint union.

Theorem 4. *Pivot partition of n numbers can be computed in the PCRAM model in time $\mathcal{O}(n/k + \log^2 k + \log w)$.*

Proof. We reduce the pivot partition problem to aggregation. For each $0 \leq i < n$ we set $M[i] = [A[i] \leq p]$ and perform aggregation with input (A, M) , getting output (t, B) . A_1 consists of the first t elements of B and A_2 of the rest of B . The reduction can be done on PCRAM in time $\mathcal{O}(n/k + \log w)$ by using a circuit to compute k values of array M at once in depth $\mathcal{O}(\log w)$ with $\mathcal{O}(kw)$ gates. Together with aggregation, this gives the desired running time. $\square$

Similarly to aggregation, pivot partition problems can be solved in bulk. The next theorem follows from Theorem 3 by the same reduction as in Theorem 4.

Theorem 5. *We can solve m independent pivot partition problems for arrays of total length n in the PCRAM model in time $\mathcal{O}(n/k + m + \log^2 k + \log w)$.*

Theorem 6. *There is a randomized algorithm in the PCRAM model which sorts n numbers in expected time $\mathcal{O}(\frac{n}{k} \log n + \text{polylog}(n, k, w))$.*

Proof. We show a modification of randomized Quicksort. We use bulk pivot partitioning from Theorem 5 and a bitonic sorter [3] of order k sorting w -bit numbers. Each comparator can be synchronously implemented by $\mathcal{O}(w \log w)$ gates in depth $\mathcal{O}(\log w)$. Therefore, the bitonic sorter has $\mathcal{O}(kw \log^2 k \log w)$ gates and depth $\mathcal{O}(\log^2 k \log w)$. Note that we can also use the bitonic sorter of order k to sort less than k numbers if we fill the unused inputs by the maximum w -bit value and it is therefore enough to only have sorting circuit of one size. The algorithm traverses the recursive tree of Quicksort algorithm in BFS manner. Whenever a subproblem has size at most k , it is sorted by the bitonic sorter. Each layer of the recursion tree corresponds to a sequence of blocks $B_1, \dots, B_m$ for some integer m to be sorted satisfying that if $b_1 \in B_j, b_2 \in B_\ell$, such that $1 \leq j < \ell \leq m$, then $b_1 \leq b_2$. In each layer, blocks of length at most k are sorted using the bitonic sorter and the remaining blocks are partitioned into two by the bulk pivot partition algorithm from Theorem 5 with pivots being chosen uniformly at random for each of the blocks.

In each layer, there can be at most n/k blocks with more than k elements. Thus, from Theorem 5 follows that the partition requires $\mathcal{O}(n/k + \log^2 k + \log w)$ time. Furthermore, there can be at most $2n/k$ blocks of size at most k . We do not have to wait for the delay of the sorting circuit before processing of the next layer. Therefore, we spend $\mathcal{O}(n/k + \log^2 k + \log w)$ time per layer.

The expected depth of the Quicksort recursion tree is $\mathcal{O}(\log n)$ [19] (note that Quicksort recursion depth is the same as the depth of a random binary search tree, which is proven to be $\mathcal{O}(\log n)$ in the cited paper). It follows that the total expected time complexity of our algorithm is $\mathcal{O}((n/k + \log^2 k + \log w) \log n + \log^2 k \log w)$. $\square$

4.3 Associative operation

Let $\otimes : \{0, 1\}^w \times \{0, 1\}^w \rightarrow \{0, 1\}^w$ be an associative operation (such as maximum or addition over a finite group). Let C_1 be a synchronous boolean circuit with g' gates, $2w$ inputs and depth d' which computes the function $\otimes$. We want to efficiently compute $\bigotimes_{i=1}^n a_i$ for input numbers $a_1, \dots, a_n \in \{0, 1\}^w$.

We can assume without loss of generality that there is a neutral element e satisfying $(\forall x \in \{0, 1\}^w) (x \otimes e = e \otimes x = x)$. If there is no such e , we extend $\otimes$ to have one. This only changes the size and depth of C_1 by a constant factor.

Problem 3 ($\otimes$ -sum).

Input: array A of n numbers $a_1, \dots, a_n \in \{0, 1\}^w$

Output: value $\bigotimes_{i=0}^{n-1} A[i]$

The idea of the algorithm is to split the input into blocks that can be processed by the circuit simultaneously by pipelining the computation. In order to do this, we arrange copies of circuit C_1 into a full binary tree. For $j \geq 1$ we inductively define circuit C_{j+1} that consists of two copies of C_j whose outputs go into a copy of C_1 .

In this section, we use $k \in \Theta(\min(\frac{G}{g'}, \frac{I}{w}))$. We assume that k is a power of two.

Theorem 7. $\otimes$ -sum of n w -bit numbers can be evaluated on PCRAM in time $\mathcal{O}(\frac{n}{k} + d' \log(kd'))$.

Proof. The algorithm repeatedly calls subroutine SUMBLOCKS which splits the input into blocks of size k and computes the $\otimes$ -sum of the block using the circuit $C_{\log k}$. The results from $C_{\log k}$ are collected in a buffer that is used as the input for the next call of the subroutine. Each call of SUMBLOCKS reduces the input size k times, therefore we have to call the subroutine $\lceil \log_k n \rceil$ times.

If the length of the input of subroutine SUMBLOCKS is not divisible by k we fill the last block to the full size with neutral element e . By an argument similar to that in Theorem 1, this can be done in constant time by having $\Theta(\log n)$ circuits, the i -th of which has size $w2^i$, has no inputs and produces 2^i copies of e .

For an input of length n' , SUMBLOCKS invokes $\lceil \frac{n'}{k} \rceil$ calls of $C_{\log k}$ and runs in time $\Theta(\frac{n'}{k})$ with delay $\Theta(d' \log k)$.

The size of the problem decreases exponentially, so the first call of SUMBLOCKS dominates in terms of time complexity. The delay is the same for each call of SUMBLOCKS. However, if the input has length $\Omega(kd' \log k)$, the individual values are computed by the circuit before they are needed in the current call of SUMBLOCKS and it is, therefore, not necessary to wait for the delay. We only wait for the delay the last $\mathcal{O}(\log_k(kd' \log k)) = \mathcal{O}(\log_k(kd'))$ executions of SUMBLOCKS, resulting in total waiting time of $\mathcal{O}(d' \log k \log_k(kd')) = \mathcal{O}(d' \log(kd'))$.

4.4 Dynamic programs

A *dynamic program* is a recursive algorithm where the computed values are being memoized, meaning that the function is not computed if it has already been computed with the same parameters. Many dynamic programs used in real-world applications have the property that the subproblems form a multidimensional grid. We then call the individual subproblems cells. For simplicity, we will focus only on dynamic programs with a rectangular grid. However, it is possible to generalise the described technique to higher dimensions.

Many dynamic programs such as those solving the longest common subsequence, shortest common supersequence and edit distance problems or the Needleman-Wunsch algorithm satisfy the following three properties:

1. Each cell depends only on a constant number of other cells.
2. Each cell C depends only on cells which are in the upper-right quadrant with C in the corner (as shown in figure 1).
3. For any cell C , the cells that C depends on have the same relative position with respect to C (as shown in figure 2).

We call such dynamic programs CULD (constant upper-left dependency).

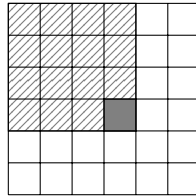


Fig. 1. The grey cell can only depend on the hatched cells in a CULD dynamic program

A CULD dynamic program has linear number of cells that are near left or top edge of the grid and, therefore, should have a dependency on a cell that does not exist (because it would have at least one coordinate negative). These cells have to have their values computed in some other way and are called base cases. As it is usually trivial to compute the values of these cells, we will not deal with computing them in this paper.

Dynamic programs can be sped up on the PCRAM model using the circuit for (1) speeding up the computation of the individual values (it has to take more than constant time to compute

each value for this to be possible) or (2) parallelizing the computation of the values (usually when the individual values can be computed in constant time). Computation of individual values of the function can be sped up in the special case of associative operation using the algorithm described in Section 4.3. In this section, we show how to use the second approach to speed up the computation of the CULD dynamic programs.

Theorem 8. *Let P be a CULD dynamic program with dependency grid of size $M \times N$. Let g' and d' be the number of gates and delay, respectively, of a circuit which computes value of a cell from the values that the cell depends on. After all base cases of the dynamic program have been computed, the remaining values in the dependency grid can be computed in time $\mathcal{O}(\frac{MN}{k} + M + N + kd'^2)$ where $k \stackrel{\text{def}}{=} \min\{I/w, G/g'\}$.*

Proof. Instead of storing the grid in a row-major or column-major layout, we store it in antidiagonal-major layout. That is, the antidiagonals are stored contiguously in memory.

Let d be the number of cells that a cell depends on. We calculate the values of k cells in parallel using a circuit. Notice that subsequent cells in the layout depend on at most d contiguous sequences of cells (see figure 2). First, we move these cells to one contiguous chunk of memory. This can be used as input to a circuit which computes the values of the k cells.

This process can be done repeatedly for all antidiagonals, computing values of all cells in the grid. Note that it is necessary to wait for the delay only if the antidiagonal that is being computed has $\mathcal{O}(kd')$ cells. The total waiting time is, therefore, $\mathcal{O}(kd'^2)$ and the total time complexity $\mathcal{O}(\frac{MN}{k} + M + N + kd'^2)$. $\square$

The circuit computing the cell value can be obtained either by using the general technique of simulating a RAM, which we show in Section 4.5, or an ad-hoc circuit can be used.

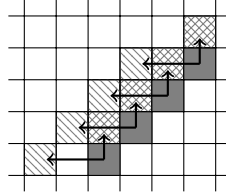


Fig. 2. Contiguous sequence of cells (grey) depend on a constant number (in this case two) of contiguous sequences of cells (hatched)

4.5 Recursive algorithms

In this section, we describe how the circuit can be used to speed up recursive algorithms in which the amount of work is increasing exponentially in the recursion level. This includes some algorithms whose time complexity can be analyzed by the Master theorem [7] as well as many exponential time algorithms. Our approach does not result in a substantial increase of memory usage.

We use the circuit to solve the problem for small input sizes. We do this by using the concept of totally computing circuits, which we introduce. Totally computing circuit is a circuit that computes a given family of functions for all inputs of length bounded by some constant. This is a natural concept in PCRAM since we often want to use the circuit to compute a function on arbitrary input that is short enough and not only for inputs of a specific length.

Definition 3 (Totally computing circuit). *Let $\{f_k\}_{k \in \mathbb{N}}$ be a family of functions such that $f_k : \{0, 1\}^k \rightarrow \{0, 1\}^k$ for every k . Boolean circuit C_n is a totally computing circuit of order n of sequence $\{f_k\}_{k \in \mathbb{N}}$ if it has $n + \lceil \log_2 n \rceil$ inputs, interpreting the first $\lceil \log_2 n \rceil$ bits of input as a binary representation of number n' and it outputs on the first k outputs the value of function $f_{n'}$ evaluated on the next n' bits of the input.*

Similar definition could be made, where the circuit would work in ternary logic with one of the values meaning “no input”, requiring that the input is contiguous and starts at the first I/O node. We stick with the definition mentioned above because this notion of totally computing circuits can be easily simulated by totally computing circuits according to Definition 3.

Definition 4 ((f, G) -recursive algorithm with independent branches). *We call an algorithm A in word-RAM model to be (f, G) -recursive with independent branches if it is of the form $A(0) = 0$ (the recursion base case) and $A(x) = f(A(g_1(x)), \dots, A(g_k(x)))$ otherwise, where the functions f and g_i have no side effects, can use randomness and $G = \{g_i\}_{i=1}^k$ is the family of functions g_i .*

Theorem 9. *Let R be an (f, G) -recursive algorithm with independent branches for some f, G in which the time spent on recursion level i increases exponentially with i . Let $C_{\mathcal{R}}$ be a family of circuits computing the same function as R such that the circuits in $C_{\mathcal{R}}$ can be generated by a Turing machine in time polynomial in their size. Then there exists an algorithm in the PCRAM that runs in time equal to the total time spent by R on problems of size smaller than $k \stackrel{\text{def}}{=} \min(\text{size}_{C_{\mathcal{R}}}^{-1}(G), K)/w$, where K is the greatest number such that $2K + \lceil \log_2 K \rceil \leq I$, plus $\text{del}_{C_{\mathcal{R}}}(kw)$.*

Proof. Notice that k is the maximum subproblem size which can be solved using the circuit. We run a recursion similar to R while using the circuit for solving subproblems of size at most k . The algorithm has two phases and it uses a buffer to store results from the first phase.

In the *first phase*, we compute results of subproblems of size at most k . We run R with the modification that results of subproblems of size at most k are computed using the circuit and stored in the buffer.

In the *second phase*, we compute results of subproblems of size more than k . We run R with the modification that when subproblem size is at most k , we obtain the result computed in phase 1 from the buffer and return it.

Both phases take asymptotically same amount of time which is spent in algorithm R on subproblems of size more than k . The second phase might have to wait for the delay of the circuit before its execution. $\square$

The memory consumption of the algorithm can be high, as we store all the results for subproblems of size at most k in the buffer. This can be avoided by running both phases in parallel with the second phase delayed by the number of time-steps equal to the delay caused by the circuit. Then, at any moment, the number of items in the buffer does not exceed the delay of the circuit and it is, therefore, enough to use a buffer of size equal to the delay of the circuit.

Corollary 1. *Let R be a recursive algorithm with independent branches whose time complexity can be analysed by the Master theorem, running in time $t(n)$. If the time spent on recursion level i of R is increasing exponentially in i , then there is an equivalent algorithm R' in the PCRAM model that runs in time $\mathcal{O}(t(n/k) + \text{del}(C_k))$ where C_k is the circuit used to solve subproblems of size at most k .*

Such problems include the Karatsuba algorithm [15] and the Strassen algorithm [22]. Other algorithms which are not analyzed by master theorem but can be sped up using this technique include some exponential recursive algorithms for solving 3-SAT, maximum independent set, as well as other exponential algorithms based on the branch-and-reduce method [23, 26].

Note that it may be possible to avoid using totally computing circuits by ensuring that the subproblems which are solved using the circuit have all the same size or only have constant number different sizes.

Simulation of RAM or a Turing machine can be used to get totally computing circuits. We say that a circuit of order n simulates given Turing Machine (word-RAM program), if it computes the same function as the Turing machine (word-RAM program) for all inputs of size n .

Observation 1. *Let A be an algorithm that runs on RAM or Turing machine and computes function $f(x)$, where x is the input. Then for any n , there is an algorithm A' which takes the first $\lceil \log_2 n \rceil$ inputs as the binary representation of n' and computes the function f on next n' inputs. Moreover, there is such A' that has the same asymptotic complexity as A .*

This observation can be used as a basis for creation of totally computing circuits as it is sufficient to simulate the algorithm A' to get a totally computing circuit.

The following theorems say that it is possible to simulate a Turing machine or RAM on a circuit. For proof of Theorem 10, see the appendix A. Theorem 11 is well known and its proof can be found in lecture notes by Zwick and Gupta [27].

Theorem 10. *A Word Random Access Machine with word size w running in time $t(n)$ using $m(n)$ memory can be simulated by a circuit of size $\mathcal{O}(t(n)m(n)w \log(m(n)w))$ and depth $\mathcal{O}(t(n) \log(m(n)w))$.*

Theorem 11. *A Turing Machine running in time $t(n)$ using $m(n)$ memory can be simulated by a synchronous circuit of size $\mathcal{O}(t(n)m(n))$ and depth $\mathcal{O}(t(n))$.*

5 Lower Bounds

There are several lower bounds known on the I/O complexity of cache-aware algorithms. We show a general theorem which gives lower bounds in PCRAM from lower bounds in the cache-aware model. Recall that the set of circuits $\mathcal{C}$ is generated by a polynomial algorithm in word-RAM. Throughout this section, we use $t_{\text{gen}}(G, I)$ to denote the time complexity of this program.

Theorem 12. *Let A be an algorithm in the PCRAM model running in time $t(n, G, I, w)$ where n is the input size. Then there is a cache-aware algorithm A' in the I/O model, simulating algorithm A with I/O complexity $\mathcal{O}(t(n, M, Bw, w) + t_{\text{gen}}(M, Bw))$.*

Note that since B and M is the number of words of cache-line and cache, respectively, they have Bw and Mw bits, respectively.

Proof. We simulate algorithm A using $\mathcal{O}(1)$ amortized I/O operations per instruction. We can achieve this by simulating the circuits of A in the cache. Instructions of PCRAM model that do not work with circuits can be trivially performed with $\mathcal{O}(1)$ I/O operations. There exists $c < 1$ such that synchronous circuits with at most cM gates and Bw I/O nodes can be simulated in the cache. At the beginning of the simulation, we generate the set of circuits $\mathcal{C}$ and load them into the cache. This takes time $t_{\text{gen}}(M, Bw)$. Any time we simulate circuit execution, we charge the I/O operations necessary for loading the circuit into the cache to the operation which caused eviction of the blocks which have to be loaded. This proves a bound of $\mathcal{O}(1)$ amortized I/O operation per simulated operation.

We can, therefore, simulate A with I/O complexity $\mathcal{O}(t(n, cM, Bw, w) + t_{\text{gen}}(cM, Bw))$. Since the speedup in the PCRAM model and t_{gen} depend on the number of gates at most polynomially then $\mathcal{O}(t(n, cM, Bw, w) + t_{\text{gen}}(cM, Bw)) = \mathcal{O}(t(n, M, Bw, w) + t_{\text{gen}}(M, Bw))$.

Corollary 2. *If there is a lower bound on a problem of $\Omega(t(n, M, B, w))$ in the cache-aware model, then the lower bound of $\Omega(t(n, G, I/w, w) - t_{\text{gen}}(G, I))$ holds in the PCRAM.*

Some lower bounds in the cache-aware model hold even if the cache can have arbitrary content independent of the input at the start of the computation. Such lower bounds in the cache-aware model imply a lower bound of $\Omega(t(n, G, I/w, w))$ in the PCRAM. This is the case for the lower bound used in Theorem 13.

The following lower bound follows from corollary 2 and a lower bound shown in a survey by Demaine [8].

Theorem 13. *In the PCRAM model, given a set of numbers, assuming that we can only compare them, the search query takes $\Omega(\log_{I/w} n)$ time.*

More lower bounds on cache-aware algorithms are known, but, like in the Theorem 13, they often make further assumptions about operations that we can perform, often requiring that we treat the input items as indivisible objects. Problems for which a non-trivial lower bound is known include sorting, performing permutation and FFT [24].

6 Open Problems

There are many problems waiting to be solved. An important direction of possible future work is to implement the algorithms described in this paper and compare experimental results with predictions based on time complexities in our model. The soundness of the model of computation needs to be empirically verified.

There are many theoretical problems that are not yet solved, including the following:

Open problem 1 (Fast Fourtier Transform). *Is there an asymptotically fast algorithm performing the Fast Fourier Transform in the PCRAM model?*

Open problem 2 (Data structures). *Are there any data structures in the PCRAM model with asymptotically faster queries than their word-RAM counterparts?*

References

- [1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM. ISBN 0-89791-099-0. doi: 10.1145/800061.808726. URL <http://doi.acm.org/10.1145/800061.808726>.
- [2] Nahid Alam. Implementation of genetic algorithms in fpga-based reconfigurable computing systems. 2009. URL https://tigerprints.clemson.edu/all_theses/618/?utm_source=tigerprints.clemson.edu%2Fall_theses%2F618&utm_medium=PDF&utm_campaign=PDFCoverPages.
- [3] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM. doi: 10.1145/1468075.1468121. URL <http://doi.acm.org/10.1145/1468075.1468121>.
- [4] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *2008 Symposium on Application Specific Processors*, pages 101–107, June 2008. doi: 10.1109/SASP.2008.4570793.
- [5] Paweł Chodowiec and Kris Gaj. Very compact fpga implementation of the aes algorithm. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 319–333, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45238-6.
- [6] G. Chrysos, E. Sotiriades, C. Rousopoulos, A. Dollas, A. Papadopoulos, I. Kirmitzoglou, V. Promponas, T. Theocharides, G. Petihakis, J. Lagnel, P. Vavylis, and G. Kotoulas. Opportunities from the use of fpgas as platforms for bioinformatics algorithms. In *2012 IEEE 12th International Conference on Bioinformatics Bioengineering (BIBE)*, pages 559–565, Nov 2012. doi: 10.1109/BIBE.2012.6399733.
- [7] Thomas H. Cormen and Charles E. Leiserson. *Introduction to algorithms*, 3rd edition. The MIT Press, 2009.
- [8] Erik Demaine. Cache-oblivious algorithms and data structures. *EEF Summer School on Massive Data Sets*, 2002.
- [9] Cristian Grozea, Zorana Bankovic, and Pavel Laskov. *FPGA vs. Multi-core CPUs vs. GPUs: Hands-On Experience with a Sorting Application*, pages 105–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16233-6. doi: 10.1007/978-3-642-16233-6_12. URL https://doi.org/10.1007/978-3-642-16233-6_12.
- [10] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 162–170, New York, NY, USA, 2004. ACM. ISBN 1-58113-829-6. doi: 10.1145/968280.968304. URL <http://doi.acm.org/10.1145/968280.968304>.
- [11] Torben Hagerup. Sorting and searching on the word ram. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *STACS 98*, pages 366–398, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-69705-3.

- [12] L. H. Harper. An $n \log n$ lower bound on synchronous combinational complexity. *Proceedings of the American Mathematical Society*, 64(2):300–306, 1977. ISSN 00029939, 10886826. URL <http://www.jstor.org/stable/2041447>.
- [13] Jennifer Huffstetler. Intel processors and fpgas-better together, May 2018. URL <https://itpeernetwork.intel.com/intel-processors-fpga-better-together/>.
- [14] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 248–255, June 2011. doi: 10.1109/AHS.2011.5963944.
- [15] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Dokl. Akad. Nauk SSSR*, volume 145, pages 293–294, 1962. URL <http://mi.mathnet.ru/dan26729>.
- [16] M. Karkooti, J. R. Cavallaro, and C. Dick. Fpga implementation of matrix inversion using qrd-rls algorithm. In *Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005.*, pages 1625–1629, October 2005. doi: 10.1109/ACSSC.2005.1600043.
- [17] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202 – 215, 2014. ISSN 0167-739X. doi: <https://doi.org/10.1016/j.future.2013.06.020>. URL <http://www.sciencedirect.com/science/article/pii/S0167739X13001349>. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.
- [18] Atabak Mahram. Fpga acceleration of sequence analysis tools in bioinformatics, 2013. URL <https://open.bu.edu/handle/2144/11126>.
- [19] Bruce Reed. The height of a random binary search tree. *J. ACM*, 50(3):306–332, May 2003. ISSN 0004-5411. doi: 10.1145/765568.765571. URL <http://doi.acm.org/10.1145/765568.765571>.
- [20] Johannes Romoth, Mario Porrmann, and Ulrich Rückert. Survey of FPGA applications in the period 2000 – 2015 (Technical Report), 2017.
- [21] Dustin Sklavos. Ddr3 vs. ddr4: Raw bandwidth by the numbers, Sep 2015. URL <https://www.techspot.com/news/62129-ddr3-vs-ddr4-raw-bandwidth-numbers.html>.
- [22] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, August 1969. ISSN 0029-599X. doi: 10.1007/BF02165411. URL <http://dx.doi.org/10.1007/BF02165411>.
- [23] Johan M.M. van Rooij and Hans L. Bodlaender. Exact algorithms for dominating set. *Discrete Applied Mathematics*, 159(17):2147 – 2164, 2011. ISSN 0166-218X. doi: <https://doi.org/10.1016/j.dam.2011.07.001>. URL <http://www.sciencedirect.com/science/article/pii/S0166218X11002393>.
- [24] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):54–63, January 2008. ISSN 1551-305X. doi: 10.1561/04000000014. URL <http://dx.doi.org/10.1561/04000000014>.
- [25] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag, Berlin, Heidelberg, 1999. ISBN 3540643109.
- [26] Gerhard J. Woeginger. Combinatorial optimization - eureka, you shrink! chapter Exact Algorithms for NP-hard Problems: A Survey, pages 185–207. Springer-Verlag New York, Inc., New York, NY, USA, 2003. ISBN 3-540-00580-3. URL <http://dl.acm.org/citation.cfm?id=885909.885927>.
- [27] Uri Zwick and Anupam Gupta. Concrete complexity lecture notes, lecture 3, 1996. URL www.cs.tau.ac.il/~zwick/circ-comp-new/two.ps.

A Simulation of Word-RAM

Theorem 10. *A Word Random Access Machine with word size w running in time $t(n)$ using $m(n)$ memory can be simulated by circuit of size $\mathcal{O}(t(n)m(n)w \log(m(n)w))$ and depth $\mathcal{O}(t(n) \log(m(n)w))$.*

Proof. We first construct an asynchronous circuit. In the proof, we will be using the following two subcircuits for writing to and reading from the RAM's memory.

Memory read subcircuit gets as input nw bits consisting of $m(n)$ words of length w together with a number k which fits into one word when represented in binary. It returns the k 'th group. There is such circuit with $\mathcal{O}(m(n)w)$ gates and depth $\mathcal{O}(\log(m(n)w))$.

Memory write subcircuit gets as input $m(n)w$ bits consisting of $m(n)$ words of length w and additional numbers k and v , both represented in binary, each fitting into a word. The circuit outputs the $m(n)w$ bits from input with the exception of the k 'th word, which is replaced by value v . There is such circuit with $\mathcal{O}(m(n)w)$ gates in depth $\mathcal{O}(\log w)$.

The circuit consists of $t(n)$ layers, each of depth $\mathcal{O}(\log(m(n)w))$. Each layer executes one step of the word-RAM. Each layer gets as input the memory of the RAM after execution of the previous instruction and the instruction pointer to the instruction which is to be executed and outputs the memory after execution of the instruction and a pointer to the next instruction. Each layer works in two phases.

In the *first phase*, we retrieve from memory the values necessary for execution of the instruction (including the address where the result is to be saved, in case of indirect access). We do this using the memory read subcircuit (or possibly two of them coupled together in case of indirect addressing). This can be done since the addresses from which the program reads can be inferred from the instruction pointer.

In the *second phase*, we execute all possible instruction on the values retrieved in phase 1. Note that all instructions of the word-RAM can be implemented by a circuit of depth $\mathcal{O}(\log w)$. Each instruction has an output and optional wires for outputting the next instruction (which is used only by jump and conditional jump – all other instructions will output zeros). The correct instruction can be inferred from the instruction pointer, so we can use a binary tree to get the output from the correct instruction to specified wires. This output value is then stored in memory using the memory store subcircuit.

The first layer takes as input the input of the RAM. The last layer outputs the output of the RAM.

Every signal has to be delayed for at most $\mathcal{O}(\log(m(n)w))$ steps. The number of gates is, therefore, increased by a multiplicative factor of at most $\mathcal{O}(\log(m(n)w))$. $\square$