A Regular VLSI Array for an Irregular Algorithm

Florent de Dinechin[1], Doran K. Wilde[2], Sanjay Rajopadhye[1], Rumen Andonov[3]

[1] IRISA, Campus de Beaulieu, 35042 Rennes France
[2] Brigham Young University, Provo Utah
[3] ISTV, University of Valenciennes, France

Abstract. We present an application specific, asynchronous VLSI processor array for the dynamic programming algorithm for the 0/1 knapsack problem. The array is derived systematically, using correctness-preserving transformations, in two steps: the standard (dense) algorithm is first transformed into an irregular (sparse) functional program which has better efficiency. This program is then implemented as a modular VLSI architecture with nearest neighbor connections. Proving bounds on buffer sizes yields a linear array of identical asynchronous processors, each with simple computational logic and a pair of *fixed size* FIFOs. A modular solution can be obtained by additional load-time control, enabling the processors to pool their buffers.

1 Introduction

The *0/1 knapsack problem* is a classic, NP-complete, combinatorial optimization problem with many applications [7, 12]. In this paper we concentrate on the *dynamic programming* approach to this problem [4, 7], since it has more regularity than the dual *branch-and-bound*. It is well known that naive dynamic programming performs a lot of redundant computation, which can be avoided by using a *sparse representation* of the data, yielding a significant improvement in the average case performance [6]. Many authors have investigated parallel solutions in the dense or sparse case. Software parallel implementations of the dense approach may be found in [11]. Lee et al. implement the sparse algorithm on a hypercube using a divide and conquer strategy [10], which takes $O(mc/q + c^2)$ time on q processors, and uses $O(mc)$ storage in the worst case[4]. Chen et al. present a pipelined linear array which uses $\Theta(mc)$ storage, $\Theta(c)$ on each of m processors [5]. These authors, however, all assume that the target is a general purpose multiprocessor, in particular, each processor has unbounded memory.

In this paper, we present a dedicated VLSI array architecture for the forward phase of the sparse algorithm. This architecture is a *wavefront array processor*

[4] Lee, Shragowitz and Sahni point out that this could be worse than the sequential algorithm [10]. The average behavior, however, is expected to be better because of sparsity.

(WAP) which is similar to a systolic array, except that the processors are asynchronous, and communicate through FIFO queues [9].

Our contributions are twofold. First, we *systematically derive* the sparse algorithm from the (dense) recurrence of the dynamic programming algorithm. Our derivation is similar to that used in [3] for the *unbounded* knapsack problem. Second, our implementation on dedicated VLSI is fully modular with respect to problem parameters. For this purpose we first show that buffer sizes are bounded by the maximum object weight. This is itself a problem parameter, but we then show how an appropriate number of PE's, each with the same amount of memory, may be configured so that they "pool" this memory (a similar idea was previously used for the dense algorithm [2]). Thus it is possible to solve a larger problem instance by simply adding more PEs to the array, without having to redesign the PE itself. Furthermore, we also discuss the problem of choosing the buffer sizes optimally.

The paper is organized as follows. In Sect.2 we introduce the problem and the sparse representation. In Sect.3 we present the transformation of the recurrence equation of the dense algorithm into a stream functional program. Sect.4 deals with the implementation of this program as a WAP, and the choice of the buffer sizes. Finally, we present our conclusions. Because of space constraints we give neither proofs nor implementation details, which may be found in [1].

2 Problem Definition

The *forward phase* of the dynamic programming algorithm for the 0/1 knapsack problem is defined by the profit function given by the recurrence equation below:

$$\begin{cases} f_k(j) = \max\big(f_{k-1}(j),\ p_k + f_{k-1}(j - w_k)\big) & \forall (k,j) \in \{1 \ldots m\} \times \{1 \ldots c\} \\ f_0(j) = f_k(0) = 0 \\ f_k(j) = -\infty \quad \forall j < 0 \end{cases} \tag{1}$$

Table 1 shows an example of the $f_k(j)$, calculated as per (1). The entry at $j = 10$ and $k = 4$ indicates that the maximum profit acheivable for this problem is 19. The backtracking phase (which we do not consider in this paper) would indicate that the maximum profit is achieved by placing objects 1, 2 and 4 in the knapsack.

Table 1. Values of $f_k(j)$ for $m = 4$; $c = 10$; $w_i = 5, 4, 6, 1$; $p_i = 7, 8, 9, 4$.

j / k	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	$\boxed{7}$	7	7	7	7	7
2	0	0	0	$\boxed{8}$	8	8	8	8	$\boxed{15}$	15
3	0	0	0	$\boxed{8}$	8	$\boxed{9}$	9	9	$\boxed{15}$	$\boxed{17}$
4	$\boxed{4}$	4	4	$\boxed{8}$	$\boxed{12}$	12	$\boxed{13}$	13	$\boxed{15}$	$\boxed{19}$

There is considerable redundency in this table. It is easy to prove [1] that each f_k (each row) is a monotonically increasing function, which can be efficiently represented as a set of *critical points* $[j, f_k(j)]$ (the boxed values in Table 1). This *sparse representation* of f_k is also illustrated by Fig.1.a.

The sparse algorithm [13] uses (1) to build this sparse representation iteratively: given the set (or sequence) S_{k-1} of critical points representing f_{k-1}, first compute an auxillary set S'_{k-1} by adding $[w_k, f_k]$ to each element of S_{k-1}. Then take the union of S_{k-1} and S'_{k-1}, which contains all the critical points of f_k, plus some that are *dominated* (not critical): a point $[j_1, f_1]$ dominates another point $[j_2, f_2]$ iff $j_1 \leq j_2$ and $f_1 \geq f_2$, (i.e., if it has less weight and more profit).

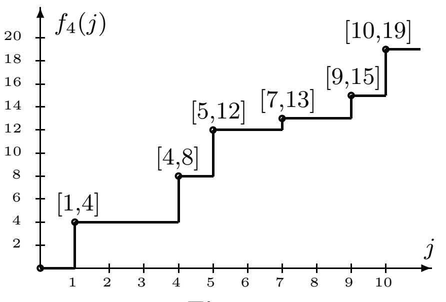

Fig.1.a
The function f_4 and its critical points

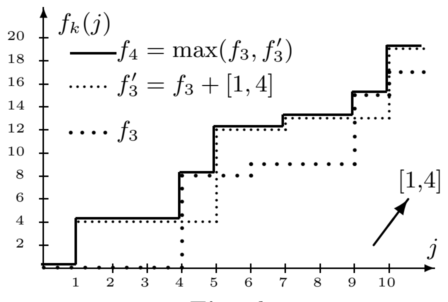

Fig.1.b
...built from the critical points of f_3

The set S_k of the critical points of f_k is obtained by removing all the dominated points from the union set, as illustrated in Fig.1.b.

3 Derivation of the Stream Functional Program

We derive a stream functional program to implement the recurrence equation (1), computing the stream (sequence) of critical points S_k from the stream S_{k-1}.

We use the following notations: braces `{ ... }` are used to group sequences. The operator `^` denotes the concatenation of an element to a sequence. Parentheses are used for function application and square brackets are used to form tuples.

The initial function $f_0 = 0$ can be trivially represented by the sequence $S_0 = \{[0,0]\}$. Then the term $p_k + f_{k-1}(j - w_k)$ is represented by the sequence S'_{k-1} obtained by adding the pair $[w_k, p_k]$ to each pair element in $[0,0]\hat{}S_{k-1}$. The resulting sequence is clipped so that no accumulated weight exceeds the knapsack capacity c. This yields $S'_k = \texttt{AddTest}([w_k, p_k],\ [0,0]\hat{}S_{k-1}, c)$, where `AddTest` is defined as follows:

```
AddTest([w,p], {}, c)       = {}
AddTest([w,p], [j,f]^S, c) = if j+w>c : {}
                              else  : [j+w, f+p] ^ AddTest([w,p], S, c)
```

The max function used in (1) is implemented as a stream operator computing the sparse sequence for the maximum of two functions given as sparse sequences. This is done by a merge sort based on the j elements (the `Merge` function below), followed by a removal of the pairs that are dominated by an earlier pair in the sequence (the `Filter` function below).

```
Merge({}, S) = S
Merge(S, {}) = S
Merge([j1,f1] ^ S1 , [j2,f2] ^ S2) =
  if       j1 < j2   :  [j1,f1] ^ Merge(S1 , [j2,f2] ^ S2)
  else if  j1 > j2   :  [j2,f2] ^ Merge([j1,f1] ^ S1 , S2)
  else if  j1 = j2   :  [j1, max(f1,f2)] ^ Merge(S1 , S2)

Filter({}, t) = {}
Filter([j,f]^S, t) =  if  f > t   :  [j,f] ^ Filter(S, f)
                          else    :  Filter(S ,t)
```

The following program then computes S_k from S_{k-1}:

```
S(0)={}
S(k) = Filter( Merge( S(k-1),
                      AddTest( [wk,pk], [0,0]^S(k-1), c ) ),
               0 )
```

4 A WAP for the Sparse Algorithm

It is well known [8] that a stream functional program corresponds to a network of processes communicating over asynchronous FIFO channels. Hence the above program for calculating the recurrence (1) can be implemented on an *asynchronous array processor* with m cells. The cell k receives the sequence S_{k-1} and computes the sequence S_k. The structure of such a cell is shown in Fig.2. The FIFO buffer between two consecutive cells is optional: its purpose is to improve the overall throughput, therefore its size depends on statistical considerations. Simulations showed, however, that small internal buffers lead to near-optimal efficiency: for random values of (w_i, p_i) in the range $(10 \cdots 100, 10 \cdots 100)$ and $c = 3000$, for instance, the optimal throughput was reached with only 10 registers.

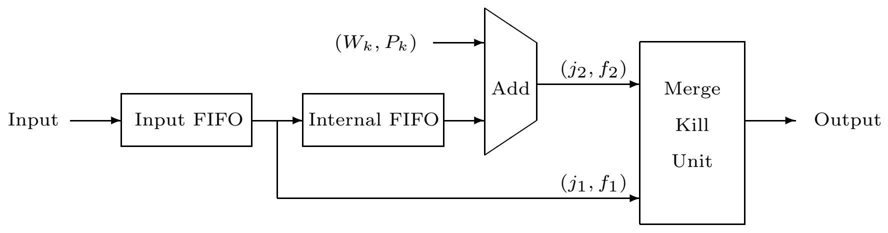

Fig. 1. Basic Processor

The internal FIFO, on the other hand, may lead to problems: as the merge-kill unit consumes $[j_1, f_1]$ from the input FIFO, a copy must be saved in the internal queue for later use in the second input of the merge-kill unit. If this queue is too small, the circuit will fail due to FIFO overflow. The crucial fact allowing a VLSI implementation using fixed size buffers is based on the fact that *the maximal size of the internal queue on any cell is* w_{max} (see proof in [1]). As this result still limits such a processor to operate on objects of weight smaller than its internal FIFO size, we extend the design, by the means of a configuration bit, to allow the FIFOs of consecutive processors to be connected to form one single logical processor[5]. Thus the array is extensible with respect to w_{max}. It is also extensible with respect to m by simply adding new processors to the array, or by using multiple passes: it is easy to implement a LPGS partitioning scheme for this WAP, with an external buffer (e.g. in a host) bounded by c, the knapsack capacity. The choice of the size α of the internal FIFO then leads to a classical trade-off: a large FIFO increases the average efficiency of the processors but also their area, leading to fewer processors on a given silicon area. The optimal value of α is very data-dependent and should thus be determined by an extensive simulation using sample data. A worst-case analysis is also possible and leads to the following equation which gives, for each value of $\beta = \log_2 \alpha$, the maximal size m of the knapsack problem which may be handled by the architecture.

$$m \leq \beta + \left\lfloor \log_2 \left(1 - \beta + \left\lfloor \frac{S}{2^\beta s_r + s_0} \right\rfloor \right) \right\rfloor \tag{2}$$

Here s_r is the area of one FIFO register, and s_0 the area of one processor, excluding the internal FIFO. An extensive search over the values of β (from 0 to 16 is obviously sufficient) gives the optimal value of α in the worst case.

5 Conclusion

We have presented a systematic derivation of a wavefront array processor from a recurrence equation specifying the (forward phase of the) dynamic programming algorithm for the 0/1 knapsack problem. The final array is interesting in its own right (although it remains an open problem whether it is possible to perform the backtracking phase with similar space/time complexity).

The main advantage of this approach is to combine the strong points of an irregular algorithm (time efficiency due to asynchrony, and space efficiency due to sparsity) with the strong points of a parallel regular array (simplicity of the design and efficiency due to potential massive parallelism). Thus the methodology consists of two systematic, independent transformations: the first transforms regular to irregular (and thus increases algorithmic efficiency), and the second transforms irregular to regular (and thus increases implementation efficiency). Problems, however, arise when we try to generalize both steps:

The first transformation uses some property of the data (here sparsity and monotonicity) to lead to an irregular algorithm, expressed as a stream functional

[5] The price to pay for this is roughly doubling the number of pins per chip.

program, which is more efficient in average than the regular one. This is based on manually proving properties of the computations, and is related to program synthesis. It is not likely that it can be automated.

The second transformation is more classical and leads from a irregular stream functional program to a regular (but asynchronous) WAP, whose parameters are derived from properties of the algorithm or data. The crucial problem here is to prove bounds on buffers in order to obtain a VLSI implementation. We conjecture that this can be done systematically, however slight variations in the algorithm can lead to crucial problems (for example, a direct adaptation of the program used in [3] yields an array where buffer sizes cannot be bounded).

We believe that our methodology may be applied to other irregular problems, such as sparse matrix computations.

References

1. R. Andonov, F. de Dinechin, S. Rajopadhye, and D. Wilde. – Systematic design of wavefront array processors : A case study. – Internal Report 743, IRISA, March 1994.
2. R. Andonov and S. Rajopadhye. – An optimal algo-tech-cuit for the knapsack problem. – Technical Report PI-791, IRISA, January 1994. – (to appear in IEEE Transactions on Parallel and Distributed Systems).
3. R. Andonov and S. V. Rajopadhye. – A sparse knapsack algo-tech-cuit and its synthesis. – In *International Conference on Application-Specific Array Processors (ASAP-94)*, pages 302–313, San Francisco, August 1994. IEEE.
4. R. Bellman. – *Dynamic Programming.* – Princeton University Press, Princeton, NJ, 1957.
5. G.H. Chen and J.H. Jang. – An improved parallel algorithm for 0/1 knapsack problem. – *Parallel Computing*, 18:811–821, 1992.
6. E. Horowitz and S. Sahni. – Computing partitions with aplications to the knapsack problem. – *Journal of the ACM*, 21(2):277–292, April 1974.
7. T. C. Hu. – *Integer Programming and Network Flows.* – Addison-Wesley, 1969.
8. G. Kahn. – The semantics of a simple language for parallel processing. – In *Proceedings of IFIP*, pages 471–475. IFIP, August 1974.
9. S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. B. Rao. – Wavefront array processor: Language, architecture and applications. – *IEEE Transactions on Computers*, C-31:1054–1066, 1982.
10. J. Lee, E. Shragowitz, and S. Sahni. – A hypercube algorithm for the 0/1 knapsack problems. – *J. of Parallel and Distributed Computing*, 5:438–456, 1988.
11. J. Lin and J. A. Storer. – Processor-efficient hypercube algorithm for the knapsack problem. – *J. of Parallel and Distributed Computing*, 13:332–337, 1991.
12. S. Martello and P. Toth. – *Knapsack Problems: Algorithms and Computer Implementation.* – John Wiley and Sons, 1990.
13. G. Nemhauser and J. Ullman. – Discrete dynamic programming and capital allocation. – *Management Science*, 15(9):494–505, 1969.