Performance Evaluation of Data Locality Exploitation

Paolo D'Alberto

Technical Report UBLCS-2000-9

July 2000

Department of Computer Science

University of Bologna

Mura Anteo Zamboni 7
40127 Bologna (Italy)

Dottorato di Ricerca in Informatica

Università di Bologna, Padova, Venezia

Performance Evaluation of Data Locality Exploitation

Paolo D'Alberto

January 2000

Coordinatore:
Prof. Özalp Babaoğlu

Tutore:
prof. Gilberto Filè

To my father Giacomo

Oct. 7 1999

Abstract

Data Locality exploitation and Performance Evaluation are important issues in compiler and algorithms so that modern architectures can be fully utilized. Multiprocessor platforms, with RISC processor and multiple level of caches, are becoming very common platforms. In spite their computational capability, they are not easy to be modeled and therefore application performance is not easy to be estimated. Indeed, it is often the case that developers code applications that can obtain very different performance on different platforms. This almost magic but unpleasant behavior is due to many factors. One of them is *Data Locality.* A smart reutilization of data, stored at the different levels of the memory hierarchy, may improves performance.

In this investigation we are looking for a machine independent definition of data locality, inherent to the application and a general technique to exploit it automatically at source code level: reorganizing the schedule of instructions or, sometime, rewriting the algorithm. Indeed, we are proposing a machine independent method to estimate data locality, *Access Complexity*, we are proposing an heuristic to schedule instructions and manage memory, then we apply our ideas on a case study, *matrix matrix multiplication.*

Acknowledgements

For all the advises I got from, for the help I got when applications could not work, and for all the neverending discussions I have to thank the following people (in alphabetical order):

- Azevedo, Ana: about optimizations and SPARC architecture consulting
- Bilardi, Gianfranco: my advisor.
- Capitanio, Andrea: suggestion for application and different subjects.
- Fiorini, Paolo: support.
- Nicolaescu, Dan: master of *gcc*, multiple load/store
- Nicolau, Alex: my advisor
- Pietracaprina, Andrea: k-marking
- Savoiu, Nicolae: without him I think I would not have run any application on Windows

Contents

List of Figures

Chapter 1

Introduction

Programmers would like available unlimited amount of zero time access memory so that their scientific application can grow in size without loosing any CPU's cycle waiting data from memory. Current memory chips are cheaper and more capable day by day, but memory latency has not followed neither the increasing capacity nor the increasing processor performances. Data Accessing is becoming a bottleneck for actual computations. An economical solution to avoid high latency time is to build up a hierarchy of memory levels with different latencies, but since applications do not access uniformly the memory space, an architecture solution can be good for an application but for others might not. Instead to find the right application for the right architecture, usual approach for embedded systems, we can *adapt*, or tune, the code of the application, which is software, to the architecture.

A common model for the current memory hierarchy is to identify a location by an address, locations in caches closer to the processor are indicated by low addresses, and so the lower it is the address location the faster it is the time to access data [2], [4], [39], [56], [58], [37]. The most used data are to fit into low addresses. This model can be generalized for parallel and distributed machine. Communications among processors can be seen as remote accesses to a memory location, communications as memory accesses within an extended range of addresses [3], [12], [10], [41], [47], [14].

Most of the *trails* to reach high performance through exploiting data locality are carried on in two no distinct subjects. Transforming source code to satisfy machine memory hierarchy is automatically carried on by a compiler [5], [7], [16], [19], [33], [45], [53], [54], [52], [66], [67], [68] or *hand coded* by a programmer, in parallel programming environments, [8], [11], [35], [62]. Experimental results indicate that these last paradigms might be no complete and some authors identify this model lack through a no correct model of memory hierarchy and interprocessor communications [42], [25].

Current software applications are based on different paradigms and technologies. The most used model it is proposed by the WWW world, which usually can be seen as dynamic client/server paradigm [51], [21], [48]. There are many applications on this paradigm where exploitation of the data locality can give interesting results. Consider an applet [6], [46], this is a semi interpreted code which a server sends to a remote host. The code cannot be written with any dependent machine optimization because hosts are unknown until the applet is sent to. Only the receiving host can perform machine dependent optimization of the code before applet execution. But very often, local compiler needs more information, which is available only from the original code, high level implementation, [1], [36]. Such a result can be amplified when applications sent are more than simple applets but more complex applications [22], [50], [36], i.e. CPU time consuming and in general resources consuming. This example underlines important aspects of data locality and in particular inherent data locality of a program (independent from any host machine). We think that data locality optimization can be performed in steps. First of all there is the exploitation of inherent data locality of the program, and then locality exploitation respect any particular support, where applications are executed.

We propose a model of computation based on computation DAG (direct acyclic graph) which is not new [39], [10], [2]. A computation DAG describes precedence relations of simple functions. A node in a DAG is a simple function whose arguments are the output of node's predecessors. We propose a technique which estimates the minimum number of accesses over a threshold location in memory. Such a minimum number is called access complexity [10], and it is a reformulation of well known I/O complexity [39]. The threshold is a parameter. We can say that our technique counts inherent data locality of the algorithm. But also it can be used to estimate data locality respect a specific architecture, just fixing the value of threshold to machine parameters.

DAGs express algorithms. We can see that a strict topological order of a DAG induces an instruction schedule and therefore it specifies a particular execution of the algorithm.

A compiler should decompose properly a DAG, in sub-DAGs, and it should induce a decomposition and a strict topological order among nodes of DAG, so that every sub-DAG can be executed sequentially and/or concurrently. We propose heuristics that automatically partitions the DAG using a divide and conquer technique and it is based only on topological properties of DAG. This algorithm suggests a way to exploit data locality. We propose also an algorithm so that for a scheduled DAG the memory can be managed properly and it can be written the *code* for the DAG.

But performance is determined not only by data locality. To maximize performance of current workstations, parallelism, such as pipelined and multiple functional units architectures, must be fully utilized. To achieve such goal the stream of instructions should be organized properly. At very fine grain, we can claim that code should have no data locality at all. We are implying that data locality exploitation is not always our goal, we have to understand when our goal changes. This is a very challenging problem, because often *when* we have to change policy it is machine dependent. For this reason, we think that a divide and conquer technique can offer the required flexibility to change policy during problem decomposition ([31]).

Any complexity model is based on estimation of single operation, or basic operation. To give an idea of the estimation problem, we propose an example. Consider a pipelined unit with nine stages. If we consider a worst case scenario, an algorithm do not utilize the pipeline structure and every instruction is finished every nine cycles. A best case is when the pipeline is fully utilized and any instruction is accomplished every cycle. There is a factor of nine. A constant factor in worst case analysis can be negligible but for real application performance is remarkable. In this work when we estimate performance we are taking in account carefully the constant factor. Data misses in memory hierarchy can affect execution time in several ways, because they do or do not stop execution of instructions. Some architectures permits to hide memory latency, in particular, latency among caches, feeding the CPUs with useful instructions while they are waiting data from memory. We know that to give a complete model of modern complex architecture is not feasible, but we want to gain a *taste* of the problem and to formalize a naive complexity model. We propose a *general approach* to measure computational performance of different platforms. From our point of view, performance of a platform is its capacity to execute computations and memory accesses. We have devised a set of performance tests to measure the characteristics of a platform. We are interested to understand average values and quantitative measure of the architecture itself. Indeed, we can measure cache sizes, cache line sizes, number of stages of a pipelined functional unit through a sequence of time measures. We had designed and written performance tests in C and we have not forgotten an important detail, we want to understand how compilers exploit hardware characteristics of processors. Very often we use native compiler, but not always is possible. So, we obtain an average value which comprises not only *hardware* but also *software* properties.

We propose an intriguing, and well known, example: *matrix multiplication* ([54] ,[24] and [27]). Indeed, this is a well known algorithm where data locality exploitation and instruction reorganization characterize execution time. We study the *good and old* divide and conquer approach ($O(n^3)$ algorithm). It exploits data locality because divides the problem in sub-problems, it solves them locally and combine the results. A non standard layout of matrixes minimizes cache misses, capacity misses and cross interference among data (Z-Morton layout). We highly optimized the algorithm reducing the overhead due to its recursive implementation, we devised a heuristic to unfold leaves and we suggested register allocation. In the literature we can find all these aspects investigated separately, but rarely they were considered together and how they can affect each other. The efficiency of our algorithm and its implementation is compared with other algorithms such as in standard libraries, in vendor libraries and libraries obtained by machine dependent approach. The experimental results are very competitive.

Chapter 2

Data Locality

2.1 Definitions and Notations.

Let $G = (V, A)$ be a directed acyclic graph (D.A.G) where V is the set of nodes and A is the set of arcs. The number of outcoming arcs from a node $v \in V$ is denoted by d_v and the number of incoming arcs to a node v is denoted by γ_v. The maximum out degree of any node in a DAG is denoted by $\Delta = \max_{\forall v \in V} d_v$, and the maximum in degree of any node is denoted by $\Gamma = \max_{\forall v \in V} \gamma_v$. If a node has no incoming arcs we say that the node is an *input node*. If a node has no outcoming arcs we say that the node is an *output node*. Any node, which is neither input nor output, is called *internal node*. For every arc $\alpha = (u, v) \in A$ we say that the node v is a *successor* of u and the node u is a *predecessor* of v. For every pair of nodes $u, v \in V$ such that there is a directed path in G from u to v, we say that the node u is an *ancestor* of v and v is a *descendant* of u. For example, in

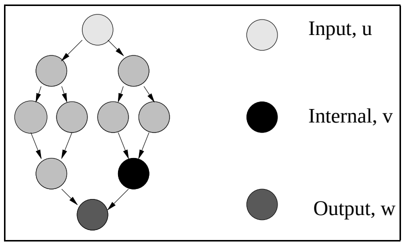

Figure 2.1: Example

Figure 2.1, the output w is successor of v and descendant of any node. The input node u is ancestor of any node.

A *computation DAG* is a directed acyclic graph G whose nodes are associated with values as follows: value of a node with in degree zero is regarded as input, while value of any other node is obtained by applying a certain function, associated with the node, to values computed at the node's predecessors. Consequently, to evaluate an internal node v, it is necessary to evaluate every ancestor of node v. At time node v is evaluated, there must be available the values of v's predecessors. It can be easily seen that to evaluate every output node we need to evaluate every node of the DAG G (otherwise no evaluated nodes can be removed safely). We define as *DAG evaluation* of DAG G, the evaluation of all output nodes in G. Every DAG evaluation must start from input nodes. Inputs have no predecessor and they cannot be evaluated as other nodes. We suppose that input values are available by a function without arguments that we call *query*. A query is performed in two steps: we look input value up, which can be into a dedicated space memory devoted to store input values, and then we store value into a location of the memory workspace of DAG evaluation. We spend no time looking inputs up, but it takes time to store input values. In every DAG evaluation it is permitted more than one query for an input node. So, there can be many copies of the same input value into different locations. Note that for a computation DAG there can be different DAG evaluations. Certain DAG evaluations evaluate every node once but other DAG evaluations evaluate some nodes more than once. The first DAG evaluation is called DAG evaluation without recomputation[12]. Such a DAG evaluation performs the minimum number of node evaluations. We consider DAG evaluations of G such that the *operations* devoted to DAG evaluation are performed sequentially and we define what we mean as operation shortly. In another words, a DAG evaluation can be described as a sequence of operations on nodes performed one at time. We consider a node evaluation as an operation. An example of node evaluations follows. Consider an internal node $u \in V$ with predecessors $v, w, x \in V$ and x is an input node. Before evaluation of u at time t_u the values of u's predecessors are available in memory. The values of v and w are stored at addresses add_v and add_w, respectively, after theirs evaluations. The value of x is stored at address add_x after a query. Node values are computed before t_u through every DAG evaluation. At time instant t_u, these values are accessed one at time and u is computed. The result value is stored at address add_u and the node evaluation is considered done. Note that we do not specify any order of accesses for operand read. *Access a value*, we mean to load a value or to store a value at a memory address. A query can be considered as a computation without arguments and therefore without arguments fetching. We consider any data moving within memory as an operation (i.e to load a value stored at an address and to put it into another one, which may be easier to access for performance purpose).

We can note that for a computation DAG G, there can be many different sequences of operations that evaluate output nodes. With symbol ξ_G we indicate ordered sequence of operations performed by DAG evaluation of G. With symbol Ξ_G we indicate the set of all ordered sequences of operations that are DAG evaluations of G. We can note that whether we take in account number and type operations performed in a DAG evaluation, we can give an estimation of the time spent to do the DAG evaluation. We are interested to investigate lower bounds of time complexity for computation DAG. We are not interested to count all operations but a specific kind that can be considered as indispensable operation. In particular we are interested on evaluations that access node values stored over a threshold location. We indicate as *evaluation access*, $q(\xi_G, T)$, the number of memory accesses over the location of address T, threshold included, performed by a DAG evaluation ξ_G. We call *Access Complexity*, $Q_G(S) = \min_{\xi_G \in \Xi_G} q(\xi_G, T)$ the minimum number of memory accesses over the location of address T for all DAG evaluations. The concept of access complexity is equivalent to the notion of I/O complexity in [39] and the access complexity in [12] but it is different the type of DAG evaluation considered. In the first paper [39], the authors consider input values stored over the threshold location at the beginning of DAG evaluation. In the second paper [12], the authors consider only DAG evaluation without recomputation. Now, we briefly recall the machine model used to perform DAG evaluation in [39] and their ideas. They describe a computational model with an unbounded memory workspace splits in two levels. The first level is bounded. It ranges from the location 0 to S-1 and the access time to any one of its locations is considered inexpensive, zero time access. The second level is unbounded. It ranges from S to infinity and access time to any one of its locations is a constant. The time spent for a node evaluation is considered zero. In this model a lower bound to time complexity for a computation DAG is access complexity $Q_G(S-1)$. The results obtained from access complexity can be applied when memory model is more complex. A *hierarchical memory model* is a possible unbounded memory such that if we access a memory location of address x then we spend time $f(x)$ ($f(x)$ is a monotone non decreasing function on x which is a positive integer number). We call such a model an *$f(x)$-memory model*, [4]. Also for a machine which has such a hierarchical memory as support and any computation time is negligible (therefore latency cannot be hidden), time complexity of a computation DAG is access complexity. In fact, the time spent by any DAG evaluation is at least $\sum_{k=0}^{M} f(k)[Q_G(k) - Q_G(k+1)]$[1] [4]. The right term of the sum, $Q_G(k) - Q_G(k+1)$, is the minimum number of accesses on memory location of address k. $f(k)$ is the latency time to access a memory location. As we did for access complexity we can define what we mean for memory workspace of a computation DAG. We call *evaluation space*, $\mu(\xi_G)$, the number of memory locations needed for a DAG evaluation ξ_G. We call *Space*, $\mathcal{M}_G = \min_{\xi_G \in \Xi_G} \mu(\xi_G)$, the minimum number of memory locations needed for all DAG evaluations.

We reformulate Hong and Kung's theory proposing a new method based on space complexity. Indeed, if we know that a computation DAG needs at least $\mathcal{M}_G$ memory locations for each DAG evaluation, then each location is accessed at least once. Therefore $Q_G(S) \geq \mathcal{M}_G - S$ with $1 \leq S < \mathcal{M}_G$. This naive method cannot take into account multiple accesses to the same location and during a DAG evaluation memory locations can be accessed several times. Suppose to observe an interval time of a DAG evaluation and to observe the accesses at a location add_x. A node value can be stored at add_x and it can be read many times, it is *reused*, but node values can replace value at add_x. We propose a technique which induces a partition in subDAGs on a computation DAG, and it permits to count part of these data replacements. In fact, we can look for access complexity of every subDAG by a lower bound to the space complexity and we can sum each contribution.

2.2 Hong and Kung's Results

The first example and characterization of access complexity for computation DAGs was formalized by Hong and Kung [39]. They associate every DAG evaluation with a so called *S-Partition* through a particular sequence of rules. Then, they look for access complexity lower bound, $Q_G(S)$, based on S-Partitions. The model of the evaluation is not equal to our model. Consider a DAG $G = (V, A)$ and suppose that before the beginning of any

[1]We define $\Delta f(m) = f(m) - f(m-1)$, $\mathbb{1}(x)$ the step function and $f(x) = 0$ $x < 0$: $f(x) = \sum_{m \leq x} \Delta f(m) = \sum_m \Delta f(m) \mathbb{1}(x-m)$ then if we indicate with n_x the number of accesses at address x the time spent in this model is $T_f = \sum_x f(x) n_x = \sum_x \sum_m \Delta f(m) \mathbb{1}(x-m) n_x = \sum_m \Delta f(m) \sum_x \mathbb{1}(x-m) n_x \geq \sum_m \Delta f(m) Q_G(m) = \sum_m [f(m) - f(m-1)] Q_G(m) = \sum_m f(m) Q_G(m) - \sum_m f(m-1) Q_G(m)$.

evaluation, input values are already in memory. We can consider such an evaluation as a DAG evaluation which performs for every input exactly one query at the beginning of the evaluation and all these operations are considered time inexpensive.

We define as *minimum set* $M(U)$ of a set $U \subset V$ the set of nodes in U that have all directed successors not in U. We define as *dominator set* D a set of nodes in V such as for every path that connects an input node to a node in W there is at least a node into the path that is into the dominator set. Since for any set of node there can be many dominator sets, we define as the *minimum dominator class* $D(W) = \{D : D$ is a dominator set of $W$and $\forall R$ dominator set of $W, |D| \leq |R|\}$. We can see any member of the minimum dominator class as a *minimum cut*. Whether the nodes of a minimum cut are removed from the DAG the inputs are disconnected from any node in W. We recall the following definition which is necessary to understand the results obtained by Hong and Kung.

Definition 2.1 *an S-Partition for V is a partition $V_1, V_2, \ldots, V_h$ of V such that:*

- $\bigcup_{i=1}^{h} V_i = V$ *and* $\forall i, j$ *such as* $i \neq j$ *and* $V_i \cap V_j = \emptyset$.
- $\forall i \in [1, h]$ *there is a* $D \in D(V_i)$ *so that* $|D| \leq S$.
- $\forall i \in [1, h]$ *it must be* $|M(V_i)| \leq S$.
- *For any $i > j$, there is no arc from a node $v \in V_i$ to a node $u \in V_j$.*

And now, we recall the result of Hong and Kung [39, Theorem 3.1]

Theorem 2.1 *Every DAG computation ξ_G on DAG G is associated with a $2S$-Partition such that $Sh \geq q(\xi_G, S) \geq S(h-1)$, where h is the number of sets in the $2S$-Partition.*

The proof of the theorem is based on the following idea. Let T be the time interval associated with an DAG evaluation ξ_G and consider a set of instants $t_1 \leq t_2 \leq \ldots \leq t_{h-1}$ that divide T into h subintervals $T_1, T_2, \ldots T_h$. This partition induces a partition $V_1, V_2, \ldots, V_h$ of V where V_i contains all nodes v satisfying the following three properties:

1. during T_i v is evaluated or its value is read from a memory location of address greater than S;
2. at the end of T_i, v is either in memory or it has a descendant in V_i;

3. v does not belong to any V_j with $j < i$.

In [39], they show that for any DAG evaluation it is possible to select instants $t_1 \leq t_2 \leq \ldots \leq t_{h-1}$ such that the induced partition $V_1, V_2, \ldots, V_h$ is an $2S$-Partition and such that $Sh \geq q(\xi_G, S) \geq S(h-1)$. The argument given in [39], is rather complex and the difficulty is represented by proving that the union of the V_i's is indeed V. They prove that for any DAG evaluation there is a $2S$-Partition. They prove that to find a lower bound to h, the cardinality of the set of the partition, it is to find a lower bound to any access evaluation, that is, the access complexity. For any $2S$-Partition of a DAG composed by h parts we can see that $h \geq \frac{|V|}{\max|V_i|}$. Any technique to estimate upper bounds to any V_i of a $2S$-Partition induces an estimation of a lower bound to h. They propose the concept of span from $2S$ values, $\rho(2S)$. They define $\rho(2S) = \max\{|W| : \forall d \in D(W) \text{ and } |d| \leq 2S\}$. It is easy to see that for all V_i in any $2S$-Partition of a DAG $|V_i| \leq \rho(2S)$. Eventually, we can write that $Q_G(S) \geq \frac{|V|}{\rho(2S)}(S-1)$. This idea is refined and simplified in several articles [2], [58].

From a motivating example, Diamond DAG, we start our investigation. For diamond DAG the authors suggest a naive lower bound. Our model of DAG evaluation is more general and it cannot be accepted their assumption on input nodes. We prove that the lower bound it is still the same but with different arguments. We can say that our method improves the method proposed but it is not better. In the sense that our technique gives tighter lower bound than the technique proposed by Hong and Kung for the diamond DAG but it does not give every time the tightest result for other DAG (i.e. FFT DAG). We investigate why such technique differs in the following.

2.3 k-closed Marking Method

Our aim is to find a topological characteristic of DAG evaluations. We want to infer a technique which is able to take care of accesses in any DAG evaluation but it can be applied directly to the DAG without any knowledge how DAG evaluations are performed. This simple idea is well known and applied. But our work differs from previous ones because we introduce the concept of *marking*. Informally, we can say that a marking is a method for choosing evaluations, that is, it is a method to select from a DAG evaluation node evaluations. We know that a node can be evaluated more than once: between two consecutive evaluations of a node, its value can be stored in a memory location or it can be removed at any time after its evaluation. We need a preliminary sieving of evaluations because we do not have to consider all evaluations. We need to focus on node evaluations used to evaluate its successors, unless it is recomputed successively. We need to focus on the evaluations carried for the evaluations of output nodes. So, we introduce the concept of *useful evaluation*. If v is an output, its first evaluation is a useful evaluation. If v has successors, then we define as useful any evaluation, or query, such that value computed is used for a useful evaluation of at least one of its successors. We do not consider isolated nodes because they are trivial DAGs. They are inputs and outputs at the same time. For our purpose, we can restrict our DAG evaluations to useful evaluations.

Lemma 2.1 *Every node v in a computation DAG G has at least a useful evaluation in every DAG evaluation of G.*

Proof: by induction on the distance from outputs of G. Every output has at least a useful evaluation in any DAG evaluation. Suppose that every node v, which has distance at most $k-1$ from an output node, has a useful evaluation in any DAG evaluation. Consider any node w at distance k from any output, that is, there is an output which has distance k from w and the other outputs have distance greater than k from w. Then, take a successor s of node w which in the DAG computation uses the node value of w. It exists otherwise we can remove the node w from the DAG without changing the meaning of the DAG. From the node s we can keep on with the same arguments above until we meet a node u which has distance at most $k-1$ from outputs. The node u has a useful evaluation in every DAG evaluation. When a useful evaluation of u is performed, there are available the values of predecessors of u. There must exist an evaluation or a query on predecessor which produces such a value, and this is a useful evaluation. If we propagate backward such a useful evaluation we reach the node w. The node w has a useful evaluation. $\square$

We can say that choosing from any DAG evaluation only useful evaluations is a first sieving. The precedence relation between nodes u and v where u is descendant of u is respected by their first useful relation as the following lemma states.

Lemma 2.2 *If in a DAG evaluation ξ_G the first useful evaluation of node u is at time instant t_u then every descendant of v has a useful evaluation after time instant t_u.*

Proof: the proof is done by contradiction. Suppose there exists a successor v of u which has a useful evaluation at time instant t_v before the time instant t_u. Node v at time instant t_v needs the value of its predecessors and therefore of u. The evaluation of u which produces such a value would be a useful evaluation and since it is performed before instant t_u it would be the first useful evaluation of u. This is a contradiction. Therefore every successor of node v has its first useful evaluation after the first evaluation of v. We can continue in the same manner with the successors of v and eventually we can find that every descendant of node v has its first useful evaluation after instant t_u. □

For sake of simplicity, in the following we omit the adjective *useful* and for us evaluation of a node means useful evaluation of a node.

The *marking* of a computation DAG $G = (V, E)$ is the choice of only one node evaluation of node v, for all $v \in V$, in each DAG evaluation ξ_G. Shortly, we indicate the evaluation selected in a DAG evaluation by a marking as a *marked evaluation* and the node, which has a marked evaluation, as *marked node*. Therefore a *marking method* on computation DAG G is a specific method to mark every node of G in any one of its DAG evaluation ξ_G.

Definition 2.2 *as k-*closed marking method *of a DAG evaluation ξ_G we define the following technique: we go backward from the last to the first operation in the DAG evaluation ξ_G and we mark an evaluation of a node v, if it is not marked yet and if at least* $\max(1, d_v - k + 1)$ *of its direct successors are marked.*

Note that the outputs have no successors in DAG G and in any DAG evaluation ξ_G they have at least an evaluation, therefore they can always be marked.

Lemma 2.3 *The k-closed marking method is a marking method.*

Proof: by construction every node is marked at most once and we can prove that every node is marked by contradiction. We know that every output is always marked. Suppose there exists a no output node v that is not marked in a DAG evaluation ξ_G. We observe the first evaluation of v at time instant t_v in ξ_G. Clearly, before time instant t_v there is no evaluation of any successor of v. Since we cannot mark the evaluation of v at t_v, there is at least one of its direct successors that is not marked after time instant t_v and therefore in ξ_G. Let u be such a no marked node and let t_u be the time instant of its first evaluation on ξ_G. Time instant t_u follows time instant t_v. Now, from the node u we can keep on searching no marked node and by the acyclicity of DAG eventually we can find out an output which cannot be marked and this is a contradiction. □

We do not state that the first evaluation is always the evaluation marked. We state that since there is the first evaluation of a node there exists an evaluation that can be marked. There can be more than one evaluations for a node v but when we go backward on the computation we will mark the first allowed.

2.4 Space Complexity Based on k-closed Set

We consider a DAG evaluation ξ_G on DAG $G = (V, A)$ and we observe at time instant t_v, which is the first evaluation of a node v. Time instant t_v splits the DAG evaluation in two parts. Then we can split the DAG into two subsets. Into the first subset there are all nodes evaluated before time instant t_v and into second subset there are all nodes evaluated no early time instant t_v. By Lemma 2.2 we know that all successors of v must belong to the second set because they are associated with the right part of the DAG evaluation ξ_G. We can obtain a bipartition of the DAG from any DAG evaluation ξ_G and any time instant in ξ_G. *k-closed set* is a set $W_k \subset V$ which can be seen as the composition of two disjoint sets I_k and B_k. For every node $v \in W_k$ if v has at least $\max(1, d_v - k + 1)$ successors into $V - W_k$ then v is into B_k and if v has at most $\min(d_v, k)$ successors in W_k then it is into I_k. We define as *k-closed bipartition* (W_k, W_k^c) in $G = (V, A)$ two subsets of V such that $W_k \cup W_k^c = V$ and W_k is k-closed.

Lemma 2.4 *Take a computation DAG $G = (V, A)$ and time instant t_0 into DAG evaluation ξ_G, the set of nodes marked by the k-closed marking method earlier than t_0, is a k-closed set.*

Proof: DAG evaluation ξ_G is marked through k-closed marking method, so any marked node v has at least $\max(1, d_v - k + 1)$ successors marked in time instants after the marked evaluation of v. There must be some nodes marked before t_0 which have at least $\max(1, d_v - k + 1)$ successors marked after t_0, we call such set of nodes B. There must be all other nodes marked before t_0 which have at most $\max(1, d_v - k)$ successor marked after t_0, otherwise the set is empty. Therefore they have at most $\min(d_v, k)$ successors marked before t_0, we call such a set I. $W = I \cup B$ and it is k-closed. □

For every DAG evaluation ξ_G and a time instant t into ξ_G we can derive a bipartition of the DAG G. The bipartition is a k-closed bipartition and set indicated as W_k has an interesting characteristic, its subset indicated as B_k, *frontier*, can derive space complexity for the DAG evaluation ξ_G.

Theorem 2.2 *For every DAG evaluation ξ_G and time instant t in ξ_G, we can derive a k-closed bipartition (W_k, W_k^c) such that $W_k = (I_k \cup B_k)$ and at t the values of the no output nodes in B_k are stored in memory.*

Proof: by Lemma 2.4, any time instant t into ξ_G can determine a k-closed bipartition

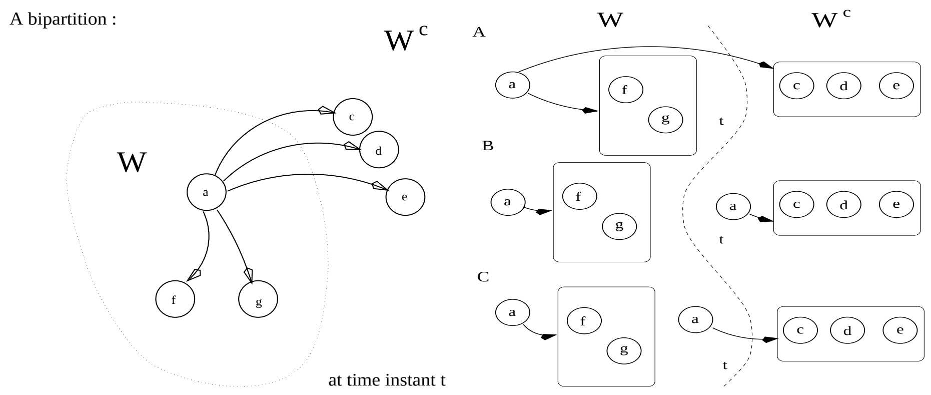

Figure 2.2: Bipartition, case B is not possible.

(W_k, W_k^c) such that $W_k = (I_k \cup B_k)$. The outputs evaluated have not to be stored until the end of DAG evaluation, so we cannot state that they are in memory at time t. If $v \in B_k$ is not an output node it has at least $\max(1, d_v - k + 1)$ successor in W_k^c then **1)** one of its successors is in W_k^c and in W_k and its value is used by both, **2)** one of its successors is in W_k, it uses v, no node in W_k^c uses it before t and it is re-evaluated after t or **3)** it is re-evaluated right before t. But the second case is not possible, because the node v would belong to W_k^c, since it has the necessary number of successors marked in W_k^c and it is evaluated after or at time instant t. □

We have now a technique to estimate the minimum space which is needed by a DAG evaluation ξ_G on computation DAG G.

Theorem 2.3 *Take a computation DAG $G = (V, A)$ with size $|V| = n$ and indicate with O the set of output node in G then the minimum space $\mathcal{M}_G$ for all DAG evaluations in Ξ_g is at least*

$$\beta(G) = \max_{1\leq k\leq \Delta}\{\max_{1\leq j\leq n}\{\min\{|B_k - O| : (W_k = I_k \cup B_k, W_k^c) \text{ } k\text{-closed bip.}, |W_k| = j\}\}\} \quad (2.1)$$

Proof: by contradiction. Suppose $\mathcal{M}_G$ is smaller than Equation 2.1 indicates. It exists a DAG evaluation ξ_G such that it has at any time instant at most $\mathcal{M}_G$ node values stored in memory. On ξ_G we apply the k-closed marking method $(1 \leq k \leq \Delta)$. Sequentially, we can find out any instant time when an node evaluation is marked. We indicate such a sequence of time instants as $\{t_j\}$ with $1 \leq j \leq n$. So, at any instant time t_i we can determine a k-closed bipartition $(W_{k,i}, W_{k,i}^c)$ such that the size of $|W_{k,i}| = i$. Note that the node values of the frontier $B_{k,i} \setminus O_G$ are in memory at t_i. Obviously, $\mathcal{M}_G$ must be greater than $|B_{k,i} \setminus O_G|$ for every $1 \leq i \leq n$, in turn, greater than $\min_{|U_k|=j}\{|B_k \setminus O| :$ B_k frontier of U_k, (U_k, U_k^c) k-closed bipartition and $|U_k| = j\}$ (for every $1 \leq j \leq n$). This is a contradiction. □

For sake of readability, hereafter we indicate as $\beta_{k,j}(G) = \min\{| B_k \setminus O |: (W_k = I_k \cup B_k, W_k^c)$ k-closed bip. of G and $| W_k |= j\}$. Furthermore we concisely define $\beta_k(G) = \max_{\forall i} \beta_{k,i}$. The Equation 2.1 can be concisely written as $\max_{1 \leq k \leq \Delta} \beta_k(G)$.

2.4.1 Partitionability

Note that we can apply formula 2.1 to achieve an estimation for the access complexity. A naive approach is to say $Q(S) \geq \mathcal{M}_G - S \geq \beta(G) - S$ where $\beta(G)$ is the estimation of the space needed by Equation 2.1. Every memory location is counted as an access. But we know that memory space can be reused and accesses may be multiple on same location for different node values. We explain a method to overcome, partially, this problem and for this purpose we introduce the concept of *partitionability* of a computation DAG $G = (V, A)$. We identify as $I(H)$ with H subDAG of $G = (V, A)$ the set of nodes in V such that either they are input nodes or they have at least a predecessor not in H. The nodes in $I(H)$ is defined as *extended ed inputs*. To consider H as a DAG, we remove every arc $a \in A(G)$ which is incoming to a extended inputs (if we perform a query to evaluate an extended input, it is not necessary any predecessor). So if the subDAG H is considered as isolated DAG, its inputs and extended inputs are the same. We identify as $O(H)$ the set of nodes in V such that either they are output nodes or they have at least a successor not in H. The nodes in $O(H)$ is defined as *extent ed outputs*. For these nodes, we remove arc outcoming from any extended output but arc which is outcoming from extended output and is incoming to extended output. Any evaluation of extended outputs are treated as output evaluations. Therefore, if we apply any k-closed marking method on any DAG evaluation ξ_H as isolated DAG to evaluate space μ_H we must remove from the frontier of every k-closed set its extended outputs (recall Theorem 2.2).

Definition 2.3 *A DAG G is a composition of j DAGs without overlapping if they respect the following rules:*

- *for every $1 \leq i \leq j$ the subDAG G_i can be seen as $I(G_i) \cup \dot{G}_i \cup O(G_i)$.*
- *for every $1 \leq i < k \leq j$ it must be $G_i \cap G_k = \emptyset$*[2]

Theorem 2.4 *Let G be obtained by the composition without overlapping of j DAGs $G_1, .., G_j$ then $Q_G(S) \geq \sum_{i=1}^{j} (\beta(G_i) - S)$.*

Proof: let ξ_G be a DAG evaluation. It is always possible to recognize j sets of subintervals of ξ_G, A_i with $1 \leq i \leq j$, so that in A_i there are all the operations (evaluations, queries and data movements) on nodes of G_i. Then we reorganize every A_i in $\dot{A}_i$ so that it becomes a time interval. The precedence relation between operations in $\dot{A}_i$, respects the precedence relation between operations in A_i. In other word, $\dot{A}_i$ are a compacted time intervals of A_i and if a node v is evaluated at time t_v and a node w at t_w, with $t_v < t_w$ and t_v and t_w are in A_i, then the corresponding evaluation time instants in $\dot{A}_i$, say t_v^1 and t_w^1, have the same precedence relation $t_v^1 < t_w^1$. We consider the DAG G_i as a single DAG. We can see that queries can be performed only on the input nodes. The inputs are into $I(G_i)$. The node in $I(G_i)$ and in $O(G_i)$ have a lesser number of constrains than the same node in G. Therefore it easy to see that there is a DAG evaluation ξ_{G_i} of G_i, as single DAG, for which to mark ξ_{G_i} or $\dot{A}_i$ gives the same result. To each DAG evaluation so determined, we can apply the estimation of a lower bound to the space complexity by k-closed marking method. There are at least $\beta(G_i) - S$ accesses over the location S. Since A_i are distinct intervals, on which are involved nodes of different subDAGs, we can add the contribution to the access complexity of each subDAG. We are done. □

2.5 Examples

We apply the k-closed marking method to different DAGs. Our aim is to show that there is no *preferential* k to estimate the space complexity of a DAG with a k-closed marking

[2] $V(G_i) \cap V(G_k) = \emptyset$

method. We propose DAGs for which different values of k must be chosen to obtain tighter lower bounds. Furthermore, we compare the goodness of our technique quantitatively (Ω notation) respect to the other techniques in literature. Such a comparison will permit to understand what is *captured* from any DAG evaluation by the k-closed marking method and what cannot be captured. We show that our technique is different from the other techniques and for particular DAGs is the best, and we show why we can obtain such a results.

2.5.1 Diamond

We identify the DAG $D = (V, A)$ as diamond if $|V| = N = n^2$, each node can be labeled with a pair of numbers $(i, j), \forall i, j \in [1, n]$ and the arcs are: $((i, j), (i+1, j)), ((i, j), (i, j+1)) \in E$ with $\forall i, j \in [1, n-1]$.

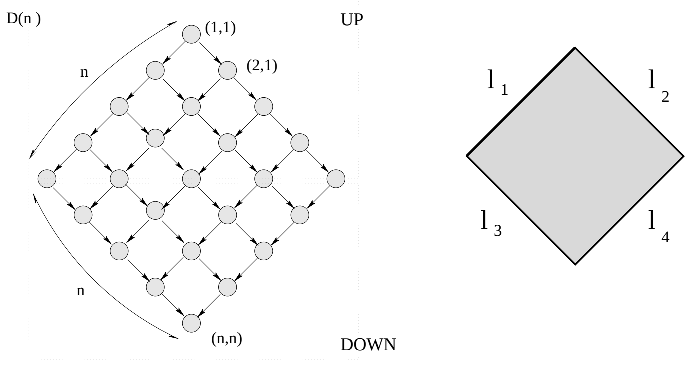

Figure 2.3: Diamond

In Figure 2.3 there is an example of a diamond $n \times n$. In the leftmost picture there are two horizontal dashed lines that split the DAG in two parts: the upper part and the lower part. In the rightmost picture the borders are indicated.

Suppose to apply the 1-closed marking method. In every 1-closed bipartition $(W_1 = B_1 \cup I_1, W_1^c)$ the frontier B_1 coincides with the minimum set of W_1 ([39]). We can note the input node is the minimum dominator of any subDAG of D. There is a DAG evaluation ξ_D such that the frontier of every 1-closed bipartition on ξ_D has at most two nodes. Suppose we have evaluated the sub-diamond $k \times k$ rooted on the input node and we have all node values in memory. The size of its frontier is one. We compute the node $(k+1, 1)$. The frontier has size two. We compute the nodes $(k, 2)$, $(k, 3)$ and we go on until node

$(k, k+1)$. The size of frontier of every set so evaluated is at most two. We evaluate the nodes $(1, k+1)$, $(2, k+1)$ till node $(k+1, k+1)$ is evaluated. The frontier size of every set is at most two. We have computed a sub-diamond $k+1 \times k+1$ from the sub-diamond $k \times k$ with frontier size at most two. Since the sub-diamond 1×1 has frontier of size one we have done. The diamond is *insensitive* to the 1-closed marking method.

Theorem 2.5 *For diamond D of size n^2, $\beta_{2,\frac{n^2}{2}}(D) = \Omega(n)$.*

Proof: let $(W_2 = B_2 \cup I_2, W_2^c)$ be a 2-closed bipartition of D such that $|W_2| = \frac{n^2}{2}$. The frontier B_2 can be seen as the composition of two sets $B_{2,i}$ and $B_{2.f}$. The former is the set of node without any predecessor in I_2. The latter is the set of nodes of the frontier such that they have at least a predecessor in I_2. We can see the set $B_{2,f}$ as the border set which determines the set I_2. $|B_2| = |B_{2,i}| + |B_{2,f}|$ and we consider $|B_2| = k$ and $|B_{2,f}| = j$. In turn, the set $B_{2,f}$ can be viewed as the set of nodes $\{b_i\}$ with $0 < i \leq j$. The set W_2 can be seen as the union of 2-closed disjoint sets $W_{2_1}, \dots, W_{2_m}, B_{2,i}$ such that they have disjoint frontiers $B_{2_1}, \dots, B_{2_m}, B_{2,i}$. By construction, $I_2 \subset \cup_i W_{2_i}$. If we connect the nodes in $B_{2,f}$ with a path, the maximum size of the path is at most of $3j$ nodes length. In other words, let u and v be the farthest nodes in $B_{2,f}$ and let P any undirected path which connects them, then the nodes that belong to the path P and to W_2 are at most $3j$. Indeed, take a node u which is not on any lower borders of the diamond. There exists a predecessor v of u which belongs to I_2. The node v has both successors in $B_{2,f}$, u and w, the distance from u to w is at most of three nodes (at most two nodes into $B_{2,f}$ have one predecessor in I_2 and they are located in the border l_3 and l_4 Figure 2.3). There exists a node b_i which has greatest distance among the others from the input node. Let cn with $2n \geq c \geq 0$ be its distance from the input node. We consider cn and $3j$ as the two sides of a no square diamond. This no square diamond has size $3cnj$ and we can see that

$$3cnk \geq 3cnj \geq |\cup_i W_{2_i}| \geq |I_2| \geq \frac{n^2}{2} - k$$

and therefore $k \geq \frac{n^2}{2(1+3cn)} = \Omega(n)$. □

It easy to prove that there is a computation of the DAG that needs only n memory locations and therefore the diamond has $\mathcal{M}_D = \Theta(n)$. Our lower bound is tight. The naive access complexity it would be $Q_D(S) = \Omega(n - S)$. But we can improve it by the partitionability property of the diamond. We can divide the diamond D in subDAGs and

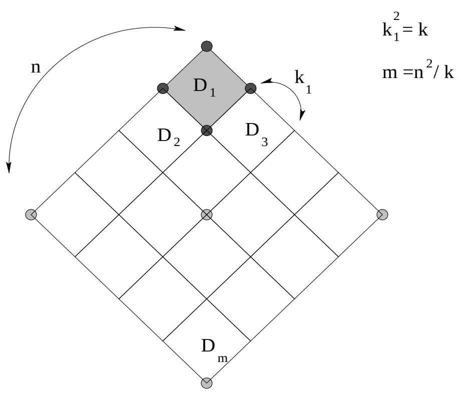

Figure 2.4: Recursive Decomposition

recursively to apply the k-closed marking method on each subDAG. Then, we maximize the contribution of each subDAG. For example, if we chose to decompose the DAG D of size n^2 into diamonds of size k, say D_j, and then to apply the Δ-closed marking method on them we can improve the naive lower bound. Indeed, $\beta_\Delta(D_j) \geq \frac{\sqrt{k}}{c}$ where c is a constant and $\beta_\Delta(D)$ is a contracted form of $\min_{\forall i} \beta_{\Delta,i}(D)$. Then $Q_D(S) \geq (\frac{\sqrt{k}}{c} - S)\frac{n^2}{k}$ and we obtain the maximum value for $\sqrt{k} = 2cS$ and therefore $Q_D(S) \geq \frac{n^2}{4c^2 S}$ for every $S < n$. Again, it is easy to see this is is an upper bound.

2.5.2 Binary Tree without Recomputation (BTWR)

We consider a binary tree and any DAG evaluation is a DAG evaluation without recomputation.

Lemma 2.5 *Let $BT = (V, A)$ be a binary tree of size $n = 2^l - 1$. The root is the input node and the leaves are the output nodes. Every output node has the input node as ancestor. If every node is evaluated once then $\mathcal{M}_{BT} \geq \beta_2 \geq l - 1$.*

Proof: Let t_0 be the time instant in any DAG evaluation without recomputation ξ_{BT} when the first leaf is evaluated. We mark the DAG evaluation ξ_{BT} and we obtain a 2-closed bipartition (W_2, W_2^c) of BT. We know that all the nodes belonging to the only path from the root to the leaf are evaluated and there are at least $l-1$ subtrees with any leaves not evaluated yet. Each tree which has the no evaluated leaves must have a root. Such a root is stored in memory and it permits the evaluation of its leaves. Such a root is stored in B_2. The set of these roots with minimum size is the path itself, without the input node. $\square$

The Lemma above says that $\mathcal{M}_{BT} = \Omega(\log n)$ for any DAG evaluation without recomputation. The naive access complexity is $Q_{BT}(S) \geq \log n - S$. But we can improve the result when we apply the partitionability of the DAG. Decompose the DAG BT in sub-BTs of size 2^{S+2}. Every sub-BT has access complexity of one. Therefore $Q(S) \geq \frac{|V|}{2^{S+2}} = \Omega(\frac{n}{2^S})$. The Lemma 2.5 can be generalized on every j-ary balanced tree.

Note that we have considered only the DAG evaluation characteristics, that is, every node is evaluated once. On this kind of DAG evaluation, we know that every k-closed marking method obtains the same bipartition ($W_k = W_j$ for all $j \neq k$) but we can note that the frontiers are different ($B_i \subseteq B_j$ for all $i < j$). We have not considered any possible 2-closed bipartition of BT to achieve the lower bound cited. In fact, if a node can be evaluated more than once the binary tree is insensitive to any k-closed marking method. It is easy to see because BT can be evaluated with constant space. This simple DAG offer a quantitative idea of different performances achieved when a DAG is evaluated without and with node re-evaluations.

2.5.3 Fast Fourier Transform

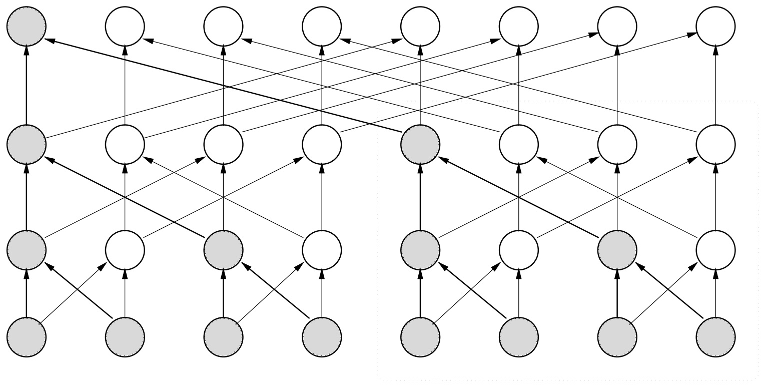

Figure 2.5: Eight Points FFT

An example of FFT DAG is drawn in Figure 2.5. The authors in [39] give $Q_{FFT}(S) = \Omega(\frac{n \log n}{\log S})$ when DAG FFT has size $n \log n$. The inputs are in memory at the beginning of the computation then the memory space is at least $\Omega(n)$ but we know that it is also an upper bound to the space memory, because the FFT can be evaluated *in place* (space memory is $\Theta(n)$). Obviously, we can perform an algorithm which performs $O(n\log(n-S)) = O(\frac{n\log n}{\log S})$ access over the S memory location. Therefore, $Q(S)$ is $\Theta(\frac{n\log n}{\log S})$. Our method cannot do better. We prove that we give a worse lower bound and we explain why.

Before to start any arguments we must introduce some definitions. A *reverse* DAG of a DAG $G = (V, A)$ is a DAG $G_r = (V, A_r)$ where for every (u, v) in A then (v, u) is into A_r and vice versa. We define as *FFT tree* the DAG $T(u) = (V, A)$ such that T is a binary tree embedded in a FFT DAG. The tree has one root u which is an output of the FFT. The leaves are the inputs of the FFT. In Figure 2.5, a FFT tree is represented by the set of blackened nodes.

Lemma 2.6 *The space memory of a DAG FFT of size $n\log n$ is $O(\log n)$ in any DAG evaluation.*

Proof: It is easy to see that to evaluate an output node u it is to evaluate the node of its FFT tree $T(u)$ from its leaves to its root. The recurrence formula to the space of a binary tree can be written as follows. If $k = 2$ then $\mathcal{R}(k) = 3$ and if $2 < k \leq n$ then $\mathcal{R}(k) = 1 + \mathcal{R}(\frac{k}{2})$. The solution is $\mathcal{R}(n) = 2 + \log n$. The algorithm which uses exactly the space above it can be recursively described as follows. It evaluates and stores in memory the left child of the root u. It evaluates the right child. It accesses the left child and evaluate the root. We can repeat the algorithm for every output. Since the inputs are queried only on demand they may not waste space. $\square$

Lemma 2.7 *The space memory of a DAG FFT of size $n\log n$ is $\Omega(\log n)$ in any DAG evaluation.*

Proof: Let $T(u) = (V, A)$ be the FFT tree with u an output node of the FFT DAG, and let (W_Δ, W_Δ^c) be any Δ-closed bipartition of $T(u)$. Then take the reverse DAG of $T(u)$, $T_r(u)$. We prove that the Δ-closed bipartition $(W_\Delta(T), W_\Delta^c(T))$ in $T(u)$ determines a 1-closed bipartition $(W_1^c(T_r), W_1(T_r))$ in $T_r(u)$ such that $W_\Delta(T) = W_1^c(T_r)$ and $W_\Delta^c(T) = W_1(T_r)$. The set $W_\Delta^c(T)$ has the property that if it is not empty, the node $u \in W_\Delta^c(T)$ has a direct successor in $W_\Delta^c(T)$ too (we can see the set of nodes $W_\Delta(T)$ is composed by completed and balanced binary trees). Therefore if a node u is into $W_1(T_r)$ then every its predecessor is into $W_1(T_r)$. It is a 1-closed set. We can apply the result of Lemma 2.5 on $T_r(u)$. $|B_2(T)| \geq |B_2(T_r)| \geq \beta_2(T_r) = \Omega(\log n)$. □

Clearly, an obvious lower bound to the access complexity is $\log n - S$ or using the partitionability property we can decompose the FFT into subFFT of size $m \log m$. Therefore $Q(S) \geq (\log m - S)\frac{n \log n}{m \log m}$ and for example with $2S = \log m$ we obtain $Q(S) \geq (n \log n)/2^{\frac{S}{2}}$.

To understand why the two approaches differ we can take the following case. We take a machine which has $1+\log n$ locations available as space memory. Take $S = \log n$. Apply the algorithm suggested in Lemma 2.6. The access complexity of the DAG evaluation associated with the execution of the algorithm is null, because it never uses the location of address $\log n$. If we substitute the value of S in the lower bound proposed by Hong and Kung we obtain an access complexity of n. This difference is due to the fact that in their model the inputs must be read at least once and they are to be present in memory at beginning of the computation. So the results are different. The *FFT* DAG is a good example in which the data might input one at time but many times. A simple example of application is a periodic signal with very high frequency for which we can pick up a sample in every period but we can sample accurately the whole waive form in different cycles. The source give n samples every n cycles and we have to choose whether store the input values or not. In this case it may be useful take the algorithm which uses a low number of locations in particular such an algorithm which uses exactly $\log n$ locations. Note that the situation changes sharply when $S < \log n$.

2.5.4 The Fan

We describe an example of DAG such that the DAG is almost insensitive to the 2-closed marking method but it is not insensitive to 1-closed marking method. We write $LA(j)$ and we refer to an oriented linear array of j nodes. The input is one end and the output is the other end of the array. The k-th node in the linear array is the node that is k nodes far from the input. We refer as $LA_k(j)$ a specific linear array from a set. The index k is discriminant index. We write $C_P(r)$ and we refer to the nodes that belong to a circle of radius r on the plane P. Since we are build a three dimensional DAG the meaning of plane is the same of plane in a three dimensional space. We introduce the DAG *Fan*, it is a parametric DAG, i.e. Figure 2.6.

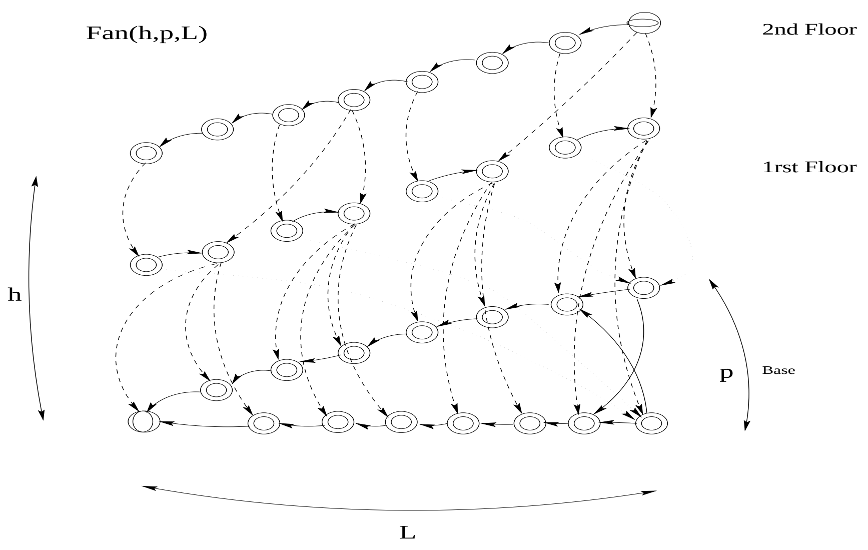

Figure 2.6: Fan

Definition 2.4 *the constructive definition of a fan with parameters p, h, L follows:*

1. *We take the 3-D space where axe x is width, axe y is depth and axe z is height. The axes system is oriented by the rule of the right hand, the thumb index is forward the axe x and the medium finger is forward the axe y.*

2. *We choose the following relations:*
   - $L = 2^l$.
   - $p = 2^m$ *e* $1 \leq p \leq L/2$.
   - $\lfloor L/2p \rfloor = L/2p$.

3. *The* Base *of the fan is plane $(x, y, 0)$ where we put a composition of linear arrays of size L. We put a node into $(0, 0, 0)$ and we lay down p $LA(L-1)$s on the Base. An arc connects each output of every $LA(L-1)$ to the node into the origin of the space. Every i-th node in the linear array, with $1 \leq i \leq L-1$, belongs to the $C_{Base}(L-i)$ (i.e. the inputs of the $LA(L-1)$'s belong to $C_{Base}(L-1)$ and the outputs to $C_{Base}(1)$). Also there is an arc from every node in $C_{Base}(L-1)$ to every node in $C_{Base}(L-2)$.*

4. *The* first floor *is a combination of $\frac{L}{2}$ $LA(2)$s which we put on the line which lays on $(x, 0, 1)$. Indeed, the input node of the $LA_i(2)$ with $0 \leq i \leq \frac{L}{2} - 1$ is into the point*

that has coordinates $(2i, 0, 1)$ *and its output node has coordinates* $(2i + 1, 0, 1)$.

5. *We connect by arcs the fist floor with the Base as follows: there is an arc from any node in* $(2i+1, 0, 1)$ *to every node in* $C_{Base}(2i) \cup C_{Base}(2i+1)$ *for every* $0 \le i \le \frac{L}{2} - 2$. *There is an arc from any node in* $(2i+1, 0, 1)$ *to an input of the* $LA(L-1)$*'s embedded in the Base as follows. Consider the* $LA_k(L-1)$ *with* $1 \le k \le p$, *there is an arc from every node in* $(2i, 0, 1)$ *for all* $0 \le i \le \frac{L}{2} - 2$ *to the input of* $LA_k(L-1)$ *such that* $1 + \lfloor \frac{i}{p} \rfloor = k$.

6. *The* second floor *is a* $LA(L)$ *such as the i-th node has coordinates* $(L - i, 0, 2)$.

7. *We connect by arcs the second and the first floor as follows: there is an arc from a node in* $(2i, 0, 2)$ *with* $0 \le i \le \frac{L}{2} - 2$ *to a node in* $(2i, 0, 2)$. *There is an arc from any node in* $(i, 0, 2)$ *such as* i *mod* $4 = 3$ *to the nodes in* $(i, 0, 1)$ *and* $(i-2, 0, 1)$.

Lemma 2.8 *In any DAG evaluation in* $\Xi_{F(h,p,L)}$ $\min_i \beta_{\Delta,i} \le h + 2p - 1$ *for all* $1 \le i \le |V|$.

Proof: it is a constructive proof, for every $i \in [1, |V|]$ we obtain a Δ-closed set such that its frontier is no greater than $h+2p-1$. $(0, 0, 0)$ belongs to W^c_Δ. We define $U_0 = \{(0, 0, 0)\}$. The complementary set of a Δ-closed set is the set of node such as whether a node x is in here it exists a path from it to the output so that every node of the path is into the set too. Let U_1 be the set $U_0 \cup \{(1, 0, 1)\}$, this set of node satisfies the necessary conditions and therefore is a complementary set of a Δ-closed set. $|B_{\Delta,n-2}|$ is $p+2$. We keep on to building in this fashion: $U_2 = U_1 \cup \{(0, 0, 1)\}$ and the correspondent bipartition has $|B_{\Delta,n-3}| = p+2$. $U_3 = U_2 \cup \{(0, 0, 2), (1, 0, 2), (2, 0, 2), (3, 0, 2)\}$ and $|B_{\Delta,n-7}| = p+1$; $U_4 = U_3 \cup C_{Base}(1)$ and $|B_{\Delta,.}| = p+1$; $U_5 = U_4 \cup C_{Base}(2)$ and $|B_{\Delta,.}| = p+2$; $U_6 = U_5 \cup \{(3, 0, 1)\}$ and $|B_\Delta| = p+2$; $U_7 = U_6 \cup \{(2, 0, 1)\}$ and $|B_{\Delta,.}| = p + 1$; $U_8 = U_7 \cup C_{Base}(3)$ and $|B_{\Delta,.}| = p + 1$. The method follows similar steps and when we reach the node of $C_{Base}(L-2)$ there is no problem because the cut edge is not the frontier. □

The number of incoming arcs into a node v in $F(h, p, L)$ is:

- $\gamma_{(0,0,0)} = p + 1$.
- Every node v in $C_{Base}(L-2)$ has $\gamma_v = p + 1$.
- Every node v in $C_{Base}(L-1)$ has $\gamma_v = \lfloor L/2p \rfloor + 1$.

- $\Gamma = \max(p + 1, \lfloor L/2p \rfloor + 1, L/2 - (p - 1)\lfloor L/2p \rfloor + 1)$.

Lemma 2.9 $\beta_1(F(p, h, L)) \geq p - 1 + \lfloor L/2p \rfloor$

Proof: let t_0 be the time instant in the computation on $F(h, p, L)$ such as only a node v in $C_{Base}(L - 1)$ is marked after t_0 and exactly at t_0, we recall that if we use the 1-closed marking method every successor of v is marked after t_0 therefore the output node $(0, 0, 0)$ is marked after t_0 too. By construction, the direct predecessors of the node v in the first floor, or their direct successors that are in the first floor too, they have every direct successor marked after time instant t_0. Immediately, we can find that $|B_1| \geq p - 1 + \lfloor \frac{L}{2p} \rfloor$. Indeed, $p - 1$ is due to the nodes in $C_{Base}(L - 1)$ and $\lfloor \frac{L}{2p} \rfloor$ is due to the direct predecessors of node v in the first floor. □

We give just a little summary of the results choosing $p = (\frac{L}{2})^{\frac{1}{4}}$

1. By Lemma 2.9, $\beta_1(F(h, p, L)) \geq (\frac{L}{2})^{\frac{3}{4}} + (\frac{L}{2})^{\frac{1}{4}} - 1$.
2. By Lemma 2.8, $\beta_\Delta \leq 2(\frac{L}{2})^{\frac{1}{4}} - 1 + h$.
3. $\Gamma = (\frac{L}{2})^{\frac{3}{4}} + 1$

We can see that for $p = (\frac{L}{2})^{\frac{1}{4}}$ the fan DAG $F(h, p, L)$ is insensitive to the Δ-closed marking method because the marking gives as result a lower bound which is lesser than the maximum in-degree Γ of the DAG. Note than the most obvious lower bound to the space complexity for any DAG evaluation is the maximum in-degree Γ of the DAG. This no simple DAG is useful because it is an example in which the Δ-closed marking method fails respect other k-closed marking.

2.5.5 k-Bridge

We have drawn in Figure 2.5.5 a k-Bridge DAG. Its name derive from its seeming with a bridge with k support columns. This DAG is insensitive to 1-closed and Δ-closed marking method but it is not insensitive to 2-closed marking method. So this DAG can be viewed as exception from the DAGs we have proposed. It states that any k-closed marking method with $1 \leq k \leq \Delta$ must be tested.

Lemma 2.10 *Let $BR = (V, A)$ be a k-Bridge DAG, then $\beta_1(BR) \leq 2$.*

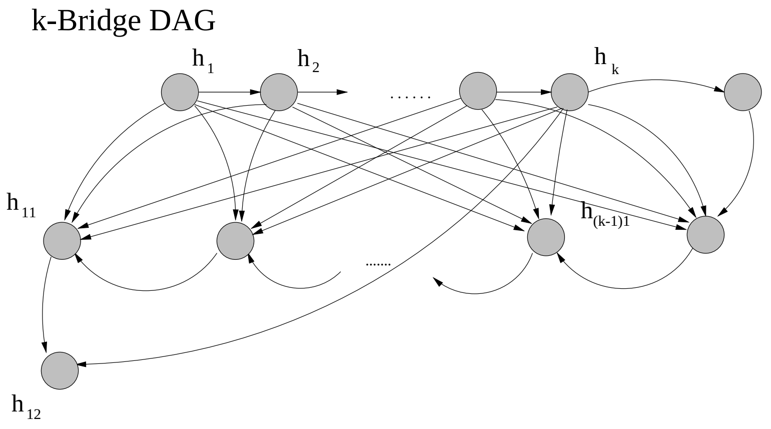

Figure 2.7: k-Bridge Directed Acyclic Graph

Proof: we show a DAG evaluation ξ_{BR} such that for every time instant t onto ξ_{BR}, the frontier of the 1-closed bipartition determined at t is $|B_1| \leq 2$. We evaluate the nodes $h_1, h_2, \ldots, h_k$ in sequence and therefore $|B_1| = 1$. Then, we evaluate the nodes $h_{k,1}, h_{(k-1),1}, \ldots, h_{1,1}$. We have done. □

Lemma 2.11 *Let $BR = (V, A)$ be a k-Bridge DAG then $\beta_\Delta(BR) \leq 2$.*

Proof: we determine for every $1 \leq j \leq |V|$ a Δ-closed bipartition such that $|B_\Delta| \leq 2$. Take the k-th node $h_{k,1}$ and in particular let $W^c_\Delta = h_{12}$ the set such that at any instant we add the following nodes: $h_k, h_{k-1,1}, h_{k-2,1}, \ldots, h_{1,1}$, after we can add the nodes $h_1, \ldots h_k$. From the arguments above it is easy to see that the frontier B_Δ is limited above by the constant value 2. □

Lemma 2.12 *Let $BR = (V, A)$ be a k-Bridge DAG then $\beta_2(BR) \geq k$.*

Proof: let t be the time instant in any DAG evaluation ξ_{BR} such that at t it is marked the last node in h_i with $1 \leq i \leq k$. Let h_m be the node which is marked last respect other node h_i (with $1 \leq j \leq k$ and $j \neq m$), The node h_m is marked at time instant t. Every node h_j is into W_2 after t. By construction at least $\max(1, d_v - 1) \geq k$ direct successor of h_m are into W^c_2. they are connected with the output node by a directed path. Every node h_j has at least k direct successor in W^c_2. They are into the frontier B_2. □

For this DAG the minimum space is naively the maximum in-degree Γ_{BR} and it is also the maximum space. It is insensitive to 1-closed and Δ-closed marking method, but the 2-closed marking method obtain a tight lower bound.

Chapter 3

Code Reorganization, Idea and Practice

Suppose we have a DAG and we want to exploit data locality and write the code which implements the DAG. In this Chapter we study the application of *Convex Decomposition Tree* (CDT) as intermediate representation to write the code, and therefore, the algorithm expressed by a DAG so that Data Locality is exploited. The CDT is a tree which recursively decompose the DAG in subDAGs rooted at internal nodes of the tree and the order from left to right of the children infers a schedule of operations of the DAG. The construction of a CDT from a DAG is equivalent to the scheduling problem ($\mathcal{NP}$-hard), therefore we know that an optimal solution can be obtained only with an exhaustive search (unless $\mathcal{P} = \mathcal{NP}$). In the following we propose an heuristic to build a CDT from a DAG and an algorithm to manage memory accesses when a CDT is available.

3.1 Scheduling of an evaluation DAG without recomputation

The problem of scheduling operations of a evaluation DAG is complex, to simplify the problem we restrict the computation model. We consider only DAG evaluation without re-computation. In other words, every node is evaluated once. When we mark a node $v \in V$ in any DAG evaluation we know several information. We know that at instant time v is marked, its predecessors are evaluated and its successors are not evaluated yet.

If we say that W_1 is the set of all ancestors of node v, then they are evaluated and marked when we mark node v. If we say W_2 is the set of all descendants of node v, then they are not evaluated yet. We say that W_3 is the set of all nodes that are neither in W_1

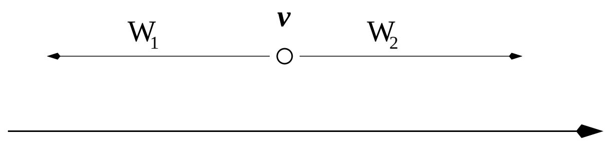

Figure 3.1: Splitting of the computation and of the DAG.

nor W_2 ($W_3 = V \setminus (W_1 \cup W_2)$, note that W_1 and W_2 are disjoint sets). We can say nothing about these nodes with respect to v, some of them can be evaluated and other cannot be evaluated yet.

Lemma 3.1 *W_1, W_2 and W_3 are convex set.*

Proof: we prove W_1 is convex. Let node w and v be in W_1 so that there exists a node $u \notin W_1$ which is descendant of w and ancestor of v. But if u is a descendant of node v by construction u would belong to W_1. We can prove in similar way that W_2 is convex. We prove W_3 is convex. Suppose there exists a node $u \in W_3$ which has a descendant in W_1 (resp. in W_2). We call this node v. Suppose v has a descendant in W_3, but u would belong to W_1 (resp. the node successor of v would belong to W_2), this is a contradiction. $\square$

Our idea is based on a simple consideration. We know that every node in W_1 is evaluated before every node in W_2 for every DAG evaluation. The bigger the sizes of the sets W_1 and W_2 are the smaller the size of the unknown W_3 should be. In fact, we know that W_3 can be evaluated after W_1 and before W_2, but it can be evaluated in several ways. Let us introduce some definitions and notation.

Definition 3.1 *A node v is a h-*balanced node *if the number of its ancestors differs from the number of its descendants by h (absolute value).*

Definition 3.2 *A node v is a* central node *if v is a h-balanced node and h is minimum.*

Definition 3.3 *A node v is a* maximum node *if v is a central node and the number of its ancestors and descendants is maximum.*

Definition 3.4 *A node v is a* left node *if v is a maximum node and the number of its ancestors is greater that or equal to the number of its descendants.*

For every left node $v \in V$ we can split the DAG. We know that the set $W1_v$, $W2_v$ and $W3_v$ are convex sets. The size of $W1_v$ and $W2_v$ is maximum.

Lemma 3.2 *Let L be the set of left node in computation DAG $G = (V, A)$. The set $W = \cup_{v \in L} W1_v$ is a convex set.*

Proof: by contradiction, suppose there is a node $w \in W$ and there exist two nodes $x \notin W$ and $y \in W$ so that x is descendant of w and y is a descendant of x. If $x \notin W$ then $\nexists v \in L$ so that $x \in W1_v$ but $\exists u \in L$ so that $y \in W1_u$ and therefore x should be in $W1_u$. $\square$

Following the similar arguments, we can prove that $W = \cup_{v \in L} W2_v$ is a convex set and $W = V \setminus (\cup_{v \in L}(W1_v \cup W2_v))$ is a convex set.

3.2 Idea

A *Convex Decomposition Tree* (CDT) is a tree based on a DAG G. From the root of a CDT it is possible to reach all the nodes in G. The leaves of a CDT are the nodes of G. SubDAGs of G are associated with internal nodes in CDT(G). A preorder visit of a CDT determines a total order for the nodes in G. Take for example an internal node v in a CDT(G) with three children v_1, v_2 and v_3 in order from left to right. Rooted at v_1, v_2 and v_3 there are CDTs that infer a total order on their leaves, and therefore on subsets of node in G. The order on the internal nodes v_1, v_2 and v_3 infers a order among the subset of nodes in G. We propose an heuristic algorithm to produce such structure, therefore a scheduler algorithm.

Algorithm *BCDT(r, V)*

1. Determine the set $L \subseteq V$ of left nodes in G.

2. Let W_1 be the set of ancestor nodes of any node in L. Let W_2 be the set of descendant nodes of any node in L. Let W_3 be the set of node $V \setminus (W_1 \cup W_2)$. Let t_l be the left child of the root t. t_l is the root of the CDT associated with the DAGs identified by W_1. The node t_m is the middle child of the root t. t_m is the root of the CDT associated with the DAGs identified by W_3. The node t_r is the right child of the root r and it is the root of the CDT associated with the DAGs identified by $W_2 \cup L$.

3. Recognize and collect the isolated DAG obtained by the decomposition (previous steps). Associate with every DAG a node at the same level in the BCDT.

4. Apply *BCDT(left,W_1)*,*BCDT(middle,W_3)* and *BCDT(right,$W_2 \cup L$)*.

♠

This algorithm infer a topological order to the nodes in W_3, which can be non optimum. Indeed, nodes in W_3 may not be evaluated between evaluations of nodes in W_1 and nodes in W_2. And in general it is an arbitrary choice to say that nodes in L splits the computation perfectly. These choices are always correct when W_3 is an empty set and L is composed by only one node. The heuristic try to reduce the size of W_3 and L, so as an erroneous topological order has minimum effect on the overall DAG evaluation.

3.2.1 Algorithms

The algorithm BCDT must be specified further so that we can achieve a better understanding of the implementative details and also achieve a complexity estimation, how many steps are required in function of the size of the problem. We start describing an algorithm to determine the left node in a DAG G.

Algorithm *Left Node Determination*, LND

1. For every node $v \in V(G)$ compute the number of its ancestors and its descendants with algorithm $PSA(G)$, store these results in the attributes v_p and v_s associated with node v.

2. Let k be the $\min_{v \in V} |v_p - v_s|$ and let l be $\max_{v \in V} v_p + v_s$. Sort the node $v \in V$ in a such way that the first nodes have $|v_p - v_s| = k$ and $v_p + v_s = l$. Let H be the set of the first nodes after the sorting.

3. From the set H obtained in the previous step we pick up the node $v \in H$ such that $v_p \geq v_s$ and we put it into L.

♠

Algorithm *Predecessor Score Algorithm, PSA*

```
PSA(G) {
    G1  = G;
    while (I(G1) is not null) {
       for any s in I(G1);
       C = Pulse(G,s);
       for each v in C {
           v.p++;
       }
       s.s = |C|;
       remove s from G1;
    }
}
```

where $I(G)$ is the set of input nodes of G. ♠

The following algorithm send an impulse to its descendants. When a node receives an impulse it is considered touched, further impulses received from the same source must be ignored.

Algorithm *Pulse, Touched = Pulse(G,s)*

```
Pulse(G,s) {
  return BFS(G,s)
}
```

♠

The algorithm $BFS(G,s)$ is a breadth-first search from s into G and returns the node touched by the search. It is easy to see that the algorithm PSA computes correctly the number of successors for every node. In every iteration, one input at time subscribes its contribution to its successors without repetition. The number of predecessors for a node v is the number of impulses received from its predecessors. We estimate the time complexity of $PSA(G)$. Let $G = (V, A)$ be a DAG with $|V| = n$ and $|A| = m \geq n - 1$. $T_{PSA(n,m)} = \sum_{i=0}^{n-1} T_{BFS(n-i)} \leq \sum_{i=0}^{n-1} K(n + m - 2i) = Kmn^2 - Kn(n-1) = O(mn^2)$. $T_{LND(G)}$ is dominated by the time spent to execute PSA, because any sorting can be performed in $O(n^2)$.

An iteration of T_{BCDT} is the execution of two steps. The first step is the determination of the left node set and the splitting of DAG G in its three parts. We know its time complexity. The second step is to recognize and collect subDAGs, obtained as side effect of the splitting. In this case we can use DFS (Depth-First search), and the family of trees identifies isolated DAGs. The time complexity is $O(n+m)$. The total time has upper bound $O(mn^2)$ (We think that this upper bound can be improved to $O(nm \log n)$).

If the CDT is balanced and $|W_3| \sim 0$ we can write a recurrence equation as follows. $T(nm) = 2T(\frac{nm}{2}) + O(mn^2)$ this has simple solution $T(nm) \leq O(mn^2 \log(mn))$. More sofisticated analisys must be devised for the general case.

3.3 Example

We propose two examples of DAG and we apply our heuristic approach.

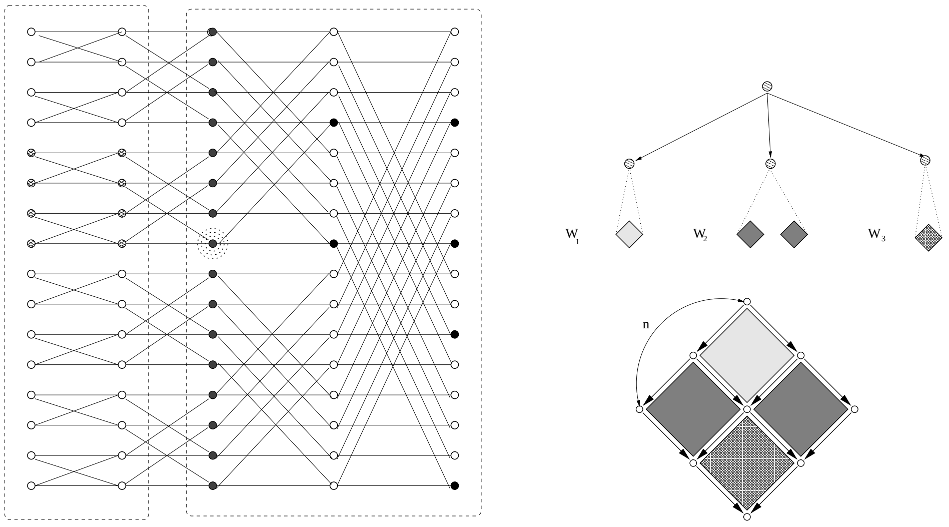

Figure 3.2: Diamond decomposition and 16-node FFT decomposition, first iteration.

For the diamond in Figure 3.2 the set L is composed by one node, the node v at coordinate $(\frac{n}{2}, \frac{n}{2})$. In fact, any central node is 0-balanced node. The number of its ancestors and descendants is $2(n-k)k$ for any $k \in [1, n-1]$. The maximum value is for $k = \frac{n}{2}$. The induced partition and computation is optimal even if set W_3 is not empty. For the FFT in Figure 3.2, set L is composed by sixteen nodes and L is a cut set for DAG G. Note that the partition suggests the well known in-place evaluation of FFT DAG.

3.4 CDT: Intermediate Code Representation

A CDT T is a tree with internal nodes and leaves. The leaves are nodes of a computation DAG $G = (V, A)$. The internal nodes are build up from a topological sort of the nodes of the DAG G. Indeed, the tree represents a topological sort of the DAG and any visit of the tree induces a strong order among any two nodes in the DAG. In this Section we study the problems related to the implementation of the *mapping and scheduling algorithm MSA*, [12] and in particular how to write the code such as we can attain a DAG evaluation of G. Shortly we recall the MSA algorithm but now we introduce some terminology and notation which will be used. We say that an internal node $u \in T$ is the *common father* for the set of nodes $U \subset V$. Lowercase letters of the alphabet always identify a single node in the CDT T and the correspondent uppercase determine the set of nodes for which the node is common father. We indicate with $S(U)$ the memory space necessary for the evaluation of the subDAG determined by a set U. Therefore with $S(V)$ we identify the space necessary to a DAG evaluation of G. Furthermore, we recall that $\Gamma(U) = \{v \in V - U | v \text{ has at least on successor in} U\}$ and it is called pre-boundary of set U. The statement of the *MSA* follows.

Algorithm It is defined recursively. If $u \in T$ is a leaf, we can execute the single operation in u. If u is an internal node it has children $u_1, u_2, \ldots, u_k$ with $k \geq 2$ and ordered from left to right. For every u_i starting from u_1 and ending to u_k. Copy the values of node $\Gamma(U_i)$ into the memory locations from $S(U_i) - |\Gamma(U_i)|$ to $S(U_i) - 1$. Execute U_i. Copy the values that are input to successive subsets into *suitable* memory locations in the range from $S(U) - \sum_1^k |\Gamma(U_i)|$ to $S(U) - 1$. ♠

The last step of the algorithm it is not completely clear, unless we specify what the authors mean for *suitable*. The left to right data dependency, which this algorithm exploits, is between sets. Instead to consider the root we consider an internal node w with children w_i with $0 \leq i \leq j - 1$. Suppose we have evaluated the node w_i (therefore the set W_i). It may be that the input nodes for the successive sets and node in W_i are not input for any child of w, but for right siblings of w. How to solve this case is not clear, because in general it cannot be claimed that data locality among sub problems can be exploited so that data are required among nodes with common father and at the same level.

A tree-like structure is a very familiar structure to generate code. A simple example is the code generation for expression in three address code. Given an internal node u, suppose the data computed by a child, v, cannot be managed by u. There must be an internal node w which is common father of u and of all the siblings of w that need the node values computed in W, i.e. x Figure 3.3. w is the right node to manage this information. This problem is known as the *least common ancestor problem.*

1. There is a node evaluation phase.
2. There is an ascending phase from the leaves to the internal nodes.
3. There is a descending phase where the data are transmitted from the common father to its children.

We can see that the ascending phase is completely determined by the least common ancestor of two nodes, the node where the datum is produced and the node where the datum is consumed. There are two possible cases describing the ascending phase. We can copy

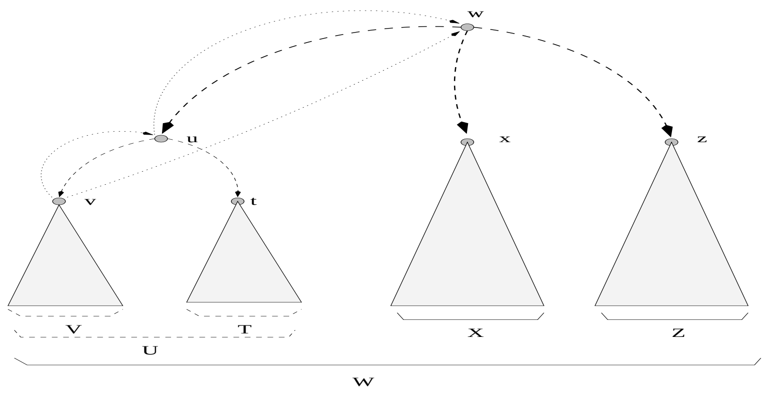

Figure 3.3: Example of copy propagation: a value computed in v is used in x, we can see how to determine the least common ancestor respectively in a direct fashion or one level at time.

directly the node values to the nearest common father of the node evaluated and of the node who uses that value. We can store node values into the local common father and when all the children are evaluated we perform a mapping to an upper level. We can see that the second solution does not follow strictly the idea of MSA algorithm. But we have to produce code, which evaluates G, we do not evaluate the tree. So we can use the second technique to determine the right mapping between nodes and their evaluations. We have to decide how to perform the copy propagation between leaves and internal nodes. If we want to write the code as soon as possible we have to determine the address location to store the result of every node evaluation just after its evaluation. An alternative solution is we can apply a back patching technique: if a leaf has to store its node value into a least common ancestor but it does not know exactly what is this node, it can delegate the problem and the solution to its father. When we reach the least common ancestor we can go back and complete the code generation of the left children. This case is represented in Figure 3.3 by the dashed path which connects the node v and node w by the node u. The final choice can be done when we define the final structure and the definitive algorithm.

3.4.1 Description

We introduce some notations such that it is easier to describe the following steps. Let T be a CDT of DAG $G = (V, E)$. The level of an internal node is its distance from the root of the CDT T (the common ancestor of every node). Two internal nodes v and w in T such that v is direct predecessor of w, if v is at level $l-1$ then w is at level l. If the tree is balanced, there are at most $O(\log|V|)$ levels (we will abuse the notation and we will write $O(\log V)$). The tree exploits a strong order relation among node in G. We can codify the node in G by an integer that we define as *key* of the node. We gave a definition to pre-boundary of a set of node W, $\Gamma(W)$ can be intuitively see as the set of nodes in G that must be evaluated and stored in memory before the evaluation of W (we remind that the DAG evaluation are restrict to be without node re-evaluation and the concept of pre-boundary is a particular dominator set or boundary for Δ-marking).

We introduce an ausiliary data structure for the CDT. It is another tree structure which is very similar to a *range tree* [9]. Each node w in a CDT is the root of a sub-tree and stores an associated trees. We identify the associated structure at every node w by $\mathcal{L}(w)$, and we call it the *canonical set* of node w. $\mathcal{L}(w)$ is a binary search tree. Consider the children of w, $w_1, \ldots, w_q$, then $\mathcal{L}(w)$ stores not only $\Gamma(W)$ but also $\Gamma(W_1), \ldots, \Gamma(W_q)$. We do not need to store $\sum_i |\Gamma(W_i)|$ keys, we need $|\cup_i \Gamma(W_i)|$ keys, without repetitions. $\mathcal{L}(w)$ stores $\cup_{i=1}^{q}\Gamma(W_i)$ and $\Gamma(W) \subseteq \cup_{i=1}^{q}\Gamma(W_i)$. $\mathcal{L}(w)$ associates each leaf with an integer number such that it expresses the number of leaves at its left. The root of $\mathcal{L}(w)$ stores the total number of leaves. We remind that W is the set of leaves rooted at w. The space required for the tree is $O((|E|+|V|)\log|V|)$ or shortly $O((E+V)\log V)$. Indeed, whether we identify the outdegree of a node v as δ_v then we must store node v in at most $1+\delta_v$

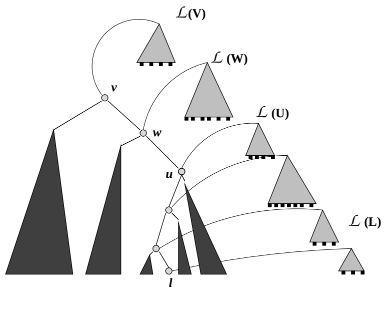

Figure 3.4: Description of the Tree and of its associated structure

canonical sets at same level, one in its leaf and the other in its direct successors in DAG G. This is done for each node v. Then every node may be present in $O(\log V)$ levels of the tree. Note that we proposed an heuristic to determine the CDT T from a DAG G so that the tree is balanced as much as it is possible.

To apply the algorithm proposed we must compute the *work space* for every sub-tree. This can be done by a bottom-up visit of the tree. Indeed, if we indicate with γ_v the indegree of node $v \in V$, the space required to evaluate a leaf in the tree is $S(v) = 1 + \gamma_v$. We can note that $\gamma_v = |\Gamma(\{v\})|$, that is, the indegree of node v is the pre-boundary cardinality of the set $\{v\}$. For each internal node w with children $u_1, \ldots, u_q$, $S(w) = \max_{i=[1,q]} S(u_i) + |\cup_{i=1}^{q} \Gamma(U_i)|$. We identify such space by $S(w) = S_w(w) + S_\Gamma(w)$, we say that $S_w(w)$ is the low part and $S_\Gamma(w)$ is the high part. In this phase we use the canonical sets to compute S_Γ. If the number of children, we proposed a technique such that $q \leq 4$, is a constant, this step can be done in $O((V + E)\log V)$ for all node in the same level. Indeed, we can read $\cup_{i=1}^{q}\Gamma(U_i)$ in $O(|\cup_{i=1}^{q} \Gamma(U_i)|)$ step and we can introduce them into $\mathcal{L}(w)$ in $O(|\cup_{i=1}^{q} \Gamma(U_i)| \log V)$ steps. A node v can be handled at most $1 + \delta_v$ times in the same level, because there are only $1 + \delta_v$ sub trees that store such nodes. At every level we handle at most V nodes and therefore it follows the upper bound. We compute not only $|S_\Gamma(w)|$ but also $\mathcal{L}(w)$ for every w at the same level of the tree. And therefore the overall work will be $O((V + E)\log^2 V)$. We can see that the time spent to make up the tree with its canonical sets is not optimal by a factor $O(\log V)$. Shortly, we describe how to use the data structure.

- Let u and w be two nodes. u is nearest predecessor of w and w is visited for the first time. By construction $\Gamma(W) \subseteq \Gamma(U)$, and if $\Gamma(W) \cap \Gamma(U) \neq \emptyset$ we must copy from $S_\Gamma(u)$ the nodes in $\Gamma(W)$ to $S_\Gamma(w)$. $S_\Gamma(w)$ is a array of locations. The canonical set

$\mathcal{L}(w)$ is used to determine the mapping value-node in $S_\Gamma(w)$. Indeed, every leaf in $\mathcal{L}(w)$ store an integer value that can be considered as offset in $S_\Gamma(w)$. This step can be accomplished in $O(|\Gamma(W)| \log V)$ steps.

- Let u be a leaf. We evaluate u. We must copy its value in all least common ancestor w in the tree T. Copying takes $O(\gamma_v \log^2 V)$. Let P be the path which connects the root of the tree with u. P identifies $O(\log V)$ sub-trees and canonical sets. We look for in each canonical set if the node u is stored. If it is in, we must perform a copy. Searching in a binary search tree takes $O(\log V)$ steps.

The code generator performs such steps in order to produce the final result and the total time is $O(V \log^2 V + E \log^2 V)$. Indeed, the work associated with the first step can be estimated saying that all nodes in the DAG G is copied ($O(V)$) from a level to another ($O(\log V)$ levels) and each copy costs $O(\log V)$. The work associated with the second step can be estimated considering that every node in DAG G is evaluated and all the arcs are tested. Then we can say that we determine in $O((E+V)\log^2 V)$ steps the *copying rules* and we are done.

3.5 Application and Copy optimization

Suppose that we want to generate code for an architecture with a three levels memory: register files, a very fast but small cache memory and a huge RAM memory. The access to a datum in the memory is determined by three constants. Each constant represents a different access time in each memory level. With this memory hierarchy we must pay attention when we copy values in different locations. Copying a node value from a memory location to another one in the same level has the only side effect to execute a useless copy. Copying must be done when source and destination belong to different levels. The CDT T has an interesting characteristic. When we have computed for each node w the parameters $S_w(w)$ and $S_\Gamma(w)$, we know the required space to compute all the nodes in W, and therefore if it is contained completely in the first level, in the second and so on. If W is contained in the first level we know that can be computed when there are available the node values of $\Gamma(W)$.

Furthermore, we know where node values are stored in memory and therefore we can compute the trade off between copying and recomputation. To evaluate a node we restore a node value from a location and this is done incrementally. We push down the values from an internal node to its children. We can compute how much is expensive to copy up and down in the memory hierarchy a node and we can compute how much expensive is to re-evaluate it.

Chapter 4

Micro-benchmarking RISC Architectures

We propose an *general approach* to measure computational performance of different platforms. Performance is the capacity to execute computations and memory accesses. We measure time to compute double precision floating point operations, integer operations and we measure time to manage data from and into memory, different level of caches and main memory. An interested reader on micro-benchmarking can find interesting the tests proposed in [40]. This Section can be seen as an investigation of the available parallelism in current general purpose workstations and also as the searching of a set of tests such that a simplified model of the architecture may be obtained automatically. Many automatically tuned software packages use this approach to produce high optimized code.

4.1 Not only FLOPS, Floating Point Unit Performance

Without loss of generality, we can say that current platforms exploit in two different ways parallelism: through pipeline architectures and multiple functional units (scalar and super-scalar platform). Number of functional units and number of stages in a functional unit may vary. In general there is a main distinction between integer functional units and floating functional units. Operations on different types can be executed in parallel, floating point operations and integer operations. Certain operation on the same type of values (i.e. integer) can be performed in parallel because there are multiple (integer) functional units. Data dependency among operations is key to measure the properties of an architecture. Indeed, two operations related each other, where there is a data dependency, they cannot be neither executed in parallel nor put into consecutive stages of a pipeline unit. This simple case is the base for every micro tests and we can go beyond, we can figure out the architecture by quantitative tests, at least a rough and simplified model. Hence its basic functionality, we pay particular attention to operation $c+ = a * b$, where a, b and c are floating point. This operation is ideally executed in two steps: first multiplication, $a * b$, and right after sum. The second cannot be started before the end of the first one. The dynamic scheduler of processor can take different policy to compute a stream of operations and our purpose is to understand and measure the execution time of the operation due to the different policies and architectures. Basically, floating point units are so structured: there are a floating point register set, a multiply unit, an add unit, a divide unit and optionally other features. In other word two multiplications cannot be executed in parallel but a multiplication and an addition can. All units are pipelined and the number of stages can be identified by an integer k. We are talking about the number of stages to execute an operation, or *latency*. So if we try to measure the execution time of a stream of operations of the same type, with a tight data dependency, we should be able to measure the time kT_c where T_c is the clock cycle of the machine. We measure the number of stages of the machine. If the stream of operation has enough independent operations, they can be pipelined and we should be able to measure the single stage of the pipeline, therefore the clock cycle. Unfortunately this measure is affected by error, we measure these properties using the machine itself. The relative measure error for our tests set can be estimated as 10%. Perhaps it is too much, but it is a result of a trade-off between accuracy and time execution of the test. Whether we know the clock cycle we can measure the number of stages of functional units and we can understand their architecture and their scheduling policy.

The tests can be described as follows. We determine a loop with two dependent operations:

```
for (i=1; i<=MAX; i++) {
        op(a,a,i);    // a = a op i;
        op(a,a,i);    // a = a op i;
}
```

Data dependency does not permit pipeline, and therefore there should be $k - 1$ empty cycles. If in the body loop we insert independent operations, these can fit the empty cycles. If we put k independent operations the execution time should increase at least by a cycle for every iteration of the loop. This simple test works when the operation is a single instruction. But our target is a composition of instructions. The behavior observed is slightly different, because dynamic scheduler and native compiler can choose different scheduling policies respect to the hardware available and the number of instructions in the body loop. Such test can be very useful to understand how much data dependency among operations can affect the execution time of an application and how much the architecture is efficiently used. For $c+=a*b$, the difference of execution time between pipelined and not pipelined stream of operations can be quantified as $(2k-1)T_c$. This suggests that at very fine grain, it can be better to have no data dependency at all, or at least longer stream of independent operations, at least k independent operations.

We remind that we are not writing test in assembly code, and therefore is duty of the compiler to optimize code, so that characteristics of processors are exploited. The complexity of the actual processor is so high that we know that modeling its behavior can be infeasible, especially if we have not control of the executable produced by a compiler. Our goal is to achieve an understanding how compiler and platform interact, and therefore to determine in a quantitative way their performance. The idea suggested about the exploitation of the pipeline architecture makes sense but there are exceptions: Pentium II and SGI R5000. When we run our tests on R5000 processors we noted that we cannot achieve the maximum performance. The compiler unrolls the body of loops and using an heuristic reorganizes operations schedule. The performance is slightly unstable because the heuristic works better for different numbers of instructions in the loop body. In other words, when we insert an instruction in the body loop, it can affect execution time in non intuitive way. When we run our test on Pentium II we noted that the pipeline architecture is not used properly. The floating point functional unit in the Pentium II uses a *stack register*. If we look at the assembly code generated by Microsoft compiler, we can see that among floating point operations there are (a lot of) register data exchanges. Such operation can be done in parallel with floating point operations. But the experimental results suggested that the pipeline properties of this functional unit is not exploited and it seems that managing the stack is the bottleneck. Furthermore execution time of the single operation is input values sensible. Indeed, the test for DELL machines was re-designed so that no overflow can occur. Quantitatively, handling exceptions is very expensive, tests suggested that with overflow the computation takes 30 times the execution without exception (these are not reported, obviously).

This scenario is very interesting. The platforms we have studied are very different and we have understood that there are different optimizations that cannot be achieved or understood only looking the *C* code. Fine grain code reorganization can be a good idea, and almost all the times it can increase performance, but such reorganization should be done by the native compiler of the machine. Indeed, the compiler is aware of real properties. For our purposes, we can reorganize the code such as whether native compiler does not perform a good work, the executable might still have good performance (our experience say that the compiler may reorganize the code in a such a way that our intent of *right organization* is completely messed up).

In Table 4.1 and following we summarized the experimental results. We can see as SPARC ULTRA Table 4.10, 4.11 and 4.12 has the most predictable behavior and as the Pentium II has the most unpredictable behavior, i.e. Table 4.13, 4.14 and 4.15.

We measure performance of three operations: integer addition, double floating point addition and double floating point add-and-multiply. In Table 4.1 there are experimental results of the integer addition. The test is split in two parts. The Table where is written *SEQUENTIAL* we introduce in the body loop 1, 2, 3, 4, 5 and 6 dependent operations (the loop contains 3, ..., 8 operations) such as the body loop increases in size and the average execution time of the body loop should increase by a constant, the latency through the number of stages of the architecture. Below the first row, we can see the differential time between basic body loop and incremented one. In the second row we can see the time expressed in nanosecond and in the third row the time expressed in number of machine cycles. If we say that T is the average time to execute the two operations body loop with data dependency and T_k is the average time to execute the body loop with $k+2$ operations, we reported the average time $T_k - T$. The Table where is written *PIPELINABLE* follows the same idea but the operation inserted are independent, no data dependency. The average values reported here are computed on the base of at least 10,000,000 iterations, the number of iterations varies from architectures.

PIPELINE	1	2	3	4	5	6
$x = y + z$	9.0 nsec	19.5 nsec	42.5 nsec	42.0 nsec	53.5 nsec	65.0 nsec
cycles	1.62 cycles	3.51 cycles	7.64 cycles	7.55 cycles	9.62 cycles	11.69 cycles

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	11.5	22.5	34.5	44.5	57.0	67.0
cycles	2	4	6	8	10	12

Table 4.1: SGI R5000 Integer addition. In the second table we can see that every two cycles it is started an operation, that is, the integer unit is a two stages integer unit. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent. We can see that the architecture is not completely utilized.

PIPELINE	1	2	3	4	5	6
$x = y + z$	0.0	0.5	39.0	6.5	11.0	17.5
cycles	0	0	7	1.1	1.98	3.1

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	22.0	45.5	67.0	90.0	112.0	133.5
cycles	3.9	8	12	16.1	20.1	24

Table 4.2: SGI R5000 Double floating point addition. In the second table we can see that every four cycles it is started an operation, that is, the floating point unit is a four stages floating point addition unit. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent. We can see that we can achieve a throughput of one operation at every cycle. Note that the behavior of the functional unit is not stable for small body loop

PIPELINE	1	2	3	4	5	6
$x += y * z$	39.5	79.0	117.0	156.5	195.5	234.0
cycles	7.1	14.2	21	28.1	35.1	42

SEQUENTIAL	1	2	3	4	5	6
$x += y * z$	39.5	78.5	157.0	156.0	195.5	235.0
cycles	7.1	14		28	35.1	42

Table 4.3: SGI R5000 Double floating point low latency multiply-addition. In the second table we can see that every seven cycles it is started an operation, that is, the floating point unit, for this operation, is composed by seven stages. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent. We can see that we could not improve the throughput.

PIPELINE	1	2	3	4	5	6
$x = y + z$	-6.27	-6.25	6.88	3.68	3.69	3.68
cycles	-$\frac{2}{3}$	-$\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	28.8	46.8	55.8	76.2	99.6	115.2
cycles	2.9	4.7	5.6	7.70	10	11.6

Table 4.4: HAL station 300 Integer addition. In the second table we can see that it seems that every two cycles it is started an operation, that is, the integer unit, for this operation, is composed by two stages. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent. We can see that the functional unit has more than one add unit

PIPELINE	1	2	3	4	5	6
$x = y + z$	-0.6	-0.2	0.2	10.0	27.4	32.2
cycles	0	0	0	1	2.7	3.2

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	48.8	96.6	145	194.4	241.8	291
cycles	4.9	9.7	14.6	19.6	24.4	29.3

Table 4.5: HAL station 300 Double floating point addition. This case is very curious, increasing the size of the body loop we decrease the overall execution, i.e. negative time, this is counterintuitive. But the idea is that the scheduler as more instructions, more parallelism to exploit and this can affect performance.

PIPELINE	1	2	3	4	5	6
$x += y * z$	-0.4	0.0	0.2	10.0	27.4	32.6
cycles	0	0	0	1	2.7	3.2

SEQUENTIAL	1	2	3	4	5	6
$x += y * z$	49.0	97	145	194	242	291
cycles	4.9	9.7	14.6	19.6	24.4	29.3

Table 4.6: HAL station 300 Double floating point addition and multiplication. In the second table we can see that every five cycles it is started an operation, that is, the floating point unit, for this operation, is composed by five stages. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent.

PIPELINE	1	2	3	4	5	6
$x = y + z$	6.8	16.2	25.9	38.8	45.2	54.6
cycles						

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	12.6	28.9	32.4	38.5	48.0	57.7
cycles						

Table 4.7: SPARC station 5 integer addition

PIPELINE	1	2	3	4	5	6
$x = y + z$	72.6	119.2	174.7	247.2	295.4	362.0
cycles						

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	54.8	115.9	185.9	237.2	308.0	361.3
cycles						

Table 4.8: SPARC station 5 double addition

PIPELINE	1	2	3	4	5	6
$x+ = y * z$	142.6	307.8	463.6	617.0	772.0	925.6
cycles						

SEQUENTIAL	1	2	3	4	5	6
$x+ = y * z$	200.5	385.3	607.8	780.2	975.4	1178.8
cycles						

Table 4.9: SPARC station 5 double addition and multiplication

PIPELINE	1	2	3	4	5	6
$x = y + z$	5.8	6.0	11.8	11.9	17.8	18.0
cycles	1	1	2	2	3	3

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	6.0	12.0	17.8	23.9	29.8	36.1
cycles	1	2	3	4	5	6

Table 4.10: SPARC ULTRA Integer addition. In the second table we can see that every cycle it is started an operation. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent. We can see that the functional unit has more than one integer add unit

PIPELINE	1	2	3	4	5	6
$x = y + z$	-0.2	-0.2	5.7	11.8	17.5	23.6
cycles	0	0	1	2	3	4

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	18.1	35.9	53.7	73.0	90.4	107.4
cycles	3	6	9	12	15	18

Table 4.11: SPARC ULTRA Double floating point addition. In the second table we can see that every three cycles it is started an operation. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent.

PIPELINE	1	2	3	4	5	6
$x+ = y * z$	-0.1	0.2	12.4	23.9	29.9	36.0
cycles	0	0	2	4	5	6

SEQUENTIAL	1	2	3	4	5	6
$x+ = y * z$	35.8	71.8	107.6	144.2	181.0	215.7
cycles	6	12	18	24	32.3	38.59

Table 4.12: SPARC ULTRA Double floating point addition and multiplication. In the second table we can see that every six cycles it is started an operation. In the first table we can see how the performance are increased by the reorganization of the code by the compiler when the instruction are independent.

PIPELINE	1	2	3	4	5	6
$x = y + z$	2.9	6.0	5.9	12.0	13.8	24.0
cycles	1	2	1.9	4	4.6	8

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	2.9	5.1	9.1	11.4	14.6	16.4
cycles	1	1.7	3	3.8	4.85	5.47

Table 4.13: DELL Dimension integer addition

PIPELINE	1	2	3	4	5	6
$x = y + z$	11.2	43.55	54.0	60.0	90.85	94.37
cycles	3.73	14.52	18	20	30.28	31.46

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	30.14	50.62	75.41	90.68	127.5	148
cycles	10	16.79	25.4	30.2	42.5	49.33

Table 4.14: DELL Dimension Double floating point addition

PIPELINE	1	2	3	4	5	6
$x+ = y * z$	16.2	50.5	73.6	73.9	101.9	413.6
cycles	5.4	16.83	24.5	24.2	33.97	137.8

SEQUENTIAL	1	2	3	4	5	6
$x+ = y * z$	42.16	83.17	110.96	129.63	178.66	206
cycles	14.05	27.72	36.99	43.2	59.55	68.67

Table 4.15: DELL Dimension Double floating point addition and multiplication.

PIPELINE	1	2	3	4	5	6
$x = y + z$	3.3	6.71	6.6	13.5	15.4	26.9
cycles	1	2	2	4	4.6	8

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	3.2	5.7	10.0	12.6	16.1	18.3
cycles	1	1.7	3	3.7	4.8	5.4

Table 4.16: DELL Poweredge integer addition

PIPELINE	1	2	3	4	5	6
$x = y + z$	13.75	54.3	64.2	71.2	105.0	108
cycles	4.13	16.31	19.28	35	36	

SEQUENTIAL	1	2	3	4	5	6
$x = y + z$	33.6	54.1	81.3	97.8	138.3	162
cycles	10	16.25	24.4	32.6	41.54	48.65

Table 4.17: DELL Poweredge: double floating point addition.

PIPELINE	1	2	3	4	5	6
$x+ = y * z$	15.7	65.1	80.87	82.02	110.3	470.7
cycles	4.71	19.5	24.2	24.6	33.12	141.3

SEQUENTIAL	1	2	3	4	5	6
$x+ = y * z$	46.85	89.82	120.3	142.17	196.1	226.55
cycles	14.7	26.97	36	42.69	58.89	68

Table 4.18: DELL Poweredge: double floating point addition and multiplication.

4.2 Memory Hierarchy, Latency

Current platforms have more than one level of cache, and therefore different access latencies. We want to design a general test which is able to measure access time on different levels of memory hierarchy. We distinguish reads and writes, because these operations are implemented differently and sometimes this difference is remarkable.

The idea of tests is to measure a memory access indirectly by two distinct measures of the same operation, with and without access to memory. The operations have data dependency so there should be no optimizations. In the following sections we describe briefly the tests to measure the access time for a three level cache architecture. We can see these tests as an interesting tool to understand what is the memory hierarchy architecture. Indeed, it is possible to devise a try and error approach that automatically understands the memory hierarchy and measure its performance.

Cache parameters:

SPARC station 1: it has 64K bytes direct mapped unified cache, line size of 16 bytes, the write policy is Write-through. This architecture can be considered an old-style memory architecture but it offer an interesting test-bed since code and data share the same cache. This architecture is harder to handle because we cannot know in advance the size of the cache dedicated to pure data.

SPARC station 5: it has a 8K bytes direct mapped data cache, a 16K bytes direct mapped instruction cache and line size of 16 bytes, the write policy is write-through. The architecture is good test-bed because has two distinct and small caches, so we can see independently the misses due to instructions and data.

SPARC Ultra: It has a two level cache. It has on-chip a 16K bytes instruction cache, 2-way associative with line size of 32 bytes. It has on-chip a 16K bytes direct mapped data cache with line size of 32 bytes. The write policy is write-through. It has an external cache of 512K-1M bytes, this is a direct mapped cache unified for data and code, its line size is 32 bytes and the write policy is write-back.

SGI: R5000 It has a two level cache. It has on-chip a 32K bytes 2-way associative instruction cache, with line size of 32 bytes. It has on-chip a 32K bytes 2-way associative data cache with line size of 32 bytes. The write policy is write-through.

It has an external cache of 512K bytes, this is a direct mapped cache unified for data and code, its line size is 32 bytes and write policy is write-back.

DELL: DELL DIMENSION XPS D333 & PowerEdge 4200 , Sirius & Heze: They have Pentium II processors at 333Mhz and at 300Mhz respectively, but the structure of the memory hierarchy is similar. There are two level of caches. The first level is composed by a 16K bytes data cache and 16K bytes instruction cache. The cache line is 32 bytes. The second level is a 512K bytes unified cache. They have different main memory. These two platforms are interesting because we can investigate how the last level of the memory hierarchy, the main memory, affects the performance. This communications are by far more expensive than other communications among different levels of the memory hierarchy. They magnify the overall performance of algorithms with a small number of accesses on the high level of the memory hierarchy (Hard disk).

Fujitsu: Hal Station 300 it has only one level of cache. A 128K bytes 4-way set associative data cache and a 128K bytes 4-way set associative instruction cache, a cache line of 128 bytes and its write policy is write back.

4.2.1 First Level L_1

This is usually an on-chip memory, the read time takes one clock cycle and it is pipelined. A very simple test to measure read time and write time at the first level cache can be designed very quickly. We take an array of size N and we read, or we write, a block of the same size of the cache. We fill the cache in and we iterate operations known. The accesses to the array block should have unitary stride. The test to measure write time is similar. Note that writing on L_1 can require other operations on successive level of memory hierarchy, this depends on the write policy used level by level. Since current platforms are pipelined, accessing cache locations should require no time because is one step in the pipe.

4.2.2 Level L_2 and Level L_3

L_i is usually an out-chip memory and reading takes at least three cycles. To measure time access we must be sure that every time we perform an access we reach the cache level desired, neither below nor above. First of all we read, or write, an array block of same size of level cache L_i. We can say that at level i cache size is at most four time bigger than cache size at level $i-1$ and at least four time smaller than cache size at level $i+1$. Suppose to read several times an array from element 0 to element k, such that the elements read fill in perfectly L_2 but L_1. Since L_1 is smaller than L_2, we can always access an element in L_2 but not in L_1, because in L_1 there are elements we have read and never element that we will read before to move out from the cache. This is true if we do not access elements with a unitary stride. The stride should be greater than cache line (to avoid space locality) and small enough so that the number of elements read is bigger than L_1 capacity. Since the particular way we access elements in the array, we can generalize. We can say that any element we access in L_i is not present in any of L_j with $j < i$. To avoid space locality due to the cache line or page size, access stride cannot be unitary but at least equal to cache line, resp. L_2, or page size, resp. L_3.

4.2.3 Implementation

We describe only one test because the other ones are similar. The test is organized in three phases. In the first part we measure the time spent to perform computations, in the second part me measure computation and reads and in the third part we measure computations and writes.

```
/** Computations */
START_CLOCK;
for (j=0; j<l1; j++) {

  for(i=0;i<l2;i+=LINE) {
    k+=LINE;
    potential(k); // to avoid unwanted optimization
  }
  potential(k); // to avoid unwanted optimization
}
END_CLOCK;
work += duration;

/** Fill cache */
for(i=0;i<RangeOnL2;i++) {
  A[i]=i;
}
```

```
/** Computations + Reads */
START_CLOCK;

for (j=0; j<l1; j++) {
  for(i=0;i<l2;i+=LINE) {
    k+=A[i];
    potential(k);
  }
  potential(k);
}
END_CLOCK;
timeRead+=duration;

/** Computations + Writes */
START_CLOCK;

/* Work */
for (j=0; j<l1; j++) {
  for(i=0;i<l2;i+=LINE) {
    A[i]=i+j;   /* if unrolled,  pay attention independent operations */
    potential(k);
  }
  potential(k);
}
END_CLOCK;
timeWrite+=duration;
```

We can see that we have chosen a stride quantified by $LINE$. Stride depends on what level we are testing. The LINE to test main memory is bigger than cache line and it should be the page size. Result of tests should be considered carefully because they are not intuitive and they depend on many factors. Table 4.19, ..., 4.21 show the experimental results of SPARC machines. Executables are produced by *gcc* compiler. Access latency from and into the first level cache is almost hidden into the stages of pipelined computation. If we do not optimize the code, experimental results are completely different. Without optimization it is possible that reads and writes can interrupt *the pipe* and therefore we would measure not only access latencies but also the the number of stages in *the pipe*. In general we can say that the first level cache can be accessed for free (at least a clock

Level	Read	Write
L1	0.58 ns	29.8 ns
Memory	55.35 ns	56.82 ns

Table 4.19: SPARC 5: the access time is expressed in nano seconds.

Level	Read	Write
L1	0.52 ns	54.98 ns (5 cycles)
Memory	37.6 ns (4 cycles)	539.12 ns (53 cycles)

Table 4.20: Hal station 300: access time expressed in nanosecond and in parenthesis number of cycles.

Level	Read	Write
L1	-2.08 ns (-.3)	-73.6 ns (-12)
L2	77.7ns (12 cycles)	-52.1 ns (-9 cycles)
Memory	593.4 ns (100 cycles)	371.11 ns (61 cycles)

Table 4.21: SPARC ULTRA, two processors

cycle) but we paid for all the other accesses. Do not be worried about negative time. The writing loop exploits independent operations and it can offer better pipeline scheduling furthermore some caches can use a write buffer, and therefore hide write latency. This is a weakness of the test that will be fixed in the future.

Even if we are testing same processors, i.e. Pentium II, with a different clock cycle, Table 4.23 and 4.24 differ. We can see that there is no difference for the number of cycles to access the first level cache, but they differ for access time on the second and third level of memory. The second level cache is off-chip and it is possible they differ for their write policies and other features. The main memory is shared by two processors in DELL Poweredge and therefore it is understandable why the access to the main memory is slower respect to the other platform.

Level	Read	Write
L1	5.32 ns (1 cycles)	28.94 ns (4.5 cycles)
L2	262.96 ns (47 cycles)	371.63 ns (66.8 cycles)
Memory	625.71 ns (112 cycles)	665.78 ns (119 cycles)

Table 4.22: SGI, R5000

Level	Read	Write
L1	3.36 ns (1 cycles)	-9.25 ns (-3 cycles)
L2	24.3 ns (8 cycles)	4.10 ns (1 cycles)
Memory	136.32 ns (45.4 cycles)	207.20 ns (69 cycles)

Table 4.23: DELL dimension (Microsoft compiler) Pentium II.

Level	Read	Write
L1	3.75 ns (1 cycles)	-10.24 ns (-3 cycles)
L2	68.9 ns (22 cycles)	69.23 ns (22 cycles)
Memory	224.79 ns (74 cycles)	544.67 ns (181 cycles)

Table 4.24: DELL Poweredge (Microsoft compiler) Pentium II processors.

Chapter 5

Fractal Matrix-Matrix Multiplication

5.1 Introduction

Matrix multiplication is a very well known application studied by many researchers along these years. As a recall a briefly description of the application follows. Let us indicate by capital letters matrixes ($A, B \ldots$ and without loss of generality we can say that $A, B \in \Re^{n\times n}$) and by subscribed lower case the elements ($a_{ij}, b_{ij} \ldots$ in a row-column notation). Any element c_{ij} in matrix C obtained by moltiplication, $C = AB$, is the result of the following summation $c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}$.

There have been proposed different algorithms to compute matrix multiplication. The most common is the ijk-loop algorithm which follows the definition of matrix multiplication and it has complexity $O(n^3)$. There are blocked algorithms with also complexity $O(n^3)$ with a better data locality (i.e. [23]) and different size of blocks. There is Strassen's algorithm with complexity $O(n^{\log_2 7})$ ([59]) and Winograd's variant which performs less additions, either one is still a blocked algorithm requiring extra space for temporary matrixes. The best known algorithm is due to Coppersmith and Winograd with complexity $O(n^{2.376})$ ([20]), but in practice it is not used by any library. We are particularly interested on the blocked algorithm with time complexity $O(n^3)$ because our goal is to study data locality and compiler-technique to exploit it, and this algorithm offer a good and simple example.

Matrix multiplication is more than a case study or a toy example as the number of papers available on the subject, directly and indirectly related, suggests. Matrix multiplication can be proven as basic operation in several applications, i.e. linear algebra, and optimal performance can be achieved by a tuned matrix multiplication routine ([43], [44]). To achieve *portable performance* there are several approaches so that peak performance can be achieved for different platforms and for different algorithms ([64], [13], [24], [27], [28]). Researchers have studied the impact on performance of several issues: data layout, data locality, latency hiding, register allocation, instruction scheduling and instruction parallelism. These are fundamental issues to achieve performance. In this paper we address all of them at once by a general approach and from source code.

Let us give an overlook on the most common optimizations performed on matrix multiply code. A general technique that today's compilers and algorithms try to exploit so that temporal locality is maximized is loop tiling ([45], [54], [66], [49], [67]). Loops are tiled and it is increased time locality, therefore data in caches can be reused reducing the so called *capacity misses*. But tile sizes are machine dependent parameters, based not only on the cache sizes but also on any technique to reduce *self interference*, for example copying elements of matrix in contiguous memory space ([33], [27]).

Vendor libraries intensively use the available information of platforms so that the parameters to determine tile sizes, scheduling instruction and other optimizations are information which can be inferred easily.

Automatically tuned packages ([64], [13]) measure these parameters by interactive tests, and then they produce machine tuned code. Automatically tuned packages work very well and in general they achieve better performance than the vendor libraries. They have the advantage that if we want to install the package on another machine this is possible, achieving maximum performance, but spending time for a new installation.

Another approach, completely different, is called *auto-blocking*. The tile sizes are not determined by any *a priori* information but they are exploited by a recursive decomposition of the problem. This approach can be proven asymptotically optimal ([32]). We find these applications appealing, and in literature are called *cache-oblivious algorithms*. Recursive-based algorithms are the most common example of cache-oblivious algorithms and they do not actually use standard layouts. Non standard layout are applied because space locality is maximized with respect to the way the problem is decomposed so that data locality are maximized at any sub-problem decomposition. This non standard layouts are called *recursive layouts* and they may exploit different features ([18], [17], [61], [30]). We have chosen to deal with a variant of the Z-Morton layout ([17]) that we called *fractal layout* (note that it is not the *bit interleaved* layout [32] because the definition is valid only for power of two matrixes) because it fits naturally the naive blocked matrix multiply algorithm and it permits to exploit some peculiarities of the computation. We have considered the problem of format conversion from and to the most common layout formats, i.e. row-major format which is by default in C and column-major format which is by default in Fortran. The conversion time is negligible for big enough matrixes ([18]) and until the device where the matrixes are stored is fast enough that the time to access the entire matrix is negligible with respect to the time spent in the matrix multiplication (i.e. if the matrixes are stored on a hard disk, the conversion time is not negligible anymore).

During our investigation and performance evaluation of our implementations, we found that an optimal cache utilization is not enough for performance purpose, a good register utilization must be achieved. The register file is the first level of memory and performance is based on register allocation. A small number of loads and stores reduces the traffic from/to the cache and latency hiding of loads and stores avoid to stall pipelined CPUs ([27], [64]). Currently, the practical way tiling is combined with other optimizations is function of the level of memory hierarchy which the code is written for, i.e. tiling for the first level of cache may be different from the tiling at register level where other optimizations are performed as loop unrolling, software pipelining, peeling and scalar replacements ([49]). But for recursive algorithms no compiler is smart enough to perform unfolding of last calls (*leaves*) and performs optimizations on the code. We investigated this problem and the impact of compiler optimizations onto unfolded leaves. We present an automatic technique to unfold and perform scalar replacement for recursive matrix multiplication algorithms. In fact, if the code may need any tuning so that to achieve an efficient implementation, then at register level we need better code. We developed parametric code generators that automatically write unfolded leaves of different sizes so that it is becoming easy tuning code for a different architecture.

In spite of all the bibliography devoted to matrix multiplication, there is no available results to optimize the index computation, that is, the computation of the indexes to access matrix elements, specially for recursive algorithm. Indeed, recursive algorithms are often based on power of two matrixes (with padding, overlapping, or peeling) where the index computation is very simplified. But in general, aside the major computation involving floating point instructions, there is the index computation involving integers necessary to load and store elements of the matrixes. We devised an approach to prune the calling tree of a recursive algorithm, saving index computations and open an interesting way to remove the recursion at once, for such a purpose we introduce the concept of *type DAG*. We discovered that in matrix-multiplication-like applications it is very common to perform the same computation over and over on different data, and it can be useful store index computation results on a small temporary space. Of course, this interferes with the data space but simulations give us insights so that we can claim that it does not affect the overall computation.

We are interested to implement a cache-oblivious tunable package, where performance is the first goal. We compare this approach w.r.t. automatically-tuned machine dependent approach ([64]) that very often outperform vendors applications. We can see that we can achieve comparable performance on different architectures just adapting the computation on the leaves of the recursion algorithm.

Note that in this paper will be marginally cited numerical stability. Stability and accuracy are important features but for machine without extended precision accumulator and with register file, the order of the computation should not affect the worst case error estimation (Lemma 2.4.1 [34] or Lemma 3.4 [38]).

5.2 $2^k \times 2^k$ Matrix Multiplication

This case study is a simplification of the matrix multiplication problem where most of the current algorithms have no optimal performance due to the high number of data conflicts in cache. In this Section we explain the *fractal layout* of data, the blocking algorithm and its data locality property.

5.2.1 Fractal Layout and Blocked Algorithm.

Let n be a power of two ($2^k = n$). We consider two dimensional matrixes $n \times n$. We indicate an element of matrix A by the usual notation a_{ij}. The indexes i and j indicate respectively the row and the column of A in which the element is located. The matrix A can be seen as the composition of four sub matrixes A_0, A_1, A_2 and A_3.

- $A_0 = \{a_{ij} \text{ with } 0 \leq i, j < \frac{n}{2} - 1\}$.

- $A_1 = \{a_{ij} \text{ with } 0 \leq i < \frac{n}{2} - 1 \text{ and } \frac{n}{2} \leq j < n - 1\}$.
- $A_2 = \{a_{ij} \text{ with } 0 \leq j < \frac{n}{2} - 1 \text{ and } \frac{n}{2} \leq i < n - 1\}$.
- $A_3 = \{a_{ij} \text{ with } \frac{n}{2} \leq j, i < n - 1\}$.

The layout of the matrix A, the way it is stored in the memory, can be recursively defined as follows. Lay out of A is the linear sequence of the lay out of A_0, then the lay out of A_1, then the lay out of A_2 and eventually the lay out of A_3. See Figure 5.1. We apply the

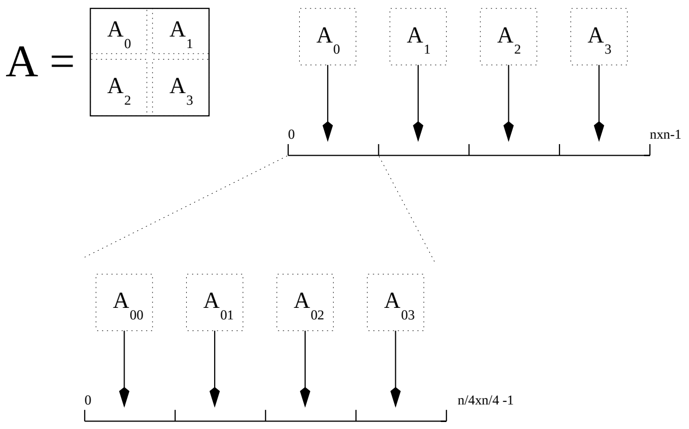

Figure 5.1: The way a matrix is split and its recursive lay out definition.

blocked matrix multiplication algorithm with $O(n^3)$ operations. Let A and B be two $n \times n$ matrixes. Let C be the matrix which is the result of the multiplication of the matrixes A and B, that is, $C = A \times B$.

Algorithm

```
fractal(A,B,C) {

 if (size(A)==size(B)==1)  /* scalar */
     C+=AB;
 else {
     S = {(A0,B0,C0),(A1,B2,C0),(A0,B1,C1),
          (A1,B3,C1),(A2,B0,C2),(A3,B2,C2),
          (A2,B1,C3),(A3,B3,C3)}

     for each (A',B',C') in S
         fractal(A',B',C');
 }
}
```

♠

The algorithm defines a tree, see Figure 5.2, the *call tree*. The tree has $\log n$ levels. In

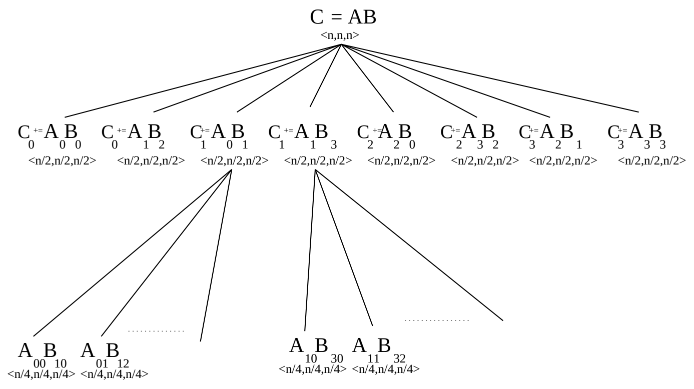

Figure 5.2: The call tree of the blocked matrix multiplication.

Figure 5.2 the matrix A_{ij} is the sub matrix $(A_i)_j$ and it can be identified in the lay out by its first element. The relation between the first element and the indexes i and j is expressed by the formula $i\frac{n^2}{4} + j\frac{n^2}{4^2}$. The size of the matrix A_{ij} is $\frac{n^2}{4^2}$. The denominator $(4^{l=2})$ is a power of four and its exponent is function of the number of indexes, i.e. two indexes i and j. The fractal scheme generate a class of fractal algorithms, is correspondence with specific ordering of execution of the recursive call. A fractal algorithm is a visit of the call tree. We focus our attention on two particular algorithms.

Definition 5.1 *We call CAB-fractal the algorithm obtained from the fractal scheme when the recursive calls are executed in the following order:* (A_0, B_0, C_0), (A_1, B_2, C_0), (A_1, B_3, C_1), (A_0, B_1, C_1), (A_2, B_1, C_3), (A_3, B_3, C_3), (A_3, B_2, C_2), (A_2, B_0, C_2).

Definition 5.2 *We call ABC-fractal the algorithm obtained from the fractal scheme when the recursive calls are executed in the following order:* (A_0, B_0, C_0), (A_0, B_1, C_1), (A_2, B_1, C_3), (A_2, B_0, C_2), (A_3, B_2, C_2), (A_3, B_3, C_3), (A_1, B_3, C_1), (A_1, B_2, C_0).

These ordering of recursive calls can be seen as Hamiltonian cycles in a three dimensional binary cube, Figure 5.3.

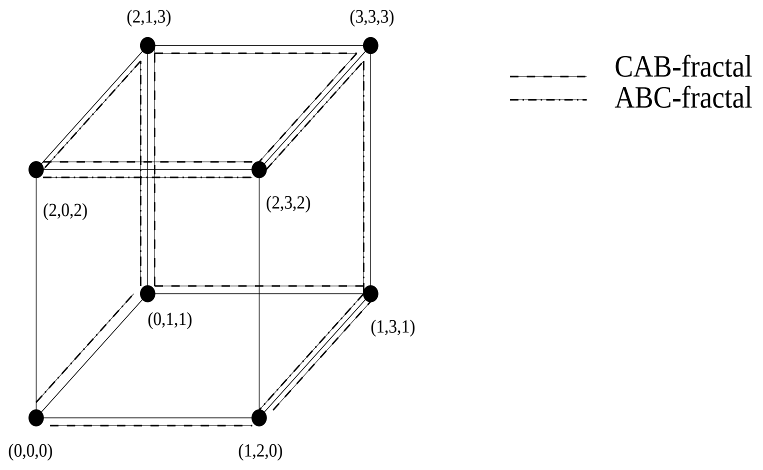

Figure 5.3: CAB-fractal and ABC-fractal as Hamiltonian cycles in a binary cube a node (i, j, k) stays for (A_i, B_j, C_k).

5.2.2 Misses of Fractal Algorithm

Proposition 5.1 *The fractal matrix multiplication has* $Q(S) = \Theta(\frac{n^3}{\sqrt{S}})$ *accesses over the location* S*, where* $2^s = S$[1] *and matrix size is* n*.*

Proof: we split the proof in two parts, in the first part we show a lower bound and in the second part we show an upper bound to the number of accesses over the location S.

The algorithm has an intrinsic access complexity. Whatever it is the way we visit the recursion tree at a certain level in the tree, the size of the matrixes are small enough that they fit the cache and their matrix multiplication can be done without any miss. Let k^2 be the size of a sub-matrix and let us consider a sub-matrix multiplication $C_r + = A_s * B_t$. Whether S is the size of the cache then $3k^2 = S$ and therefore $k = \lfloor\sqrt{\frac{S}{3}}\rfloor$. Even if k is always a power of two (i.e. $k = 2^{k_0}$), here, for sake of explanation, we consider $\sqrt{\frac{S}{3}}$ as an opportune integer. We have chosen a sub-multiplication and it identifies a sub-tree with k^3 operations and there are $\frac{n^3}{k^3}$ sub-trees. In any sub-tree there is no misses except the read misses to fill in the cache with appropriate data. Let m be a coefficient $0 \leq m \leq 3$ of locality among consecutive leaves (w.r.t. the visit policy). It identifies the number of common sub-matrixes to two consecutive sub-multiplications. We must read $(3 - m)k^2$ elements going from a leaf to another one. We identify the length of the cache line by the

[1]because the size of caches is power of two

integer l. In fact, the number of reads outside the cache will be affected by the length of the cache line, $Q(S) \geq \frac{m}{l}\frac{n^3}{k^3}k^2$. When we substitute the value of k, we obtain that $Q(S) \geq \frac{3-m}{l}\sqrt{\frac{3}{S}}n^3$ (i.e. the miss ratio is $\frac{3-m}{l}\sqrt{\frac{3}{S}} = \Omega(\frac{1}{\sqrt{S}})$). A general proof of the inequality can be found also in [39].

To estimate an upper bound we suppose that the cache is a direct mapped cache and we give an estimation to the conflict misses, this is sufficient since there is no self interference. Take eight consecutive sub-multiplications with k^3 operations each, as above. To visualize how they interfere we consider the pre-order visit of the recursive tree. $C_0+ = A_0B_0$, $C_0+ = A_1B_2$, $C_1+ = A_0B_1$, $C_1+ = A_1B_3$, $C_2+ = A_2B_0$, $C_2+ = A_3B_2$, $C_3+ = A_2B_1$, $C_3+ = A_3B_3$. We know that C_i, A_j and B_k (almost perfectly) fit the cache. Without loss of generality suppose that $C_0+ = A_0B_0$ has no interference, that is C_0 is in the first k^2 cache slots, A_0 is in the second k^2 cache slots and B_0 is in the last k^2 cache slots. Since the matrixes are stored sequentially we can note that in $C_0+ = A_1B_2$ there should be no interference, but in $C_1+ = A_0B_1$, matrix C_1 interfere with matrix A_0. In Figure 5.4 we

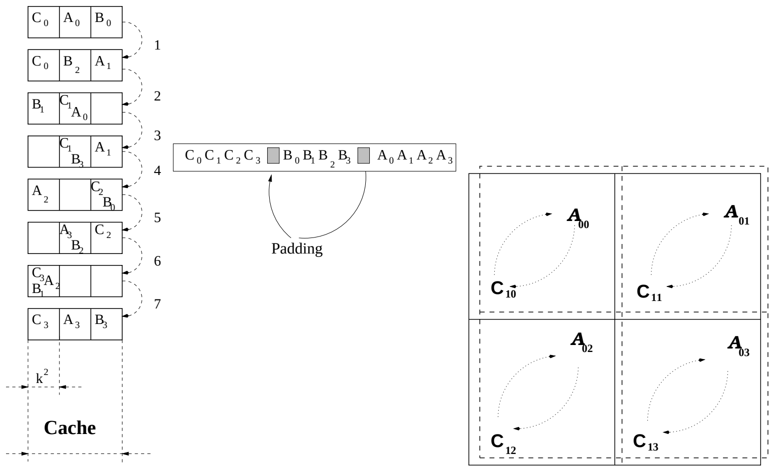

Figure 5.4: From left to right: in the first picture we can observe the layout in the cache of the sub-matrixes involved in the computation, in the second the layout in memory with a padding between consecutive matrixes and in the third how matrix C_1 and matrix A_0 can interfere in the same slot.

can see the pattern of interference among matrixes, when S is $3\frac{n^2}{4}$. An alignment due to a padding of $\frac{S}{3}$ elements permits to achieve such pattern. We indicate the interference in these eight multiplications as $I(S)$. In Figure 5.4 we can see in more detail as the two sub-matrixes C_1 and A_0 share the same set of locations in the cache. We indicate as $I_B(\frac{S}{3})$ the conflict misses related to such *lay out*. The sub-multiplications affected by conflict misses are $C_{10} = A_{00}B_{20}$, $C_{11} = A_{01}B_{23}$, $C_{12} = A_{02}B_{20}$ and $C_{13} = A_{03}B_{23}$. The number of misses is expressed by the following equation: $I_B(\frac{S}{3}) = 4I_B(\frac{S}{3*4})$ and $I_B(1) <= 3$. This equation has simple solution $I_B(\frac{S}{3}) = 3\frac{S}{3} = 3k^2$. So if we indicate by i the number of sub-matrix pairs that share the same cache slots, the number of conflict misses is $I(S) \leq \frac{n^3}{8k^3}i3k^2$ which, in turns, can be expressed as $I(S) = \frac{3i\sqrt{3}}{8\sqrt{S}}n^3$ (i.e. the miss ratio is $O(\frac{1}{\sqrt{S}})$). In Figure 5.4, i is seven but if we transpose B, i becomes 6. $\square$

If we want to get an quantitative idea of $Q(S)$ we can compute a simple case, for a data cache of 16KBytes, direct mapped, 32 byte line size and an element of the matrix is 8 bytes. If we add the interference misses and the capacity misses we obtain $Q(S) = O((\frac{1}{2} + \frac{3*7}{8})\sqrt{\frac{3}{2048}}) \sim 0.12$. Experiments with the Shade simulator show that we can achieve a $Q(S)$ ranging between 0.061 and 0.104, respectively with and without padding, which satisfies our estimation.

5.3 Square Matrix Multiplication

We generalize the approach applied for power of two matrixes to square matrixes and square matrix multiplication.

5.3.1 Square Matrixes, Fractal Layout.

We identify a square matrix of size k simply by $k \times k$. An integer k identifies a unique sequence $k_0, k_1, \ldots k_{\log k}$ where $k_0 = \lceil \frac{k}{2} \rceil$, $k_i = \lceil \frac{k_{i-1}}{2} \rceil$, $\ldots$, $k_{\log k} = 1$. We indicate, without any further specification, the first element of the sequence based on integer k as k_0. A square matrix $k \times k$, A, can be split in four sub matrixes A_0, A_1, A_2 and A_3. Matrixes A_1

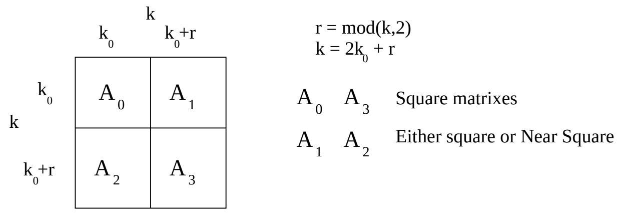

Figure 5.5: Splitting matrix A $k \times k$

and A_2 are *near square* since the number of row and the number of columns may differ by one, i.e. k is odd then A_1 is a $k_0 \times (k_0+1)$ matrix and A_2 is a $(k_0+1) \times k_0$ matrix. We will show that a fractal decomposition of a near square matrix offers a set of near square matrixes.

Proposition 5.2 *Let A be a $k \times h$ matrix with $h = k+1$, then A_0, A_1, A_2 and A_3 are near square matrix.*

Proof: the proof is done by complete enumeration of the possible cases. $k = 2k_0 + mod(k,2)$ and $h = 2h_0 + mod(h,2)$. If k is even $h_0 = k_0$. If k is odd $h_0 = k_0 + 1$.

k is even, $h_0 = k_0$	k is odd, $h_0 = k_0 + 1$
$\begin{array}{c\|c\|c} & h_0 & h_0+1 \\ \hline k_0 & A_0 & A_1 \\ \hline k_0 & A_2 & A_3 \end{array}$	$\begin{array}{c\|c\|c} & h_0 & h_0 \\ \hline k_0 & A_0 & A_1 \\ \hline k_0+1 & A_2 & A_3 \end{array}$

□

The proof of the other case $k+1 \times k$ is similar. We are going to investigate if it is possible determine a sub set of simple near square matrixes so that every square matrix can be tiled with. Let $\mathcal{T}$ be a set of matrixes $\{2\times 2, 2\times 3, 3\times 2, 3\times 3, 4\times 3, 3\times 4\}$.

Proposition 5.3 *Every square matrix and near square matrix A can be tiled with matrixes in $\mathcal{T}$.*

Proof: easy, by induction on k and enumeration of possible cases. □

Fractal Layout for multidimensional Arrays

Definition 5.3 *A fractal layout of a k-dimensional array $A \in \mathcal{R}^{ld_0 \times ld_1 \dots ld_{k-1}}$ with $\max_j ld_j - \min_j ld_j \le 1$, is the decomposition of A in 2^k linear arrays so that*

$$A_i = A_{\sum_{m=0}^{k} i_m 2^{k-m-1}} = \{a_{j_0,\dots,j_{k-1}} | \forall 0 \le l \le k-1 \textbf{if } i_l = 0 \quad \begin{array}{l} \textbf{then } 0 \le j_l \le \lceil \frac{ld_l}{2} \rceil - 1 \\ \textbf{else } \lceil \frac{ld_l}{2} \rceil \le j_l \le ld_l - 1 \end{array} \quad \}$$

with i_m either 1 or 0 for $0 \le m \le k$.

$i_0, \dots, i_{k-2}, i_{k-1}$	size of A_i	Start point of A_i
$0, \dots, 0, 0$	$\lceil \frac{ld_0}{2} \rceil \dots \lceil \frac{ld_{k-2}}{2} \rceil \lceil \frac{ld_{k-1}}{2} \rceil$	0
$0, \dots, 0, 1$	$\lceil \frac{ld_0}{2} \rceil \dots \lceil \frac{ld_{k-2}}{2} \rceil \lfloor \frac{ld_{k-1}}{2} \rfloor$	$\lceil \frac{ld_0}{2} \rceil \dots \lceil \frac{ld_{k-2}}{2} \rceil \lceil \frac{ld_{k-1}}{2} \rceil$
$0, \dots, 1, 0$	$\lceil \frac{ld_0}{2} \rceil \dots \lfloor \frac{ld_{k-2}}{2} \rfloor \lceil \frac{ld_{k-1}}{2} \rceil$	$\lceil \frac{ld_0}{2} \rceil \dots \lceil \frac{ld_{k-2}}{2} \rceil ld_k$
$\dots$		
$1, \dots, 1, 1$	$\lfloor \frac{ld_0}{2} \rfloor \dots \lfloor \frac{ld_{k-2}}{2} \rfloor \lfloor \frac{ld_{k-1}}{2} \rfloor$	$\prod_j ld_j - \lfloor \frac{ld_0}{2} \rfloor \dots \lfloor \frac{ld_{k-2}}{2} \rfloor \lfloor \frac{ld_{k-1}}{2} \rfloor$

Table 5.1: How to determine the stating point and size of for the first level in the fractal decomposition of a k-dimensional array.

We can see that for $k = 2$ we obtain the fractal layout for matrixes. In general the start point of a fractal sub-matrix A_i $(0 \leq i \leq 2^k - 1)$ can be computed in $O(2^{k-1}k)$ steps. In several applications it is required accessing single element in matrixes, i.e. $a_{i,j}$. For a matrix stored either by row major of by column major the identification of an element $a_{i,j}$ is done by a simple computation, i.e. $i{*}ld{+}j$ where ld is the leader dimension of the matrix. For a multidimensional array the computation is slightly more complex, but the number of operations to compute the location required is linear respect the number of dimensions. For example $a_{i_0,i_1,\dots i_k}$ is $a[\sum_{j=0}^{k} i_j \prod_{l=j+1}^{k} ld_l]$. In practice, on fractal matrixes we should perform a variation of a bitonic search and the number of steps is $O(2^{k-1}k \log_2(\max_j ld_j))$. If we consider that usually the dimensions k of the array is fixed and if for $0 \leq j \leq k-1$ we have $ld_j = N$, then array size is N^k and the access time is $O(\log_2(N))$. Note that this simplification works only if the number of dimensions is a constant and small enough w.r.t. N (Similar searching problems can be found in other applications and in the following of this thesis, Sparse Matrixes).

5.3.2 From Call Tree to Type DAG

A recursive tiling of a near square matrix is indicated with $\mathcal{T}$, this set of tiles infers also a set of basic multiplications, that is, a set of kernels that any matrix multiplication uses. With $\mathcal{M}(\mathcal{T})$ we indicated the set of possible basic matrix multiplications based on the set $\mathcal{T}$. For example for $\mathcal{T} = \{2 \times 2, 2 \times 3, 3 \times 2, 3 \times 3, 4 \times 3, 3 \times 4\}$ the basic matrix multiplications $\mathcal{M}(\mathcal{T})$ is $\{<2,2,2>, <2,3,2>, <3,2,2>, <2,3,3>, <3,2,3>, <3,3,2>, <3,3,3>, <3,4,3>, <4,3,3>, <3,3,4>, <3,4,4>, <4,3,4>, <4,4,3>\}$. The reason there is no basic computation as $<2,3,4>$ is because one of the matrix involved in the computation should not be a near square matrix, i.e. 2×4 does not belong to $\mathcal{T}$. If we look at the fractal algorithms, the set basic computations $\mathcal{M}(\mathcal{T})$ indicate when we can stop the recursion and we can prune the call tree.

Definition 5.4 *$Y(k) = \{< r, s, t >: r, s, t \in \{k, k-1\}\}$ then $\mathcal{M}_{2,4}(\mathcal{T}) \cup_{i=2}^{4} Y(k)$. We call $\mathcal{M}_{2,4}(\mathcal{T})$* kernel dictionary *of interval* $[2, 4]$.

We can see that for any k and j so that $j \geq 2k$, $\mathcal{M}_{k,j}(\mathcal{T})$ is a good kernel dictionary, that is, any square matrix multiplication involves routines from this set. For performace purpose the value of k and j may vary, and the lay out of the tiles may vary as well. We noticed that a general good kernel dictionary is $\mathcal{M}_{4,8}(\mathcal{T})$ where each tile is fractally tiled up to the single element. This choice has the advantage that layout and kernel dictionary can be decoupled. But other sets can be used as well, and for example a very interesting one is $\mathcal{M}_{16,32}(\mathcal{T})$, where each tile is laid out in a row-major. This is appealing because we can reuse software from other libraries, which are very optimized for this particular format. The disadvantage of this choice is that the layout of matrixes is function of the computation that matrixes are involved with. An example will explain this exception. Cosider the problem $< 33, 32, 32 >$ that is C is 33×32, A is 33×32 and B is 32×32. If we use the kernel dictionary $\mathcal{M}_{16,32}(\mathcal{T})$, we can see that matrix B must be decomposed further because A and C have to, as they are they do not belong to $\mathcal{T}$.

We can force a comparison with related works in literature where loop tiling and loop unrolling are applied to ijk-loop algorithms. The recursive tiling can be seen as the equivalent of tiling and kernel dictionary, that is, the unfolding of recursive algorithms, can be seen as loop unrolling.

By construction the types of the call tree leaves are limited, they are a constant number. Also the computation indicated by sub trees in the call tree are very similar to each other and most of them represent the same computation on different data. We introduce the definition of type dag which extracts from the call tree the sub tree without repetition.

Definition 5.5 *given a fractal algorithm $\mathcal{A}$, an input problem of sizes $< M, N, P >$ and the call tree $T = (V, E)$, the type dag $D = (U \subset V, F \subset E)$ is obtained from T applying the two following rule:*

- *every v_i in V which solves a problem of size $< m, n, p >$ is merged in only one u in U of type $< m, n, p >$, only one tree is maintained rooted at u.*

- *links among merged v_is and their parents are maintained, unless their parents will be merged.*

The size of the type dag D for square matrix multiplication grows logarithmically with the matrix sizes, therefore it is becoming more appealing for efficient implementation.

Proposition 5.4 *In the call tree of a square matrix multiplication of sizes $< n, n, n >$, the type of each call tree node at distance $0 \leq d \leq \lceil \log n \rceil$ from the root belongs to $Y(n_d)$.*

Proof: by induction on the distance from the root d. For $d = 0$ $Y(n_0) = Y(n)$ the statement holds. For the closure property of the fractal decomposition of square matrixes (sub matrixes of near square matrixes are near quare matrixes), all matrixes blocks at level $d+1$ have dimensions $\lfloor (n_d - 1)/2 \rfloor$ and $\lceil n_d/2 \rceil$, but $\lfloor (n_d - 1)/2 \rfloor = \lceil n_d/2 \rceil - 1$. We have that all types at level $d+1$ belong to $Y(\lceil n_d/2 \rceil)$ and therefore to $Y(n_{d+1})$. $\square$
In practice the size of the type dag is asymptotically bounded by its height and therefore by $O(\log n)$. The worst case scenario is when for every i, n_i is odd, because every element in $Y(n_{i+1})$ is in D and therefore $|U| = \sum_{i=0}^{\lceil \log n \rceil -1} |Y(i)| = 8\lceil \log n \rceil$. To reduce further the call tree, and therefore the type DAG, we can prune the call tree enlarging the set $\mathcal{T}$. A C-code definition of *type DAG* follows[2]

```
typedef void (*LEAF)(element *, element *, element *);
typedef struct type_dag_node_structure NS;

struct type_dag_node_structure {
  NS *p0,*p1,*p2,*p3,*p4,*p5,*p6,*p7;;   /* Sub-problem links */

  LEAF leaf; /* link to one element in the kernel dictionary */

  int p,n,m,order; /* sizes and type of the problem C+=AB */
  int i1,i2,i3;    /* C1, C2, C3 start points w.r.t. C*/
  int j1,j2,j3;    /* A1, A2, A3 start points w.r.t. A*/
  int k1,k2,k3;    /* B1, B2, B3 start points w.r.t. B*/
};
```

In Figure 5.6 we can see an example of type dag for a problem of size $< 17, 17, 17 >$. The construction of a type DAG can be logically decomposed in two steps. Supppose that the smallest leaf has size $< k, k, k >$, we can build up a call tree using space $O((\frac{n}{k})^3)$. A leaf in the call tree is determined by a field, a function pointer, which specifies the leaf computation. Then we visit the pruned call tree and we construct the type DAG.

[2]the notation for indexes i_l, j_l and k_l may be misleading, indeed, they are offsets, but this is a notation inherited from our first implementation, since it is not harmful we have not changed.

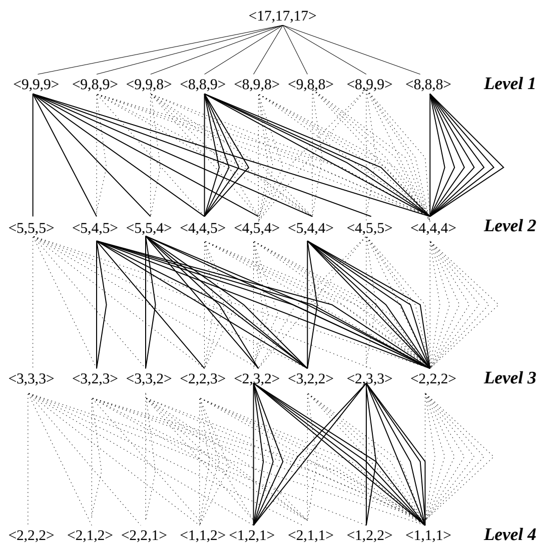

Figure 5.6: A $< 17, 17, 17 >$ Call Tree reduced to a $< 17, 17, 17 >$ type DAG.

We end up with a type DAG of size $O(\log \frac{n}{k})$. In fact, the current implementation binds these two steps together, so we decrease the work space ($O(\log n)$) and the execution time ($O(\log^2 n)$, we use a linear list as auxiliary data structure of size at most of $O(\log n)$ and we perform $O(\log n)$ accesses). A matrix multiplication is the visit of a type DAG, and an example follows:

```
void computeABC(NS *tree, element *c, element *a, element *b) {
  if (tree->leaf)
    (*(tree->leaf))(c,a,b);
  else  {
      computeABC(tree->p0,c,a,b);                           /* C0 += A0B0 */
      computeABC(tree->p1,c+ tree->i1,a,b+tree->k1);        /* C1 += A0B1 */
      computeABC(tree->p2,c+ tree->i3,a+tree->j2,b+tree->k1);  /* C3 += A2B1 */
      computeABC(tree->p3,c+ tree->i2,a+tree->j2,b);        /* C2 += A2B0 */
      computeABC(tree->p4,c+ tree->i2,a+tree->j3,b+tree->k2);  /* C2 += A3B2 */
      computeABC(tree->p5,c+ tree->i3,a+tree->j3,b+tree->k3);  /* C3 += A3B3 */
      computeABC(tree->p6,c+ tree->i1,a+tree->j1,b+tree->k3);  /* C1 += A1B3 */
      computeABC(tree->p7,c,a+tree->j1,b+tree->k2);         /* C0 += A1B2 */
  }
}
```

5.3.3 Scalar Replacement of Array Element

Unfolding a recursive algorithm gives a chance to a compiler to perform optimizations, basically removing the recursion. The tail calls of recursive algorithms can be unfolded so that computations of the leaves can be written as a sequence of instructions, then the code is fed into an optimizing compilers. We explain how the fractal algorithms can be unfolded and we present three different approaches that can be applied. The first is based on the conjunction of two fractal algorithms, the second is based on the on-chip matrix multiply in the ATLAS package and the third one is based on a generalization of cache tiling. Even if each approach is implemented in C-code, we can perform some optimizations at source level as scalar replacement, software pipelining, unrolling. These optimizations reduce the work of compilers, but they have still a lot to do. Indeed, scalar replacement suggests only an efficient register allocation, but the compiler codifies the register allocation and scheduling of instructions. The compiler can affects in several ways the performance of final applications and it is not easy to predict. Automatic packages try to solve this problem using timing tests to chose the champion over a set of algorithms. This approach overcome the problem choosing the fastest executable, no matter it is the best algorithm. This is not our point of view. We have a set of procedures decided in advance, the leaves of the type DAG. We need good executables from good algorithms. We need a good compiler. To facilitate this tedious, but necessary, job the genertators of code generates also an estimation of the performance of the single procedure, counting the number of loads and stores in the procedure itself. If we inspect the assembly code produced by any compiler we can check if there is any spilling and therefore, iterate with different optimizations or different compilers.

We propose different unfolding techniques but none of them is always the best. Many factors must be considered: the size of the problem, the number of scalar variables, the instruction set, the number of pipeline stages. We can see as a mixed approach must be followed so that peak performance can be achieved.

Fractal Heuristic for Register Allocation

The fractal approach to unfold the computation of the leaves is appealing because it is the approach applied at higher level in the memory hierarchy, and it permits to decouple lay-out from the computation in a very elegant way. With the notations $\uparrow CAB$-fractal and $\downarrow CAB$-fractal we indicate the following algorithms.

```
**Algorithm:** $\uparrow CAB$-fractal
if $f(|A|,|B|) < R$, then gen(A,B,C,FORWARD).
else {
    $\uparrow C_0A_0B_0$-fractal
    $\downarrow C_0A_1B_2$-fractal
    $\uparrow C_1A_1B_3$-fractal
    $\downarrow C_1A_0B_1$-fractal
    $\uparrow C_3A_2B_1$-fractal
    $\downarrow C_3A_3B_3$-fractal
    $\uparrow C_2A_3B_2$-fractal
    $\downarrow C_2A_2B_0$-fractal
}
```

```
**Algorithm:** $\downarrow CAB$-fractal
if $f(|A|,|B|) < R$, then gen(A,B,C,BACKWARD).
else {
    $\uparrow C_2A_2B_0$-fractal
    $\downarrow C_2A_3B_2$-fractal
    $\uparrow C_3A_3B_3$-fractal
    $\downarrow C_3A_2B_1$-fractal
    $\uparrow C_1A_0B_1$-fractal
    $\downarrow C_1A_1B_3$-fractal
    $\uparrow C_0A_1B_2$-fractal
    $\downarrow C_0A_0B_0$-fractal
}
```

A similar algorithm can be found in [30] but for data locality at cache level. We use the same idea but for data locality at register level. The advantage of this decomposition might be explained intuitively. Two consecutive leaves, i.e. v and u, have a common matrix, if we have chosen an instruction scheduling which manipulates only a part of the common matrix at any time, at the end of leaf v some values are in registers. Leaf u can avoid to load again the common values if it performs its computation so that it can exactly access the last part of the common matrix exploiting temporal re-use. Otherwise even if the two leaves are accessing the same matrix it may be that are accessing different parts of it. The order of the computation of the two recursive algorithms permits to exploit such locality between any two consecutive leaves. Another difference from previous work is that, for performance purpose, we do not load equally sized sub matrixes, leaves are unbalanced, that is, a different number of scalar variables are used to replace different elements in different matrixes. Register file is not a cache and we can force a more opportune replace policy. Let (A, B, C) be a leaf which can be computed at register level. We can graphically describe the CAB-fractal as in Figure 5.8 and ABC-fractal algorithm as in Figure 5.7.

In the pictures we can see an high level description of the computation, where only the liveness of the variables are alighted. Suppose to have R scalar variables (registers) and we want to write the code which computes the multiplication, i.e. in Figure 5.7, applying scalar replacement. It follows a pseudo-implementation of a code generator. Note that matrix C has been replaced by $\frac{4R}{7}$ scalar variables, A has been read in four steps in $\frac{R}{7}$ scalar variables and B has been read in two steps in $\frac{2R}{7}$ scalar variables:

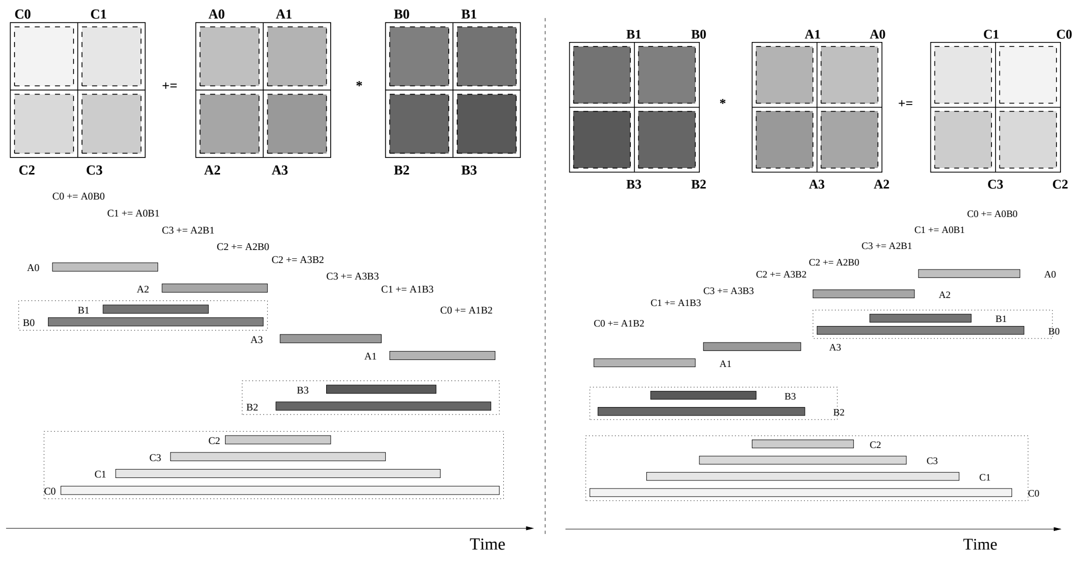

Figure 5.7: *ABC*-fractal algorithm in forward (left picture) and backward order (right picture).From the top to the bottom: the layout of the matrixes marked with different level of gray, the computation and the *liveness* of matrix elements during the computation.

Algorithm gen(A,B,C,order)

```
if (order==FORWARD)
   If it not loaded in the previous problem, Load C_0, C_1, C_2, C_3.
   If it not loaded in the previous problem, Load A_0.
   If it not loaded in the previous problem, Load B_0, B_1.
   Compute C_0+=A_0B_0.
   Compute C_1+=A_0B_1.
   Dis-allocate A_0 and Load A_2.
   Compute C_3+=A_2B_1.
   Compute C_2+=A_2B_0.
   Dis-allocate A_2 and Load A_3.
   Dis-allocate B_1 and B_0, Load B_2, B_3.
   Compute C_2+=A_3B_2.
   Compute C_3+=A_3B_3.
   Dis-allocate A_3 and Load A_1.
   Compute C_1+=A_1B_3.
   Compute C_0+=A_1B_2.
   If it will not be reuse in the next problem Write C_0, C_1, C_2, C_3,
      Dis-allocate C_0, C_1, C_2, C_3.
   If it will not be reuse in the next problem Dis-allocate B_2, B_3.
   If it will not be reuse in the next problem Dis-allocate A_1.
else
   If it not loaded in the previous problem, Load C_0, C_1, C_2, C_3.
   If it not loaded in the previous problem, Load A_1.
   If it not loaded in the previous problem, Load B_2, B_3.
   Compute C_0+=A_1B_2.
   Compute C_1+=A_1B_3.
   Dis-allocate A_1 and Load A_3.
   Compute C_3+=A_3B_3.
   Compute C_2+=A_3B_2.
   Dis-allocate A_3 and Load A_2.
   Dis-allocate B_2 and B_3, Load B_0, B_1.
```

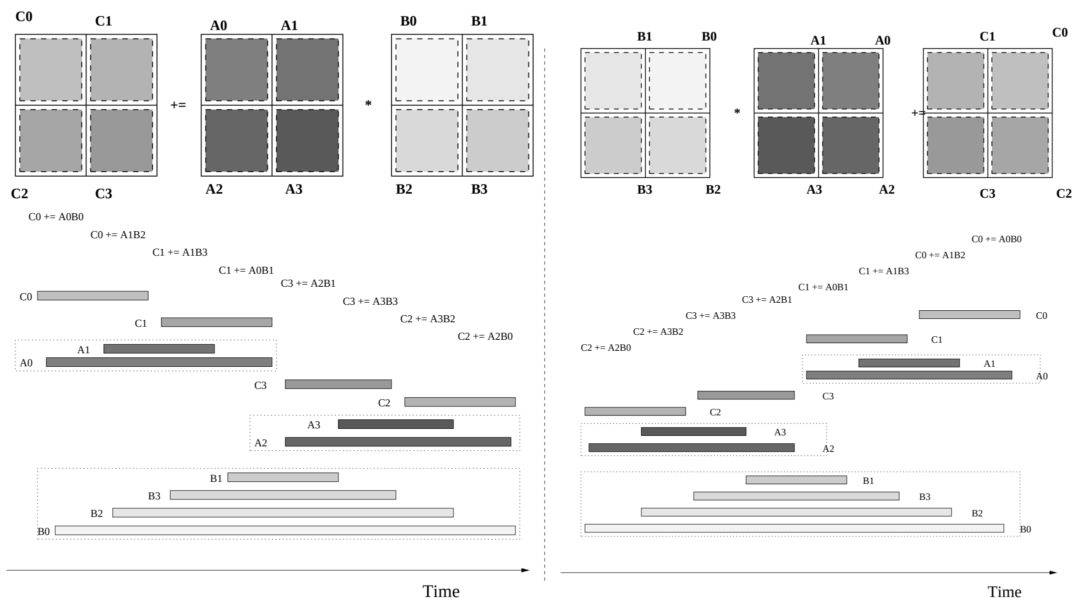

Figure 5.8: We can see the CAB-fractal algorithm in forward (left picture) and backward order (right picture). From the top to the bottom: the layout of the matrixes marked with different level of gray, the computation and the *liveness* of matrix elements during the computation.

```
Compute C_2+=A_2B_0.
Compute C_3+=A_2B_1.
Dis-allocate A_2 and Load A_0.
Compute C_1+=A_0B_1.
Compute C_0+=A_0B_0.
If it will not be reuse in the next problem Write C_0, C_1, C_2, C_3,
   Dis-allocate C_0, C_1, C_2, C_3.
If it will not be reused in the next problem Dis-allocate B_0, B_1.
If it will not be reused in the next problem Dis-allocate A_0.
```

♠

In general the order of the computation does respect an order that it is not related to the size of the problems we compute. For example the sub-problem $C_3+=A_3B_3$ may be computed before sub-problem $C_2+=A_3B_1$, the former may be smaller than the latter (i.e. (A_0, B_0, C_0) fits the number of scalar variables, then every problem does[3].). The latter sub-problem may require *to spill* scalar variables (as registers) and therefore it is necessary to define a policy. As we can see the code generator explicity allocates and

[3]W.l.o.g. consider two problems $< m_i, n_i, p_i >$ and $< m_{i+1}, n_{i+1}, p_{i+1} >$ so that $p_i m_i + \lceil \frac{m_i}{2} \rceil \lceil \frac{n_i}{2} \rceil + p_i \lceil \frac{n_i}{2} \rceil \leq R$ and $p_{i+1} m_{i+1} + \lceil \frac{m_{i+1}}{2} \rceil \lceil \frac{n_{i+1}}{2} \rceil + p_{i+1} \lceil \frac{n_{i+1}}{2} \rceil > R$ where R is the number of scalar variables. Note that only one parameter differs between two subproblems, because they have a common matrix, i.e. C, and therefore n changes by at most one. Any sub-problem of $< m_{i+1}, n_{i+1}, p_{i+1} >$ is characterized by $< m = \lceil \frac{m_{i+1}}{2} \rceil, n = \lceil \frac{n_{i+1}}{2} \rceil, p = \lceil \frac{p_{i+1}}{2} \rceil >$ and therefore uses $pm + \lceil \frac{m}{2} \rceil \lceil \frac{n}{2} \rceil + p \lceil \frac{n}{2} \rceil \leq p_i m_i + \lceil \frac{p_i}{2} \rceil \lceil \frac{n_i}{2} \rceil + m_i \lceil \frac{n_i}{2} \rceil \leq R$.

dis-allocates scalar variables to elements of the matrixes involved. The policy to up-date the scalar replacement is the *last loaded* policy, that is, the scalar variable loaded for last it is eligible to be used for another assignment, loosing the previous link between variable and matrix element. Indeed, the check at the beginning and at the end of the routine can be solved since the code generation is performed on leaf for which the size problem is known. This access pattern permits to decrease self interference misses, maximizing the cache-line effect and, for certain architecture it is possible to perform multiple loads and stores in a single instruction, exploiting spatial locality at register file.

We implemented the following two approaches.

- ABC-fractal algorithm is applied at leaves level, that is at `gen(A,B,C,order)` (see Figure 5.7) where the whole matrix C is stored in scalar variables; the leaves are called by CAB-fractal order, so C utilization is maximized, reducing number of loads and, more important, writes.

- CAB-fractal is implemented at leaves level (B is stored completely in scalar variables, Figure 5.8) and the leaves are organized as BAC-fractal.

Any fractal algorithm is not unique, it can be implemented differently in function of the number of scalar variables and therefore registers. If we use a fixed implementation with R scalar variables (i.e. `gen(A,B,C,order)` may require $R = 28$ scalar variables) and, for example, a problem can be solved with $R + 1$, then we decompose the problem and we will re-use only one fourth of the available registers. We cannot change the decomposition factor, but we can decide how many scalar variables are required at the leaves in a more flexible way. We implemented three different leaf algorithms to solve a problem of size $< m, n, p >$. They may use a different number of scalar variables, and they are all variations of `gen(A,B,C,order)` using the following number of scalar variables:

1. $R = mp + n\lceil\frac{n}{2}\rceil + p\lceil\frac{n}{2}\rceil$, i.e. $R = 32$.

2. $R = mp + \lceil\frac{m}{2}\rceil\lceil\frac{p}{2}\rceil + p\lceil\frac{n}{2}\rceil$, i.e. $R = 28$.

3. $R = \lceil\frac{m}{2}\rceil\lceil\frac{p}{2}\rceil + \lceil\frac{m}{2}\rceil\lceil\frac{n}{2}\rceil + \lceil\frac{n}{2}\rceil\lceil\frac{p}{2}\rceil$, i.e. $R = 12$.

Of course, we could introduce more cases, but for practical purpose we have chosen only these three cases. This permits a smoother decomposition of the problem with the idea

32 scalar vars	CAB-fractal	BAC-fractal	Tiling
$< 4,4,4 >$	1	1	1
$< 5,5,5 >$	1.22	1.07	0.80
$< 6,6,6 >$	1.00	0.87	0.79
$< 7,7,7 >$	0.79	0.96	0.93
$< 8,8,8 >$	0.70	0.84	0.74
$< 9,9,9 >$	1.20	0.97	0.66
$< 10,10,10 >$	1.12	1.00	0.76
$< 11,11,11 >$	1.02	0.90	0.80
$< 12,12,12 >$	0.94	0.83	0.73
$< 13,13,13 >$	0.81	0.98	0.76
$< 14,14,14 >$	0.75	0.93	1.16
$< 15,15,15 >$	0.70	0.87	1.15
$< 16,16,16 >$	0.66	0.82	1.14
$< 32,32,32 >$	0.66	0.82	N/A

Table 5.2: Theoretical performance: ratio of loads and stores over the number of basic operations $c+ = a * b$.

that the values in scalar variables will be reused across sub-problem, even if the local problem might not be optimally solved.

This approach is automated and we can investigate the solution space and achieve an estimation of performance by counting the number of loads and stores with respect of the number of basic operations $c+ = a * b$. This measure offers a partial, but indicative, estimation of performance. We show in the following Table 5.2 a comparison among three different algorithms for register allocation. With *Tiling* we indicate an approach based on the most common tiling technique used on caches but with an exhaustive search (See page 77). Table 5.2 represents a architecture with a 32 registers file.

This technique is an elegant extension of the fractal approach to the register file and, in fact, it is the implementation of the same idea at every level of the memory hierarchy. This approach is a general approach to data locality. The major disadvantage is the explosion in size of the code. For small problems and for big enough instruction cache the technique offers good performance but code explosion is a characteristic that must be considered very carefully.

ATLAS Register Allocation

An interesting alternative to the fractal approach for loop code generation is found in the ATLAS [64] package. The register allocation does not follow a divide and conquer technique but it follows an approach based on a three-loops algorithm: the basic routine is `ATL_dNBmm()` and we studied the implementation for Ultra Sparc 2 with 32 double floating point registers.

Algorithm let $C_{(i,j)}$ be a 4×4 matrix, let $A_{(i,k)}$ be a 4×2 matrix and let $B^t_{(k,j)}$ be a 4×2 matrix and they are sub-matrixes respectively of C, A and B^t ($N_B \times N_B$ matrixes)

```
for i=0 to NB/4
  for j=0 to NB/4 {
    load C(i,j);
    load A(i,0);
    load B(0,j);
    for k=1 to NB-1 step 2 {
      C(i,j) += A(i,k)*B(k,j);
      load A(i,k);
      load B(k,j);
    }
    C(i,j) += A(i,k)*B(k,j);
    Store C(i,j);
  }
```

The number of accesses to the memory are $M_{\text{on-chip}} = (8N_B + 16 + 16)\frac{N_B}{4}\frac{N_B}{4}$. If we are computing the matrix multiplication $< N, N, N >$ and N is a multiple of N_B, the ratio memory accesses over number of basic operations is:

$$R = \frac{(\frac{N}{N_B})^3 M_{\text{on-chip}}}{N^3} = \frac{1}{2} + \frac{2}{N_B}$$

For this architecture $N_B = 44$ and therefore $R_0 \sim 0.55$[4] ♠

[4] We did not count performed accesses to copy the sub-matrixes so that the matrixes fit the second level of cache.

The advantages are spelled out, the code is simple, it is small in size, the number of loads and stores are half of the number of the basic instructions. We generalize the approach so that the number of available registers is a parameter, and also the size of the leaf is a parameter as well. If we generalize the approach, we require $O(r^2 + r)$ registers. For a

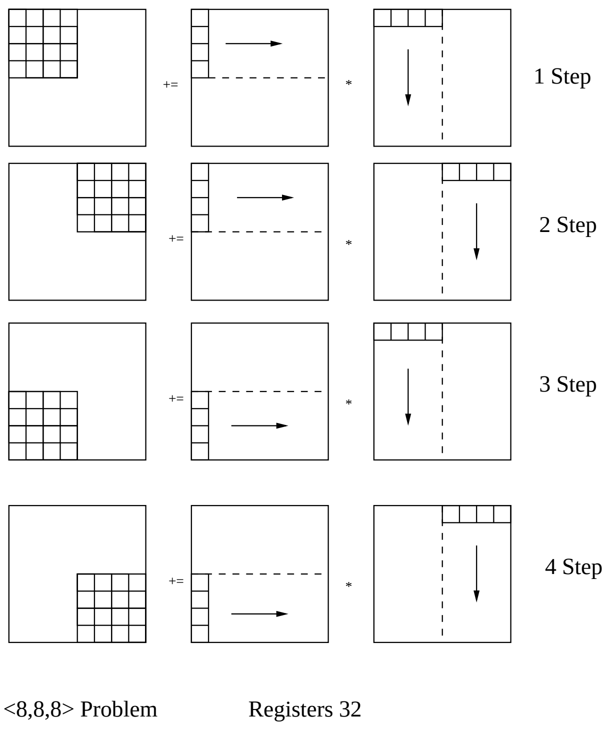

Figure 5.9: Register allocation for power of two register files and general near square matrix multiplication

problem of size $< K, K, K >$ the ratio is $R = \frac{2}{r} + \frac{2}{K}$. A trivial generalization was devised for near square matrix multiplication (Figure 5.9). We developed a code generator which given the problem sizes $< M, N, P >$ and the number of registers, generates the necessary C code to solve the problem using this approach. This implies a change of strategy from the general fractal approach to evaluate the leaves in the recursion tree. It also implies a different lay out of the data. Indeed, the computations on the leaves involve sub-matrixes stored in row-major (column-major). To decouple layout and computation we stop the recursion when the size of the problem is between K and $2K$ and the fractal decomposition of the matrix is stopped between L and $2L$ with $2L = K$[5]. This is a good example for further study, because this three loop routine can be achieved from the ijk-loop by the application of code optimizations such as tiling, loop-unrolling, scalar replacement and software pipeline. This routine is a very good example for two basic performance issues: minimization of traffic from/to the cache and latency hiding. The last issues is really interesting, indeed, we can see how the loads performed in one iteration are used only at the successive one.

Parametric Tiling

Tiling for register file is a subject rarely discussed and indeed, there is always a not very detailed description. But tiling for caches it is always well explained. In this Section we apply the tiling technique commonly published for caches and we apply to register file. We will see how the problem can be formalized and how the solution comes naturally from the definition of the problem. Unfortunately, we introduce new notations.

Definition 5.6 *we indicate with $\mathcal{P}_0(M, N, P)$ a matrix-matrix multiplication $< M, N, P >$ so that matrix A is loaded at register files by $k_1 > 0$ row(s) at once, then the number of accesses to load and write registers is $f_0(k_1) = 2PM + MN + \lceil \frac{M}{k_1} \rceil NP - (\lceil \frac{M}{k_1} \rceil - 1)k_2 k_3$.*

$\mathcal{P}_0(M, N, P)$ reads and writes C once, it reads A once and it reads B $\frac{M}{k_1}$ times.

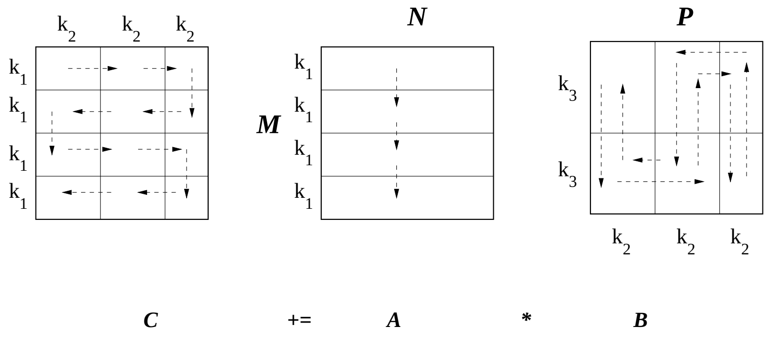

Figure 5.10: A parametric tiling of matrixes C, A and B, the parameters are k_1, k_2 and k_3. In this picture is depicted out the solution when matrix A is split by rows, a similar tiling techniques can be applied when B is split by columns. The arrows indicate the computation order.

[5]The current implementation leaves the lay-out of the leaves in row-major order, only for power of two leaves this implementation is correct. This can be seen as a flaw but our goal was to prove that register allocation is key for performance

Definition 5.7 *we indicate with $\mathcal{P}_1$ a matrix-matrix multiplication $< M, N, P >$ so that matrix B is loaded at register files by $k_2 > 0$ column(s) at once, then the number of accesses to load and write registers is $f_1(k_2) = 2PM + PN + \lceil \frac{P}{k_2} \rceil NM - (\lceil \frac{P}{k_2} \rceil - 1)k_1k_3$.*

$\mathcal{P}_1(M, N, P)$ reads and writes C once, it reads B once and it reads A $\frac{P}{k_1}$ times. Any architecture has its own characteristic parameters such as number of registers, $\mathcal{R}$, number of stages of the functional point unit (FU), $\mathcal{Z}$, and instructions set. We assume the single multiply-add instruction, i.e. *madd*, is available.

Suppose we have $k_1 * k_2$ independent operations, the throughput of a FU for $k_1 * k_2$ independent operations can be summarized as follows

Definition 5.8 **if** $\mathcal{Z} \leq k_1k_2$, **then** $\tau_0(k_1, k_2) = \frac{\mathcal{Z} - k_1k_2}{k_1k_2} + 1$
else $\tau_0(k_1, k_2) = 1$.

If problem $< M, N, P >$ is decomposed so that all the floating point operations are gathered in groups of k_1k_2 independent operations the following definition determines the number of cycles taken to compute the MNP flops.

Definition 5.9 $f_2(k_1, k_2) = \tau_0(k_1, k_2)MNP$

Our goal is to find a scalar replacement so that we can solve the following problem.

$$\boxed{\begin{array}{l} \mathcal{F} = \min_{i=[0,1]} \min_{k_1,k_2,k_3} f_i(k_1, k_2, k_3) + f_2(k_1, k_2) \\ k_1, k_2, k_3 \text{ are positive integers.} \\ k_1k_2 + k_1N + k_2k_3 \leq \mathcal{R}. \text{ or} \\ k_1k_2 + k_2N + k_1k_3 \leq \mathcal{R} \end{array}}$$

In Figure 5.10 we can see that k_1k_2 is the block size in C, k_1N is the *slice* of A and k_2k_3 is the block size of B.

Consider again $f_2(k_1, k_2)$ as in Figure 5.11: $f_2()$ should be considered as the contribution of four parts, $f_2(k_1, k_2) = \sum_{i=0}^{3} f_{2i}(k_1, k_2)$

- $f_{20}(k_1, k2) = \tau(k_1, k_2)(M - M\mathbf{mod}k_1)(P - P\mathbf{mod}k_2)N$,
- $f_{21}(k_1, k2) = \tau(k_1, P\mathbf{mod}k_2)(M - M\mathbf{mod}k_1)(P\mathbf{mod}k_2)N$,
- $f_{22}(k_1, k2) = \tau(M\mathbf{mod}k_1, k_2)(M\mathbf{mod}k_1)(P - P\mathbf{mod}k_2)N$,
- $f_{23}(k_1, k2) = \tau(M\mathbf{mod}k_1, P\mathbf{mod}k_2)(M\mathbf{mod}k_1)(P\mathbf{mod}k_2)N$

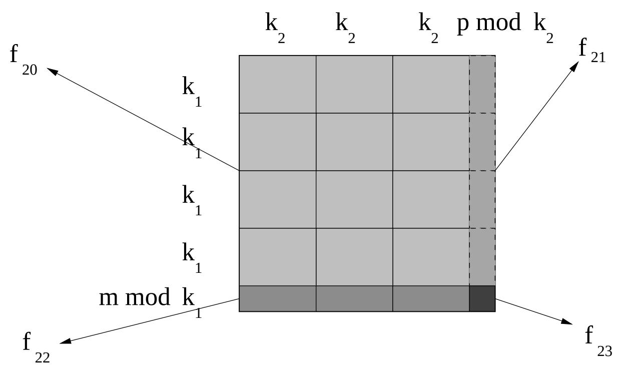

Figure 5.11: Matrix C tiling with (k_1, k_2) tile and its effect on the computation.

Suppose that $P = M = N = 8$, $\mathcal{Z} = 4$ and $\mathcal{R} = 32$ and to understand the range of k_1 let us assign to $k_2 = k_3 = 1$, we can see that $k_1 \leq 3$ (from $k_1k_2 + k_1N + k_2k_3 \leq \mathcal{R} \rightarrow k_1 + 8k_1 + 1 \leq 32$).

- $k_1 = 1$. There is minimum and it can be achieved by the triple $(k_1 = 1, k_2 = 4, k_3 = 5)$ and its value is $\mathcal{F}_0 = 19 * 8 * 8$.
- $k_1 = 2$. There is minimum by the triple $(k_1 = 2, k_2 = 2, k_3 = 5)$ and its value is $\mathcal{F}_0 = 15 * 8 * 8$.
- $k_1 = 3$. There is minimum by the triple $(k_1 = 3, k_2 = 2, k_3 = 1)$ and its value is $\mathcal{F}_0 = (11 + \lceil \frac{8}{3} \rceil) * 8 * 8$.

Let us interpreter the solution suggested by the triple $(k_1 = 3, k_2 = 2, k_3 = 1)$. Matrix A is split in three horizontal parts: A_0, A_1 and A_2. A_0 and A_1 are 3×8 matrixes and A_2 is a 2×8 matrix. Matrix B is read three times and it is split by two column item and each item is split row by row. For each block of C there are twelve *madd*s and six different destinations. The ratio of memory accesses over number of computations is $\frac{f_1(k_1)}{f_2(k_1,k_2)} = \frac{3+\lceil 8/3 \rceil}{8} = 6/8 = 0.75$. For the same size of the problem $< 8, 8, 8 >$ we know that this is not an optimal solution (Table 5.2), because this approach is not able to exploit reuse in subproblems but when the problem is small enough it works fine.

5.3.4 Unfolding: notations and considerations

Unfolding of recursive algorithm is a common problem/technique a programmer has to deal with when performance of a recursive algorithm is wanted. Unfolding does not imply to remove completely the recursion, but it may involve only a sub-computation of a recursive algorithms. A recursive matrix multiplication algorithm is our case study: we present a formal description of our work and a possible generalization to similar problems.

A common miss understanding is that recursive algorithms are slower than non recursive ones, because at any recursive call it is popped/pushed data in the stack, i.e. formal parameters are allocated in the stack.

- Compilers are becoming smart enough that if it is possible, formal parameters are allocated in registers, especially if these parameters are constant, and therefore avoiding any stack allocation.

- The stack utilization can be very useful for performance purpose, because it may avoid re-computation.

- The height of the call tree in a recursive algorithms can be customized reducing the common overhead due to the calling convention.

A common pitfall when recursive algorithms are optimized is to remove the recursion completely and reducing the work space, trying to avoid the emulation of the stack. For example, consider a recursive algorithm $\mathcal{A}$, it may be we can rewrite algorithm $\mathcal{A}$ removing the recursion without emulating the stack and therefore reducing the working space of the routine but without writing *another algorithm*, we identify such algorithm with $\mathcal{W}(\mathcal{A})$. We wrote several algorithms $\mathcal{A}_i$ for blocked matrix multiply and we removed systematically the recursion, $\mathcal{W}(\mathcal{A}_i)$. These algorithms are interesting because we can reduce the work space to only a few registers. Consider $\mathcal{A}_{2^n}$ the fractal algorithm for power of two matrixes, we developed $\mathcal{W}(\mathcal{A}_{2^n})$. $\mathcal{W}(\mathcal{A}_{2^n})$ has an extraordinary regularity and it has no need of stack at all, to perform the computation the routine needs only four integer registers and three of them are for index computation. $\mathcal{W}(\mathcal{A}_{2^n})$ requires bit-wise-operations that usually are expensive operations in a RISC machine, but nonetheless $\mathcal{W}(\mathcal{A}_{2^n})$ is slightly faster than $\mathcal{A}_{2^n}$. The same approach can be applied to the fractal algorithm for square matrix multiply, $\mathcal{A}_n$. $\mathcal{W}(\mathcal{A}_n)$ requires 6 integer registers, 4 registers are for index computation, the other two are used to emulate the *code stack*, that is, as reminder which sub problem has been computed and which has not. Surprisingly, $\mathcal{W}(\mathcal{A}_n)$ is slower than $\mathcal{A}_n$. Indeed, the non recursive algorithm has to recompute values that the recursive one has only to read from the stack. In this application re-evaluation is more time expensive than load. Quantitatively we can explain this case with an example extracted from the fractal algorithm: suppose we have to compute three indexes i, j and k, an up-date of any index takes, in average, three instructions, a total of 9 instructions. On a SPARC there are two ALUs and therefore the nine instructions can be scheduled so that they can take no less than 5 cycles. On a SPARC multiple loads are allowed and caches and memory are pipelined, for common application the worst case scenario the stack can be stored in the second level of cache, which has latency of 3 cycles. Therefore a latency of 5 cycles is required to load the three indexes from the second level of cache. But current compilers schedule load instructions so that latency is partially hidden and, even so, best scenario is a latency of 1 cycle for each load. For the same computation, index computation, the two approaches can offer different performance and, in fact, for SPARC architecture we discovered that it is *faster* read than recompute.

Leaves Unfolding

We indicate as $\mathcal{X}$ the input set for an algorithm. Given $\mathcal{X}$ with $|\mathcal{X}| = k$, the call tree of $\mathcal{A}(\mathcal{X})$ is fixed, therefore the computations and the order of the computations of $\mathcal{A}(\mathcal{X})$ are known. Given $\mathcal{X}$, unfolding $\mathcal{A}(\mathcal{X})$ is writing the sequence of instructions of $\mathcal{A}$ on $\mathcal{X}$. Then, compilers can perform the common optimizations on the *unfolded code*. We can substitute every call to $\mathcal{A}(\mathcal{X})$ s.t. $|\mathcal{X}| = k$ with a call to an unfolded and optimized routine. We call *leaves unfolding* the technique to choose a set of leaves, where the size of the problem is known, and substitute the recursive call with an unfolded and optimized routine. The advantage is twofold: we can reduce the height of the recursion and we can easily apply optimizations, i.e. register allocation.

Leaves unfolding for any recursive algorithm $\mathcal{A}$ can be performed by two distinct steps.

1. We need a set of inputs, or subproblems, $\mathcal{M} = \{\mathcal{X}_i\}$ so that every decomposition of $\mathcal{X}$ by $\mathcal{A}$ reaches a element in $\mathcal{M}$ (we called *kernel dictionary*).

2. For each element in $\mathcal{M}$, we unfold and optimize the code but we need to define a order of computation, based on a sort of optimality.

This approach is feasible only if the size of the kernel dictionary, $\mathcal{M}$, is constant and small (i.e. for blocked matrix multiplication we can claim that the $|\mathcal{M}|$ is a small constant.

In other words, the set $\mathcal{M}$ should be independent from the size of the problem. For a divide and conquer algorithm, which by definition, decomposes the problem in smaller subproblems, the way to split the problem is in-coded in the algorithm itself and therefore fixed. The definition of the algorithm can give the required information about the size of $\mathcal{M}$ and therefore its feasibility. It is not clear how to determine this set yet.

The call tree of a recursive algorithm can be seen as a Convex Decomposition Tree, CDT, (any computation without re-computation of the node determines a set which is convex). Actually the implementation of a CDT is the result of a proper decomposition of the computation so that a sort of optimality, number of inputs vs size of the computation, is achieved. In this particular case we start already with a CDT based on the definition of the algorithm and what we can do is *to rewrite* the order of the computation. We devised an heuristic technique that we called *fractal scalar replacement.* This technique is a rewriting technique, which tunes the calling order in the recursive algorithms and rewrite the *particular case*, or final case. Let us write the matrix multiply algorithm with a fancier, and more obscure, notation:

```
Algorithm $\mathbf{F}(\mathcal{X}, l, w, ord)$
if $(\mathcal{B}(\mathcal{X}, R) = \mathbf{true})$ then $\mathcal{P}_c(\mathcal{X}, R, l, w, ord))$;
else {
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(0,ord)}), l, \mathcal{W}(\mathcal{X}, \pi_{g(0,ord)}, \pi_{g(1,ord)}), h(0, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(1,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(0,ord)}, \pi_{g(1,ord)}), \mathcal{W}(\mathcal{X}, \pi_{g(1,ord)}, \pi_{g(2,ord)}), h(1, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(2,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(1,ord)}, \pi_{g(2,ord)}), \mathcal{W}(\mathcal{X}, \pi_{g(2,ord)}, \pi_{g(3,ord)}), h(2, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(3,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(2,ord)}, \pi_{g(3,ord)}), \mathcal{W}(\mathcal{X}, \pi_{g(3,ord)}, \pi_{g(4,ord)}), h(3, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(4,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(3,ord)}, \pi_{g(4,ord)}), \mathcal{W}(\mathcal{X}, \pi_{g(4,ord)}, \pi_{g(5,ord)}), h(4, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(5,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(4,ord)}, \pi_{g(5,ord)}), \mathcal{W}(\mathcal{X}, \pi_{g(5,ord)}, \pi_{g(6,ord)}), h(5, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(6,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(5,ord)}, \pi_{g(6,ord)}), \mathcal{W}(\mathcal{X}, \pi_{g(6,ord)}, \pi_{g(7,ord)}), h(6, ord))$;
$\mathbf{F}(L_z(\mathcal{X}, \pi_{g(7,ord)}), \mathcal{L}(\mathcal{X}, \pi_{g(6,ord)}, \pi_{g(7,ord)}), w, h(7, ord))$;
}
```

where

- $\pi = \{i : i \in [0, 7]\}$ is a set of integers which describes the order of the decomposition of the recursive algorithm (as it is specified, let us consider the sequence $0, 1, 2, 3, 4, 5, 6, 7$ associated with the algorithm $C_0 += A_0B_0$, $C_0 += A_1B_2$, $C_1 += A_0B_1$, $C_1 += A_1B_3$, $C_2 += A_2B_0$, $C_2 += A_3B_2$, $C_3 += A_2B_1$, $C_3 += A_3B_3$) therefore with $\pi_i = i$.

- $g : \pi \times \rho \to \pi$ represents a parametric permutation.

- $L_z : \mathcal{X} \times \pi \rightarrow \mathcal{X}$ is the Z-Morton decomposition of the subproblem with input $\mathcal{X}$ for the π-th case.
- $\mathcal{L} : \mathcal{X} \times \pi \times \pi \rightarrow \mathcal{X}$ is the common data between two subproblems, data may be not loaded twice.
- $\mathcal{W} : \mathcal{X} \times \pi \times \pi \rightarrow \mathcal{X}$ is the common data between two subproblems, data may be not written twice.
- $h : \pi \times \rho \rightarrow \rho$ identifies the order of the recursive calls at the same level,$\rho = \{0, 1\}$.
- $\mathcal{B} : \mathcal{X} \times R \rightarrow \{\mathbf{true}, \mathbf{false}\}$ this is the test to halt the recursion and apply a particular case.
- $\mathcal{P}_c(\mathcal{X}, R, l \in \mathcal{X}, w \in \mathcal{X}, ord \in \rho)$ case when recursion stops.
- R is the number of registers.

♠

A possible approach is to generate all possible solutions and then choose, even if the approach is exponential, we are working with a *fixed* and small problem. Instead, we have chosen the following bottom-up policy: we devised by hand for very small size of the problem different algorithms (i.e. we developed different solutions for problem of size up to $< 4, 4, 4 >$). We indicate them with $\mathcal{P}_c^i$. We choose one at time, $\mathcal{P}_c^i$, in $\mathbf{F}$ and we infer $g()$ and $h()$, simplifying further the structure of the functions. A quantitative measure of optimality must be given so that the order of the recursive calls and the implementation of $\mathcal{P}_c$ can be guided. Consider a node u as in Figure 5.12 in the call tree, if we define $f(u)$

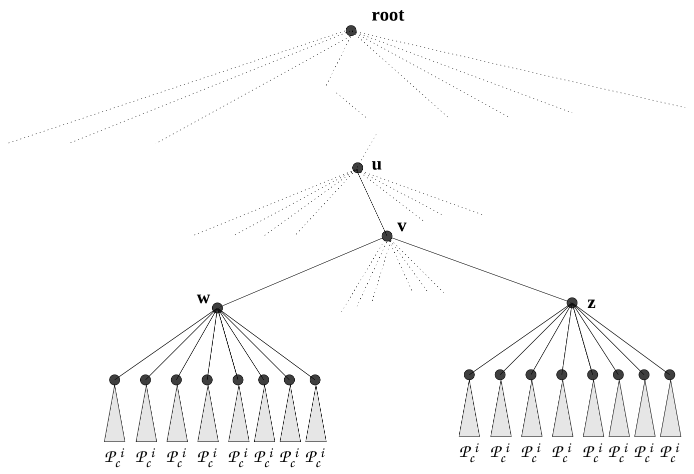
root
u
v
w
z
$\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$
$\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$ $\mathcal{P}_c^i$

Figure 5.12: Excerpt from a call tree

as $|l| + \sum_{i=1}^{7} |\mathcal{L}(\mathcal{X}, \pi_{g(i-1,ord)}, \pi_{g(i,ord)})| + |w| + \sum_{i=0}^{6} |\mathcal{W}(\mathcal{X}, \pi_{g(i,ord)}, \pi_{g(i+1,ord)})|$. f offers a measure of the loads and stores we may avoid exploiting locality at node u. In f, as it is defined, there is no indication how the choice of $\mathcal{P}_c^i$ affects the overall computation, but in practice it does. Indeed, how we implement $\mathcal{P}_c^i$ determine the number of memoty accesses for sub-problem and determine quantitatively the number of matrix elements re-used among sub-problems.

We devised two algorithms, $\mathcal{P}_c^1$ and $\mathcal{P}_c^2$. $\mathcal{P}_c^1$ allocates more space to matrix C and schedules instructions following the CAB-fractal approach. $\mathcal{P}_c^2$ allocates more space to matrix B and schedules instructions following the BAC-fractal approach. For both we have found

$h(x,0) = h(x,1)$ **if** x is even, $h(x,*) = 0$
else $h(x,*) = 1$

$h()$ is a very simple function which describes when a sub problem should follow a forward or backward order to call subproblems. Indeed, the function $g()$ has the following property: $g(x,1) = g(7-x,0)$. So if we use $\mathcal{P}_c^1$ for leaves computation then $g([0,1,2,3,4,5,6,7],0) = [0,1,3,2,6,7,5,4]$ otherwise if we use $\mathcal{P}_c^2$ then $g([0,1,2,3,4,5,6,7],0) = [0,4,6,2,3,7,5,1]$. This heuristic is based on the assumption that if $\mathcal{P}_c^i$ is optimal for all the leaves, a proper reordering of the computations offers a optimal solution.

Optimality

In this section we claim that a divide and conquer technique for register allocation is not optimal. We show that the lower bound of memory accesses is larger than the upper bound of memory accesses obtained from the technique proposed in ATLAS package, by a constant factor.

Lemma 5.1 *In any fractal decomposition there are at most four consecutive sub problems that have in common the same sub matrix.*

Proof: by contradiction. Suppose there are five, or more, consecutive subproblems, i.e. $< k,k,k >$ (v_0, v_1, v_2, v_3, v_4). There should be three consecutive problems of double size, i.e. $< 2k,2k,2k >$ (u_0, u_1 and u_2) and therefore two consecutive problems, i.e. of size $< 4k,4k,4k >$ (t_0 and t_1). But if we look at the computation as a tree, then four sub problems should be at the border of the trees associated with two $< 2k,2k,2k >$ problems, indeed, they should be consecutive. This makes not feasible any schedule so that five sub consecutive problems can share the same matrix, as Figure 5.13 shows. □

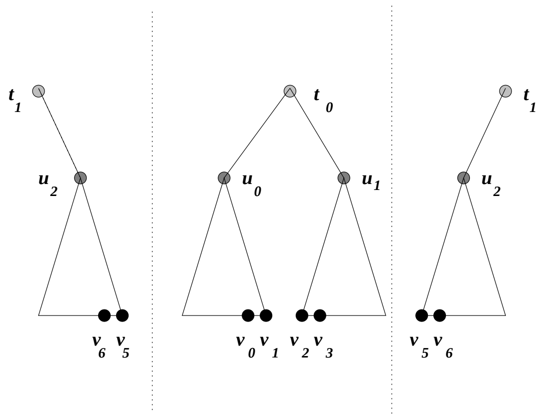

Figure 5.13: Example of possible schedule for five sub problems of equal size.

Lemma 5.2 *Let u_0, u_1, u_2 and u_3 be four consecutive sub problems with a common matrix, and let x_0 be the problems that precede u_i problems (w.l.o.g. x_0, u_0, u_1, u_2, u_3) then x_0 has in common with u_0 a sub matrix which can be common only to the previous sub problem of x_0.*

Proof: if there is other two sub problems y_0 and y_1 so that y_0, y_1, and x_0 are consecutive sub problems with common sub matrix, by construction and from the proof of Lemma 5.1, they have to belong to the same problem and same tree, and it is not possible. □

The same consideration can be applied to the first sub problem that follows a sequence of consecutive sub problems. Maximum advantage of this data locality can be achieved if the common matrix is C, that is, the data that must be read and written. Of course, the effective number of elements exchanged between two consecutive sub problems depends on the number of registers where these elements are stored.

Lemma 5.3 *Four consecutive sub problems of size $< k, k, k >$, the minimum number of memory accesses is $7k^2$.*

Proof: Let us call the four sub problems u_0, u_1, u_2 and u_3, they *share* matrix C. It may be that u_0 have common element from previous computation. u_0 has to read at least two operands, one of them is C. u_1 and u_2 have to read two operands as well and u_3 has to read two operand and write C, then we need $(2+2+2+3)k^2$ memory accesses. $\square$

A problem of size $< k, k, k >$ can be solved with $4k^2$ memory accesses with a minimum number of registers as $R = k^2+k+1$. This is possible only loading in registers the elements of one matrix, with the approximation that $k \sim \sqrt{R}$. But consecutive sub problems try to have in common not the same matrix requiring to move elements in registers and therefore increasing the number of memory accesses for the single problem, up to $5k^2 - k$, for single problem. Unfortunately using $R = k^2 + k + 1$, the minimum number of accesses for each sub problem is $O(3k^2)$ which gives a total number of memory accesses of $\frac{3n^3}{k}$, substituting $k \leq \lfloor\sqrt{R}\rfloor$ we can find that the ratio of memory accesses over the number of operation is $Ratio_H \geq \frac{3}{\lfloor\sqrt{R}\rfloor}$.

Theorem 5.1 *A fractal solution of a problem* $< n, n, n >$ *with any subproblem of size* $< k, k, k >$ *solved optimally by itself with* $4k^2$ *memory accesses, requires at least* $\frac{19n^3}{8k} + k^2$ *memory accesses.*

Proof: Any problem of size $< k, k, k >$ must read at least all three operands and write the partial result, therefore it needs $4k^2$ memory accesses (no less), independently from the number of registers. Suppose for sake of explanation that problem $< n, n, n >$ is evenly divisible in subproblem of size $< k, k, k >$, and we can see the solution of $< n, n, n >$ as a sequence of $< k, k, k >$. From Lemma 5.1, 5.2 and 5.3 we can write the following equation for the number of memory accesses: $5\frac{n^3}{8k^3}2k^2 + 3\frac{n^3}{8k^3}3k^2 + k^2 = \frac{19n^3}{8k} + k^2$. $\square$

We proposed an approach which can exploit locality for matrix C so that $R = (1 + 1/2 + \gamma)k^2$ where $\gamma < 1/2$. We do not consider the side effect on the number of accesses for a single sub problem. $Q(R) = \frac{19}{8\lfloor(\sqrt{\lfloor\frac{2R}{3+2\gamma}\rfloor})\rfloor}n^3 + \lfloor(\sqrt{\lfloor\frac{2R}{3+2\gamma}\rfloor})\rfloor$ and therefore $Ratio_{\text{fractal}} \geq \frac{19}{8\lfloor(\sqrt{\lfloor\frac{2R}{3+2\gamma}\rfloor})\rfloor}$. We can see that this bound is tighter than the previous one ($Ratio_H$). For the common case when there are 32 registers, and $n \to \infty$ $Ratio_{\text{fractal}}$ is 0.59 (the implementation achieves only a 0.66). In the ATLAS package we can find that $Ratio_{\text{ATLAS}} = \frac{2}{\lfloor\sqrt{1+R}-1\rfloor} + \frac{2}{n}$. For $R = 32$ and $n \to \infty$ $Ratio_{\text{ATLAS}}$ is 0.5, and this is not a lower bound but an upper bound. Since the functions we have found so far are complex functions, it is better to be not confident of the *impression* and, we figure out the behavior for cases taken from real architectures and problems. In the following we will

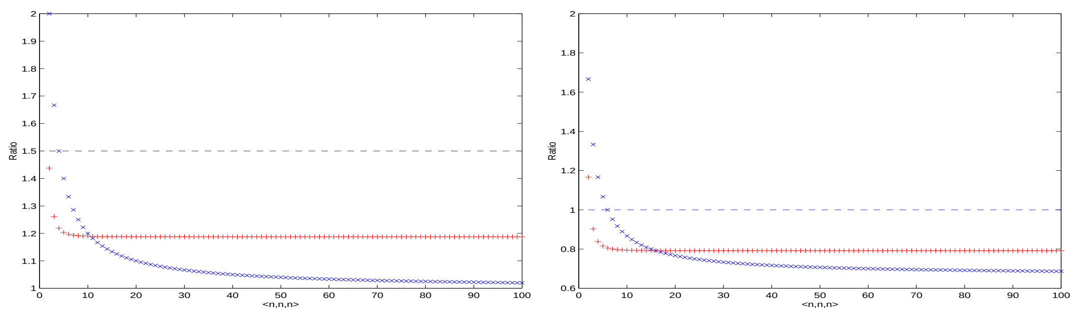

Figure 5.14: On the left the ratio when the number of registers is 8 and on the right when is 16. With $+$ is depicted $Ratio_{\mathrm{fractal}}$, with $\times$ $Ratio_{\mathrm{ATLAS}}$ and with dashed line $Ratio_H$

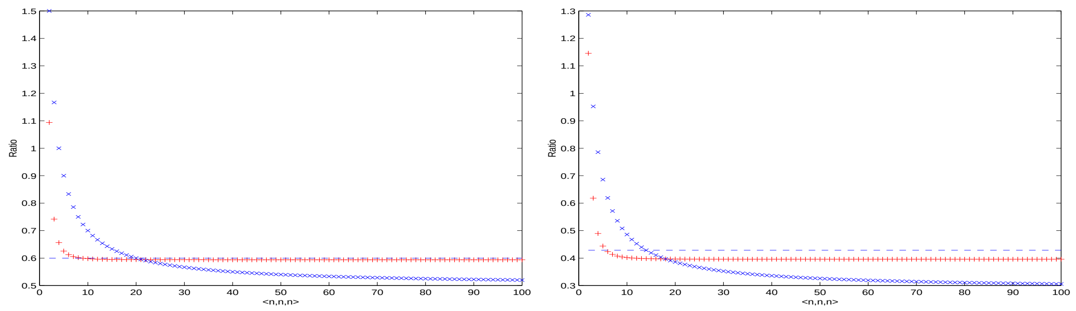

Figure 5.15: On the left the ratio when the number of registers is 32 and on the right when is 64. With $+$ is depicted $Ratio_{\mathrm{fractal}}$, with $\times$ $Ratio_{\mathrm{ATLAS}}$ and with dashed line $Ratio_H$

show the Ratio when the number of registers is fixed (Figure 5.14 and 5.15) and when the problem is fixed and we can vary the number of registers (Figure 5.16 and 5.17). With plus ($+$) we indicate the $Ratio_{\mathrm{fractal}}$, with $\times$-mark ($\times$) we indicate the $Ratio_{\mathrm{ATLAS}}$ and with dashed line $Ratio_H$: If we want to summarize the results, we can say that when the problem is big enough the approach proposed in the ATLAS package is the best, and this is true whatever is the fractal order we choose. If we consider problems with sizes within $< 4, 4, 4 >$ and $< 8, 8, 8 >$, the fractal approach is optimal and in particular we discovered that it may be we can obtain performance improvement as suggested by the diagrams of $Ratio_H$, but it is not optimal in general.

5.4 Experimental Results

When we collected the experimental results for our matrix multiplication on different architectures and we compared them with the experimental results obtained by the ATLAS

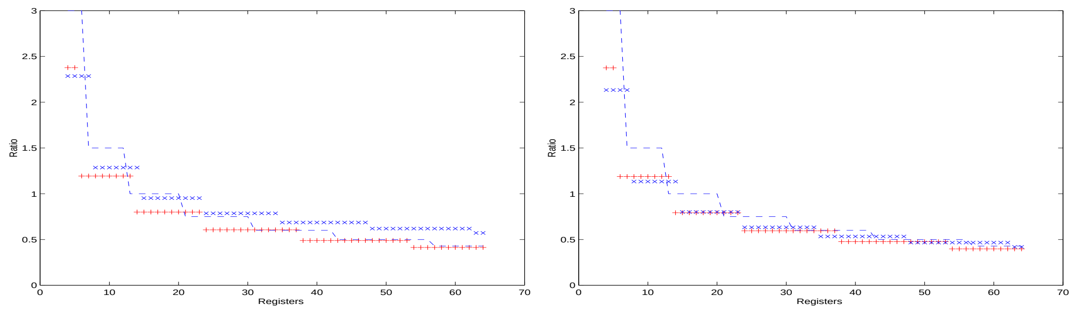

Figure 5.16: On the left the ratio when the problem has size $< 8, 8, 8 >$ and on the right when is $< 16, 16, 16 >$. With $+$ is depicted $Ratio_{\text{fractal}}$, with $\times$ $Ratio_{\text{ATLAS}}$ and with dashed line $Ratio_H$

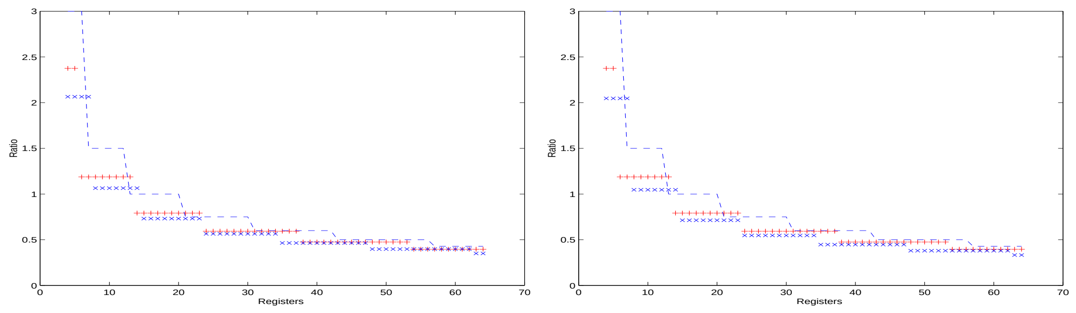

Figure 5.17: On the left the ratio when the problem has size $< 32, 32, 32 >$ and on the right when is $< 44, 44, 44 >$. With $+$ is depicted $Ratio_{\text{fractal}}$, with $\times$ $Ratio_{\text{ATLAS}}$ and with dashed line $Ratio_H$. The size $< 44, 44, 44 >$ has been chosen because is the basic case in ATLAS for SPARC ULTRA architectures.

([64]) package, we have found how much register allocation plays a basic role. In Figure 5.18 we tested the performance when the leaves are implemented using the fractal approach. The sizes of the leaves are among $< 8, 8, 8 >$ and $< 4, 4, 4 >$. The peak performance are good for R5000 IP32 180MHz, comparing with ATLAS, but we cannot achieve comparable performance on SPARC architectures. So we tested the performance when the sizes of the leaves are among $< 32, 32, 32 >$ and $< 16, 16, 16 >$ and we implemented the scalar replacement suggested by ATLAS, Figure 5.19 and 5.20. Each test measure the execution time of a single iteration of matrix multiplication.

As we can see the performances in Figure 5.19 and 5.20 are now comparable, they differ for $\frac{1}{10}$ (theoretically there should be at least a ratio 1.05 due to a different number of loads and stores) for small matrixes. When matrixes are large enough we can see as the fractal approach takes advantage of its space locality, specially when disk is involved or very high level of the memory is accessed for very large number of times.

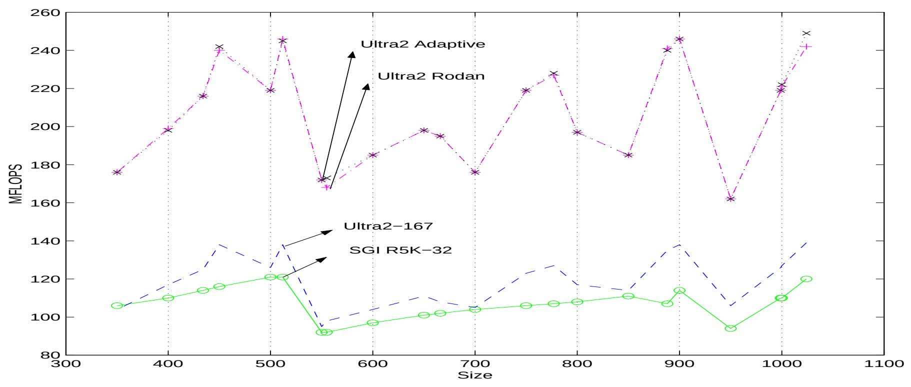

Figure 5.18: Performance evaluation on four platforms: SGI R5000 IP32, Ultra 2-170, Ultra 2-250 (Rodan-Adaptive), when fractal approach is used at every level

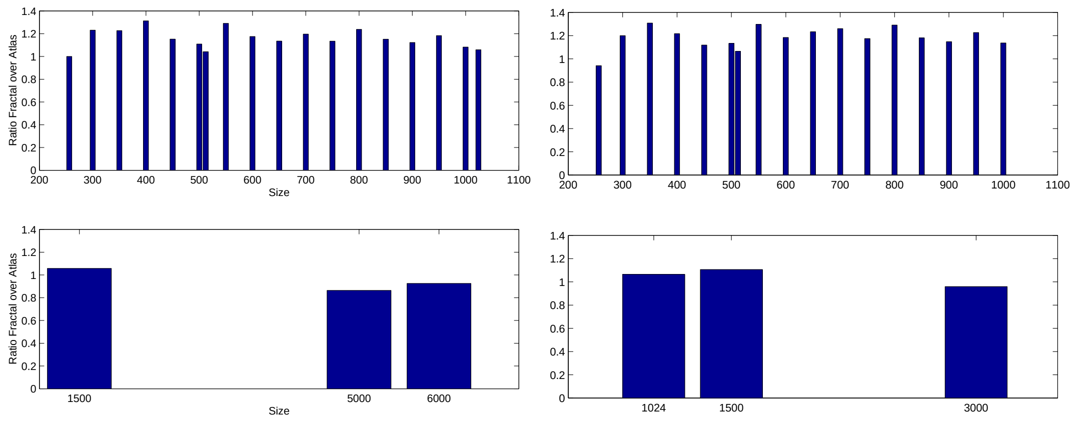

Figure 5.19: Ratio fractal over ATLAS when we change strategy: On the left for Ultra 2-170 on the right for Ultra 2-250

5.4.1 Cache Simulation

In the following tables there is an excerpt from simulation results by *Shade* for a real case memory hierarchy:

- On chip cache (L1) composed of an instruction cache (I1) and a data cache (D1), I1 is a 2-way associative cache of size 16K bytes with a line of 32 bytes, D1 is direct mapped, write through, no allocation cache of size 16K bytes with a line of 32 bytes, a least recently used replacement policy is applied.

- Off chip cache (L2), a unified cache (U2) direct mapped, write through (no a real

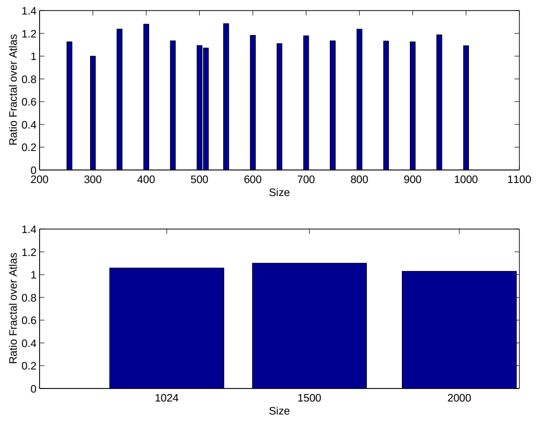

Figure 5.20: Ratio fractal over ATLAS when we change strategy for Ultra 5-10

case, indeed) of size 1M bytes with a line of 32bytes (64 Bytes is the most common case).

Every number is normalized to the number of basic operations, i.e. n^3, so that the results are more concise. The tables are laid out so that we can compare the simulated cache behavior of our fractal algorithm and the algorithm in ATLAS DGEMM. As we can see, the data cache behavior for the two algorithms is practically the same. The instruction cache behavior is different and the fractal approach, due to the code explosion of the leaves, is penalized. From the cache point of view, the two approaches have the same miss ratio, but different execution time.

500×500	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write mis
Fractal	4.13	0.05	1.17	0.06	0.87	0.05	0.30	0.01
ATLAS	3.85	1.58e-5	0.73	0.06	0.64	0.04	0.08	0.01

Table 5.3: L1: Instruction cache is 16 KB, line 32 B, 2-way lru and data cache 16 KB,line 32 B, direct write through, any number is a normalized value to

500×500	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write mis
Fractal	4.73e-2	7.30e-4	3.43e-1	1.31e-2	0.05	6.59e-3	0.30	6.54e-3
ATLAS	1.58e-5	1.48e-5	1.26e-1	1.86e-2	0.05	9.69e-3	0.08	8.93e-3

Table 5.4: L2: unified 1MB, line 32 B, direct write through

The case $< 500, 500, 500 >$ and case $< 1000, 1000, 1000 >$ are investigated to show how much the code explosion of the code for the leaves affects the instruction cache and the overall memory traffic. The case $< 1024, 1024, 1024 >$ is reported. The DGEMM in ATLAS package reduces cache conflicts performing a block copy of the matrixes it has to deal with. The fractal algorithm uses a particular padding of the matrixes (statically allocated) function of the cache size (if S is the cache size, there are introduced $\sqrt{\frac{S}{3}}$ elements) so that the three matrixes have a starting point of addresses not aligned; of course, if the matrixes are dynamically allocated no padding is required because we cannot make any assumption on the location of any matrix.

1000×1000	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write mis.
Fractal	3.61	4.73e-2	1.08	5.65e-2	8.09e-1	4.85e-2	2.69e-1	8.02e-3
ATLAS	3.35	1.97e-6	0.64	5.11e-2	5.85e-1	3.98e-2	5.51e-2	1.13e-2

Table 5.5: L1: Instruction cache is 16 KB, line 32 B, 2-way lru and data cache 16 KB,line 32 B, direct write through, any number is a normalized value to

1000×1000	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write mis.
Fractal	4.73e-2	8.84e-4	3.18e-1	9.93e-3	4.85e-2	5.95e-3	2.69e-1	3.68e-3
ATLAS	1.97e-6	1.94e-6	9.49e-2	1.53e-2	3.98e-2	1.04e-2	5.51e-2	4.82e-3

Table 5.6: L2: unified 1MB, line 32 B, direct write through

1024×1024	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write mis.
Fractal	3.60	1.83e-6	1.08	5.10e-2	7.92e-1	4.36e-2	2.90e-1	7.36e-3
ATLAS	3.33	2.01e-6	0.63	8.27e-2	5.81e-1	7.17e-2	5.47e-2	1.10e-2

Table 5.7: L1: Instruction cache is 16 KB, line 32 B, 2-way lru and data cache 16 KB,line 32 B, direct write through, any number is a normalized value to

1024×1024	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write mis.
Fractal	1.83e-6	1.70e-6	3.34e-1	1.23e-2	4.36e-2	9.01e-3	2.90e-1	3.33e-3
ATLAS	1.70e-6	1.84e-6	1.26e-1	1.42e-2	7.17e-2	9.24e-3	5.47e-2	4.92e-3

Table 5.8: L2: unified 1MB, line 32 B, direct write through

A more complete experimental results set may offer more details but the general case does not differ too much from the cases shown. Indeed, an estimation of factor γ it can be obtained. γ summarizes the miss ratio for a ideal cache of S elements size and line l (in $Q(S) = \gamma\frac{3\sqrt{3}}{l\sqrt{S}}$), indeed, $1 < \gamma < 4$.

5.5 Multiple Load and Store in a Single Cycle

Spatial locality can be exploited at register level by some architecture in a very particular and intriguing way. Loads and stores can involve contiguous registers and contiguous locations, for example registers reg_x and reg_{x+1} with x even and two contiguous memory locations $M[y]$ and $M[y+1]$ with y an opportune address. If the original code require to load $M[y]$ to reg_x and $M[y+1]$ to reg_{x+1}, it might be that the code can be modified so that instead of two loads only one is performed: $Load(reg_x, reg_{x+1}, M[y], M[y+1])$ spending the same number of cycles as a single load.

Typical constrains to apply this multiple load (and store) are the following:

- The registers should be consecutive.
- The number of registers involved should be a power of two (k=1,2,4..).
- The first register has number divisible by the number of register involved ($reg_x, \dots reg_{x+k}$ with $x\mathbf{mod}k == 0$).
- The starting memory address we start loading in should be divisible (*aligned*) by the number of elements we want to read from ($\mathbf{address}(M[y])\mathbf{mod}(k*\text{byte-per-element})$).

No every architecture is able to play this *trick* and furthermore no every scientific application exploits such space locality. It is required to divide the register file in contiguous set of registers, in general this is not possible but ABC-**fractal** approach exploits maximally this register file partition-ability. In the following we will compare three implementations for $< 1024, 1024, 1024 >$ so that matrixes have single precision (float) elements on an ULTRA 5: the algorithm available in the ATLAS package, the ABC-**fractal** algorithm with leaves implemented with single load/store and with double load/store. We compare the three algorithms in three different ways: time complexity, cache behavior and number of instructions. The execution time for the three implementations is depicted in Table 5.9.

ATLAS	Fractal
4.71s (455MFLOPS or 1.46 cycles per madd)	10.15s (211MFLOPS or 3.15 c/madd)
Fractal with double load/store	Peak Performance
6.88s (312MFLOPS or 2.13 c/madd)	3.22s (666MFLOPS or 1c/madd)

Table 5.9: Performance comparison on Ultra 5

The caches utilization can be summarized in Table 5.10 and 5.11 where every number is normalized with respect to the total number of basic instructions (1024^3). We can see that even if the fractal approach is oblivious about the memory hierarchy it can achieve about the same number of misses. We complete the performance evaluation showing

Algorithm	Instructions	Ins. Misses	Data	Data Misses
ATLAS	3.12	0	5.69e-1	5.40e-2
Frac. double l/s	2.71	0	3.92e-1	4.37e-2
Frac. single l/s	3.32	0	1.00e-0	5.81e-2

Table 5.10: ULTRA 5 I1 = 16KB, 2-way, 32Bytes line, D1 = 16KB, direct, 32Bytes line

Algorithm	Instructions	Ins. Misses	Data	Data Misses
ATLAS	0	0	7.71e-2	8.65e-3
Frac. single l/s	0	0	2.94e-1	1.55e-2
Frac. double l/s	0	0	9.74e-2	1.30e-2

Table 5.11: ULTRA 5 U2 = 512KB, direct, 32Bytes line

how many operations are performed (see Table 5.12, 5.13 and 5.14). We can see that reducing the number of loads and stores we do not affected negatively the cache miss behavior, we reduced the misses at the first level of cache and we reduced the number of communications with the caches. The performance is getting closer to the algorithm performance available in ATLAS package. From this comparison we can see a quantitative measure of the improvement obtained by a better memory access policy, we improved by thirty per cent. This approach fits particularly well our algorithm and the particular size of the problem. Matrixes are power of two and if a matrix is aligned then every sub matrix in the fractal decomposition has starting element aligned and therefore it is eligible for a multiple load and store. In general this is not the case, it is easy to find a counterexample

opcode	#exec	%exec	#annulled
fadds	1074790400	32.1903%	0
fmuls	1073741824	32.1589%	0
ldf	564167560	16.8970%	153089
add	156530628	4.6881%	6782
mulscc	103811880	3.1092%	9437378
subcc	76257389	2.2839%	1319
bpne,pt	65885919	1.9733%	0
sethi	34621536	1.0369%	16
stf	27296269	0.8175%	288
or	24703036	0.7399%	2466
jmpl	12600884	0.3774%	0
call	12598839	0.3773%	0
lduw	9505669	0.2847%	4626
be,a	9456598	0.2832%	0

Table 5.12: SGEMM from ATLAS package solving $< 1024, 1024, 1024 >$

in matrix $A = 6 \times 6$, indeed address of the start element of A_1 is $A+9$, which is not aligned (whatever is $k > 0$). To overcome this problem we can introduce dummy elements so that to force alignment of the starting element of any submatrix. The number of dummies is function of how many time we need to decompose fractally the matrix in sub-matrixes and the number of consecutive elements (K) we would like to load/store. We indicate with $D(N)$ the number of dummies elements we introduce in a matrix $N \times N$. $D(N)$ can be specified as follows:

$$D(N) \leq 3(K-1)\sum_{j=0}^{i-1} 4^i + 4^i D(N_i)$$

where $N_i = \lceil N_{i-1}/2 \rceil$ and $N_0 = N$ and where $D(n) = K - n$ for $n \leq K$. We can see that $D(N) \leq (K-1)[(\frac{N}{n})^2 - 1] + (\frac{N}{n})^2(K-n)$. In practice we decompose the matrix with coarser granularity based on the size of the leaves we want to compute and we do not pad inside those sub-matrixes at all. If we say that L is the size of the leaves in the computation of the matrix multiplication, $D(N) \leq K\frac{N^2}{L^2}$ where $4 \leq L \leq 8$. But in every case the number of dummies introduced is $O(N^2)$.

opcode	#exec	%exec	#annulled
fadds	1073741824	30.1979%	0
fmuls	1073741824	30.1979%	0
ldf	766509075	21.5573%	0
stf	279969795	7.8739%	0
mulscc	103811880	2.9196%	9437378
sethi	34612775	0.9734%	14
or	32841198	0.9236%	2424
lduw	19688054	0.5537%	154257
jmpl	19182721	0.5395%	0
add	16972516	0.4773%	1026
call	14989573	0.4216%	0
be,a	9456571	0.2660%	0
andncc	9438049	0.2654%	0
sll	8555718	0.2406%	1951
subcc	7459716	0.2098%	2098471

Table 5.13: ABC-**fractal** with single load/store solving $< 1024, 1024, 1024 >$

5.6 Dense Matrix Multiply Applications

We introduce in this section two basic applications where matrix multiplication might be basic operation: *Block LU factorization* and *Multiple Right Hand sides* [34].

5.6.1 Multiple Right Hand sides

Definition 5.10 *Multiple right hand side problem is to determine the unknown* $X \in \mathbb{R}^{n \times q}$ *into the system* $LX = B$ *where* $L \in \mathbb{R}^{n \times n}$ *is lower triangular and* $B \in \mathbb{R}^{n \times q}$.

Let us decompose the system as depicted as in Figure 5.6.1 and suppose all sub matrixes are square matrixes ($\mathbb{R}^{q \times q}$). We solve the system $L_{11}X_1 = B_1$ determining X_1 and then we remove the first column of L. Then we can solve $L_{22}X_2 = B_2 - L_{21}X_1$ and we can keep on in a similar fashion and eventually we can solve X_N.

Algorithm **rightSide**(L,X,B)

opcode	#exec	%exec	#annulled
fadds	1073741824	37.0349%	0
fmuls	1073741824	37.0349%	0
lddf	321912856	11.1032%	0
mulscc	103811880	3.5806%	9437378
stdf	67108884	2.3147%	0
sethi	34612775	1.1938%	14
or	32841200	1.1327%	2424
lduw	19688054	0.6791%	154256
jmpl	19182721	0.6616%	0
call	14989573	0.5170%	0
add	14875359	0.5131%	1025
be,a	9456569	0.3262%	0

Table 5.14: *ABC*-**fractal** with double load/store solving $< 1024, 1024, 1024 >$

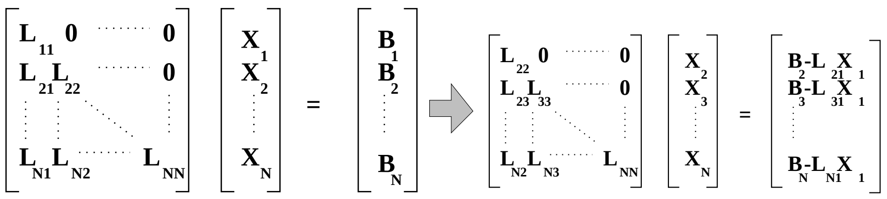

Figure 5.21: $LX = B$ block decomposition and columns elimination

```
for j=1:N
   Solve LjjXj=Bj
   for i=j+1:N
      Bi += LijXj;
```

♠

Of course, the choice of the size $q \times q$ is machine dependent. We apply a recursive algorithm which decomposes at any level the problem by four to near square matrixes. It follows a description of the algorithm in a pseudo language:

Algorithm **RSL**(B,L)

```
                        /* B = L X; X -> B */
if (|B| !=1) then
   RSL(B0,L0);          /* B0 = L0 X0; X0 -> B0 */
   RSL(B1,L0);          /* B1 = L0 X1; X1 -> B1 */
   submult(B3,L2,B1); /* B3 -= L2 X1 */
   submult(B2,L2,B0); /* B2 -= L2 X0 */
   RSL(B2,L3);          /* B2 = L3 X2; X2 -> B2 */
   RSL(B3,L3);          /* B3 = L3 X3; X3 -> B3 */
else
   B /= L;
```

♠

Aside of any line there is a comment that indicates the matrix computation each call performs. If we look at the matrix computations performed we can see easily that the type dag used to pre-compute indexes for matrix multiplication, it can be used here as well, as Figure 5.22 describes concisely. For performance purpose it is better to stop

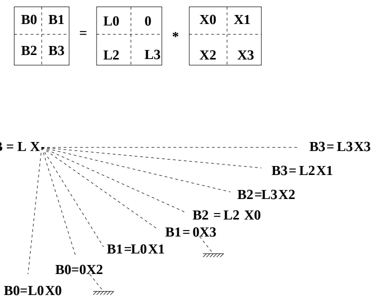

Figure 5.22: Recursive decomposition of RSL and first level of the type DAG associated with the matrix multiplication $B = LX$

recursion when the size of the problem is bigger than one. In fact, we stop the recursion when the number of rows in B is between four and eight. The computation is a explicit forward substitution, with some optimizations as loop-unrolling, scalar replacement and parallelism at instruction level.

5.6.2 LU-factorization without Pivoting

Definition 5.11 $A \in \mathbb{R}^{n\times n}$ *has LU factorization if there are a lower triangular matrix* L *and an upper triangular matrix* U *so that* $A = LU$.

A blocked algorithm for LU-decomposition can be described as follows. Matrix A can be decomposed as A_0, A_1, A_2 and A_3 where $A_0 \in \mathbb{R}^{q \times q}$, $A_1 \in \mathbb{R}^{q \times n-q}$, $A_2 \in \mathbb{R}^{n-q \times q}$ and $A_3 \in \mathbb{R}^{n-q \times n-q}$ and therefore A can be decomposed as in Figure 5.23. We can factorize

$$\begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \\ \mathbf{A}_2 & \mathbf{A}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{L}_0 & \mathbf{0} \\ \mathbf{L}_2 & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{U}_0 & \mathbf{U}_1 \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

Figure 5.23: $LU = A$ block decomposition

$A_0 = L_0U_0$. Then we can solve two multiple right hand side triangular systems: $L_0U_1 = A_1$ for U_1 and $L_2U_0 = A_2$ for L_2. We compute $B = A_3 - L_2U_1$ and we repeat the procedure on B.

Algorithm

FactorizeLU(A)

```
                         /* A = LU */
if (|A|>1) {
   FactorizeLU(A0);      /* A0 = L0U0 */
   RSL(A1,A0);           /* A1 = L0U1 */
   RSU(A2,A1);           /* A2 = L2U0 */
   submult(A3,A2,A1);    /* A3 -= L2U1*/
   Factorize(A3);        /* A3 = L3U3 */
}
else {
   L=1;
   U=A;
}
```

♠

Matrix multiplicaion is dominant operation on either application. This can justify a preliminary assumption so that square matrixes should be stored using the fractal layout, boosting the performance of matrix multiplies and slightly slow-down (i.e. N is the size of matrixes involved, the slow-down factor should be $O(\log_2 N)$) operations requiring single matrix element access (i.e. pivoting and backward/forward substitutions).

Experimental Results

We compare performances for LU factorization against LU factorization with partial pivoting available on *Sun Performance Library* (in ATLAS it is not implemented directly any LU factorization but the optimized matrix multiplication can be used in blocked algorithms). We show either cache behaviors (Table 5.15 ... 5.18) and execution time ratios (Figure 5.24) for Ultra 5.

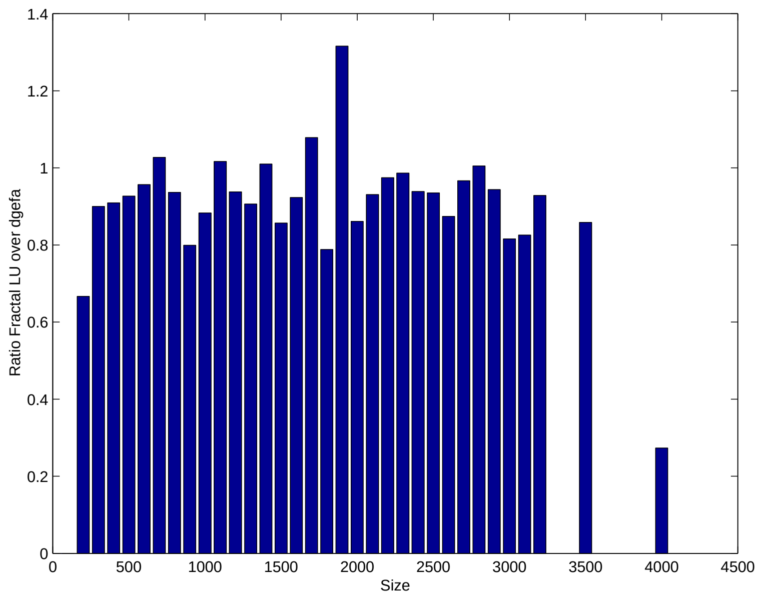

Figure 5.24: Performance evaluation on Ultra 5-10 (Lola), when fractal approach is used at every level

Note in Table 5.17 how the data miss ratio is really high for both algorithms. The reason can be found in the size of the matrix, power of two, and since both algorithms are blocked algorithms they perform operations on sub matrixes of the matrix, that may be power of two as well increasing cross interference among data. Unfortunately, the only way to reduce cross interference is padding the matrix introducing dummy elements: even if this technique can be applied it is not as for matrix multiplication, the padding should be embedded in the matrix layout (for matrix multiplication we could just allocate more space for the matrix elements and then shift the first element of the matrix, keeping the matrixes contiguously laid-out).

When we unfold the leaves of the recursive algorithms, we write in sequence the in-

LU 700×700	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write Mis.
LU-fractal	3.80	2.66e-2	6.58e-1	2.72e-2	5.48e-1	2.29e-2	1.10e-1	3.81e-3
dgefa	2.83	3.31e-5	6.18e-1	2.70e-2	5.55e-1	2.70e-1	6.29e-2	3.55e-3

Table 5.15: L1: Instruction cache is 16 KB, line 32 B, 2-way lru and data cache 16 KB,line 32 B, direct write through, any number is a normalized value to $\frac{2}{3}700^3$

LU 700×700	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write Mis.
LU-fractal	2.67e-2	2.45e-4	1.33e-1	5.88e-3	2.34e-2	3.33e-3	1.10e-1	2.54e-3
dgefa	3.31e-5	2.23e-5	8.99e-2	7.46e-3	2.70e-2	5.17e-3	6.29e-2	2.29e-3

Table 5.16: L2: unified 512 KB, line 32 B, direct write through

LU 1024×1024	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write Mis.
LU-fractal	2.98	3.53e-4	4.88e-1	5.76e-2	4.05e-1	3.52e-2	8.26e-2	2.24e-2
dgefa	2.47	1.23e-5	5.51e-1	7.52e-2	5.03e-1	5.26e-2	4.77e-2	2.26e-2

Table 5.17: L1: Instruction cache is 16 KB, line 32 B, 2-way lru and data cache 16 KB, line 32 B, direct write through, any number is a normalized value to $\frac{2}{3}700^3$

LU 1024×1024	Ins	Ins.Mis.	Data	Data Mis.	Read	Read Mis.	Write	Write Mis.
LU-fractal	3.53e-4	2.11e-5	1.18e-1	1.23e-2	3.52e-2	7.71e-3	8.26e-2	4.56e-3
dgefa	1.23e-5	7.56e-6	1.00e-1	9.88e-3	5.26e-2	8.28e-3	4.77e-2	1.60e-3

Table 5.18: L2: unified 512 KB, line 32 B, direct write through

structions that the compiler will optimize. The bigger is the size of the leaf to unfold the bigger is the sequence of instructions. The instruction cache misses may increase. In Table 5.15 we can find such a situation where all procedures in $Y(8)$ for the matrix multiplication are called. The number of instructions and the number of instruction cache misses for the fractal algorithm is by far larger than its adversary. The large size of the code is a problem that we have not attempted to solve yet for two reasons: 1) it is difficult because we want to perform scalar replacement and register allocation, 2) we may find solution forcing a register allocation, i.e. gcc permits to suggest a register allocation, and decompose the sequence of instruction by calls to functions with predefined register allocation, but this is not standard and it is based on a particular compiler which is not in general the best compiler.

5.7 Sparse Matrix

We want to introduce a data structure so that we can apply a fractal algorithm for matrix multiplication to sparse matrix. The main idea is to use a so called *quad tree* [29], shortly Q-tree, to accesses matrix elements. This section offers a preliminary introduction to the concepts and possible implementation of Q-trees.

Definition 5.12 *The quad tree is a tree. Every internal node has four children and every leaf can store either an element or a sub-matrix.*

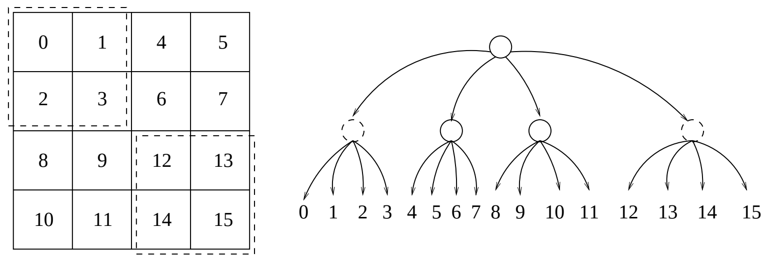

Figure 5.25: Quad Tree and Fractal Matrix Decomposition.

It is intuitive how the Q-tree can help to access matrix elements and therefore elements of matrix fractally stored. For square matrix the decomposition in four sub-matrixes, which, in turn, are divided recursively, is intuitive and easy to see. But Q-trees can be used also for non square matrixes, and the decomposition might be more irregular. In Figure 5.25 the leaves of the structure contain only one element. But we can increase the number of element stored in each leaf such as the size of the Q-tree can be considered negligible with respect to the matrix size.

There are different implementations for Q-trees. We consider the case when Q-tree is embedded in the matrix and when it is separated from the matrix associated with. Let us consider when Q-tree and matrix are separated structures. In Figure 5.26 we can see an example of the data structure organization. Indeed, there are two arrays. One is used for the Q-tree and the other is for the matrix. Formally, we can define the data structure as follows.

- An internal node of the Q-tree is composed by four pointers to four sub-matrixes.
- A leaf is a pair: a number of matrix elements pointed and a pointer to the matrix (or an offset value).

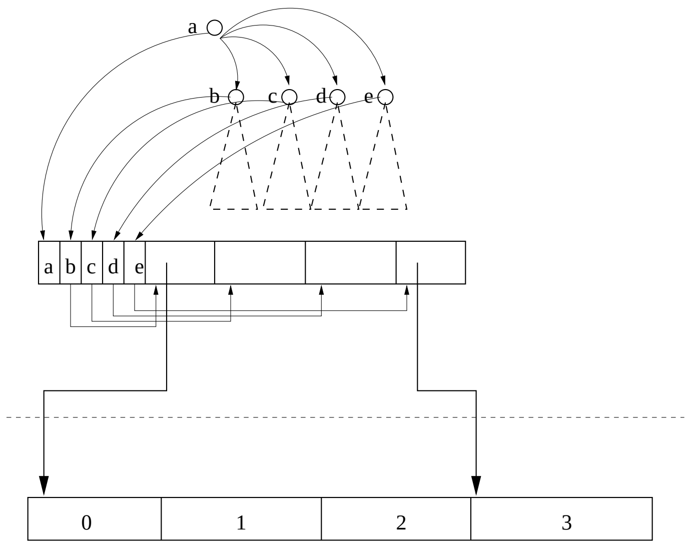

Figure 5.26: Quad tree and matrix are stored in different space storage.

This data structure arises naturally, it is intuitive and it decouples Q-tree from the matrix associated with. From a *Software Engineering* point of view, this permits to maintain different types for the two structures. The matrix element type can be a *short, integer, floating point with single* and *double precision* or *complex*, but usually we access the element of an array by integer indexing. This offers better maintenance, visualization of the solution and, if the matrix is not sparse, we can remove the Q-tree and access the matrix directly. We can see only one disadvantage, data interference. The access to the different storage spaces increases the interference among accesses to elements of same matrix.

Let us consider when the Q-tree is embedded in the matrix associated with, see Figure 5.27.

- An internal node has four pointers to internal nodes of the tree.
- A leaf is a $k+1$-tuple: an integer number where we store the number of matrix elements accessed from a leaf (k) and a sequence of k matrix elements stored as *fractal matrix* (or whatever else way).

We can see that we can avoid self interference because the access of matrix element and tree nodes is consecutive. Intuitively we can see that we can obtain some benefits to reduce cross interference among data using the technique proposed for matrix multiplication with dense matrixes. This layout has some disadvantages. It is dedicated, if the matrix is not

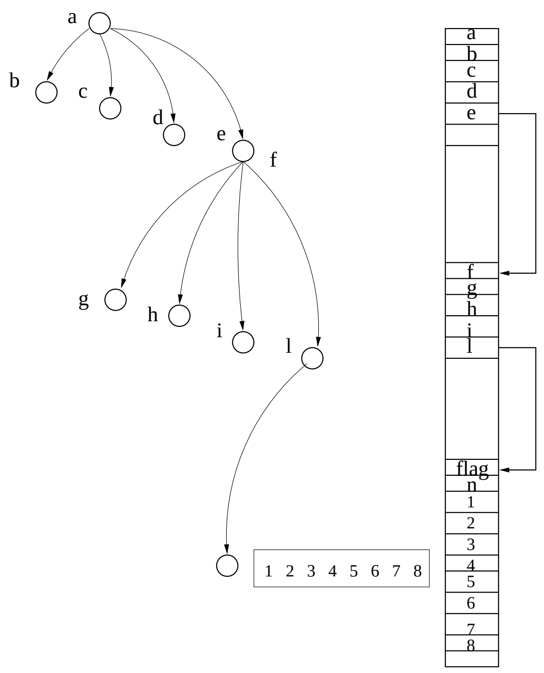

Figure 5.27: Q-tree and matrix are stored in the same space storage.

sparse we double the matrix size, the data stored have different information, and therefore different types but they share the same storage space. So visiting this structure should be done more carefully, doing opportune casts and data conversions.

Our purpose is to combine performance and portability, therefore we cannot drop a priori any of the data structures proposed. We need to have some experimental feed-backs to understand quantitatively how performance is affected by our choices.

We have not defined yet what we mean for sparse matrix.

Definition 5.13 *We say that a square matrix of sizes n^2 is a* sparse matrix *if it has at most $\frac{n^2}{k}$ non zero elements, where $k > 4$.*

In this case the overall space to store the matrix should be not greater than n^2.

There is a little problem. When we multiply two sparse matrixes the result might be dense. Indeed, the multiplication of two dense matrixes can give as result a sparse matrix, and the multiplication of two sparse matrixes can give as result a dense matrix.

5.7.1 Fractal Approach based on Q-tree

The data structures proposed previously can be solved in a single data structure. A node of a Q-Tree is 6-tuple. A C definition follows.

```
#define Leaf Element*
typedef  struct tnode *PQT;

typedef struct tnode {  /* Root Matrix A */
        PQT x0;         /* Root of Matrix A0 */
        PQT x1;         /* Root of Matrix A1 */
        PQT x2;         /* Root of Matrix A2 */
        PQT x3;         /* Root of Matrix A3 */
        PQT father;     /* Predecessor in the tree */
        LeafDatum submatrix; /* If A is leaf, this is a submatrix */
} QuadTree;
```

It can be used on top of fractal matrixes as a separated structure when the matrix is already stored, attaching on *sub-matrix* attribute a pointer to a fractal sub-matrix. It can be used during the construction of a fractal matrix allocating dynamically space to the *sub-matrix* attribute. In the same application we can use differently the structure. Take a matrix-matrix multiplication $C = A * B$ and suppose A and B are already in a fractal layout form but C is empty. We can avoid to duplicate the matrixes A and B simply attaching the right pointers to the leaves of Q-trees, obtained dynamically. C can be a sparse matrix and if it does not affect the performance, we can build matrix C dynamically. The lay out affects the miss ratio, since a Q-tree as auxiliary structure which is separated from the matrix introduces a new source for self interference (and cross interference) among data.

Suppose we have a fractal matrix A and we want to create dynamically its Q-tree. Suppose the matrix is a general matrix, $A \in \Re^{m\times p}$, we can always see the matrix as a matrix $Q \in \Re^{2^n\times 2^n}$ with n an opportune integer and introducing an opportune number of zero. Q has a natural fractal layout, where A is embedded, and on this matrix we can build up a Q-tree. The zero in A and the zero introduced so far in Q can be hidden, especially if they are gathered in blocks. A pure random distribution of zero offers a natural but worse

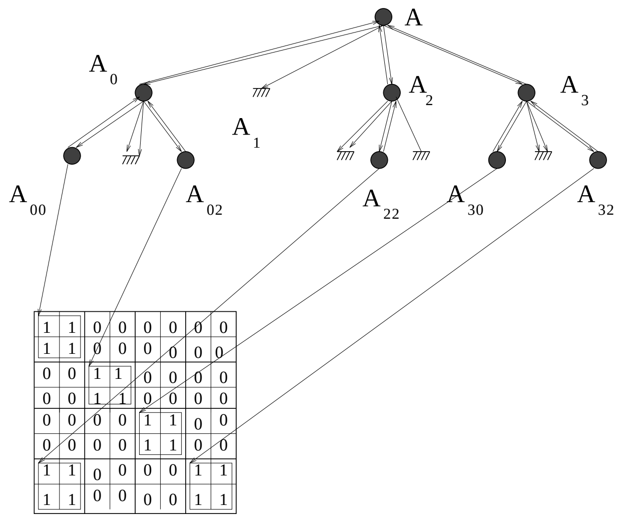

Figure 5.28: From a fractal matrix we build up a Q-Tree.

scenario, in practice it does not permit any pruning of the Q-tree, and therefore we may increase the space used to store the matrix. In Figure 5.28 we can see a quad tree so that every leaf can point/store a consecutive sub-matrix 2×2. We can see that sub-matrix A_1 is not stored, since it is a zero matrix. A very important issue is that the contents of the leaves, the sub-matrix, can be different from a simple vector of elements. We can change strategy at leaf level, i.e. use a compressed vector. Of course, if we change structure at leaf level we should change the matrix-matrix multiplication algorithm on leaves. Indeed for $2^k \times 2^k$ dense matrixes, it is natural to apply our fractal algorithm. But if there are other techniques that use registers and/or memory space in a more efficient way, we can always change approach.

The fractal-ABC matrix-matrix multiplication follows and we use an explicit recursive algorithm.

Algorithm

```
int  FHH_AB_lity_QT_Recursive(PQT cb, PQT ab, PQT bb) {
  int  r = 0;
  int i;
  static int l=0;

  if (isFullLeaf(ab) && isFullLeaf(bb)) {
    /* We have reached two leaves that contains two no empty
       submatrixes, we can multiply
       */
    makeLeaf(cb); /* we prepare the space */
    leafComputation(cb->submatrix,ab->submatrix,bb->submatrix);
    /* if the result is not a zero matrix */
    for(i=0; i<LEAFSIZE; i++)  {
      if (*(cb->submatrix+i))
          return 1;
    }
    /* otherwise free the space allocated */
    free(cb->submatrix);
    cb->submatrix=0;
    return 0;
  } else if (!isLeaf(ab) && !isLeaf(bb)) {
    /* we have reached two internal nodes
       */
    /* we prepare the sub-Quad-Tree */
    makeTree(cb->x0,cb);
    makeTree(cb->x1,cb);
    makeTree(cb->x2,cb);
    makeTree(cb->x3,cb);
    r |= FMM_AB_1ity_QT_Recursive(cb->x0, ab->x0, bb->x0);
    r |= FMM_AB_1ity_QT_Recursive(cb->x1, ab->x0, bb->x1)<<1;
    r |= FMM_AB_1ity_QT_Recursive(cb->x3, ab->x2, bb->x1)<<3;
    r |= FMM_AB_1ity_QT_Recursive(cb->x2, ab->x2, bb->x0)<<2;
    r |= FMM_AB_1ity_QT_Recursive(cb->x2, ab->x3, bb->x2)<<2;
    r |= FMM_AB_1ity_QT_Recursive(cb->x0, ab->x1, bb->x2);
    r |= FMM_AB_1ity_QT_Recursive(cb->x1, ab->x1, bb->x3)<<1;
    r |= FMM_AB_1ity_QT_Recursive(cb->x3, ab->x3, bb->x3)<<3;
    /* we prune the tree */
    if ((r&1)==0) {
      free(cb->x0);
      cb->x0=0;
    }
    if ((r&2)==0) {
      free(cb->x1);
      cb->x1=0;
    }
    if ((r&4)==0) {
      free(cb->x2);
      cb->x2=0;
    }
    if ((r&8)==0) {
      free(cb->x3);
      cb->x3=0;
    }
    return (r)?1:0;
    }
  /* other cases ? we have done nothing but
     if the current Q-tree of C is not empty
     we should not prune this one
     */
  return (cb!=0);
}
```

♠

The fractal-CAB algorithm revised permits a more *local* construction of C. Indeed, we can use the heap as a stack.

Algorithm

```
int  FHH_C_lity_QT_Recursive(PQT cb, PQT ab, PQT bb) {
  short r;
  short i;

  if (isFullLeaf(ab) && isFullLeaf(bb)) {
    makeLeaf(cb);
    leafComputation(cb->submatrix,ab->submatrix,bb->submatrix);

    for(i=0; i<LEAFSIZE; i++)  {
    if (*(cb->submatrix+i))
       return 1;
    }
    free(cb->submatrix);
    cb->submatrix=0;
    return 0;

  } else if (!isLeaf(ab) && !isLeaf(bb)) {
    /*  C0 */
    makeTree(cb->x0,cb);
    r =FHH_C_lity_QT_Recursive(cb->x0, ab->x0, bb->x0);
    r |=FHH_C_lity_QT_Recursive(cb->x0, ab->x1, bb->x2);
    if (!r) { free(cb->x0);  cb->x0=0;  }
    /*  C1 */
    makeTree(cb->x1,cb);
    r =FHH_C_lity_QT_Recursive(cb->x1, ab->x1, bb->x3);
    r |=FHH_C_lity_QT_Recursive(cb->x1, ab->x0, bb->x1);
    if (!r) { free(cb->x1);  cb->x1=0; }
    /*  C2 */
    makeTree(cb->x2,cb);
    r =FHH_C_lity_QT_Recursive(cb->x2, ab->x3, bb->x2);
    r |=FHH_C_lity_QT_Recursive(cb->x2, ab->x2, bb->x0);
    if (r==0) { free(cb->x2);  cb->x2=0;    }
    /*  C3 */
    makeTree(cb->x3,cb);
    r =FHH_C_lity_QT_Recursive(cb->x3, ab->x2, bb->x1);
    r |=FHH_C_lity_QT_Recursive(cb->x3, ab->x3, bb->x3);
    if (r==0) { free(cb->x3);  cb->x3=0;   }
    return (r!=0)?1:0;
  }
  return (cb)?1:0;
}
```

♠

We ask to allocate memory, and release memory, as in a stack. This approach is sufficient to store the matrix C in a fractal way even if it is not dense matrix. Any advantage obtained from the fractal-CAB algorithm can be exploited only if the matrix C is used in further matrix-matrix multiplication as operand. Note that if we do $C = A * B$ and $C+ = D * E$, we cannot assure that matrix C is stored in any fractal way, because we are adding elements dynamically.

Tacitly, our algorithm compute multiplication on matrix $Q \in \Re^{2^n \times 2^n}$, the lay out of non zero elements in the matrixes specifies what is the genre, rectangular, square etc. This solution is very flexible but it might be not optimal.

We used the recursive algorithm but it can be developed a non recursive one, indeed, we can see why in the structure is still present the pointer *father* to the precedence node in the tree.

5.7.2 Sparse Matrix Multiplication, Related Works

The basic concept of sparse matrix is that zero elements may not be stored or, at least, it is minimized the storing space for zero elements. Multiplication of sparse matrix must take advantage from this data structure to avoid useless multiplications, i.e. multiplication by zero. In literature there are a lot of data structures dedicated to different layouts and therefore there are different kernels routines devised for particular data structures. Therefore it is very common to find a set of routines gathered in libraries. We have found up to three different approaches: data structure oriented, kernel oriented and linear algebra application oriented, and shortly we explain these three points. These distinctions are not always clear and they are subjective.

Data Structure Oriented: SPARSKIT is an example, [57]. They offer sparse matrix multiplication and they propose an algorithm based on only one basic structure, and a collection of format change routines. The purpose of the module is offer a flexible environment to perform matrix multiplication on sparse matrixes. The developer must be aware about sparse matrix formats and apply the opportune transformations. The format of the matrixes depends on the problem domain but the solution, the algorithm, is independent. The efficiency and performance in this environment is clearly based on the algorithm, more than the data structure.

Kernel Oriented: Sparse BLAS is an example, [55], [15]. They propose a complete set of routines that perform sparse matrix-matrix multiplication for different data structures and layouts. In other words, they offer different kernels, algorithms, based on specific layout and structures. The developer has to produce the data structure and use their routines. The basic operation is $C+=AB$ where A is sparse and C and B are stored as dense matrixes. In particular the number of single/double precision multiplications is proportional to the non zero elements of both matrixes. It is clear that more it is detailed the information about contents of the sparse matrix better performance can be achieved.

Application Oriented: basic operations are used for Linear Algebraic problems, most of the times basic operation is not matrix-matrix multiplication but matrix-vector multiplication. The implementation of the basic operations can be different but usually they respect the interface and the semantic of the 3-BLAS library.

If we want to summarize our technique, we have proposed a general data structure and a general algorithm. It is similar to SPARSKIT but we do not produce a complete set of change format routines. In practice we have chosen a different structure and, consequentially, a different algorithm. But we compare our technique with Sparse BLAS for the following reasons. Sparse BLAS is becoming a *de facto* standard for sparse matrix computations therefore it is clearly evaluable obtaining a quantitative comparison. We can understand how much a generalization of data structure and algorithm, as the fractal layout proposes, looses in performance. We already know the performance of Q-tree implementation of the fractal algorithm for dense matrixes and we have understood that this approach has good performance, and if it is comparable with Sparse BLAS we can say that our approach offer a general approach for general matrix-matrix multiplication.

The main problem is to decide a test bed such that we can compare performances. In other words we must devise a set of matrixes on which we measure the execution time of matrix multiplication. Suppose we have a random generator of number $\mathcal{N}(T_1, T_2)$ that generates integer numbers between $[T_1, T_2]$ with uniform distribution, any number generated has probability $\frac{1}{T_2-T_1+1}$ to happen. This generator is used to select not only the data of the matrix but also its shape (i.e. rectangular matrix) and its sparseness.

- Generally sparse: any element of the matrix is generated by $\mathcal{N}(-T_1, T_1)$.
- Blocked Sparse: the two sizes of tiles are chosen randomly, $l_1 \times l_2$, and every element in a tile has the same value taken from $\mathcal{N}(-T_1, T_1)$.
- Diagonal Sparse: the number of diagonals above and below the principal diagonal is chosen randomly and every element in the diagonal is chosen from $\mathcal{N}(-T_1, T_1)$.

We fix the maximum size of the matrix (i.e. 512×512) and for each type of matrix we generate 1000 sparse matrixes (500 of size $P \times N$ and 500 $N \times M$). These sets are used as performance test. We can achieve in this way an *average* value determined as follows. At the i-th iteration we perform a matrix multiplication A_iB_i with A_i a matrix $P_i \times N_i$ and B_i

a matrix $N_i \times M_i$. We measure the time of each multiplication T_i. The maximum number of operations can be estimated as $N_o = \sum_{i=1}^{500} P_i N_i M_i$ and the total time is $T = \sum_{i=0}^{500} T_i$. A simple and average estimation of the performance can be expressed by the ratio $\frac{N_o}{T}$.

For our implementation the data structure is fixed but to utilize Sparse BLAS we use different data structures for different types of matrixes. We will use the CSR (Compressed Sparse Row) point entry form and BSR (Block Compressed Sparse Row). The CSR data structure, compressed sparse row, is composed by four arrays and it manages single entry point of the matrix, non zero elements. The basic idea is that the matrix is a row-major ordered matrix where only no zero elements are stored. There are different implementation of this data structure and we present the data structure proposed by Sparse BLAS.

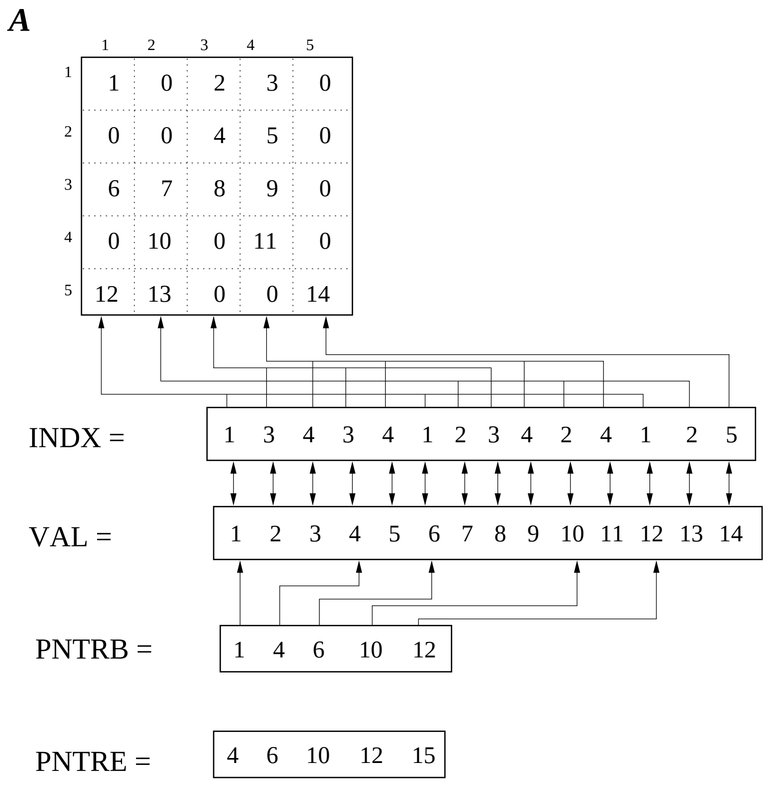

Figure 5.29: Example of CSR structure on a real case.

- VAL is the element array, the non zero elements are stored by row. We store in VAL non zero elements of the first row, followed by non zero elements of the second row and, eventually, non zero elements of the last row.

- $INDX$ is an integer array with the same number of elements of VAL, indeed, $INDX[i]$ indicates in what column in A the element $VAL[i]$ is located.

- $PNTRB$ is an integers array of size equal to the number of rows in A, and $PNTRB[i] - PNTRB[1] + 1$ indicates the index in VAL where row i starts. A row is empty if $PNTRB[i] = PNTRB[i+1]$.

- $PNTRE$ is an integers array of size equal to the number of row in A, and $PNTRE[i] - PNTRB[1]$ indicates the index in VAL where row i ends. The last element in $PNTRE$ indicates the number of non zero elements in matrix A plus one.

(I noted that in general $PNTRE$ is used but this array is redundant, same information is available in $PNTRB$).

The BSR data structure, constant block compressed sparse row, is composed by four arrays. We tile the matrix A by constant sizes tile, the matrix should be a multiple of the basic tile and we store only blocks in which there are non zero elements. The block is considered a basic element and we can see this data structure as a generalization of CSR, where instead of an entry point there is a sub-matrix.

The idea used is very similar to the fractal layout. Every sub-matrix or block is stored in column-major order, that is, every block is laid out consecutively. The layout of blocks is in a row-major order. In Figure 5.30, matrix A is tiled with 2×2 tiles. If we consider each block as an single entry we can see matrix A as a 3×3 matrix $\dot{A}$. This Matrix is stored in a compressed sparse row way, in the first row there is the first entry and the third entry that should be stored consecutively. We can see that we are storing matrix A in a such a way that accessing row blocked can be done by access a vector by constant stride and consecutively. Let ne be the number of non zero blocks in A and let p_b and l be the blocked rows of A and the size of the square tiles.

- VAL is an integer of length $ne * l * l$. It contains the elements of sparse matrix A, at $(i-1)l^2$ starts the i-th non zero blocks in VAL of the sparse matrix A, the blocks are stored in a row-major order..

- $BINDX$ is an integer array of length ne, the number of non zero blocks in A. $BINDX[i]$ is the column in the blocked matrix $\dot{A}$ where the i-th block in VAL is located.

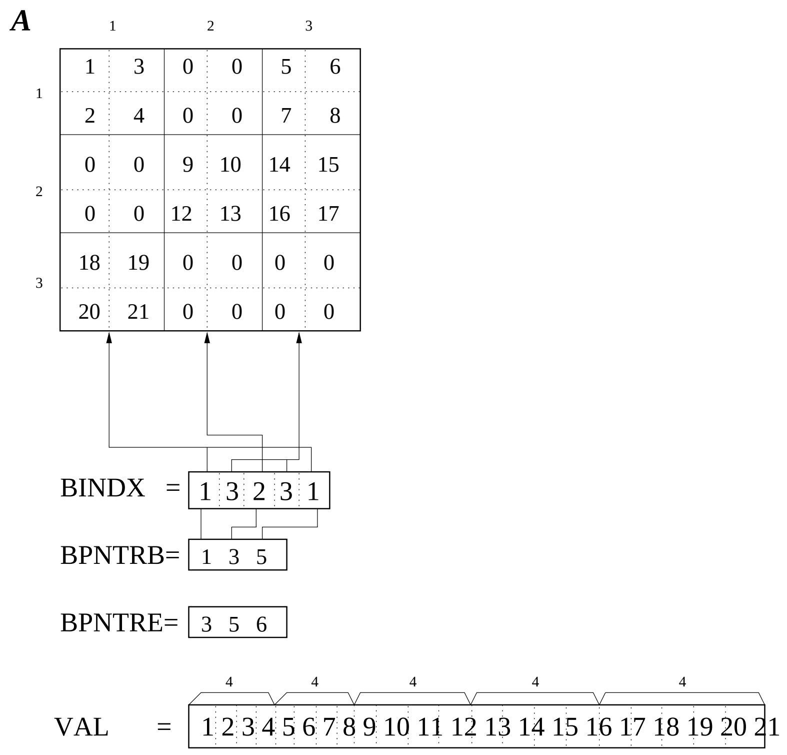

Figure 5.30: Example of BSR structure on a real case.

- $BPTNRB$ is an integer array of length equal to the number of rows in matrix $\dot{A}$, i.e. in Figure 5.30 this is three. Differently from the CSR case, $BPTNRB[i] - BPTNRB[1] + 1$ is the location in $BINDX$ of the first block in the i-th blocked row. In practice, $BINDX[BPTNRB[i] - BPTNRB[1] + 1]$ identifies the column of the first non zero block on i-th row in sparse matrix A. If $BPTNRB[i] = BPTNRB[i+1]$ then the i-th row is empty.

- $BPTNRE$ is an integer array as $BPTNRB$ but identifies the last block.

These structures are enough general so that we can devise two simple routines producing the data structure easily from dense matrixes.

Chapter 6

Consideration

Data locality at every level of memory hierarchy is a difficult to achieve and important issue for scientific applications. In this work we investigate such an issue exploiting the concept of data locality and giving particular importance to performance evaluation. Data locality of an algorithm expressed by a DAG can be quantitatively represented by its Accesse Complexity and, for DAGs, we presented an approach to schedule operations so that data locality can be exploited through the concept of Convex Decomposition Tree.

We investigated the current trend to use micro tests (microbenchmarking) to measure performance of single feature of complex architectures. This technique can be useful when it is required to produce *fine tuned code* and it is required more information about the host architecture for a dedicated application. We discovered it is hard to define a unique set of micro tests that works for a set of architectures, not only because the difference among architectures but also because different among compilers. But we could achieve insights about the most common techniques to obtain performance in the current architectures.

Square matrix matrix multiplication is our case study where we try to exploit data locality at every level of the memory hierarchy, but reducing to minimum the number of optimization based on the particular architecture. We have shown that we can achieve comparable perfomance with vendors and machine tuned packages and the memory hierarchy is optimally utilized. We discovered that a recursive-based algorithm has its waekness on the most important level of memory hierarchy: the register file. Other issues of matrix multiplication are exposed: space locality at register level is a side effect that can be exploited in some architectures. Applications of matrix multiplication have been investigated, from LU decomposition to sparse matrix multiplication. Our studies end suggesting that for performance purpose there is no magic word to solve locality optimally and other characteristics must be considered such as latency hiding, and code explosion.

References

[1] A.Azevedo, J.Hummel, D.Kolson and A.Nicolau *Annotating the Java Bytecodes in Support of Optimization* technical report.

[2] A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir *A model for Hierarchical Memory.* Proc. of 19th Annual ACM Symposium on the Theory of Computing, New York, 1987,305-314.

[3] A. Aggarwal, A.K. Chandra and M. Snir *Communication Complexity of PRAMs.* Theoretical Computer Science 71(1990) 3-28.

[4] A. Aggarwal, A.K. Chandra and M. Snir *Hierarchical Memory with Block Transfer.* 1987 IEEE.

[5] A.V.Aho, J.D.Ullman and R.Sethi *Compilers: Principles, Tecniques and Tools.* Addison-Wesley Pub.Co.1985

[6] K.Arnold and J.Gosling *The Java Programming Language.* Addison-Wesley Pub.Co. 1996

[7] U.Banerjee, R.Eigenmann, A.Nicolau and D.Padua *Automatic Programm Parallelization.* Proceedings of the IEEE vol 81, n.2 Feb. 1993.

[8] A.Beguelin, J.Dongarra, A.Geist, W.Jiang, R.Mancheck and v.Sunderam *PVM: Parallel Virtual Machine* The MIT Press, Cambridge, Massachusetts, London, England.

[9] M. de Berg, M. van Kreveld, M.Overmars and O. Schwarzkopf *Computational Geometry: Algorithms and Applications*, Springer-Verlag 1997

[10] G.Bilardi and F.Preparata *Horizon of Parallel Computation.* Jurnal of Parallel and Distributed Computing 27, 172-182 (1995).

[11] G.Bilardi, K.T.Herley, A.Pietracaprina, G.Pucci, P.Spirakis *BSP vs LogP* Proc. of 8th Ann.ACM Symposium on Parallel Algorithms and Architectures (SPAA'96), June 1996, pages. 25-32.

[12] G.Bilardi and F.Preparata *Processor-Time Trade offs under Bounded-Speed Message Propagation.* manuscript, Feb 1995. manuscript, Feb 1995.

[13] J.Bilmes, Krste Asanovic, C.Chin and J.Demmel *Optimizing Matrix Multiply using PHiPAC: a Portble, High-Performance, Ansi C Coding Methodology*, International Conference on Supercomputing, July 1997.

[14] G.E.Blelloch, C.E.Leiserson, B.M.Maggs, C.G.Plaxton, S.J.Smith and M.Zagha *A comparison of Sorting Algorithms for the Connection MAchine CM-2* Proc. of 3rd ACM Symposium on on Parallel Algorithms and Architectures (SPAA'93), pages. 3-16.

[15] S.Carney, M.A.Heroux, G.Li, R.Pozo, K.A.Remington and K.Wu *A revised Proposal for Sparse BLAS Toolkit*

[16] S.Carr and K.Kennedy *Compiler Blockability of numerical algorithms* Proceedings of Supercomputing Nov 1992, pg.114-124.

[17] S.Chatterjee, V.V.Jain, A.R.Lebeck and S.Mundhra *Nonlinear Array Layouts for Hierarchical Memory Systems* Proc. of ACM international Conference on Supercomputing, Rhodes,Greece, June 1999

[18] S.Chatterjee, A.R.Lebeck, P.K.Patnala and M.Thottethodi *Recursive Array Layout and Fast Parallel Matrix Multiplication* Proc. 11-th ACM SIGPLAN, June 1999.

[19] K.D.Cooper, M.W.Hall, R.T.Hood, K.Kennedy, K.S.McKinley, J.M.Mellor-Grummey, L.Torczon and S.K.Warren *The Parascope Parallel Programming Environment* Proceedings of the IEEE vol 81, n.2 Feb. 1993.

[20] D.Coppersmith and S.Winograd *Matrix Multiplication via Arithmetic Progression* In Poceedings of 9th annual ACM Symposium on Theory of Computing pag.1-6, 1987.

[21] *The Common Object Request Broker: Architecture and Specification* OMG document revision 2.00.

[22] M.Cordioli, M.Gusella e M.Muner *Active Message Passing.* manuscript in italian.

[23] T.H.Cormen, C.E.Leiserson and R.L.Rivest *Introduction to Algorithms* MIT press, Cambridge, Massachusetts London, England

[24] M.J.Dayde and I.S.Duff *A Blocked Implementtion of Level 3 BLAS for RISC Processors* `TR_PA_96_06`, available on line `http://www.cerfacs.fr/algor/reports/TR_PA_96_06.ps.gz` Apr. 6 1996

[25] P.D'Alberto and A.Montresor *BSP on PVM.* manuscript

[26] J.Dongarra, A.Lumsdaine, X.Niu, R.Pozo and K.Remington *A Sparse Matrix Library in C++ for High Performance Architectures*

[27] N.Eiron, M.Rodeh and I.Steinwarts *Matrix Multiplication: a Case Study of Algorithm Engineering* Proceedings WAE'98, Saarbrücken, Germany, Aug.20-22, 1998

[28] Engineering and Scientific Subroutine Library `http://www.rs6000.ibm.com/resource/aix_resource/sp_books/essl/`

[29] P.Flajolet, G.Gonnet, C.Puech and J.M.Robson *The Analysis of Multidimentional Searching in Quad-Tree.* Proceeding of the second Annual ACM-SIAM symposium on Discrete Algorithms, San Francisco, 1991, pag.100-109.

[30] J.D.Frens and D.S.Wise *Auto-blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code* Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming, SIGPLAN Not. 32, 7 (July 1997), 206–216.

[31] M.Frigo and S.G.Johnson *The Fastest Fourier Transform in the West* MIT-LCS-TR-728 Massachusetts Institute of technology, Sep. 11 1997.

[32] M.Frigo, C.E.Leiserson, H.Prokop and S.Ramachandran *Cache-Oblivious Algorithms* submitted for publication http://supertech.lcs.mit.edu/cilk/paper/FrigoLePr99.ps.gz

[33] E.D.Granston, W.Jalby and O.Teman, *To copy or not to copy: a compile-time technique for assesing when data copying should be used to eliminate cache conflicts*, Proceedings of Supercomputing Nov 1993, pg.410-419.

[34] G.H.Golub and C.F.van Loan *Matrix Computations* Johns Hopkins editor 3-rd edition

[35] M.Goudreau, K.Lang, S.Rao, T.Suel and T.Tsantilas. *Toward Efficiency and Portability: Programming with the BSP Model* Proc. of 8th Ann.ACM Symposium on Parallel Algorithms and Architectures (SPAA'96), June 1996, pages 1-12.

[36] C.A.Hsieh, M.T.Conte, T.L.Johnson, J.C.Gyllenhaal and W.WHwu *Optimizing NET compilers for Improved Java Performance.* Computer Innovative technology for computer professionals vol.30, n.6 June 1997

[37] J.L.Hennesy and D.A.Patterson *Computer Architecture a Quantitative Approach.* Morgan Kaufman 1996.

[38] N.J.Higham *Accuracy and Stability of Numerical Algorithm* ed. SIAM 1996

[39] Hong Jia-Wei and T.H.Kung *I/O complexity :The Red-Blue pebble game.* Proc.of the 13th Ann. ACM Symposium on Theory of Computing Oct.1981,326-333.

[40] R.Iyer, N.M.Amato, L.Rauchwerger and L.Bhuyan *Comparing the Memory System Performance of the HP V-class and SGI Origin 2000 Multiprocessors using Microbenchmarks and Scientific Applications* ICS'99 Rhodes, Greece.

[41] J.Jaja *An introduction to Parallel Algorithms.* Addison-Wesley Pub.Co. 1992

[42] B.H.H.Juurlink and H.A.G.Wijshoff *A quantitative Comparison of Parallel Computation Models* Proc. of 8th Ann.ACM Symposium on Parallel Algorithms and Architectures (SPAA'96), June 1996, pages 13-24.

[43] B.Kågström, P.Ling and C.Van Loan *Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues* ACM transactions on Mathematical Software, Vol24, No.3, Sept.1998, pages 303-316

[44] B.Kågström, P.Ling and C.Van Loan *GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation BenchmarK* ACM transactions on Mathematical Software, Vol24, No.3, Sept.1998, pages 268-302.

[45] M.Lam, E.Rothberg and M.Wolfe *The cache performance and optimizations of blocked algorithms*, Proceedings of the fourth international conference on architectural support for programming languages and operating system, Apr.1991,pg. 63-74.

[46] D.Lea *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Pub.Co. 1996

[47] F.Thomson Leighton *Introduction to Parallel Algorithms and Architectures: Arrays.Trees.Hypercubes.* Morgab Kaufmann, San Mateo, CA, 1992.

[48] M.Leppinen, P.Pulkkinen and A.Rantiainen *Java and CORBA-based Network Management* Computer Innovative technology for computer professionals vol.30, n.6 June 1997

[49] S.S.Muchnick *Advanced Compiler Design Implementation* Morgan Kaufman

[50] J.P.Munson and P.Dewan *Sync: A Java Framework for Mobile Collaborative Applications.* Computer Innovative technology for computer professionals vol.30, n.6 June 1997

[51] R. Orfali and D.Harkey *Client/Server Programming with Java and CORBA* Jhon Wiley and Sons 1997

[52] P.K.Pancake *Multithreaded Languages for Scientific and Tecnical Computing.* Proceedings of the IEEE vol 81, n.2 Feb. 1993.

[53] P.R.Panda, N.D.Dutt and A.Nicolau *Memory Organization for Improved Cache Performance in Embedded Processors.* International Symposium on System Syntesis (ISSS'97) La Jolla, November 1996.

[54] P.R.Panda, H.Nakamura, N.D.Dutt and A.Nicolau *Improving Cache Performance Through Tiling and Data Alignment.* Solving Irregularly Structured Problems in PArallel Lecture Notes in Computer Science, Springer-Verlag 1997.

[55] R.Pozo, K.A.Remington *SparseLib++ v.1.5* Sparse matrix Class Library Reference Guide

[56] A.L.Rosenberg and I.H.Sudborough *Bandwith and Pebbling.* Computing 31, 115-139 (1983).

[57] Y.Saad *SPARSKIT: a basic toolkit for sparse matrix computations.*

[58] John E.Savage *Space-Time tradeoff in Memory Hierarchies.* Tecnical report Oct 19, 1993.

[59] V.Strassen *Gaussian Elimination is not optimal* Numerische Mathematik 14(3):354-356, 1969.

[60] S.Toledo *Locality of Reference in LU Decomposition with Partial Pivoting* SIAM J.Matrix Anal. Appl. Vol.18, No. 4, pp.1065-1081, Oct.1997

[61] M.Thottethodi, S.Chatterjee and A.R.Lebeck *Tuning Strassen's Matrix Multiplication for Memory Efficiency* Proc. SC98, Orlando,FL, nov.1998 (http://www.supercomp.org/sc98).

[62] L.G.Valiant *A Bridging Model for Parallel Computation* Communications of the ACM, 33(8):103-111,August 1990.

[63] A.Waheed and D.T.Rover *A structured Approach to Instrumentation System Development and Evaluation.*

[64] R.C.Whaley and J.J.Dongarra *Automaticlly Tuned Linear Algebra Software* http://www.netlib.org/atlas/index.html

[65] D.S.Wise *Undulant-Block Elimination and Integer-Preserving Matrix Inversion* Technical Report 418 Computer Science Department Indiana University August 1995

[66] M.Wolfe *More iteration space tiling*, Proceedings of Supercomputing, Nov.1989, pg. 655-665.

[67] M.Wolfe *High Performance Compilers for Parallel Computing.* Addison-Wesley Pub.Co.1995

[68] H.P.Zima and B.M.Chapman *Compiling for Distributed-Memory Systems* Proceedings of the IEEE vol 81, n.2 Feb. 1993.