Task Ordering and Memory Management Problem for Degree of Parallelism Estimation

Sergiu Carpov[1,2], Jacques Carlier[2], Dritan Nace[2], and Renaud Sirdey[1]

[1] CEA, LIST,
Embedded Real Time Systems Laboratory,
Point Courrier 94, 91191 Gif-sur-Yvette Cedex, France
[2] UMR CNRS 6599 Heudiasyc,
Université de Technologie de Compiègne,
Centre de recherches de Royallieu,
BP 20529, 60205 Compiègne Cedex, France

Abstract. This paper is devoted to the problem of estimating the *achievable degree of parallelism* for a parallel algorithm with respect to a bandwidth constraint. In a compiler chain for embedded parallel microprocessors such an estimation can be used to fix an appropriate target for parallelism reduction "tools". Informally, our problem consists in task ordering and memory management for an algorithm, so as to minimize the number of memory accesses. After a brief survey of the literature, we prove the $\mathcal{NP}$-hardness of this problem and introduce a polynomial special case. We then present a branch and bound procedure for the general case along with computational results interpretation demonstrating its practical relevance.

Keywords: task scheduling, memory management, parallel processor.

1 Introduction

In this paper, we investigate a series of problems related to efficient memory bandwidth management in embedded parallel processor architectures. In particular, we are interested in estimating the memory bandwidth required for the sequential execution of a parallel algorithm so as to estimate the number of tasks which may execute in parallel with respect to an external memory bandwidth constraint. In a compiler chain proceeding by parallelism reduction (via task fusion [9]), such an estimation can be used to fix an appropriate target for the degree of parallelism. This estimation can also be used within an Algorithm-Architecture Adequation framework to perform an initial assessment.

Let us consider an embedded parallel processor architecture which consists of many processing cores which share a common memory space. The system in which this processor is used, has an external memory for storing application data and instructions. External memory locations are accessed without direct core involvement, i.e. it only initiates and finishes the data transfers. In this

B. Fu and D.-Z. Du (Eds.): COCOON 2011, LNCS 6842, pp. 592–603, 2011.

context, a typical bottleneck for many algorithms[1] is the external memory access bandwidth, which must be carefully managed in order to keep the processing cores busy enough.

An algorithm is composed of a set of tasks that are using external memory data. Let Δ be the total amount of data (i.e. number of bytes) used by all algorithm tasks. The total sequential algorithm execution time T_{exec}, $T_{exec} = T + T_\Delta$, consists of a fixed part T and of a variable part T_Δ. The fixed part T corresponds to the execution itself, that is the time the processing cores are busy. The variable part T_Δ corresponds to the time needed to load all the external memory data Δ. The data loading time is variable because several data could be loaded two or more times in function of the caching strategy.

In this study, we propose a method to estimate the *achievable degree of parallelism* for an algorithm constrained by the external memory bandwidth, that is the ratio Λ/λ between the external memory bandwidth Λ to the average bandwidth λ required by an optimal (in some sense defined later) sequential execution of the algorithm. We suppose that we are dealing with parallel algorithms, thus their intrinsic structure in terms of parallelism is not an issue, only the limitations of the chip (here, in terms of external memory bandwidth) influence the estimation. The average bandwidth λ is defined as the total amount of data divided by the total execution time, $\lambda = \Delta/T+T_\Delta$. Let us reverse the bandwidth formula, we obtain $\lambda^{-1} = T/\Delta + k$, where k is the duration of loading one unit of data (within a certain degree of accuracy we suppose $T_\Delta = k\Delta$). It is straightforward to see that the average bandwidth is proportional to the amount of data loaded from the external memory, thus loading less data will reduce the bandwidth.

Our goal is to find a task execution order which, combined with an appropriate data management (via data reuse), minimizes the average bandwidth and in turn maximizes the achievable degree of parallelism. A branch and bound procedure is introduced to deal with this problem. One practical utility of the exact resolution approach we propose is for finding optimal solutions which could be used for evaluating heuristic methods, as obtaining tight global lower bounds for a particular case of this problem is already a difficult task [1].

After a survey on the existing work, we give a formal definition of our problem and state on its complexity. We then present the issue of on-chip memory management for a fixed task calculation order, and finally, we introduce the branch and bound algorithm.

2 Related Work

Previous work related to our problem is quite rare. Although our study is on the limitations induced by the external memory bandwidth on algorithm parallelism, we have identified the following works which are indirectly related to our problem. The similarities mainly consist in the tools that are used to address related though different problems.

[1] Especially signal and image processing algorithms.

It appears that [5] were the first to study a problem relatively close to ours. In their work they intend to reorder program instructions so as to improve program data access locality. The goal of locality minimization is to reduce the distance between instructions that are using the same data, thereby to augment data reuse. However, instead of using an optimal cache management policy for calculating the distances, a sufficient-only condition is used. The latter requires that the *time distance*, which is the number of instructions between two accesses, is bounded by a constant in order to ensure a cache hit. This problem is modeled as a bandwidth-minimization problem. In their paper, the authors describe a tool that reorders program instructions and provide experimental results for a set of benchmark programs.

Some earlier works, [14,11], describe methodologies for optimizing data reuse in program loops. In a series of two papers [4,6] describe two program transformations: *locality grouping*, which reorders program data accesses in order to improve data temporal reuse, and *dynamic data packing*, which consists in reorganizing data layout so as to improve data spatial reuse. Other papers [12,13] describe similar approaches of data locality improvement and provide benchmark analysis.

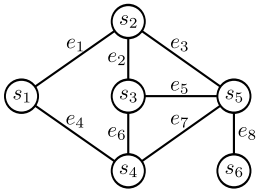
3 Problem Formulation and Complexity

An algorithm is a set of tasks. Each task uses external memory data which needs to be fetched before the task execution can start. The smallest unit of data is called an *input*, hereafter we use the terms of external memory data and input equivalently. Let $A = (S, E, \delta)$ denote an algorithm where S are algorithm's tasks, E represent the set of inputs used by the algorithm and $\delta : S \to 2^E$ is a function that associates to any task s, $s \in S$, a set of inputs $\delta(s)$ needed for its calculation. We suppose that there are no precedence relations between algorithm tasks[2]. Although this model seems to be limited the methods described further can be easily modified to take into account the precedence constraints.

Let $\gamma : S \to 2^E$ be a function that associates a set of inputs $\gamma(s)$ to any task s, $s \in S$. The set of inputs $\gamma(s)$ gives the on-chip memory state at the beginning of task s calculation. It is evident that for any task s, $\gamma(s)$ contains at least all the inputs used by this task, $\delta(s) \subseteq \gamma(s)$. Remaining inputs that do not belong to $\delta(s)$, come from data reuse. *Data reuse* is the process of reusing inputs already present in memory, originating from previously calculated tasks. Throughout this paper we suppose that available on-chip memory size is equal to C, thus condition $|\delta(s)| \leq |\gamma(s)| \leq C$ must be verified for any $s \in S$. Without loss of generality we suppose that the total number of inputs is larger than the on-chip memory size, i.e. $|E| > C$.

The task ordering and memory management problem consists in finding a permutation π of tasks S such that the number of external memory accesses, given by cost function (1) is minimized.

[2] This case can be interpreted as a coarse-grained view of algorithm task graphs where the set of tasks producing an outcome are grouped into a single task.



$$\Rightarrow \begin{array}{l} \delta(s_1) = \{e_1, e_4, e_1^d, e_2^d\} \\ \delta(s_2) = \{e_1, e_2, e_3, e_3^d\} \\ \delta(s_3) = \{e_2, e_5, e_6, e_4^d\} \\ \delta(s_4) = \{e_4, e_6, e_7, e_5^d\} \\ \delta(s_5) = \{e_3, e_5, e_7, e_8\} \\ \delta(s_6) = \{e_8, e_6^d, e_7^d, e_8^d\} \end{array}$$

Fig. 1. Illustration of graph transformations used to obtain an instance of our problem

$$\left|\gamma\left(s_{\pi(1)}\right)\right| + \sum_{i=2}^{n} \left|\gamma\left(s_{\pi(i)}\right) \setminus \gamma\left(s_{\pi(i-1)}\right)\right| \tag{1}$$

Proposition 1. *Task ordering and memory management problem is $\mathcal{NP}$-hard.*

Proof. Let $G = (V, A)$ be an arbitrary non-oriented graph. The problem of existence of a Hamiltonian path in graph G is $\mathcal{NP}$-complete [7, p.199]. We show that Hamiltonian path existence problem can be reduced to a special case of our problem, by using the following transformations.

Each vertex of graph G becomes a task in our problem $S = V$ (see Figure 1 for an illustration). To each edge of graph G an input is associated. The on-chip memory size equals to the maximum degree of graph vertices, $C = \max_{s \in V} \deg(s)$. For any task $s \in S$, $\delta(s)$ contains the set of input-edges that are adjacent to s in the graph, plus a set of $C - \deg(s)$ dummy inputs specific for each task. In this way $\gamma(s) = \delta(s)$ and $|\gamma(s) \cap \gamma(s')| = 1$ for any pair of adjacent task $s, s' \in S$.

The problem obtained in such a way is a special case of our problem. It has a solution of cost $n \cdot C - n + 1$ (n being the number of tasks) if and only if it exists a Hamiltonian path in graph G. As far as the question of existence of a Hamiltonian path in graph G is $\mathcal{NP}$-complete, our problem is $\mathcal{NP}$-hard. □

4 Memory Management for a Fixed Sequence of Tasks

We start by investigating the on-chip memory data management separately from the task ordering as it is an important issue of our problem. Let $A = (S, E, \delta)$ be an algorithm. Suppose that a permutation of tasks $s_1, \ldots, s_n$ is given. We recall that for any task $s \in S$, $\delta(s)$ are the inputs that must be loaded into the on-chip memory before s starts. It is obvious that the duration of loading inputs $\delta(s)$ depends on the position of task s in this sequence and on the content of the on-chip memory. Namely, on already loaded inputs originating from data reuse process. We are interested to find the on-chip memory states $\gamma(s)$ for any s, $s \in S$, such that the number of external memory accesses (1) is minimal.

For solving the memory management problem, we introduce an incremental[3] algorithm, see Algorithm 1, which updates the optimal memory states for a

[3] The incremental formulation is useful in the branch and bound procedure we introduce in the next section, where it is necessary to recalculate the data reuse induced only by the last task.

sequence of tasks $s_1, \ldots, s_{k-1}$ with the data reuse induced by a task s_k added to the end of this sequence. The complexity of this algorithm is $O(k \cdot m)$, where m is the number of inputs. The algorithm is based on the principle used in optimal cache replacement algorithm proposed by [2]. In our context[4], it may be informally stated as follows: "when a memory location is needed for a given input and the on-chip memory is full, free space should be obtained by dropping out the input which is to be used in the farthest future".

In order to find the optimal memory management for a sequence of tasks $s_1, \ldots, s_n$ the algorithm is executed $n-1$ times, each time adding task s_i, $i = 2 \ldots n$, to the end of previously computed sequence. The global complexity of this procedure is $O\left(n^2 \cdot m\right)$. The minimal number of external memory accesses can be found using expression (1) from the calculated memory states.

Algorithm 1. Incremental on-chip memory management algorithm (for the sake of simplicity we consider $\gamma_i = \gamma(s_i)$ and $\delta_i = \delta(s_i)$)

Input: $s_1, \ldots, s_{k-1}$ - task ordering with already computed memory states $\gamma_1, \ldots, \gamma_{k-1}$
Input: s_k - new task to be added to the end of the ordering
Output: $\gamma'_1, \ldots, \gamma'_k$ - updated memory states

```
p = k − 1
while p > 1 and|γ_p| < C do {Find last task with full on-chip memory, if it exists}
    p = p − 1
end while
γ'_i = γ_i for i = 1, ..., p
for i = p + 1 tok − 1 do
    L = (δ_k \ γ_i) ∩ γ'_{i−1} {Potential inputs to reuse}
    l = min(|L|, C − |γ_i|)
    γ'_i = γ_i ∪ {First l elements of L according to a linear order}
end for
γ'_k = δ_k
```

The optimal on-chip memory data management based on the calculated memory states is easily found. For first task s_1 all the inputs $\gamma(s_1)$ are loaded into the on-chip memory. For next tasks s_i, $i \geq 2$, before the execution of task s_i starts, inputs $\gamma(s_i) \setminus \gamma(s_{i-1})$ are loaded into the on-chip memory in place of inputs $\gamma(s_{i-1}) \setminus \gamma(s_i)$.

5 Task Ordering and Memory Management

The fact that the task ordering and memory management problem is $\mathcal{NP}$-hard legitimates the use of a branch and bound procedure for solving it. In what follows we describe the proposed branch and bound algorithm as well as each of the used components. The branch and bound algorithm starts with an empty

[4] We do not use the algorithm described in Belady's paper, because in their model at each step only a single memory location is loaded. Contrary to our model, where at each step we can load more than one input.

sequence of tasks. At each branching decision it adds to the end of this sequence a new task from the set of not yet ordered ones. A leaf is obtained when all the tasks are ordered. Lower bounds as well as a dominance relation are used in algorithm in order to reduce the search space.

Before describing the branch and bound procedure we introduce some useful definitions. Let us denote by (I, π) a *permutation of tasks* or *task ordering*, where $I \subseteq S$ is a set of tasks and $\pi : \{1, \ldots, |I|\} \rightarrow \{1, \ldots, |I|\}$ is a permutation of tasks of I. In the case when $I = S$ the task permutation is called a *complete permutation*, when it is not - a *partial permutation.*

A triplet $\omega = (I, \pi, \gamma)$, where (I, π) is a task ordering and $\gamma : I \rightarrow 2^E$ are optimal memory states at the beginning of each task calculation, is a *solution.* When the solution contains a partial ordering, we call it *a partial solution*, and when not *a complete solution.* Let Ω be the set of all possible partial and complete solutions.

A task ordering (I', π') *begins with* the partial task ordering (I, π) if the following relations are satisfied: $I' = I \cup K$ for $K \subseteq S \setminus I$ and $\pi'(i) = \pi(i)$ for any $s_i \in I$. Respectively, solution (I', π', γ') begins with the partial solution (I, π, γ) when (I', π') begins with (I, π).

Let $f : \Omega \rightarrow \mathbb{R}_0^+$ be a bijection that assigns to any solution $\omega \in \Omega$ a non-negative value $f(\omega)$, where $f(\omega)$ represents the minimum number of external memory accesses of solution ω, calculated using Algorithm 1.

In what follows we describe each of the components used in the branch and bound algorithm.

5.1 Branching Rule

A partial or a complete task ordering (I, π_I) is respectively a node or a leaf of the search tree, here I denotes the set of already ordered tasks and π_I is a permutation of tasks I. The root node of the search tree is $(\emptyset, \pi_\emptyset)$ and it corresponds to an empty task ordering. At a node (I, π_I) of the search tree, for each task $s \in S \setminus I$ the nodes $\left(I \cup \{s\}, \pi_{I \cup \{s\}}\right)$ beginning with (I, π) are created. Thus, branching from node (I, π_I) creates a number of $|S \setminus I|$ new nodes.

5.2 Lower Bounds

Let $\omega = (I, \pi, \gamma)$ be a partial solution associated to the node (I, π) of the search tree. Two lower bounds are proposed for computing an underestimated value of current partial solution. Because of limited space the proofs of the lower bounds are omitted (see [3] for complete proofs).

Proposition 2 (Lower bound 1). *Let $\omega = (I, \pi, \gamma)$ be a solution. Suppose that $\omega' = (I', \pi', \gamma')$ is a new solution that begins with ω, where $I' = I \cup \{s_d\}$ and s_d is a dummy task containing all the inputs of not yet ordered tasks, i.e. $\delta(s_d) = \bigcup_{s \in S \setminus I} \delta(s)$. Then $g_1(\omega) = f(\omega')$ is a lower bound of solution ω.*

In the following proposition a computationally cheaper lower bound, but weaker than the previous one is described.

Proposition 3 (Lower bound 2). *Let $\omega = (I, \pi, \gamma)$ be a solution. Then $g_2(\omega)$ is a lower bound of ω:*

$$g_2(\omega) = f(\omega) + \left| \bigcup_{s \in S \setminus I} \delta(s) \setminus \delta\left(s_{|I|}\right) \right| - B$$

where

$$B = \min \left(C - \left|\delta\left(s_{|I|}\right)\right|, \left| \left(\bigcup_{s \in S \setminus I} \delta(s) \cap \bigcup_{s' \in I \setminus \left\{s_{|I|}\right\}} \delta(s') \right) \setminus \delta\left(s_{|I|}\right) \right| \right)$$

5.3 Dominance Relation

A dominance relation, described in Proposition 4, is proposed. It is applied before the exploration starts, so as to divide the search space into several independent search spaces.

Proposition 4. *Suppose that the set of tasks S can be divided into p distinctive sets of tasks $S_1, \ldots, S_p$ that are not using common inputs, thus verifying relations: $S_i \cap S_j = \emptyset$ and $\bigcup_{s \in S_i} \delta(s) \cap \bigcup_{s' \in S_j} \delta(s') = \emptyset$ for any $i \neq j$. Then any complete solution $\omega = (S, \pi, \gamma)$ is dominated by solution $\omega' = (S, \pi', \gamma')$ that is built from p independent permutations of tasks (S_k, π_k), $k = 1 \ldots p$, that is to say $(S, \pi') = [(S_1, \pi_1), (S_2, \pi_2), \ldots, (S_p, \pi_p)]$.*

In order to divide the set of tasks S into p distinctive sets $S_1, \ldots, S_p$, a graph theory algorithm for finding connected components is used [10]. This algorithm is applied on the input dependency graph $G_{ID} = (V, A, t)$, defined as follows. An *input dependency graph* is an undirected graph $G_{ID} = (V, A, t)$, where the set of nodes V represent algorithm tasks (i.e. $V = S$), the set of edges A represents dependencies on inputs between the tasks and $t : A \to \mathbb{N}^*$ is a weighting function assigning to each graph edge $(s_i, s_j) \in A$ a positive number, $t(s_i, s_j) = |\delta(s_i) \cap \delta(s_j)|$. For any pair of tasks $s_i, s_j \in V$, edge (s_i, s_j) belongs to the graph G_{ID} if and only if $\delta(s_i) \cap \delta(s_j) \neq \emptyset$.

The fact that the sets $S_1, \ldots, S_p$ are not using common inputs allows us to apply the branch and bound algorithm separately on each of these sets. In this way, the number of solution search space is reduced from $|S|!$ to $\sum_{i=1}^{p} |S_i|!$. After the application of the branch and bound procedure onto each set of task sets $S_1, \ldots, S_p$, p partial solutions $\omega_1, \ldots, \omega_p$ are obtained. Complete solution $\bar{\omega}$ is obtained by joining together these partial solutions. The cost of complete solution $\bar{\omega}$ is $f(\bar{\omega}) = \sum_{i=1}^{p} f(\omega_i)$.

5.4 Selection Rules

Before describing selection rules we introduce an useful definition. For a search tree node $\omega = (I, \pi)$, let K be a set that contains the tasks from the neighborhood of already ordered tasks I in the input dependency graph:

$$K = \{s_k \in S \setminus I : (s_i, s_k) \in A, s_i \in I\}.$$

The next node to be examined is selected in a greedy fashion, the immediate profit is privileged. The next three rules are applied successively on the set of feasible nodes. The first rule is applied until there are no nodes whose last two tasks are adjacent in the input dependency graph, the second rule is applied until the set K is empty, and afterwards the third one is applied to the remaining nodes.

1. Select the node (I, π) with the largest edge cost $t\left(s_{\pi(|I|-1)}, s_{\pi(|I|)}\right)$, such that $\left(s_{\pi(|I|-1)}, s_{\pi(|I|)}\right) \in A$, in the input dependency graph $G_{ID} = (V, A, t)$.
2. Select the currently active node (I, π) with the largest edge cost $t\left(s, s_{\pi(|I|)}\right)$, such that $s_{\pi(|I|)} \in K$ and $s \in I$, in the input dependency graph $G_{ID} = (V, A, t)$.
3. Select the currently active node $\omega = (I, \pi)$ with the least lower-bound cost $g(\omega)$.

5.5 Computational Results

Random Generated Instances. In the first part of computational results section we investigate the performance of the proposed branch and bound method applied on randomly generated problem instances.

A randomly generated problem instance is characterized by five parameters: the number of tasks $n \in \{10, 20, 30\}$, the number of algorithm inputs $m \in \{1/2, 1, 3/2, 2\} \cdot n$, the average number of inputs per task (IpT) $\mu \in \{1/2, 1/4, 1/8\} \cdot m$, the standard deviation $\sigma \in \{1/2, 1/4, 1/8\} \cdot \mu$ of IpT and the distribution used for generating IpT. Three random distributions are used for IpT value generation: $\mathcal{U}(\mu - \sigma, \mu + \sigma)$, $\mathcal{N}\left(\mu, \sigma^2\right)$ and *Exponential* $\left(\mu^{-1}\right)$. An instance is generated as follows:

- a number x of inputs per task is randomly generated using one of the distribution functions with parameters μ and σ, and x is rounded to the nearest integer (eventually x is floored or ceiled in order to verify $1 \leq x \leq m$),
- for each task s the set of inputs $\delta(s)$ is uniformly drawn from the set of all inputs $e_1, \ldots, e_m$ such that the length of $\delta(s)$ is x.

For each random problem instance we define the minimal size of on-chip memory size as $C_{min} = \min_i |\delta(s_i)|$. In order to be feasible problem's on-chip memory size C must be bigger than C_{min}. In our experiments we use six values for C, $C = r \cdot C_{min}$ where $r \in \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5\}$. For each combination of parameters three instances are generated, in total 5832 problem instances are obtained.

The branch and bound algorithm is executed on each randomly generated problem instance with a time limit of 20 minutes. We are considering as optimal solutions the solutions for which the branch and bound method explored the entire search tree. In reality the number of optimal solutions could be larger because in tree search methods much time is spent for optimality proof.

Table 1 presents the number of found optimal solutions for each parameter. From the total of 5832 problem instances 3854 were solved to optimality, which

Table 1. Optimal number of solutions (in percents) for each parameter

(a) On-chip memory size ratio r.

1.0	1.1	1.2	1.3	1.4	1.5
56%	60%	64%	68%	72%	76%

(b) Number of tasks n.

10	20	30
100%	63%	35%

$1/2n$	n	$3/2n$	$2n$
75%	66%	64%	59%

(c) Number of inputs m.

$1/2m$	$1/4m$	$1/8m$
70%	59%	70%

(d) Average number of IpT μ.

$1/2\mu$	$1/4\mu$	$1/8\mu$
71%	65%	63%

(e) Standard deviation of IpT σ.

Uniform	Normal	Exponential
50%	61%	87%

(f) IpT distribution function.

corresponds to approximatively 67%. When the on-chip memory is larger the number of optimal solutions increases as well, we can see that augmenting C by 50% ($r = 1.5$) the number of optimal solutions increases from 56% to 76%. All the instances built of 10 tasks are solved to optimality, this percentage decreases for instances of 20 and 30 tasks to 63% and respectively 35%. Instances with larger input sets (number of inputs m) become more difficult, which is counterintuitive as one can think that with the increase of number of inputs the problem becomes more decoupled because the number of common inputs between tasks decreases. Another interesting fact is that for exponential number of inputs per task, 87% of instances are solved to optimality. Which is explained by relatively large on-chip memory sizes when compared to the average number of inputs per task, thus the data reuse is privileged.

Image Processing Algorithm. A good example of easily parallelizible algorithms are the image processing ones. Nowadays image processing algorithms work with considerable amount of data, e.g. one of the smallest image resolution being 640×480 pixels. Because of this fact as well as, of course, the $\mathcal{NP}$-hardness of our problem, exact resolution of practical instances is out of reach of even the most sophisticated methods. Still, we can apply the branch and bound proposed earlier on small instances so as to probe the structures of an optimal solutions for real world image processing algorithms. Thereafter, we do so for a classical image processing primitive, the image convolution (see [8] for a detailed description).

Image processing algorithm input and output parameters are images. An algorithm task uses several pixels from the input image to calculate a pixel for the output one. We search to order the execution of tasks so as to minimize the number of external memory accesses. Then, guided by the obtained results we try to find patterns in task execution order and to generalize them to higher resolution images.

Image convolution algorithm calculates the convolution product of an image I with a kernel K:

$$\sum_i \sum_j I\left[p-i, q-j\right] \cdot K\left[i, j\right]$$

It computes the value of an output image pixel in function of its neighborhood pixels in the input image, in our experiments we use a 3×3 square neighborhood (kernel). Output image pixels belonging to image boundaries are not calculated. Figure 2a illustrates the calculation order of output image pixels found by the branch and bound procedure. The used image convolution problem instance has the minimum possible on-chip memory size, $C = 9$, being also the most constrained one, the number of input image pixels is 49 and of output image pixels 25. The optimal number of external memory accesses for this instance is 81 and it was solved in approximately 2 hours on a standard desktop PC. As often observed with branch and bound algorithms, the minimal solution is rapidly found and most of the calculation time is spent on optimality proof completion.

It is easy to see that two consecutively calculated output image pixels are either horizontal or vertical neighbors. Thus, we conjecture that if the calculation order of output image pixels satisfies the last rule, then this order is an optimal one. It is straightforward to deduce output image pixels calculation order for higher resolution images based on this rule, e.g. a possible order is given in Figure 2b. We note that for each output image pixel, 3 input image pixels must be loaded, except the first output image pixel for which 9 pixels must be loaded. Thus, the number of external memory accesses is $3 \cdot (M-2) \cdot (N-2) + 6$ for a $N \times M$ input image.

Current processor technology admits bigger on-chip memory sizes than we have used for the above image convolution instance. Therefore, we carry out an experiment that aims to find the minimal on-chip memory size allowing to load only once each one of input image pixels. Several convolution instances with different image dimensions, varying from 5×5 to 8×8 pixels, were tested. The results are presented in Table 2. We can see that if the on-chip memory

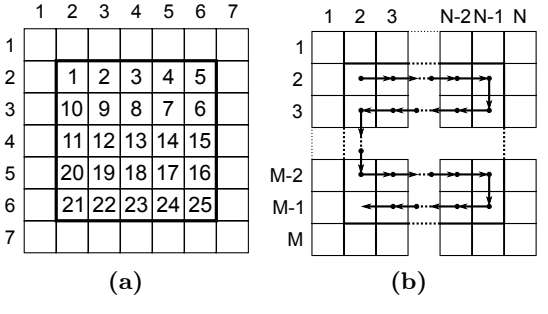


Fig. 2. Image convolution task processing order

Table 2. On-chip memory sizes allowing to load input image pixels only once

Image size	6×6	7×7	8×8	6×8	7×8	8×6	8×7
Memory size	15	17	19	15	17	15	17
Calculation order	No matter which			Vertical		Horizontal	

size equals to $2 \cdot \min(N, M) + 3$ for a $N \times M$ input image then the number of external memory accesses equals to $N \cdot M$. E.g. for a 640×480 image, if the on-chip memory size can store 963 ($2 \cdot 480 + 3$) pixels then each input image pixel will be accessed exactly once, i.e. 307200 external memory accesses; contrary to a 9 pixel on-chip memory for which approximatively 3 times more external memory accesses are needed (914898). We shall note that two different calculation orders of output image pixels were found by the branch and bound algorithm. The first one is horizontal, here output image pixels are calculated line by line[5] and the second one is vertical order, output pixels being calculated column by column.

6 Conclusion

In this paper, we have examined the task ordering and memory management problem. The aim of the last is to find a task execution order and an external memory data loading strategy so as to minimize the total number of external memory accesses for an algorithm. The main constraint is that the on-chip memory is limited in size, thus an optimal data management strategy is necessary. This problem is used to find the achievable degree of parallelism for a parallel algorithm, which in a compiler chain proceeding by parallelism reduction will help to fix an appropriate target.

Initially we have supposed that we are given a task ordering, so as to study the issue of on-chip memory data management separately from the ordering problem. We have proposed a polynomial algorithm for its resolution. This algorithm relies on Belady's principle for virtual memory management.

We have proposed a branch and bound algorithm for the task ordering and memory management problem and described its building blocks (lower bound, dominance relation etc.). Firstly we have performed computational experiments with randomly generated problem instances. The branch and bound procedure found optimal solution for more than two thirds (67%) of the cases. Afterwards, several tests were done with an image processing algorithm: the image convolution. The branch and bound algorithm was not able to solve instances of image processing algorithms applied on real, high resolution images, because the size of the search space is unimaginably huge for an exact algorithm. However we solved instances with low resolution input images and generalized the results for high resolution images.

[5] In Figure 2b a horizontal order is illustrated.

References

1. Allahverdi, A., Ng, C.T., Cheng, T.C.E., Kovalyov, M.Y.: A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187(3), 985–1032 (2008)
2. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
3. Carpov, S.: Optimisation du préfetch et du parallélisme pour plateforme MPSoC. Master's thesis, Université de Technologie de Compiègne (2008)
4. Ding, C., Kennedy, K.: Improving cache performance in dynamic applications through data and computation reorganization at run time. SIGPLAN Not. 34(5), 229–241 (1999)
5. Ding, C., Kennedy, K.: Improving effective bandwidth through compiler enhancement of global cache reuse. J. Parallel Distrib. Comput. 64(1), 108–134 (2004)
6. Ding, C., Orlovich, M.: The potential of computation regrouping for improving locality. In: SC 2004: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 13. IEEE Computer Society, Washington, DC (2004)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
8. Gonzalez, R., Woods, R.: Digital Image Processing. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
9. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII, New York, NY, USA, pp. 151–162 (2006)
10. Korte, B., Vygen, J.: Combinatorial Optimization: Theory and Algorithms. Springer, Heidelberg (2002)
11. McKinley, K., Carr, S., Tseng, C.W.: Improving data locality with loop transformations. ACM Trans. Program. Lang. Syst. 18(4), 424–453 (1996)
12. Pingali, V., McKee, S., Hsieh, W.C., Carter, J.: Restructuring computations for temporal data cache locality. Int. J. Parallel Program. 31(4), 305–338 (2003)
13. Strout, M., Carter, L., Ferrante, J.: Compile-time composition of run-time data and iteration reorderings. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 91–102. ACM, New York (2003)
14. Wolf, M., Lam, M.: A data locality optimizing algorithm. In: PLDI 1991: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (1991)