A Regular VLSI Array for an Irregular Algorithm

Florent de Dinechin\textsuperscript{1}, Doran K. Wilde\textsuperscript{2}, Sanjay Rajopadhye\textsuperscript{1},
Rumen Andonov\textsuperscript{3}

\textsuperscript{1} IRISA, Campus de Beaulieu, 35042 Rennes France
\textsuperscript{2} Brigham Young University, Provo Utah
\textsuperscript{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 $O(m)$ storage in the worst case\textsuperscript{4}. Chen et al. present a pipelined linear array which uses $O(mc)$ storage, $O(c)$ 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

\textsuperscript{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:

\[
\begin{align*}
    f_k(j) &= \max (f_{k-1}(j), p_k + f_{k-1}(j - w_k)) \quad \forall (k, j) \in \{1, \ldots, m\} \times \{1, \ldots, c\} \\
    f_0(j) &= f_k(0) = 0 \\
    f_k(j) &= -\infty \quad \forall j < 0
\end{align*}
\]

(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 achievable 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>
<thead>
<tr>
<th>( k )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

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