Microprocessors and Microsystems 50 (2017) 189–201

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Scalable shared-memory architecture to solve the Knapsack 0/1 problem

Fernando A. Escobar[a,*], Anthony Kolar[b], Naim Harb[c], Filipe Vinci Dos Santos[a], Carlos Valderrama[c]

[a] Advanced Analog Systems Chair, CentraleSupélec, 3 Rue Joliot Curie, F-91192 Gif-Sur-Yvette Cedex, France
[b] Group of Electrical Engineering Paris, UMR CNRS 8507, CentraleSupélec, Univ. Paris-Sud, Univ. Paris-Saclay, Sorbonne Univ., UPMC Univ Paris 06, F-91192 Gif-Sur-Yvette Cedex, France
[c] 31 Boulevard Dolez, 7000 Mons, Belgium

ARTICLE INFO

Article history:
Received 26 September 2016
Revised 31 March 2017
Accepted 2 April 2017
Available online 3 April 2017

Keywords:
Knapsack 01
Shared memory
Dynamic programming
FPGA
SoC

ABSTRACT

Dynamic Programming (DP) is used to solve combinatorial optimization problems and constitutes one of the 13 High Performance Computing (HPC) patterns. DP suffers from irregular, data-dependent memory accesses that deteriorates performance. The Knapsack 0/1 belongs to the simplest DP algorithms which is called Serial Monadic and has been treated in software with cache-efficient algorithms as well as parallel threads, OpenMP or MPI.

In this paper we propose a shared memory, parametrizable architecture to compute the DP matrix for the Knapsack 0/1. Our system has a parallel runtime of $\Theta(mC/q)$ for a knapsack of capacity C with m items and q operators. Using additional off-chip space and DMA transfers it can solve knapsacks of any size. The architecture is implemented on the ZYNQ-7020 System On Chip (SoC) that contains a dual-core ARM plus Artix FPGA fabric. Under such architecture we make use of 64-bit High Performance ports for off-chip transfers and asymmetric 32-bit write/64-bit read BRAMs to minimize data exchange times. We also exploit computation synchronization to minimize BRAM address propagation and reduce routing congestion. We present results for a base system with 70 Processing Elements (PEs) capable of solving problems with a maximum item weight $\omega_{\max} = 1024$. For more complex instances we configure the architecture with 58 PEs and $\omega_{\max} = 6144$, where a single BRAM is shared among 13 computing units. We thus solve problems with $6\times$ bigger weights than previous works, attain a $16\times$ speed-up versus an optimized software on an Intel Xeon E5 and get the highest efficiency per core versus other architectures. We achieve between $2.4 - 3.3\times$ acceleration versus previous FPGA solutions.

1. Introduction

Memory related issues are currently the main bottleneck for high performance systems [1,2]. Most CPU technology like prefetching, branch prediction, memory disambiguation, pipelining, etc, suffer significant efficiency losses with irregular and unpredictable access patterns. Dynamic Programming (DP) is one of the algorithmic families whose performance degrades due to such addressing. DP is a technique employed in combinatorial optimization where overlapping sub-problems are progressively solved to build the global solution. As stated, its memory access is unpredictable (data dependent) therefore it hardly profits from cache-based systems.

DP algorithms are classified into 4 categories: Serial Monadic (SMDP), Serial Polyadic (SPDP), Non-serial Monadic (NMDP) and Non-serial Polyadic (NPDP) [3]. There are two approaches to implement the DP technique: *bottom-up* or *top-down*. The first one attempts to solve the smallest problems first and stores the results in a matrix (*memoize*) to solve bigger ones afterwards. On the contrary, the *top-down* solution employs a recursive strategy that starts from the biggest problem and go down to the base case to obtain the answer. The work presented in this article belongs to the *bottom-up*, matrix-based approach.

A classic example of a Dynamic Programming application is the Knapsack 0/1 problem which belongs to the SMDP group. It consists in taking a set of items or objects which have a weight (ω_i) and a profit (p_i), to fill a knapsack of a total weight capacity C. The items are taken from a pool of m units (m is an integer bigger than zero) and their sum of weights cannot be bigger than C. Moreover, the optimization objective is to maximize their sum of profits un-

* Corresponding author.
E-mail addresses: fernando.escobar@centralesupelec.fr (F.A. Escobar), filipe.vinci@centralesupelec.fr (F. Vinci Dos Santos).

http://dx.doi.org/10.1016/j.micpro.2017.04.001
0141-9331/© 2017 Elsevier B.V. All rights reserved.

der the previous constraint. A formal definition of the Knapsack 0/1 problem along with its proposed *bottom-up* DP solution is provided in Section 2.

In general terms, the DP algorithm operates by first filling a profit matrix of size $m \times C$ and then backtracking over the values from bottom-right to top-left. This approach has been a subject of study for VLSI and FPGA implementations for many years. Most of the previous solutions have been based on Systolic Arrays with minor variations. For instance, Chen et al. [4] proposed a ring-connected systolic array of q processing elements (PE) for parallel computation of the m rows; however, their solution required full table storage ($m \times C$) and thus more memory units. Afterwards, in a similar work [5], further developed in [6], Andonov et al. presented another architecture with less memory requirements and scalable with respect to m, C and $\omega_{\max}$ for the general Knapsack problem. Interestingly, the scalability with respect to $\omega_{\max}$ was solved by allowing PEs to have a special configuration to propagate data along them if needed. As presented, those architectures were pure systolic arrays that required internal buffers of size $\omega_{\max} = 1023$ on each PE.

Another study proposed a system to solve the 0/1 Knapsack problem using less space [7]. Authors exploited the fact that only a binary matrix called *keep-table* is required to perform the backtracking and introduced it into their systolic array. Moreover, employing a divide-and-conquer algorithm, they processed the whole DP table in strips of $p \times c$ with $p < m$ and $c < C$. Their solution was also limited to problems with $\omega_{\max} = 1023$. Finally, a hypercube architecture based on a scheduling algorithm to propagate the $\omega_{\max}$ through its connections was proposed by gol [8]. Once more, all the previous proposals were limited to problems with small item's weights with $\omega_i < 1024$ or $\omega_{\max} = 1023$.

Systolic arrays have also been the preferred architecture used in other DP problems. In sequence alignment (NMDP) they have been integrated with data compression for higher efficiency as reported in [9,10] and [11]. Moreover, for the Floyd-Warshal algorithm (SPDP), a systolic array composed of B PEs capable of processing matrices of size $B \times B$ ($B = 32$) was proposed by Bondhugula et al. [12,13] for graphs of up to $16,384$ nodes. Lastly, in more complex DP algorithms (NPDP) like the Nussinov algorithm for secondary-structure RNA prediction, Jacob et al. proposed a formal space-time method to develop systolic arrays for sequences of $n = 100$ maximum [14] and $n = 273$ [15] (very small problem sizes). All these efforts impose limitations to the problem instances, like small amount of nodes or short sequences, limiting the advantages of their parallel architectures.

Graphic Processing Units (GPUs) have also been employed to accelerate DP algorithms. For the Knapsack problem, a GPU implementation based on data compression and parallel row computation in each thread was presented in [16]. Reported results were restricted to $\omega_{\max} \leq 1000$ although it is unclear whether it was a platform limitation or not. Similar works on different GPUs have also easily outperformed high-end CPUs by more than 40 × [17,18]. Nevertheless, due to the algorithm low arithmetic complexity, high memory access and conditional execution, GPUs entail a high power consumption with low efficiency (i.e. ratio between the sustained floating point operations per second (FLOPs) over the peak capacity) for the DP technique.

In this article we analyse the memory access of the Knapsack 0/1 algorithm and propose a shared-memory architecture to efficiently process the DP matrix in parallel. Our system can handle problems with bigger weights than previous FPGA solutions by maximizing on-chip storage utilization. We also overlap computation and communication through hardware-managed DMA transfers to maximize parallelism and data exchange throughput. The proposed structure can be synthesized on FPGA or Systems On Chip (SoCs) and is scalable (since more PEs can be added for better performance in bigger devices). It also features a linear runtime with respect to the number of items of the problem instance. We present experimental results for two configurations of the proposed architecture implemented on a ZYNQ-7020 SoC [19]. The first one has 70 PEs, runs at 150 MHz and can solve problems with $\omega_{\max} = 1024$. It attains a 10% speed-up versus a reference GPU implementation [16], 3 × versus previous FPGA solutions [7] and 23 × compared to our sequential, optimized software model. The second configuration runs at 133 MHz and supports $\omega_{\max} = 6144$ (6 × bigger than all previous works) and reaches up to 16 × speed-up against an optimized sequential execution on a Xeon PC. It attains an average of 1.4 × higher performance than previous FPGA solutions and is only 0.25 × slower than the same GPU reference which has 3.3 × more units, 9 × higher frequency and more than 1000 × bigger on-chip memory. The flexibility of our system allows many more configurations to be easily created by changing input parameters and re-synthesizing the architecture.

The rest of the article is divided as follows: In Section 2 we introduce the algorithm and the problem to be solved. Section 3 presents the proposed solution and theoretical analysis to model the system performance and obtain maximum scalability. Then in Section 4 we present the obtained results and analysis to contrast them with similar works. Section 5 concludes the article and summarizes our perspectives for future work.

2. DP algorithm for the Knapsack 0/1 problem

The Knapsack 0/1 problem seeks to choose a set of items from an m-item pool (m is an integer bigger than zero), each one with a profit p_i and weight ω_i to fill a bag (knapsack) of capacity C. Moreover, the selected set of items should maximize the overall profit they provide. More formally the optimization problem is presented in Eq. (1).

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^{m} p_i \cdot x_i \quad | \quad x_i \in \{0,1\} \\ &\text{subject to} && \sum_{i=1}^{m} \omega_i \cdot x_i \leq C \end{aligned} \tag{1}$$

Using the *bottom-up* DP technique, an $m \times C$ profit matrix is constructed since this technique progressively considers all knapsack weights from 0 to C. The matrix is filled adding to the knapsack one item at a time (row by row) and the cells store the accumulated profit of the objects included. More formally, the algorithm fills every cell $M(i, j)$ as follows:

$$M_{(i,j)} = \begin{cases} 0 & \text{if } j = 0 \quad (2) \\ M_{(i-1,j)} & \text{if } j < \omega_i \quad (3) \\ \max(M_{(i-1,j)}, M_{(i-1,j-\omega_i)} + p_i) & \text{if } j \geq \omega_i \quad (4) \end{cases}$$

To compute a matrix cell $M_{(i,\, j)}$, only two cells from the previous row are needed, that is $M_{(i-1,j)}$ and $M_{(i-1,j-\omega_i)}$. This implies that a buffer of at least ω_i words from row $i-1$ is needed to compute the cells of row i. Consider Table 1 where we present the matrix M for a problem of $C = 12$ with $m = 6$ items. The two accesses to compute the circled cells (previously presented in Eq. (4)) are exemplified by the arrows. These values are computed as follows:

- $M(2,1) = \max(M_{(1,1-\omega_1)} + p_1, M_{(1,1)}) = \max(M_{(1,0)} + 8, 0) = 8$
- $M(3,8) = \max(M_{(2,8-\omega_3)} + p_3, M_{(2,8)}) = \max(M_{(2,3)} + 5, 21) = 26$
- $M(6,12) = \max(M_{(5,12-\omega_6)} + p_6, M_{(5,12)}) = \max(M_{(5,8)} + 6, 38) = 44$

After the profit matrix M is filled, the maximum profit attainable can be found in the bottom-right position $M_{(m,\, C)}$. Afterwards, a backtracking from $M_{(m,\, C)}$ to $M_{(0,\, 0)}$ is carried on to determine

Table 1
Profit table for a knapsack of capacity $C = 12$ and $m = 6$ items. The maximum profit possible is 44 (bottom-right value). The arrows show some dependencies required to compute the circled cells. At every row i, each cell (i, j) is computed as the maximum between the cell above $(i-1, j)$ and the value ω_i columns to the left $(i-1, j-\omega_i)$ added to the item profit p_i.

Item \ Kn.Weight	0	1	2	3	4	5	6	7	8	9	10	11	12
A($\omega_1 = 2$, $p_1 = 13$)	0	0	13	13	13	13	13	13	13	13	13	13	13
B ($\omega_2 = 1$, $p_2 = 8$)	0	8	13	21	21	21	21	21	21	21	21	21	21
C ($\omega_3 = 5$, $p_3 = 5$)	0	8	13	21	21	21	21	21	26	26	26	26	26
D ($\omega_4 = 3$, $p_4 = 17$)	0	8	13	21	25	30	38	38	38	38	38	43	43
E ($\omega_5 = 7$, $p_5 = 9$)	0	8	13	21	25	30	38	38	38	38	38	43	43
F ($\omega_6 = 4$, $p_6 = 6$)	0	8	13	21	25	30	38	38	38	38	44	44	44

$-\omega_2$ $-\omega_3$ $-\omega_6$

Table 2
Keep-table for the proposed example where the backtracking determines the items to include. In this example items A, B, D and E provide a 44 profit.

Item \ Kn.Weight	0	1	2	3	4	5	6	7	8	9	10	11	12
A ($\omega_1 = 2$)	0	0	1	1	1	1	1	1	1	1	1	1	1
B ($\omega_2 = 1$)	0	1	0	1	1	1	1	1	1	1	1	1	1
C ($\omega_3 = 5$)	0	0	0	0	0	0	0	0	1	1	1	1	1
D ($\omega_4 = 3$)	0	0	0	0	1	1	1	1	1	1	1	1	1
E ($\omega_5 = 7$)	0	0	0	0	0	0	0	0	0	0	0	0	0
F ($\omega_6 = 4$)	0	0	0	0	0	0	0	0	0	0	1	1	1

$-\omega_2$ $-\omega_4$ $-\omega_6$

the items to be included on the knapsack. However, an alternate approach can be used to perform the backtracking and avoid storing the profit matrix M. In fact, the backtracking can be done on a *keep or leave* binary table (K) which is simultaneously computed with M as shown in Eq. (5). This way, at each iteration i, the DP algorithm only needs two profit matrix lines (M_{i-1} and M_i) and can store the (much smaller) keep-table K.

$$K_{(i,j)} = \begin{cases} 1 & \text{if} \quad M_{(i-1,j-\omega_i)} + p_i \geq M_{(i-1,j)} & (5) \\ 0 & \text{otherwise} & (6) \end{cases}$$

For the current example, K is presented in Table 2. Table K is d_{size}-times smaller than the profit table M for d_{size}-bit profit values p_i. For example, using 32-bit data, K is 32 × smaller than M. In general, due to the irregular addressing imposed by Eq. (4) and the profit matrix M size ($m \times C$), most of the time is spent filling K and thus we focus on this step. Since traditional, cache-based, architectures are designed to exploit data locality, irregular accesses deteriorate performance due to the increased data exchange with off-chip storage.

In the following section we explain the row memory dependencies and the strategy adopted to parallel the computations described by Eq. (4).

3. Proposed architecture

Given the limitations of the previous related works, namely, forcing $\omega_{\max} < 1024$, in this section we provide the reasoning followed to develop our proposal. Afterwards, we explain the different components of our architecture and examine theoretical aspects regarding performance and scalability.

3.1. Global data dependency

To compute $M_{(i,j)}$, only $M_{(i-1,j)}$ and $M_{(i-1,j-\omega_i)}$ are required. This means that a value located ω_i columns to the left of the current column j must be retrieved to fill the current cell $M_{(i,j)}$. Since ω_i is a problem input data, such access is unpredictable and thus has been limited to a *predictable range*. In the cited works that use systolic arrays, each PE computes a single row and has a local buffer that stores $\omega_{\max} = 1024$ words from the previous row [6,7]. In other words, these proposals implement inter-row parallelism and make the *predictable range* equal to 1024. If problems with $\omega_{\max} = x \cdot 1024$ need to be solved (x integer bigger than zero), these architectures should increase x times their buffer sizes, thus augmenting their on-chip storage by the same proportion.

To address such a drawback in a more space-efficient way, intra-row parallelism is also possible, for example, as implemented on GPU threads by boy [16]. Rows can be split into separate seg-

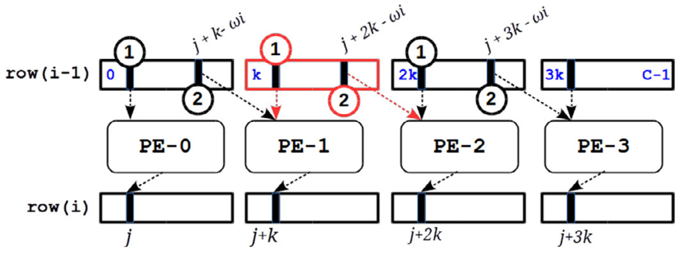


Fig. 1. Dependencies between columns of the previous and current row for PEs that compute k columns each. If each segment is on a physically separated memory, at most 2 PEs read from the same one when they are synchronized. A predictable access (1) and an unpredictable one (2) occur due to input data ω_i.

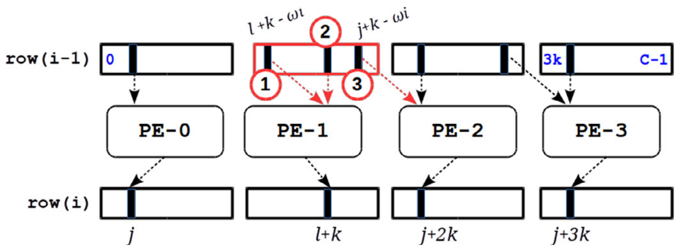


Fig. 2. Unsynchronised PEs lead to 3 simultaneous reads to each k-column segment. In this case PE-1 is not synchronized with the remaining units hence generating 2 concurrent access to the second memory for a total of 3 readings to the same segment.

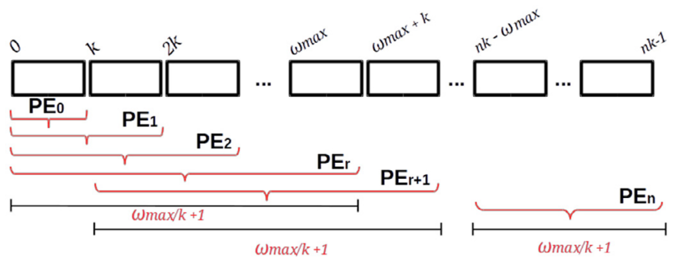


Fig. 3. The reading scope of each PE increases to the right of the table. If $r = \omega_{\max}/k$, from the rth PE, a total space of $r+1$ memories are shared by the subsequent PEs.

ments (memories) of k columns that are concurrently written by a set of PEs. If they work synchronously, that is, all start and finish at the same time, at most two simultaneous reads to the same segment will occur as shown in Fig. 1. The shaded regions denote the read cells $(i-1, j)$ (numbered 1) and $(i-1, j-\omega_i)$ (numbered 2) of PEs 1–3 for a given ω_i. When column j in row i is being computed on PE-0, column $j+k$ is computed on PE-1, $j+2k$ on PE-2 and so on. A total of two concurrent accesses per k-column segment would be required in order to implement a shared memory system with this processing strategy.

If PEs worked asynchronously, that is if they started at different times, there could be up to 3 concurrent read accesses to each k-column segment as shown in Fig. 2. The segment $[k, 2k]$ is accessed by PE-1 with two reads and by PE-2 with one read.

Since the position $j-\omega_i$ is unpredictable, PEs that process any cell j (from row i) must have access to at most $\omega_{\max}$ columns to its left, that is the range $[j-\omega_{\max}, j]$ (from row $i-1$). In the example shown in Table 1, only 1 column to the left was required to compute the cells from row 2. On the contrary, to compute row $i=3$, a total of 5 columns to the left of each cell $(3, j)$, $j \geq 5$ were needed. Thus, the bigger ω_i, the longer the jump. More generally, in Fig. 3 we present a graphic description of the just described dependencies. In this case the PEs reading range increases until it reaches $r+1$ memories, where $r = \omega_{\max}/k$, (r is any integer bigger than zero). This implies that a single memory must be shared at most among $r+1$ PEs, a fact that will be exploited in our architectural development.

In summary, three facts must be taken into account to develop a shared-memory architecture for this DP algorithm:

1. Only the profit table row $M_{(i-1)}$ is needed to produce the keep-table row $K_{(i)}$.
2. PEs must work in a synchronized way to minimize concurrent access to k-column segments which are to be implemented as separate memory units.
3. The number of shared memories is directly related to $\omega_{\max}$.

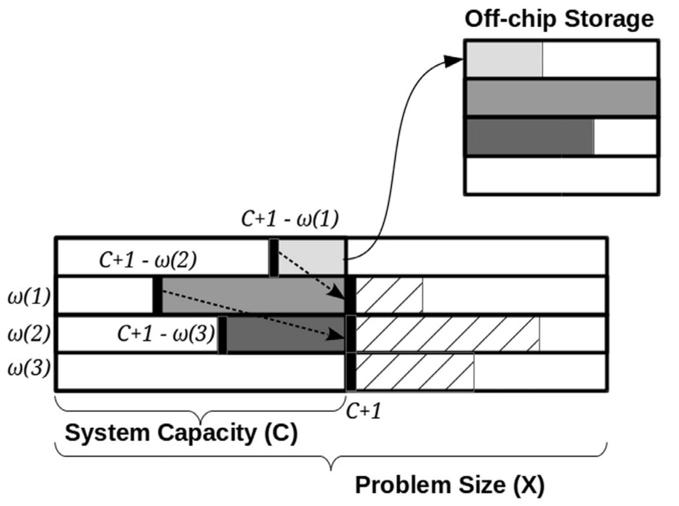


Fig. 4. When the profit matrix M has more columns X that surpass a system capacity C, additional off-chip space can be used as temporal buffer. For each row i, ω_i values from row $i-1$ produced during the computation of columns $[0-C]$, will be needed to process columns $[C+1, X]$.

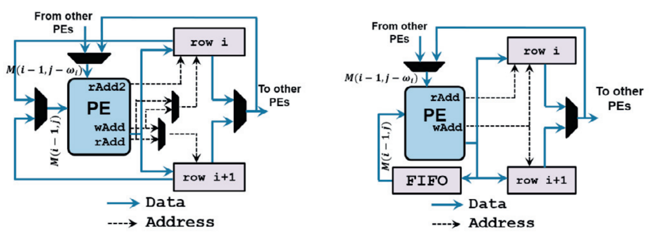
On the other hand, certain problem instances may require processing a profit table M that surpasses the system capacity. In general, a system with n PEs that compute k columns will attain a processing capacity equal to $C = nk$. Therefore, to solve bigger problems, extra off-chip buffering can be used to split the computations. Consider a problem with a knapsack of capacity $X > C$ and $m = 4$ items. The resulting matrix M for this problem has a dimension of $4 \times X$ as shown in Fig. 4. The matrix can be first processed from top to bottom for columns 0 to C. During this phase, for each row i processed, their last ω_{i+1} words will be transferred to off-chip storage. After all lines from the first $0-C$ columns have been processed, the matrix will be processed again from row 0 but now for columns $C+1$ to X. Ideally before each iteration i, the system would pre-fetch the corresponding ω_i values automatically to compute the new row columns.

With this procedure, any problem size can be treated as long as there is enough off-chip storage space, that is, $m \times \omega_{avg}$, where $\omega_{avg} = E[\vec{\omega}]$ (E is the expected value function). For uniformly distributed weights ω_i on the interval $[0, \omega_{\max}]$, $\omega_{avg} = \omega_{\max}/2$.

Summarizing, continuous data exchange with off-chip storage is required to make the system capable of computing profit matrices M that exceed the system capacity C. This solution calls for Direct Memory Access (DMA) capabilities to perform pre-fetching and minimize stalls.

3.2. Knapsack solver architecture

The two basic techniques used in the cited works [7,16], i.e. the two-line on-chip buffering and keep-table computation for backtracking, are used in this work. We extend previous FPGA solutions



(a) Implemented with 2 address and 1 data multiplexer.

(b) Implemented with FIFO for less routing congestion.

Fig. 5. Interleaved access on local PE. Both upper and lower memories act as rows i or $i + 1$ at different times. Option (b) is chosen due to its reduced logic and routing complexity.

by proposing a shared-memory system (similar to the GPU architecture) with several parallel units that feature coalesced readings and exclusive writings. As it will be shown later in the text, this configuration allows solving problems with bigger ω_i compared to traditional systolic arrays. We also propose to automate data transfers through hardware-controlled DMA units that require minimum configuration.

Following, we present and describe the details of the proposed architecture in the following order:

- **Data sharing between PEs**: Explains the interconnection between the system memory units and the processing logic.
- **Processing architecture**: Describes the main blocks composing the knapsack solver along with their main tasks.

3.2.1. Data sharing between PEs

PEs compute the row values using Eq.(4) on a k column segment. Each k-column row segment shown in Fig. 1 has two simultaneous accesses: one from the local PE and one from the r neighbouring ones. The hardware implementation of these row segments are equivalent to dual port memories. Moreover, due to the two-line requirement explained in Section 3.1, two dual port memories are needed for each k-column segment.

Consider Fig. 5a where a possible architecture that implements the memory access shown in Fig. 1 is presented. The PE produces the values $M_{(i,\,k)}$ to $M_{(i,j+k)}$ and writes them into its local memory which is the upper one at one iteration, the lower one at the next one and so on. The PE gets the value $M_{(i-1,j-\omega_i)}$ from its local k-column segment or from other PE memories (Fig. 1) depending on the value ω_i.

On the contrary, the value $M_{(i-1,j)}$ is always read from its local segment (access “1” Fig. 1) thus we studied two implementations: In Fig. 5a two multiplexers are used to alternate between reading and writing addresses to each storage unit. An additional multiplexer is used to choose the corresponding data output from the two-line row segments (upper and lower memories) due to their alternating behaviour. A second configuration presented in Fig. 5b reduces the logic and routing complexity by using a FIFO instead of the multiplexers. Interestingly, FIFOs are very useful since access numbered “1” in Fig. 1 is sequential. Considering that our shared memory structure will necessarily include several multiplexers to choose between many memory segments, we select configuration Fig. 5b for our implementation.

3.2.2. Processing architecture

The system’s remaining blocks have been added to maximize data throughput to off-chip memory whilst performing additional computations. Since big knapsacks ($X > C$) require intensive data transfers with main memory (Fig. 4) we included a second 2-line profit memory (per PE) for off-chip profit values transmission exclusively. This way, data can be sent off chip by a separate module without stalling PEs by accessing the same memory units.

To maximize throughput we used asymmetric port-width memories so that PEs write 32-bit values and transmission units do 64-bit readings, hence, employing half the time. In addition, the second transfer-intensive procedure, that is the keep-table transmission from the FPGA to main memory (DDR), is also optimized with asymmetric port-width units. Notice that since columns are processed in parallel, keep-table bits are produced in a non-sequential way: bits 0, k, $2k$, etc. are produced at t_0 while bits 1, $k+1$, $2k+1$, etc. are produced at t_1 and so on. A special handling is required to store data in an ordered way to ease the transmission procedure.

To illustrate our knapsack solver architecture, we depict a block diagram in Fig. 6 of a system with n PEs capable of solving problems with $\omega_{\max} = r \cdot k$, where k is the number of columns processed by each PE and r is any integer bigger than zero. Note that $\omega_{\max}$ depends on the input data and thus it changes from problem to problem, therefore, the system must be synthesized to support a specific maximum weight. In this section we describe the system structure for a general case with $r > 0$ but later in Section 4 we show results valid for two different architectures that support $r = 2$ and $r = 12$.

Without loss of generality, at this point k represents an arbitrary number of columns. In the figure, Processing Elements (PE0-PEn) produce profit values that are shared with others through the Profit Table memories (PT0_1-PTn_1) as previously shown in Fig. 5b. The second group of Profit Table memories (PT$(n-r)_2$-PTn_2) are the ones used to transfer data off-chip and they have 64-bit outputs to minimize transfer times. Keep-table memories (KT$_0$-KT$_{n/32}$) are 32 × smaller than profit ones when using 32-bit data for profit values. Therefore one KT is used every 32 PEs and they use 64-bit outputs to minimize transfer times. In this case, the *keep-leave* bit (Eq.(5)) sent by each PE, feeds a serial-to-parallel shift register (SR) whose output is written to the KT memory.

Input FIFOs are added to store profit values retrieved from off-chip storage when solving problems where $X > C$ as shown in Fig. 4. Other units composing the system are pipelined multiplexers that receive memory segment values and select the appropriate input for the PE. Notice the heavily pipelined interconnect structure between every component to maximize the processing frequency.

As stated in [20], interconnection complexity and congestion are key issues for shared-memory architectures. We seek to remove these obstacles with PE synchronization: Notice from Fig. 1 (synchronized scenario) that the addresses generated by PEs are shifted by multiples of k, that is: $j + k - \omega_i$, $j + 2k - \omega_i$, and so on. These addresses are however translated into local memory address $j - \omega_i$ on separated memories of size k. This way, PE1 reads the position $j - \omega_i$ from memory 0 (PT0_1), PE2 reads the same position but from memory 1 (PT1_1) and so on. It thus suffices to connect the read address generated by each PEx to their memory PTx_1 and only propagate each memory output to the remaining PEs.

To further reduce space and routing congestion, the system can be synthesized to support a maximum weight $\omega_{\max}$. In this case, FIFOs and transmission memories PTx_2 will be created only for the necessary PEs. The system in Fig. 6 is synthesized for $\omega_{\max} = rk$ (r ∈ positive integers), then PE$_0$-PEr will have input FIFOs since they compute the first rk columns of the row. PEr will get the value $j - \omega_{\max}$ from column 0 from PT0_1. Equally, only PE$n-r$-PEn will have transmission Profit Tables (PT$(n-r)_2$-PTn_2) as they compute the last rk columns (Fig. 3). As aforementioned, this set of memories feature 64-bit outputs and are used for data transmission only.

The system diagram also contains three Finite State Machines (FSMs), in charge of controlling data transmission, reception and PE synchronized execution. Finally, an additional 32-bit input is used to receive the item’s weight (ω_i) and profit (p_i) values, which

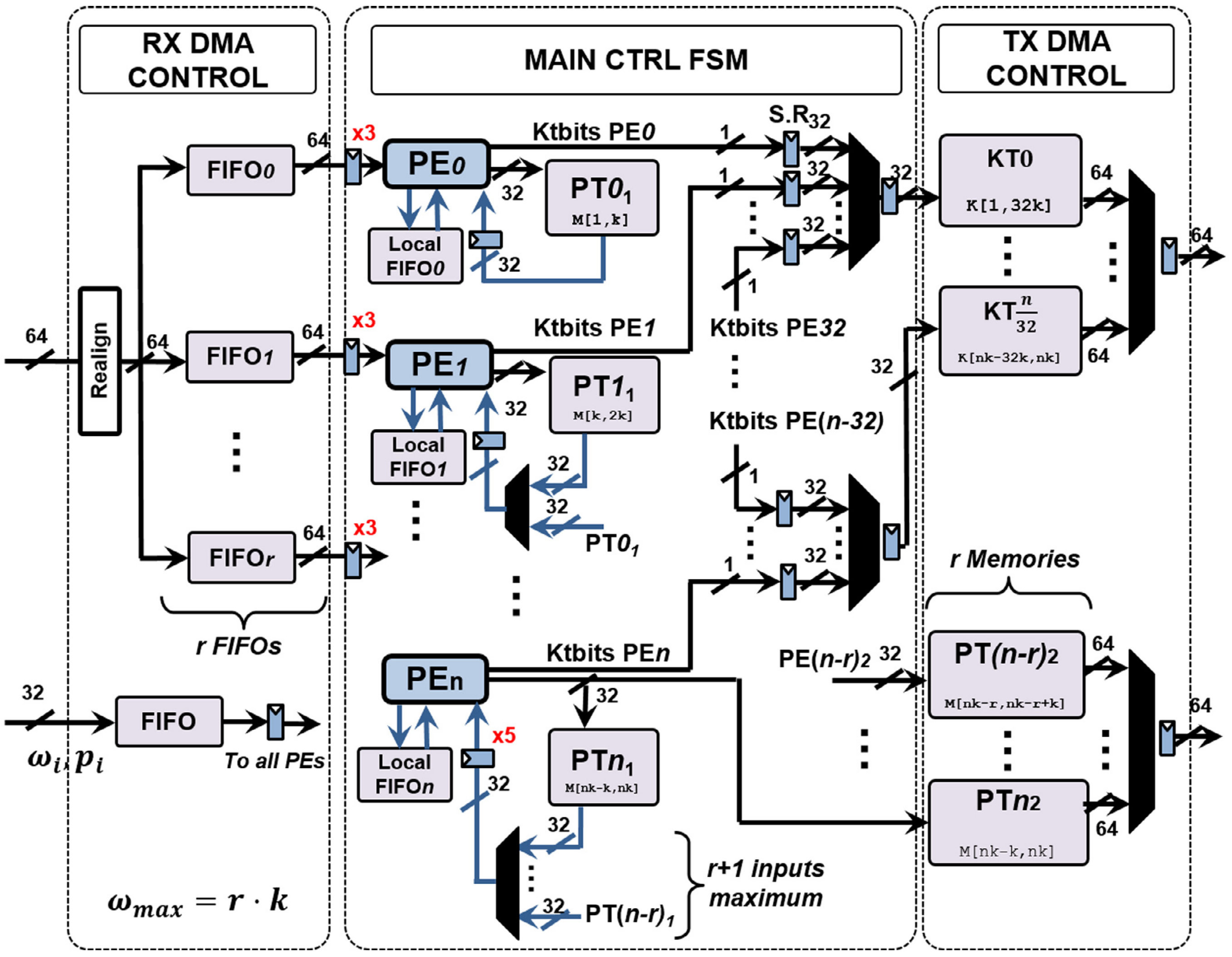


Fig. 6. Knapsack solver structure for parallel column processing with n PEs. Each processing element (PE_x) writes the same data to shared memory (PTx_1) and to a transmission one (PTx_2). Pipelined interconnects are represented with the smallest boxes with a numeral indicating the number of stages (x3,x5). They are used to maximize the processing frequency.

are stored in a small FIFO. In this case a 32-bit wide port suffices as they are retrieved once per row processed, thus, with sufficient time to be pre-fetched.

In this section we described in detail the strategy to solve the DP algorithm along with each of the units composing our architecture. We now present theoretical analysis regarding its size and expected performance with respect to resource consumption requirements.

3.3. System size and performance analysis

To estimate the system performance we modelled the four procedures that can occur in parallel. In general, for an architecture composed of n PEs of k size, thus capacity $C = nk$, we provide Eqs. (7)–(9):

- t_1 (Eq. (7)): One C-bit *keep-row* needs to be stored off-chip after an item is processed. Keep-table bits are stored as 32-bit words and read as 64-bit values for off-chip transmission. Keep-table size is then $C/32$ but at every clock cycle 2×32-bit words, that is 64 bits, are read. For this reason, the whole transmission will be done in $C/32/2 = nk/64$ clock cycles.
- t_2 (Eq. (8)): When the problem size X exceeds the system capacity C, for uniformly distributed weights, on average $\omega_{avg} \cdot$ 32-bit words must be sent off-chip per item processed as explained before. Again, since PTx_2 memories use 64-bit outputs, this takes $\omega_{avg}/2$ clock cycles.
- t_2 (Eq. (8)): For the same case ($X > C$) on average, ω_{avg} words must be retrieved at every iteration to fill the input FIFOs. The same clock cycles are used.
- t_3 (Eq. (9)): Finally, the PE latency has been established to be $k + 12$ clock cycles (caused by pipelining). This is the time it takes to produce all row values.

$$t_1 = m \cdot \frac{\left(\frac{C}{32}\right)cycles}{2 \cdot f_{clk}} \tag{7}$$

$$t_2 = m \cdot \frac{(\omega_{avg})cycles}{2 \cdot f_{clk}} \tag{8}$$

$$t_3 = m \cdot \frac{(k + 12)cycles}{f_{clk}} \tag{9}$$

First we establish the value k. We seek to maximize the system processing capacity C so that fewer iterations are required when the problem capacity X is bigger than C. This is accomplished

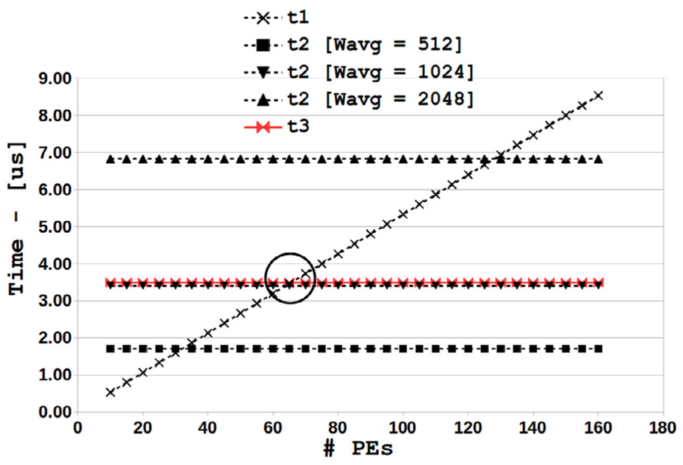


Fig. 7. Time estimations at 150 MHz when the PE size is $k = 512$. For $\omega_{avg} \leq 1024$, data transfers affect the system runtime after 65 PEs where transmission times surpass the PE computing time. Bigger ω_{avg} significantly increases the global execution time due to the increment of data exchange with main memory.

through an optimum use of fabric memories which for Xilinx FPGAs is presented in units as small as 18 Kbit (2KB). Under this restriction, PE size k should be a multiple of 512 since 512×32-bits equals 2KB. Any value $k < 512$ would result in underutilized BRAM space, hence, reducing the system capacity C.

The bigger the value k, the fewer the amount n of PEs required to reach any capacity C. In addition, bigger k lead to higher row processing times ($k + 12$ cycles). We thus assume that, under no bandwidth restrictions, the running time-per-row can be roughly computed as the slowest value among all parallel procedures as shown in Eq. (10).

$$ExecTime_{ROW} = \max(t_1, t_2, t_3) \tag{10}$$

In Fig. 7 we plot the previous equations for $f_{clk} = 150$ MHz which is a typical working frequency on FPGAs. The comparison presented still holds for other frequencies since Eqs. (7)–(9) are all divided by f_{clk}. We also vary the value $\omega_{avg} = 0.5 \cdot \omega_{\max}$ to analyse its scalability. The lines show that when the system size increases, the execution time is determined by t_1 since more cycles are required to transfer the whole keep-table. For $\omega_{avg} \leq 1024$ (or $\omega_{\max} \leq 2048$), the architecture is scalable with no timing penalties up to about 65 PEs.

To estimate the number of PEs n that can fit on a device like an FPGA or SoC, we compute the number of memory units consumed by the architecture. As a matter of fact, PEs do not perform complex arithmetic computations and logic resources are much more abundant than memory units on these devices. As it will be shown in the following section, the system limiting factor is memory. Based on the diagram presented in Fig. 6, we have that:

- Each PE requires a k-word FIFO and a $2 \times k$-word memory to process the matrix (Fig. 5b).
- A $2k$-word memory every 32-PEs is needed for keep-table storage.
- A total of $r = \omega_{\max}/k$ input FIFOs are used.
- A total of $r = \omega_{\max}/k$ transmission memories (PTx$_2$) are used transmission profit values.

More formally, this can be expressed as:

$$m_{18Kb} = n + 2n + 2 \cdot \lceil \frac{n}{32} \rceil + 4 \cdot \frac{\omega_{max}}{k} \tag{11}$$

The last two terms (keep-table (KT), input FIFO and PTX$_2$), are doubled because memories with 64-bit ports require $2 \times$ 18Kb units (on Xilinx devices). From synthesis metrics we know the AXI DataMovers consume between 13 and 15 units for 7-series FPGAs and SoCs.

For instance, on a low-end Xilinx ZYNQ-7020 device, Eq. (11) states that it would be possible to synthesize $n = [76, 86]$ PEs when $\omega_{\max} = [1024, 8192]$. However, notice from Fig. 7 that when $\omega_{avg} = 512$ and $\omega_{avg} = 1024$, there is about a 30% runtime penalty for $n = 86$ compared to when $n = 65$, which is the upper limit to avoid timing losses due to data transfers. Considering the previous analysis and following the discussion from Section 3.2, we conclude that the overall runtime will be a function of (1) the system capacity $C = nk$, (2) the problem instance capacity X, (3) the number of items m, (4) the average item weight ω_{avg} and (5) the data transfer penalty when $n > 65$ PEs. Formally, this quantity can be expressed as:

$$RT = m \cdot \left(\frac{X}{C}\right) \cdot \max(t_1, t_2, t_3) \tag{12}$$

Thus, considering the scenarios shown in Fig. 7, we can model the runtime (Eq. (12)) as:

$$RT = \begin{cases} m \cdot \left(\frac{X}{C}\right) \cdot t_3 & \text{if} \quad n \leq 65, \omega_{avg} \leq 1024 \quad (13) \\ m \cdot \left(\frac{X}{C}\right) \cdot t_1 & \text{if} \quad n > 65, \omega_{avg} \leq 1024 \quad (14) \\ m \cdot \left(\frac{X}{C}\right) \cdot t_2 & \text{if} \quad \omega_{avg} > 1024 \quad (15) \end{cases}$$

Taking into account Eqs. (7)–(9), the runtime for a system with n PEs solving a knapsack instance problem of capacity X, with m items whose average weight is ω_{avg}, presents the following behaviour:

$$RT = \begin{cases} \Theta\left(\frac{mX}{n}\right) & \text{if} \quad n \leq 65, \omega_{avg} \leq 1024 \quad (16) \\ \Theta(mX) & \text{if} \quad n > 65, \omega_{avg} \leq 1024 \quad (17) \\ \Theta\left(\frac{mX\omega_{avg}}{n}\right) & \text{if} \quad \omega_{avg} > 1024 \quad (18) \end{cases}$$

Compared to all the previous referenced works that include FPGA technology, our system runtime is affected by ω_{avg}, whenever $\omega_{avg} \geq 1024$ (or $\omega_{\max} \geq 2048$ for uniformly distributed weights) as shown in Eq. (18). However, unlike them, this architecture is able to solve problem instances with a wider range of ω_i. Strictly comparing the kind of problems that can be solved by the previous architectures, that is when $\omega_{\max} \leq 1023$, our runtime is scalable with the number of PEs (Eq. (16)) up to a limit. When $n > 65$, it is affected by the system size due to the keep-table transfers to off-chip memory (Eq. (17)). We thus propose a chained configuration (similar to a systolic array) of several knapsack solvers to reduce the impact of data transfer times t_1 and t_2.

3.4. Chained solvers configuration

Taking into account the timing penalties after $n = 65$ PEs, a chained structure can be more efficient to reach better runtimes. On a single-solver system, the overall runtime is given by Eq. (10). Now consider Fig. 8 where two knapsack solvers are chained. The first block computes columns $[0, C]$ and the second one $[C + 1, P]$, where $(P - C) \leq C$. This configuration is chosen whenever the cost of transferring data from solver-1 to solver-2 is less than the runtime of a single solver with more than 65 PEs, that is when $t_2 \leq t_1$.

Notice from Fig. 7 that $t_2 \leq t_1$ will depend on the value ω_{avg}. For instance if $\omega_{avg} = 512$, this equation is satisfied when $n = 32$. In other words, whenever there is FPGA space to synthesize $n > 65 + 32 = 97$ PEs, a single-solver system should be split into a chained configuration where every solver should have $n \leq 97$ PEs. Following the same reasoning, for a system solving problems with an $\omega_{avg} = 1024$, the condition $t_2 \leq t_1$ is satisfied for $n = 65$. This means that solvers should be chained in blocks of maximum size $n = 65 + 65 = 130$ PEs. Similar analysis can be done for other ω_{avg}.

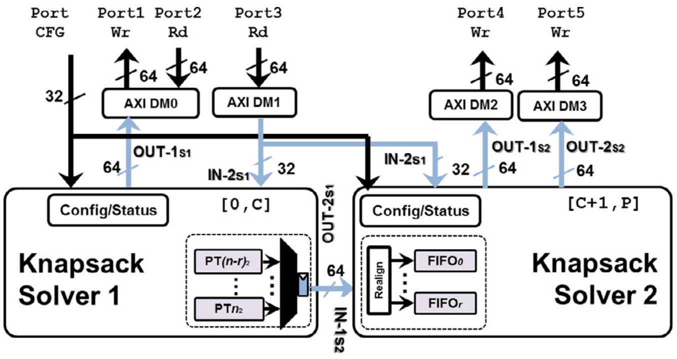


Fig. 8. Two knapsack solvers can be chained to overcome sequential data transfers and improve the runtime on bigger test devices. An additional transfer port per solver added is necessary.

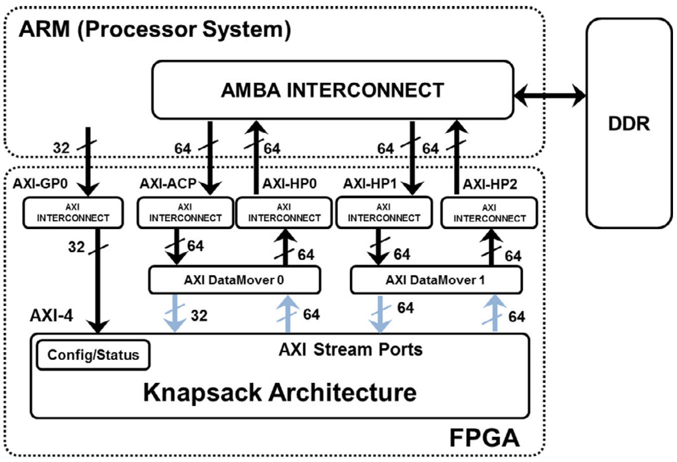


Fig. 9. System block diagram using the Zynq-7000 SoC from Xilinx. DataMover transfers use the AXI Stream (AXIs) interface and are fully controlled by the architecture.

Note however that this increment also requires an additional access port to main memory since each additional solver must transfer a portion of the keep-table.

According to Fig. 6 our system uses four 64-bit ports to send and retrieve data. To see the limits of our architecture on a target device, consider the ZYNQ architecture from Xilinx [19] which has five 64-bit ports plus two 32-bit ports. Assuming each port is used in a single direction (read-/write-only) to reduce routing congestion, only three additional solvers can be chained as there are three free ports. For instance, if $\omega_{avg} \leq 512$, we can synthesize up to 4 chained solvers with $n_{1,2} = 97$ PEs, $n_{3,4} = 48$ PEs, and a total capacity $C = n_{total} \cdot 512 = 292 \cdot 512 = 149504$. Keep in mind that the capacity of the two latter is half since their transfer time is longer as they use 32-bit ports. The same straightforward analysis can be applied to other ω_{avg} thus we skip it.

In the following section we present the measured results after implementing the system on a ZedBoard card that includes the ZYNQ-7020 chip.

4. Implementation results and discussion

Real world gains of our proposed knapsack solver architecture ultimately depend on matching the functional units to the target platform. Our solver is intended to provide power-efficient hardware acceleration (co-processing) on embedded computing systems. After examining the platforms available, we selected a typical ARM-based, low-cost, low-power SoC to validate the architecture functionality and measure its performance: the ZYNQ-7020 (Z-7020) SoC from Xilinx [19]. The Z-7020 is composed of a dual core ARM A9 processor and Artix FPGA fabric. For this purpose we utilize AVNET's ZedBoard [21] which includes a Z-7020 chip, 512MB DDR3 and an interfacing serial port, among other peripherals. To connect the architecture data exchange ports with main memory, we use two Xilinx AXI DataMovers [22] that transform AXI stream transactions into memory mapped ones.

We present our proposed embedded system structure in Fig. 9. As previously mentioned, to alleviate routing congestion we use each port on a single direction (read-only or write-only) as follows:

- CPU-to-FPGA communication is performed through the AXI General Purpose (AXI GP0) port for IP configuration which includes knapsack size, number of items, buffer addresses to store/retrieve information and status monitoring.
- The AXI Accelerator Coherency Port (AXI ACP) is used to retrieve items information (ω_i, p_i).
- The AXI HP0 is employed to send the *keep-table* to the external memory.
- The AXI HP1 is used to retrieve profit values when the problem capacity is bigger than the system one ($X > C$) as previously explained.
- The AXI HP2 is used to send the profit values to main memory when $X > C$ as shown in Fig. 4.

Notice that keep and profit matrix transfers are done in 64-bits but the items information (ω_i, p_i) is received in 32-bit words. This decision is taken because item's data is continuously pre-fetched with enough time in advance so it does not affect the performance.

We synthesized two system configurations to analyse the knapsack solver performance. First, a base configuration (KP-1024) capable of solving problems with $\omega_{\max} = 1024$ was tested and compared with previous works. The second one (KP-6144) takes full advantage of our conceptual architecture in the Z-7020 and it is able to treat problems with $\omega_{\max} = 6144$, by exploiting the shared-memory structure proposed.

4.1. KP-1024 Solver

In the first system a total of $n = 70$ PEs of $k = 512$ columns and thus capacity $C = nk = 35,840$ were implemented on the FPGA. According to Fig. 3 this implies that PEs require access to at most $r + 1 = \omega_{\max}/k + 1 = 3$ BRAMs, or in other words, a single BRAM is shared between 3 PEs at most. The structure presented in Fig. 6 is replicated from PE-0 to PE-69 but note that only the first ($r = 2$) PEs (PE-0, PE-1) include input FIFOs and only the last two ones (PE-68, PE-69) include transmission memories ($PT68_2$, $PT69_2$).

Since there are 280 × 18Kb units on the Z-7020, a total of 86 PEs could be synthesized according to Eq. (11) for $\omega_{\max} = 1024$. However, to operate at the maximum frequency allowed for AXI Datamovers (150 MHz), we only managed to include $n = 70$ PEs and reached a 90% BRAM utilization due to congestion and timing issues.

Using embedded hardware counters and the ARM processing system timers we established the execution times for randomly generated correlated problems where:

- $\omega_i, i \in \{1, \dots m\}$ randomly drawn in [1..1024].
- $p_i = \omega_i + 100, i \in \{1, \dots m\}$.
- $X = \{C, 2C, 3C, 4C\}$, with $C = nk = 35,840$.

Tests were carried on with $m = [1000 - 4000]$ items due to the platform space constrains (DDR= 512MB). From this values we obtained an execution time model using the runtime obtained with

Table 3
Area consumption for the main parts of the whole embedded system described in Fig. 9 on a ZYNQ-7020 device.

Block	LUT	LUTRAM	Registers	BRAM
AXI Datamover 0	1795 [3.4%]	17 [0.1%]	2631 [2.4%]	8.5 [6%]
AXI Datamover 1	1349 [2.5%]	17 [0.1%]	2087 [1.9%]	4 [2.8%]
Single PE	286 [0.1%]	0 [0%]	383 [0.3%]	1.5 [1%]
Knapsack Arch.	33,568 [63.1%]	676 [3.8%]	64,689 [60.8%]	114 [81.4%]
Full System	**37,502 [70.5%]**	**739 [4.3%]**	**70,344 [66.1%]**	**126.5 [90.4%]**

$m = 1000$ items which was 3580 µs. We present Eqs. (19) and (20) to generalize the system execution time.

$$t_{exe} = \left[\left(\frac{m}{1000}\right) \cdot \lceil \frac{X}{C} \rceil \cdot t_{1000}\right] \mu s \quad (19)$$

$$t_{exe} = \left[\left(\frac{m}{1000}\right) \cdot \lceil \frac{X}{35840} \rceil \cdot 3850\right] \mu s \quad (20)$$

The runtime obtained is linear with respect to the number of items m and increases in multiples of C. This can be understood by considering the procedure explained in Fig. 4. When the problem size X is bigger than the system capacity C, the table must be processed from top to bottom for the new columns $C - 2C$, $2C - 3C$, etc.

In the following subsection we present a more flexible system that can solve problems where $\omega_i > 1024$. Comparisons with similar works are also included for further analysis.

4.2. KP-6144 Solver

The aim of the second system was to solve knapsacks with ω_i values which exceeded those handled by systolic arrays as reported in Section 1. Using the comparatively modest resources offered by the Z-7020, we managed to include 58 PEs obtaining a capacity $C = nk = 29,696$ and supporting weights up to $\omega_{\max} = 6144$. Unlike the previous case, the highest frequency reached for this system was 133 MHz which can be explained due to the increased routing complexity. As a matter of fact, for this case each BRAM is shared between at most $r + 1 = \omega_{\max}/k + 1 = 13$ PEs creating a much denser data path network. Moreover, in this case the first $r = 12$ PEs (PE-0 to PE-11) include input FIFOs and the last 12 ones (PE-45 to PE-57) have transmission memories ($PT45_2$ to $PT57_2$). The global (post-implementation) resource consumption is presented in Table 3 and differs from the previously shown system by less than 5%.

From Table 3 we notice that the limiting factor is BRAM availability which is approximately 90%. Apart from it, LUT and register consumption also limited the realization of a bigger system on our test device. On the contrary, LUTRAM and DSP (zero usage) consumption is negligible.

The same tests performed in the previous case were executed for this architecture. In addition, we used $\omega_{\max} = \{1024, 2048, 3072, 4096, 5120, 6144\}$ to measure the performance degradation due to longer transfers. As in the previous case our experimental results showed a linear relation between execution time and number of items m. Nonetheless, the linearity ratio increased with ω_{avg} because bigger weights imply higher off-chip data exchange and more stalls are produced by DDR issues (bank conflicts). We plotted on Fig. 10 the metrics retrieved after the benchmark execution. Dedicated counters were included to discriminate stall times where the system is only transmitting or receiving data.

From the plotted results we derived a model to establish the system behaviour on problems any size X, with any number of items m and with any $\omega_{\max} \le 6144$. The system running time is

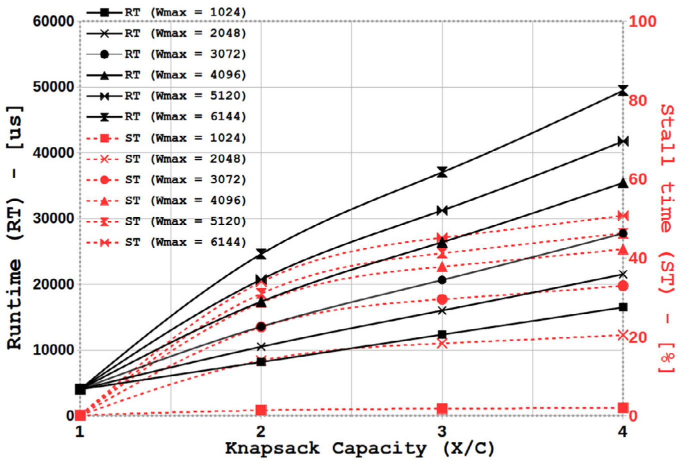


Fig. 10. Runtime (RT) for knapsack sizes $[C - 4C]$ ($C = 29,696$) with 1000 items and different $\omega_{\max}$. Stall percentages (ST) due to transfers increase consume up to 50% of the runtime when $\omega_{\max} = 6144$.

described by Eq. (22) which can be considered an extension of Eq. (20):

$$t_{exe} = \left[\left(\frac{m}{1000}\right) \cdot \lceil \frac{X}{C} \rceil \cdot t_{1000}\right] \cdot \left[1 + \left(0.35 \cdot \left(\frac{\omega_{max}}{1024} - 1\right)\right)\right] \mu s \quad (21)$$

$$t_{exe} = \left[\left(\frac{m}{1000}\right) \cdot \lceil \frac{X}{29696} \rceil \cdot 4049\right] \cdot \left[1 + \left(0.35 \cdot \left(\frac{\omega_{max}}{1024} - 1\right)\right)\right] \mu s \quad (22)$$

As expected from Fig. 7, there are no timing losses for $\omega_{\max} = 1024$ or $\omega_{avg} = 512$ for uniformly distributed values, but stalls appear for bigger weights. These are modelled as the second term in Eq. (22). Contrary to Fig. 7, when $\omega_{\max} = 2048$ (or $\omega_{avg} = 1024$), there is a 30% time increase caused by DDR access to different banks and/or bandwidth sharing with the CPU. In general, for $\omega_{\max} = [2048, 6144]$, communication tasks use between 30 – 50% of the global execution time. Compared to the theoretical modelling presented in Section 3.3, the implementation results differ in about 5 – 25% which is consistent with the penalties incurred by DDR access. Such result can also be verified by comparing the similarity of Eq. (12) with Eqs. (20) and (22).

To conclude the system characterization we established the overall bandwidth reached for every test case in Fig. 10. We added the total number of bits transmitted corresponding to the keep-table and profit values (when $X > C$) plus (1) the number of bits received due to item profits and weights and (2) the received profit table values (when $X > C$). Afterwards, we divide the previous addition by the total computation time obtained on each scenario and we present this behaviour in Fig. 11. Globally, a peak of 1.8GB/s is reached using three High Performance ports and the Accelerator Coherency one, which corresponds to about half the DDR3 peak (4.26GB/s). Nevertheless, note that the maximum theoretical peak

Table 4
Comparison with related works and optimized software execution on the ZYNQ's ARM and an Intel XEON PC. Speed-up values are scaled with the slowest one, the ARM processor.

	ARM (baremetal)	PC	FPGA [7]	KP-1024	GPU [16]	KP-6144
Device	ARM Cortex-A9	XEON E5	VirtexII	Z-7020	NVIDIA GTX-260	Z-7020
Cores	2	4	64	58	190	70
Structure	Multi-core	Multi-core	Systolic array	Shared memory	Many-core	Shared memory
Frequency [MHz]	666	2800	N.R.[a]	133	1400	150
Speed-up [Times]	1	6.5	45	108	135	150
Algorithm	Bellman	Bellman	Recursive	Bellman	M.Toth	Bellman
Max ω_i	ANY	ANY	1024	6144	1024	1024
OnChip memory	256KB	10MB	N.R.[a]	560KB	896MB	560KB
Speed-up/Core	0.5	1.625	0.703	**1.86**	0.71	**2.14**

[a] Not reported

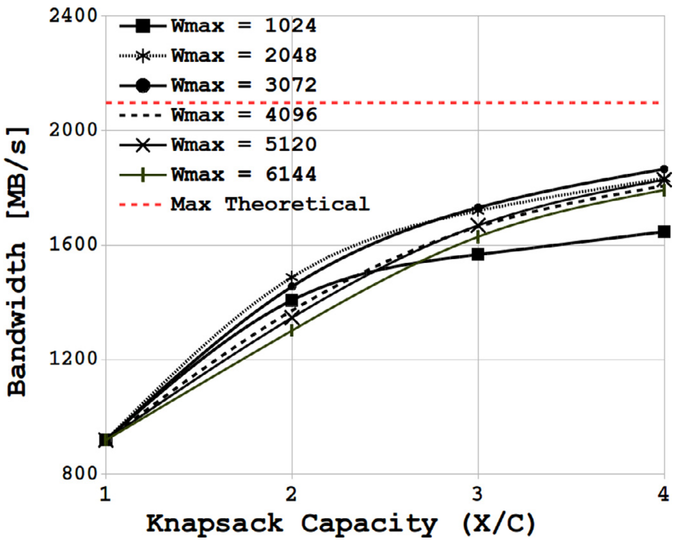


Fig. 11. Computed bandwidth reached for each of the previous test scenarios. The peak value surrounds 1.8GB/s which is about 15% less than the 2.1GB/s theoretical peak for the proposed system at 133 MHz.

for this architecture is 2.1GB/s at 133 MHz. This value is obtained by computing the global execution time using Eq. (10). Formally, the theoretical bandwidth is presented in Eq. (23).

$$BW_{max} = \frac{bitsRx_{PT} + bitsTx_{PT} + bitsTx_{KT}}{t_{exec}} \tag{23}$$

$$BW_{max} = \frac{\omega_{avg} \cdot 32 + \omega_{avg} \cdot 32 + nk}{\max(t_1, t_2, t_3)} \tag{24}$$

$BW_{\max}$ is maximized when the PE number is $n = 65$ (see Fig. 7) and at 133 MHz is 2.1GB/s. However, this value can increase to 2.4GB/s at 150 MHz and to 3.16GB/s at 200 MHz on bigger devices like the Z-7045. Note that saturations can also be explained due to congestion on the DDR chip generated by concurrent reads and writes to the same banks.

In order to benchmark these results, the system performance is compared with previously published works for FPGA and GPU. Using the reported information from previous works [7,16] and Eqs. 20–22, the global comparison is shown in Table 4. We also compare our solution with our own optimized (-O3) sequential software implementation on the ZYNQ's ARM (baremetal) as well as in a desktop PC (Intel Xeon E5-1607 running Linux Ubuntu 14.04). Measurements on the ARM were done using its 64-bit counter while Xeon metrics used the clock function from the *time.h* C++ library.

In the table, the execution time of the ZYNQ's ARM processor (single core running at 666 MHz) is used to scale the rest of the values as it was the slowest one. We included characteristics of each platform and a final efficiency measure to determine the contribution of each PE or core to the global speed-up. All the solutions presented use the same two-line and keep-table strategy. However, in the GPU work authors include a compressing technique to minimize data transfers to main memory. They also use a slightly different algorithm that reduces thread computation. However, this is not critical in our case since computation-wise, every row is solved in $k + 12$ (524 cycles).

Notice we can attain a 10% better runtime using the KP-1024 configuration which supports $\omega_{\max} = 1024$. Nonetheless, when $\omega_{\max} = 6144$, the KP-6144 solver performance falls to a 20% slowdown due to profit value transfers off-chip. It was not possible to compare their runtime when $\omega_{\max} > 1024$ since authors do not report experiments with that configuration.

As is the case in many parallel systems, we outperform sequential executions on CPUs. The single-core, baremetal, ARM speed is greatly exceeded by our system which worked at a 5 × lower frequency and with 58 × more parallel units. On the other hand, although we used the maximum optimization possible on the Xeon CPU (-O3), we were also 16 × faster than this architecture. In fact, the desktop was also running an OS (Linux) hence it did not have 100% dedicated resources to the knapsack application as is the case on GPU and FPGA solutions. The performance difference with our system is explained also by noting we had 14 × more processing units, DMA and a custom interconnect structure to minimize on-chip latency.

Furthermore, compared to the previous FPGA solution we achieve a better runtime $(2.4 - 3.3\times)$ and support 6 × bigger weights, thus we can solve bigger problems. Although there is a chip technology improvement, to the best of our knowledge there are no other recent FPGA-based results to compare to this test case. Note that unlike the typical Systolic Array (SA) solution, where a buffer as big as $\omega_{\max}$ is required per PE, we managed to use a constant size $(k = 512)$ BRAM per PE to share data through pipelined interconnections. In other words, an equivalent SA architecture to solve the problems tested on our system would require 6 × more memory per PE when $\omega_{\max} = 6144$.

Interestingly, the cited solutions employ slightly different versions of the DP algorithm. On one hand, the FPGA proposal [7] is built over an algorithm that processes the profit table M in small strips of a few rows and columns. Their strategy consists in solving partial problems whose results are gathered to obtain the final outcome. Although they incur in additional iterations, it is compensated by reduced off-chip transfers, in part due to their limited $\omega_{\max} < 1024$. On the other hand, the GPU work [16] uses an algorithm where some columns are not evaluated since they are not a feasible part of the solution. Using such algorithm in our architecture would imply that certain PEs will not execute their computations. However, the final row processing time will not be affected because it is fixed to $k + 12 = 524$ cycles.

Other published works on the Knapsack 0–1 problem include a similar GPU implementation on a NVIDIA Tesla M2050 [18] which claims to have improved the work of boy [16] by 20%. Unfortunately since authors do not provide timing metrics it was not possible for us to contrast this work against theirs. Finally, our literature review found in [23] that 500 MIPSR14000 processors spent 80.12 s to process a knapsack of size $C = 5000$ and $m = 10000$ which is by far slower than our results. Using Eq. (22), our proposed architecture would only take 4 ms.

According to the results obtained, we note a compromise between processing time and $\omega_{\max}$ due to the increase of data exchange off-chip. In the following subsection we study key aspects where the proposed system can be enhanced to increase its scalability and ameliorate the global runtime.

4.3. Improvements for higher scalability

As seen in Section 3.3 and in the results presented here, the current system is constrained by two factors: BRAM availability and data transmission times. The first one can be easily overcome using a bigger device (Z-7045, Virtex-7, etc). For the second case there are different options to enhance the system runtime.

- According to Fig. 7, when $\omega_{avg} \leq 1024$, the system can have up to $n = 65$ PEs without any data transmission penalties. On the contrary, when these values increase, they affect the global runtime and become a limitation in terms of runtime scalability. Since there are a lot of redundant data to transfer in both the keep and profit table, the strategy to decrease these times would be to transfer only non-redundant values. For instance, for the keep-table it is possible to verify that for any column $j < \omega_i$, the keep bit will be zero since the item does not fit (thus it is not kept). Moreover, keep-table bits tend to stabilize in the value '1' to the right of the table hence generating several redundant transfers. This fact was exploited by boy [16] to reach the reported execution time which is very close to ours with no compression. However, it required pre-arranging data in decreasing order p_i/ω_i. Using such a strategy, theoretically our system could decrease the time t_1 employed for keep-table transmission and more PEs could be included without any stalls.
- The profit values transfer can also increase the global runtime specially when $\omega_{avg} > 1024$ (Fig. 7). As in the keep-table case, these values tend to repeat themselves to the right of the table (specially for big ω_i values), generating redundant data exchanges. For such a case, a run-length encoding algorithm can be used before transmission to compress the words to transfer.

The systems presented are just two examples of the many different configurations that can be constructed using the proposed architecture. As a matter of fact, problems with weight values $\omega_{\max} \neq 1024$ and $\omega_{\max} \neq 6144$ can also be solved by synthesizing the system with the desired parameters.

5. Conclusions and future work

In this paper we have presented a parametrizable and scalable system to process the DP matrix for the Knapsack 0/1 with $\omega_{\max} \geq 1024$ using parallel operators that share memory. The obtained runtime is proportional to the amount of operators and can be configured in different ways to include more processing units and overcome data transfer limitations due to sequentiality.

The proposed architecture can be synthesized in any FPGA or SoC device or even used on dedicated chips. We focus on memory oriented enhancements by using DMA access, asymmetric-port memories and a deeply pipelined interconnections. To check our expectations experimentally, we synthesized two system configurations on a ZYNQ-7020 chip: The first one (KP-1024) can solve problems with $\omega_{\max} = 1024$, runs at 150 MHz, includes 70 processing elements (PEs) and achieves 23 $\times$ speed-up versus a sequential yet optimized execution on an Intel Xeon CPU. It also attains 10% speed-up versus a previously published GPU work.

In the second system (KP-6144) we included 58 PEs that run at 133 MHz and we reached a 16 $\times$ speed up versus the Xeon PC. We were also 1.4 $\times$ faster than previous reported FPGA works and only 0.2 $\times$ slower than previous GPU solutions. In addition, we attain 1.8GB/s bandwidth on the ZYNQ-7020 chip by using three of the 64-bit high performance ports for off-chip memory exchange close to the theoretical limit (2.1GB/s). This architecture is capable of solving problems with $\omega_{\max} = 6144$, that is, 6 $\times$ bigger than systolic array counterparts.

In this work we show that by maximizing on-chip memory utilization, using small, replicable and synchronized processing units, FPGAs can provide an equivalent (or even higher) performance than GPUs at much lower frequencies (thus with less power).

We presented a way of handling designs with unpredictable memory accesses by dedicating routing resources specifically for data propagation. On one hand we have used synchronization to virtually share a single memory block into several PEs without the use of arbiters. On the other, we profited from heavy pipelines to maximize the system operating frequency and reduce the effects of its interconnection complexity.

In the future we aim to explore data width reduction using 16-bit values and data and compression techniques to decrease redundant data transmissions that currently affect the system performance. Moreover we intend to integrate our system with memory optimized algorithms.

References

[1] R. Bittner, E. Ruf, A. Forin, Direct gpu/fpga communication via pci express, Cluster Comput. (2013) 1–10, doi:10.1007/s10586-013-0280-9.
[2] G. Ballard, Avoiding Communication in Dense Linear Algebra, EECS Department, University of California, Berkeley, 2013 Ph.D. thesis.
[3] A. Grama, A. Gupta, G. Karypis, V. Kumar, Chapter 12: Dynamic Programming, in: Addison-Wesley (Ed.), Introduction to Parallel Computing, 2nd Ed., Pearson, 2003, pp. 515–532.
[4] G. huey Chen, M. sheng Chern, J.H. Jang, Pipeline architectures for dynamic programming algorithms, Parallel Comput. 13 (1990) 111–117.
[5] R. Andonov, P. Quinton, S.V. Rajopadhye, D. Wilde, A shift registered-based systolic array for the unbounded knapsack problem, Parallel Process. Lett. 5 (1995) 251–262.
[6] R. Andonov, S. Rajopadhye, Knapsack on vlsi: from algorithm to optimal circuit, Parallel Distrib. Syst. IEEE Trans. 8 (6) (1997) 545–561.
[7] K. Nibbelink, S. Rajopadhye, R. McConnell, 0/1 knapsack on hardware: A complete solution, in: Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on, 2007, pp. 160–167.
[8] An efficient parallel algorithm for solving the knapsack problem on hypercubes, J. Parallel Distrib. Comput. 64 (11) (2004) 1213–1222.
[9] B. Sahoo, S. Padhy, A reconfigurable accelerator for parallel longest common protein subsequence algorithm, in: Advance Computing Conference, 2009. IACC 2009. IEEE International, 2009, pp. 260–265.
[10] M.N. Isa, M.I. Ahmad, S.A.Z. Murad, R.C. Ismail, K. Benkrid, Biological sequence alignments: A review of hardware accelerators and a new pe computing strategy, in: Region 10 Symposium, 2014 IEEE, 2014, pp. 39–44.
[11] X. Chang, F.A. Escobar, C. Valderrama, V. Robert, Optimization strategies for smith-waterman algorithm on fpga platform, in: Computational Science and Computational Intelligence (CSCI), 2014 International Conference on, 1, 2014, pp. 9–14.
[12] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, P. Sadayappan, Parallel fpga-based all-pairs shortest-paths in a directed graph, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 2006a, p. 10.
[13] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, P. Sadayappan, Hardware/software integration for fpga-based all-pairs shortest-paths, in: Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on, 2006b, pp. 152–164.
[14] A. Jacob, J. Buhler, R.D. Chamberlain, Accelerating nussinov rna secondary structure prediction with systolic arrays on fpgas, in: Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on, 2008, pp. 191–196.

[15] A.C. Jacob, J.D. Buhler, R.D. Chamberlain, Rapid rna folding: Analysis and acceleration of the zuker recurrence, in: Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, 2010, pp. 87–94.
[16] Solving knapsack problems on gpu, Comput. Oper. Res. 39 (1) (2012) 42–47. Special Issue on Knapsack Problems and Applications.
[17] D.M. Quan, L.T. Yang, Solving 0/1 knapsack problem for light communication sla-based workflow mapping using cuda, in: Computational Science and Engineering, 2009. CSE '09. International Conference on, 1, 2009, pp. 194–200.
[18] B. Suri, U.D. Bordoloi, P. Eles, A scalable gpu-based approach to accelerate the multiple-choice knapsack problem, in: 2012 Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 1126–1129.
[19] Xilinx, Zynq-7000 all programmable soc overview, 2014, URL http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.
[20] T.D. Linh, T. de Souza-Daw, T.M. Hoang, N.T. Dzung, Parallel random access memory in a shared memory architecture, in: Communications and Electronics (ICCE), 2014 IEEE Fifth International Conference on, 2014, pp. 364–369, doi:10.1109/CCE.2014.6916731.
[21] AVNET, Zedboard: Zynq evaluation and development hardware user's guide, 2014, URL http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.
[22] Xilinx, Axi datamover v5.1. logicore ip product guide, 2015, URL http://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v5_1/pg022_axi_datamover.pdf.
[23] D.E. Baz, M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 01 knapsack problem, J. Parallel Distrib. Comput. 65 (1) (2005) 74–84.

Fernando A. Escobar was born in Bogotá, Colombia on July the 30st 1985. He obtained his BSc in Electronics Engineering from University of Los Andes (Bogotá) in 2008. In 2011 he received his MSc in Elctronics and Computer engineering from the same university. In June 2016 he obtained his PhD at UMONS, Belgium, with a research focused in High Performance Computing (HPC) using FPGAs. Currently he works as a post-doctoral researcher at CentraleSupelec in France. His research interests include digital design, embedded systems, hardware verification and HPC architectures.