Algorithms in the Ultra-Wide Word Model

Arash Farzan[1], Alejandro López-Ortiz[2], Patrick K. Nicholson[3], and Alejandro Salinger[4(✉)]

[1] Facebook Inc., New York, NY, USA
afarzan@fb.com
[2] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
alopez-o@uwaterloo.ca
[3] Max-Planck-Institut Für Informatik, Saarbrücken, Germany
pnichols@mpi-inf.mpg.de
[4] SAP SE, Walldorf, Germany
alejandro.salinger@sap.com

Abstract. The effective use of parallel computing resources to speed up algorithms in current multi-core parallel architectures remains a difficult challenge, with ease of programming playing a key role in the eventual success of various parallel architectures. In this paper we consider an alternative view of parallelism in the form of an ultra-wide word processor. We introduce the Ultra-Wide Word architecture and model, an extension of the word-RAM model that allows for constant time operations on thousands of bits in parallel. Word parallelism as exploited by the word-RAM model does not suffer from the more difficult aspects of parallel programming, namely synchronization and concurrency. For the standard word-RAM algorithms, the speedups obtained are moderate, as they are limited by the word size. We argue that a large class of word-RAM algorithms can be implemented in the Ultra-Wide Word model, obtaining speedups comparable to multi-threaded computations while keeping the simplicity of programming of the sequential RAM model. We show that this is the case by describing implementations of Ultra-Wide Word algorithms for dynamic programming and string searching. In addition, we show that the Ultra-Wide Word model can be used to implement a non-standard memory architecture, which enables the sidestepping of lower bounds of important data structure problems such as priority queues and dynamic prefix sums. While similar ideas about operating on large words have been mentioned before in the context of multimedia processors [27], it is only recently that an architecture like the one we propose has become feasible and that details can be worked out.

1 Introduction

In the last few years, multi-core architectures have become the dominant commercial hardware platform. The potential of these architectures to improve performance through parallelism remains to be fully attained, as effectively using all cores on a single application has proven to be a difficult challenge. In this

R. Jain et al. (Eds.): TAMC 2015, LNCS 9076, pp. 335–346, 2015.
DOI: 10.1007/978-3-319-17142-5_29

paper we introduce the Ultra-Wide Word architecture and model of computation, an alternate view of parallelism for a modern architecture in the form of an ultra-wide word processor. This can be implemented by replacing one or more cores of a multi-core chip with a very wide word Arithmetic Logic Unit (ALU) that can perform operations on a very large number of bits in parallel.

The idea of executing operations on a large number of bits simultaneously has been successfully exploited in different forms. In Very Long Instruction Word (VLIW) architectures [14], several instructions can be encoded in one wide word and executed in one single parallel instruction. Vector processors allow the execution of one instruction on multiple elements simultaneously, implementing Single-Instruction-Multiple-Data (SIMD) parallelism. This form of parallelism led to the design of supercomputers such as the Cray architecture family [26] and is now present in Graphics Processing Units (GPUs) as well as in Streaming SIMD Extensions (SSE) to scalar processors.

In 2003, Thorup [27] observed that certain instructions present in some SSE implementations were particularly useful for operating on large integers and speeding up algorithms for combinatorial problems. To a certain extent, some of the ideas in the Ultra Wide Word architecture are presaged in the paper by Thorup, which was proposed in the context of multimedia processors. Our architecture developed independently and differs on several aspects (see discussion in full version [15]) but it is motivated by similar considerations.

As CPU hardware advances, so does the model used in theory to analyze it. The increase in word size was reflected in the word-RAM model in which algorithm performance is given as a function of the input size n and the word size w, with the common assumption that $w = \Theta(\log n)$. In its simplest version, the word-RAM model allows the same operations as the traditional RAM model. Algorithms in this model take advantage of bit-level parallelism through packing various elements in one word and operating on them simultaneously. Although similar to vector processing, the word-RAM provides more flexibility in that the layout of data in a word depends on the algorithm and data elements can be packed in an arbitrary way. Unlike VLIW architectures, the Ultra-Wide Word model we propose is not concerned with the compiler identifying operations which can be done in parallel but rather with achieving large speedups in implementations of word-RAM algorithms through operations on thousands of bits in parallel.

As multi-core chip designs evolve, chip vendors try to determine the best way to use the available area on the chip, and the options traditionally are an increased number of cores or larger caches. We believe that the current stage in processor design allows for the inclusion of an architecture such as the one we propose. In addition, ease of programming is a major hurdle to the eventual success of parallel and multi-core architectures. In contrast, bit parallelism as exploited by the word-RAM model does not suffer from this drawback: there is a large selection of word-RAM algorithms (see, e.g., [2,11,19,21]) that readily benefit from bit parallelism without having to deal with the more difficult aspects of concurrency such as mutual exclusion, synchronization, and resource contention. In this sense, the advantage of an on-chip ultra-wide word

architecture is that it can enable word-RAM algorithms to achieve speedups comparable to those of multi-threaded computations, while at the same time keeping the simplicity of sequential programming that is inherent to the RAM model. We argue that this is the case by showing several examples of implementations of word-RAM algorithms using the wide word, usually with simple modifications to existing algorithms, and extending the ideas and techniques from the word-RAM model.

In terms of the actual architecture, we envision the ultra-wide ALU together with multi-cores on the same chip. Thus, the Ultra-Wide Word architecture adds to the computing power of current architectures. The results we present in this paper, however, do not use multi-core parallelism.

Summary of Results. We introduce the Ultra-Wide Word architecture and model, which extends the w-bit word-RAM model by adding an ALU that operates on w^2-bit words. We show that several broad classes of algorithms can be implemented in this model. In particular:

- We describe Ultra-Wide Word implementations of dynamic programming algorithms for the subset sum problem, the knapsack problem, the longest common subsequence problem, as well as many generalizations of these problems. Each of these algorithms illustrates a different technique (or combination of techniques) for translating an implementation of an algorithm in the word-RAM model to the Ultra-Wide Word model. In all these cases we obtain a w-fold speedup over word-RAM algorithms.
- We also describe Ultra-Wide Word implementations of popular string searching algorithms: the Shift-And/Shift-Or algorithms [3,28] and the Boyer-Moore-Horspool algorithm [22]. Again, we obtain a w-fold speedup over the original algorithms.
- Finally, we show that the Ultra-Wide Word model is powerful enough to simulate a non-standard memory architecture in which bytes can overlap, which we shall call FS-RAM [16]. This allows us to implement data structures and algorithms that circumvent known lower bounds for the word-RAM model.

Due to space constraints, we only present a high-level description of our results. The full details can be found in the full version of this paper [15].

2 The Ultra-Wide Word-RAM Model

The Ultra-Wide word-RAM model (UW-RAM) we propose is an extension of the word-RAM model. The word-RAM is a variant of the RAM model in which a word has length w bits, and the contents of memory are integers in the range $\{0, \ldots, 2^w - 1\}$ [19]. This implies that $w \geq \log n$, where n is the size of the input, and a common assumption is $w = \Theta(\log n)$ (see, e.g., [7,24]). Algorithms in this model take advantage of the intrinsic parallelism of operations on w-bit words. We provide a more detailed description of the word-RAM in the full version [15].

The Ultra-Wide word-RAM model extends the word-RAM model by introducing an ultra-wide ALU with w^2-bit *wide words*. The ultra-wide ALU supports the

basic operations available in a word-RAM on the entire word at once. As in the word-RAM model, the available set of instructions can be assumed to be those of the restricted, multiplication, or the AC^0 models. For the results in this paper we assume the instructions of the restricted model (addition, subtraction, left and right shift, and bitwise boolean operations), plus two non-standard straightforward AC^0 operations that we describe at the end of this section.

The model maintains the standard w-bit ALU as well as w-bit memory addressing. In general, we use the parameter w for the word size in the description and analysis of algorithms, although in some cases we explicitly assume $w = \Theta(\log n)$. In terms of real world parameters, the wide word in the ultra-wide ALU would presently have between 1,000 and 10,000 bits and could increase even further in the future. In reality, the addition of an ALU that supports operations on thousands on bits would require appropriate adjustments to the data and instruction caches of a processor as well as to the instruction pipeline implementation. Similarly to the abstractions made by the RAM and word-RAM models, the UW-RAM model ignores the effects of these and other architectural features and assumes that the execution of instructions on ultra-wide words is as efficient as the execution of operations on regular w-bit words, up to constant factors.

Provided that the UW-RAM supports the same operations as the word-RAM, the techniques to achieve bit-level parallelism in the word-RAM extend directly to the UW-RAM. However, since the word-RAM assumes that a word can be read from memory in constant time, many operations in word-RAM algorithms can be implemented through constant time table lookups. With words of w^2 bits, we cannot expect to achieve constant time lookups since the size of the tables would be prohibitive. However, the memory access operations of our model allow for the implementation of simultaneous table lookups of several w-bit words within a wide word, as we shall explain below.

We first introduce some notation. Let W denote a w^2-bit word. Let $W[i]$ denote the i-th bit of W, and let $W[i..j]$ denote the contiguous subword of W from bit i to bit j, inclusive. The least significant bit of W is $W[0]$, and thus $W = \sum_{i=0}^{w^2-1} W[i] \times 2^i$. For the sake of memory access operations, we divide W into w-bit blocks. Let W_j denote the j-th contiguous block of w bits in W, for $0 \leq j \leq w-1$, and let $W_j[i]$ denote the i-th bit within W_j. Thus, $W_j = W[jw..(j+1)w-1]$. The division of a wide word in blocks is solely intended for certain memory access operations, but basic operations of the model have no notion of block boundaries. Figure 1 shows a representation of a wide word, depicting bits with increasing significance from left to right. In the description of operations with wide words we generally refer to variables with uppercase letters, whereas we use lowercase to refer to regular variables that use one w-bit word. Thus, shifts to the left (right) by i are equivalent to division (multiplication) by 2^i. In addition, we use $\mathbf{0}$ to denote a wide word with value 0. We use standard C-like notation for operations AND ('&'), OR ('|'), NOT ('$\sim$') and shifts ('<<','>>').

Memory Access Operations. In this architecture w (not necessarily contiguous) words from memory can be transferred into the w blocks of a wide word W in constant time. These blocks can be written to memory in parallel as well.

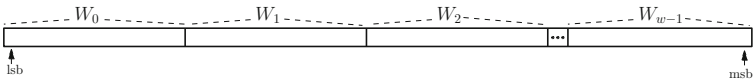

Fig. 1. A wide word in the Ultra-Wide Word architecture. The wide word is divided in w blocks of w bits each, shown here in increasing number of block from left to right.

As with PRAM algorithms, the memory access type of the model can be assumed to allow or disallow concurrent reads and writes. For the results in this paper we assume the Concurrent-Read-Exclusive-Write (CREW) model.

The memory access operations that involve wide words are of three types: block, word, and content. We describe read accesses (write accesses are analogous). A *block access* loads a single w-bit word from memory into a given block of a wide word. A *word access* loads w contiguous w-bit words from memory into an entire wide word in constant time. Finally, a *content access* uses the contents of a wide word W as addresses to load (possibly non-contiguous) words of memory simultaneously: for each block j within W, this operation loads from memory the w-bit word whose address is W_j (plus possibly a base address). The specifics of read and write operations are shown in Table 1.

Note that accessing several (possibly non-contiguous) words from memory simultaneously is an assumption that is already made by any shared memory multiprocessing model. While, in reality, simultaneous access to all addresses in actual physical memory (e.g., DRAM) might not be possible, in shared memory systems, such as multi-core processors, the slowdown is mitigated by truly parallel access to private and shared caches, and thus the assumption is reasonable. We therefore follow this assumption in the same spirit.

In fact, for w equal to the regular word size (32 or 64 bits), the choice of w blocks of w bits each for the wide word ALU was judiciously made to provide the model with a feasible memory access implementation. w^2 lines to memory are well within the realm of the possible, as they are of the same order of magnitude (a factor of 2 or 8) as modern GPUs, some of which feature bus widths of 512 bits (see, e.g., [1,18]). We note that a more general model could feature a wide word with k blocks of w bits each, where k is a parameter, which can be adjusted in reality according to the feasibility of implementation of parallel memory accesses. Although described for w blocks, the algorithms presented in this paper can easily be adapted to work with k blocks instead. Naturally, the speedups obtained would depend on the number of blocks assumed, but also on the memory bandwidth of the architecture. A practical implementation with a large number of blocks would likely suffer slowdowns due to congestion in the memory bus. We believe that an implementation with k equal to 32 or 64 can be realized with truly parallel memory access, leading to significant speedups.

UW-RAM Subroutines. A procedure called *compress* serves to bring together bits from all blocks into one block in constant time, while a procedure called *spread* is the inverse function[1]. Both operations can be implemented by straight-

[1] These operations are also known as PackSignBits and UnPackSignBits [27].

Table 1. Wide word memory access operations of the UW-RAM. MEM denotes regular RAM memory, which is indexed by addresses to words, and *base* is some base address.

Name	Input	Semantics
read_block	W, j, base	$W_j \leftarrow$MEM[base+j]
read_word	W, base	for all j in parallel: $W_j \leftarrow$MEM[base+j]
read_content	W, base	for all j in parallel: $W_j \leftarrow$MEM[base+W_j]
write_block	W, j, base	MEM[base+j]$\leftarrow W_j$
write_word	W, base	for all j in parallel: MEM[base+j]$\leftarrow W_j$
write_content	W, V, base	for all j in parallel: MEM[base+V_j]$\leftarrow W_j$

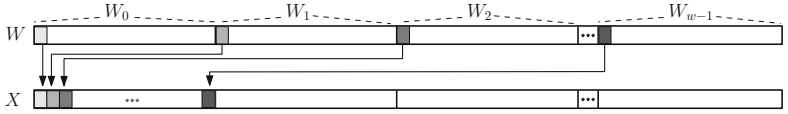

Fig. 2. The *compress* operation takes a wide word W whose set bits are restricted to the first bit of each block and compresses them to the first block of a wide word.

forward constant-depth circuits. We will also use parallel comparators, a standard technique used in word-RAM algorithms [19] (see details in full version [15]). Although these are all the subroutines that we need for the results in this paper, other operations of similar complexity could be defined if proved useful.

- **Compress:** Let W be a wide word in which all bits are zero except possibly for the first bit of each block. The compress operation copies the first bit of each block of W to the first block of a word X. I.e., if $X = \text{compress}(W)$, then $X[j] \leftarrow W_j[0]$ for $0 \leq j < w$, and $X[j] = 0$ for $j \geq w$ (see Fig. 2).
- **Spread:** This operation is the inverse of the compress operation. It takes a word W whose set bits are all in the first block and spreads them across blocks of a word X so that $X_j[0] \leftarrow W[j]$ for $0 \leq j < w$.

Relation to Other Models. We provide a discussion of similarities and differences between the UW-RAM and other existing models in the full version [15].

3 Simulation of FS-RAM

In the standard RAM model of computation memory is organized in registers or words, each word containing a set of bits. Any bit in a word belongs to that word only. In contrast, in the FS-RAM model [16]—also known as Random Access Machine with Byte Overlap (RAMBO)—words can overlap, that is, a single bit of memory can belong to several words. The topology of the memory, i.e., a specification of which bits are contained in which words, defines a particular variant of the FS-RAM model. Variants of this model have been used to sidestep lower bounds for important data structure problems [9,10].

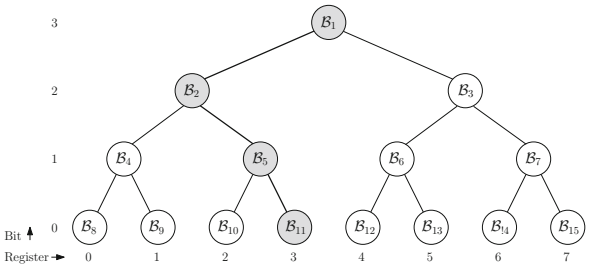

Fig. 3. Yggdrasil memory layout [9]: each node in a complete binary tree is an FS-RAM bit and registers are defined as paths from a leaf to the root. For example, register 3 contains bits $\mathcal{B}_{11}, \mathcal{B}_5, \mathcal{B}_2$, and $\mathcal{B}_1$ (shaded nodes).

We show how the UW-RAM can be used to implement memory access operations for any given FS-RAM of word size at most w bits in constant time. Thus, the time bounds of any algorithm in the FS-RAM model carry over directly to the UW-RAM. Note that each FS-RAM layout requires a different specialized hardware implementation, whereas a UW-RAM architecture can simulate any FS-RAM layout without further changes to its memory architecture.

Let $\mathcal{B}_1, \ldots, \mathcal{B}_B$ denote the bits of FS-RAM memory. A particular FS-RAM memory layout can be defined by the registers and the bits contained in them [8]. For example, in the *Yggdrasil* model in Fig. 3, reg[0]=$\mathcal{B}_8\mathcal{B}_4\mathcal{B}_2\mathcal{B}_1$, and in general reg[$i$].bit[$j$]= $\mathcal{B}_k$, where $k = \lfloor i/2^j \rfloor + 2^{m-j-1}$ ($m = 4$ in the example) [9].

In order to implement memory access operations on a given FS-RAM using the UW-RAM, we need to represent the memory layout of FS-RAM in standard RAM. Assume an FS-RAM memory of r registers of $b \leq w$ bits each and $B \leq br$ distinct FS-RAM bits. We assume that the FS-RAM layout is given as a table $\mathcal{R}$ that stores, for each register and bit within the register, the number of the corresponding FS-RAM bit. Thus, if reg[i].bit[j]= $\mathcal{B}_k$, for some k, then $\mathcal{R}[i, j] = k$. We assume $\mathcal{R}$ is stored in row major order. We simply store the value of each FS-RAM bit $\mathcal{B}_i$ in a different w-bit entry of an array A in RAM, i.e., $A[i] = \mathcal{B}_i$.

Given an index t of a register of an FS-RAM represented by $\mathcal{R}$, we can read the values of each bit of reg[t] from RAM and return the b bits in a word in constant time using the parallel reading and compress operations. Let reg[t]= $\mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$. The read operation first obtains the address in A of each bit of register t from $\mathcal{R}$. Then, it uses a content access to read the value of each bit $\mathcal{B}_{i_j}$ into block W_j of W, thus assigning $W_j \leftarrow A[\mathcal{R}[t, j]]$. Finally, it applies one compress operation, after which the b bits are stored in W_0. In order to implement the write operation reg[t]$\leftarrow \mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$ of FS-RAM, we first set $W_0 \leftarrow \mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$ and perform a spread operation to place each bit $\mathcal{B}_j$ in block W_j. We then write the contents of each W_j in $A[\mathcal{R}[t, j]]$. Both read and write take constant time. We describe these operations in pseudocode in the full version [15].

Since the read and write operations described above are sufficient to implement any operation that uses FS-RAM memory (any other operation is implemented in RAM), we have the following result (see [15] for the proof).

Theorem 1. *Let $\mathcal{R}$ be any* FS-RAM *memory layout of r registers of at most b bits each and B distinct* FS-RAM *bits, with $b \leq w$ and $\log B \leq w$. Let A be any* FS-RAM *algorithm that uses $\mathcal{R}$ and runs in time T. Algorithm A can be implemented in the* UW-RAM *to run in time $O(T)$, using $rb + B$ additional words of* RAM.

Constant Time Priority Queue. Brodnik et al. [9] use the Yggdrasil FS-RAM memory layout to implement priority queue operations in constant time using $3M - 1$ bits of space ($2M$ of ordinary memory and $M - 1$ of FS-RAM memory), where M is the size of the universe. This problem has non-constant lower bounds for several models, including the RAM model [5]. For a universe of size $M = 2^m$, for some m, the Yggdrasil FS-RAM layout consists of $r = M/2$ registers of $b = \log M$ bits each and $B = M - 1$ distinct FS-RAM bits (Figure 3 is an example with $M = 16$). Thus, by Theorem 1 we obtain the following result:

Corollary 1. *The discrete extended priority queue problem can be solved in the* UW-RAM *in $O(1)$ time per operation using $2M + w(M/2) \log M + w(M - 1)$ bits, thus in $O(M \log M)$ words of* RAM.

Constant Time Dynamic Prefix Sums. Brodnik et al. [10] use a modified version of the Yggdrasil FS-RAM to solve the dynamic prefix sums problem in constant time. This problem consists of maintaining an array A of size N over a universe of size M that supports the operations $update(j, d)$, which sets $A[j]$ to $A[j] \oplus d$, and $retrieve(j)$, which returns $\oplus_{i=0}^{j} A[i]$ [10,17], where $\oplus$ is any associative binary operation. This FS-RAM implementation sidesteps lower bounds on various models [17,20]. See the full version [15] for more details.

Corollary 2. *The operations of the dynamic prefix sums problem can be supported in $O(1)$ time in the* UW-RAM *with $O(M^{\sqrt{\log N}})$ bits of* RAM.

4 Dynamic Programming

In this section we describe UW-RAM implementations of dynamic programming algorithms for the subset sum, knapsack, and longest common subsequence problems. A word-RAM algorithm that only uses bit parallelism can be translated directly to the UW-RAM. The algorithm for subset sum is an example of this. In general, however, word-RAM algorithms that use lookup tables cannot be directly extended to w^2 bits, as this would require a mechanism to address $\Theta(w^2)$-bit words in memory as well as lookup tables of prohibitively large size. Hence, extra work is required to simulate table lookup operations. The knapsack implementation that we present is a good example of such case. We note that these problems have many generalizations that can be solved using the same techniques and describe them further in the full version [15].

Subset Sum. Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers (weights) and an integer t (capacity), the subset sum problem is to find $S' \subseteq S$ such that

$\sum_{a_i \in S'} a_i = t$ [12]. This problem is NP-hard, but it can solved in pseudopolynomial time via dynamic programming in $O(nt)$ time, using the following recurrence [6]: for each $0 \leq i \leq n$ and $0 \leq j \leq t$, $C_{i,j} = 1$ if and only if there is a subset of elements $\{a_1, \ldots, a_i\}$ that adds up to j. Thus, $C_{0,0} = 1$, $C_{0,j} = 0$ for all $j > 0$, and $C_{i,j} = 1$ if $C_{i-1,j} = 1$ or $C_{i-1,j-a_i} = 1$ ($C_{i,j} = 0$ for any $j < 0$). The problem admits a solution if $C_{n,t} = 1$.

Pisinger [25] gives an algorithm that implements this recursion in the word-RAM with word size w by representing up to w entries of a row of C. Using bit parallelism, w bits of a row can be updated simultaneously in constant time from the entries of the previous row: C_i is updated by computing $C_i = (C_{i-1} \mid (C_{i-1} >> a_i))$ (which might require shifting words containing C_{i-1} first by $\lfloor a_i/w \rfloor$ words and then by $a_i - \lfloor a_i/w \rfloor$) [25]. Assuming $w = \Theta(\log t)$, this approach leads to an $O(nt/\log t)$ time solution in $O(t/\log t)$ space.

This algorithm can be implemented directly in the UW-RAM: entries of row C_i are stored contiguously in memory; thus, we can load and operate on w^2 bits in $O(1)$ time when updating each row. Hence, the UW-RAM implementation runs in $O(nt/\log^2 t)$ time using the same $O(t/\log t)$ space (number of w-bit words).

Knapsack. Given a set S of n elements with weights and values, the knapsack problem asks for a subset of S of maximum value such that the total weight is below a given capacity bound b. Let $S = \{(w_i, v_i)\}_{i=1}^n$, where w_i and v_i are the weight and value of the i-th element. Like subset sum, this problem is NP-hard but can be solved in pseudopolynomial time using the following recurrence [6]: let $C_{i,j}$ be the maximum value of a solution containing elements in the subset $S_i = \{(w_k, v_k)\}_{k=1}^i$ with maximum capacity j. Then, $C_{0,j} = 0$ for all $0 \leq j \leq b$, and $C_{i,j} = \max\{C_{i-1,j}, C_{i-1,j-w_i} + v_i\}$. The value of the optimal solution is $C_{n,b}$. This leads to a dynamic program that runs in $O(nb)$ time.

The word-RAM algorithm by Pisinger [25] represents partial solutions of the dynamic programming table with two binary tables g and h and operates on $O(w)$ entries at a time. More specifically, $g_{i,u} = 1$ and $h_{i,v} = 1$ if and only if there is a solution with weight u and value v that is not dominated by another solution in $C_{i,*}$ (i.e., there is no entry $C_{i,u'}$ such that $u' < u$ and $C_{i,u'} \geq v$). Pisinger shows how to update each entry of g and h with a constant time procedure, which can be encoded as a constant size lookup table T. A new lookup table T^α is obtained as the product of α times the original table T. Thus, α entries of g and h can be computed in constant time. Setting $\alpha = w/10$, an entire row of g and h can be computed in $O(m/w)$ time and $O(m/w)$ space [25], where m is the maximum of the capacity b and the value of the optimal solution. The optimal solution can then be computed in $O(nm/w)$ time.

Compared to the subset sum algorithm, which relies mainly on bit-parallel operations, this word-RAM algorithm for knapsack relies on precomputation and use of lookup tables to achieve a w-fold speedup. While we cannot precompute a composition of $\Theta(w^2)$ lookup tables to compute $\Theta(w^2)$ entries of g and h at a time, we can use the same tables with $\alpha = w/10$ as in Pisinger's algorithm and use the *read_content* operation of the UW-RAM to make w simultaneous lookups to the table. Since the entries in a row i of h and g depend only on entries in row $i - 1$, then there are no dependencies between entries in the same row.

One difficulty is that in order to compute the entries in row i in parallel we must first preprocess row $i-1$ in both h and g, such that we can return the number of one bits in both $g_{i-1,0}, ..., g_{i-1,j}$ and $h_{i-1,0}, ..., h_{i-1,j}$ in $O(1)$ time for any column $j \in \{0, m-1\}$. That is, the prefix sums of the one bits in row $i-1$. Note that this is *not* the same as the dynamic problem described in Sect. 3, but it is a static prefix sums problem. We describe how to compute the prefix sums of a row of g and h in $O(m/w^2)$ time in the full version [15]. Then, each row of g and h takes $O(m/w^2)$ time to compute, and since there are n rows, the total time to compute g and h (and hence the optimal solution) on the UW-RAM is $O(nm/w^2)$. This achieves a w-fold speedup over Pisinger's word-RAM solution.

Longest Common Subsequence. The final dynamic programming problem we examine is that of computing the longest common subsequence (LCS) of two string sequences (see the full version [15] for a definition). This problem can be solved via a classic dynamic programming algorithm in $O(nm)$ time [12]. In [15] we describe a UW-RAM algorithm for LCS based on an algorithm by Masek and Paterson [23]. We note that there exist other approaches to solving the LCS problem with bit-parallelism (e.g., [13]) that could also be adapted to work in the UW-RAM. The approach we show here is a good example of bit parallelism combined with the parallel lookup power of the model, which we use to implement the Four Russians technique. We obtain the following results:

Theorem 2. *The length of the LCS of two strings X and Y over an alphabet of size σ, with $|X| = m$ and $|Y| = n$, can be computed in the* UW-RAM *in $O(\frac{nm}{w^2}\log\sigma + m + n)$ time and $O(\frac{\min(n,m)}{w}\log\sigma)$ words in addition to the input.*

Theorem 3. *The length of the LCS of two strings X and Y of length n over an alphabet of size σ can be computed in the* UW-RAM *in $O(n^2\log^2(\sigma)/w^3 + n\log(\sigma)/w)$ time. For $\sigma = O(1)$ and $w = \Theta(\log n)$ this time is $O(n^2/\log^3 n)$.*

5 String Searching

Another example of a problem where a large class of algorithms can be sped up in the UW-RAM is string searching. Given a text T of length n and a pattern P of length m, both over an alphabet Σ, string searching consists of reporting all the occurrences of P in T. We assume in general that $n \gg m$. We use two classic algorithms for this problem to illustrate different ways of obtaining speedups via parallel operations in the wide word. More specifically, we obtain speedups of $w = \Omega(\log n)$ for UW-RAM implementations of the Shift-And and Shift-Or algorithms [3,28], and the Boyer-Moore-Horspool algorithm [22].

Shift-And and Shift-Or. These algorithms simulate an $(m+1)$-state nondeterministic automaton that recognizes P starting from every position of T. For a window $T[i-m+1..i]$ in T, the j-th state of the automaton ($0 \le j \le m$) is active if and only if $P[1..j] = T[i-j+1..i]$. These algorithms represent the automaton as a bit vector and update the active states using bit-parallelism. Their running time is $O(mn/w + n)$, achieving linear time on the size of the text for small

patterns. We describe in the full version [15] two UW-RAM algorithms for Shift-And that illustrate different techniques, noting that the UW-RAM implementation of Shift-Or is analogous. We obtain the following theorem:

Theorem 4. *Given a text T of length n and a pattern P of length m, we can find the occ occurrences of P in T in the* UW-RAM *in time $O(nm/w^2+n/w+occ)$.*

Boyer-Moore-Horspool. BMH [22] keeps a sliding window of length m over the text T and searches backwards in the window for matching suffixes of both the window and the pattern. The worst case running time of BMH is $O(nm)$ (when the entire window is checked for all window positions) but on average the window can be shifted by more than one character, making the running time $O(n)$ [4]. In the UW-RAM, we can take advantage of the wide word to make several character comparisons in parallel, thus achieving a w-fold speedup over the worst case behaviour of BMH. Full details are described in [15].

Theorem 5. *Given T of length n and P of length m over an alphabet of size σ, we can find the occurrences of P in T with a* UW-RAM *implementation of BMH in $O(mn \log \sigma/w^2 + 1)$ time in the worst-case and $O(n)$ time on average.*

6 Conclusions

We introduced the Ultra-Wide Word architecture and model and showed that several classes of algorithms can be readily implemented in this model to achieve a speedup of $\Omega(\log n)$ over traditional word-RAM algorithms. The examples we describe already show the potential of this model to enable parallel implementations of existing algorithms with speedups comparable to those of multi-core computations. We believe that this architecture could also serve to simplify many existing word-RAM algorithms that in practice do not perform well due to large constant factors. We conjecture as well that this model will lead to new efficient algorithms and data structures that can sidestep existing lower bounds.

References

1. AMD: AMD FirePro W9100 Workstation Graphics. http://www.amd.com/Documents/FirePro_W9100_Data_Sheet.pdf. Acessed 20 Nov 2014
2. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM **54**(3), 13 (2007)
3. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Commun. ACM **35**(10), 74–82 (1992)
4. Baeza-Yates, R.A., Régnier, M.: Average running time of the Boyer-Moore-Horspool algorithm. Theoret. Comput. Sci. **92**(1), 19–31 (1992)
5. Beame, P., Fich, F.: Optimal bounds for the predecessor problem and related problems. J. Comput. Syst. Sci. **65**, 2002 (2002)
6. Bellman, R.: Dynamic Programming, 1st edn. Princeton University Press, Princeton (1957)

7. Bose, P., Chen, E.Y., He, M., Maheshwari, A., Morin, P.: Succinct geometric indexes supporting point location queries. In: Proceedings of SODA, pp. 635–644 (2009)
8. Brodnik, A.: Searching in Constant Time and Minimum Space. Ph.D. thesis, University of Waterloo (1995), also available as Technical Report CS-95-41
9. Brodnik, A., Carlsson, S., Fredman, M.L., Karlsson, J., Munro, J.I.: Worst case constant time priority queue. J. Syst. Softw. **78**(3), 249–256 (2005)
10. Brodnik, A., Karlsson, J., Munro, J., Nilsson, A.: An O(1) solution to the prefix sum problem on a specialized memory architecture. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) TCS 2006. IFIP, vol. 209, pp. 103–114. Springer, Boston (2006)
11. Chan, T.M.: Point location in o(log n) time, Voronoi diagrams in o(n log n) time, and other transdichotomous results in computational geometry. In: Proceedings of FOCS, pp. 333–344 (2006)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
13. Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J., Reid, J.F.: A fast and practical bit-vector algorithm for the longest common subsequence problem. Inf. Process. Lett. **80**(6), 279–285 (2001)
14. Fisher, J.A.: Very long instruction word architectures and the ELI-512. SIGARCH Comput. Archit. News **11**, 140–150 (1983)
15. Frazan, A., López-Ortiz, A., Nicholson, P.K., Salinger, A.: Algorithms in the Ultra-Wide Word Model (2014). http://arxiv.org/pdf/1411.7359v2
16. Fredman, M., Saks, M.: The cell probe complexity of dynamic data structures. In: Proceedings of STOC, pp. 345–354 (1989)
17. Fredman, M.L.: The complexity of maintaining an array and computing its partial sums. J. ACM **29**(1), 250–260 (1982)
18. GeForce: GeForce GTX 285 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285/specifications. Accessed 20 Nov 2014
19. Hagerup, T.: Sorting and searching on the word RAM. In: Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 366–398. Springer, Heidelberg (1998)
20. Hampapuram, H., Fredman, M.L.: Optimal biweighted binary trees and the complexity of maintaining partial sums. SIAM J. Comput. **28**(1), 1–9 (1998)
21. Han, Y.: Deterministic sorting in O(nlog logn) time and linear space. J. Algorithms **50**, 96–105 (2004)
22. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. **10**(6), 501–506 (1980)
23. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. **20**(1), 18–31 (1980)
24. Munro, J.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
25. Pisinger, D.: Dynamic programming on the word RAM. Algorithmica **35**, 128–145 (2003)
26. Russell, R.M.: The CRAY-1 computer system. Comm. ACM **21**(1), 63–72 (1978)
27. Thorup, M.: Combinatorial power in multimedia processors. SIGARCH Comput. Archit. News **31**(4), 5–11 (2003)
28. Wu, S., Manber, U.: Fast text searching: allowing errors. Commun. ACM **35**(10), 83–91 (1992)