A methodology for speeding up loop kernels by exploiting the software information and the memory architecture

Vasilios Kelefouras[a], Angeliki Kritikakou[b], Costas Goutis[a]

[a]*Department of Electrical and Computer Engineering, University of Patras (kelefouras@ece.upatras.gr)*
[b]*Education and Research Department in Computer Science and Electrical Engineering, University of Rennes 1*

Abstract

It is well-known that today's compilers and state of the art libraries have three major drawbacks. First, the compiler sub-problems are optimized separately; this is not efficient because the separate sub-problems optimization gives a different schedule for each sub-problem and these schedules cannot coexist as the refining of one, causes the degradation of another. Second, they take into account only part of the specific algorithms information. Third, they take into account only a few hardware architecture parameters. These approaches cannot give an optimumal solution.

In this paper, a new methodology/pre-compiler is introduced, which speeds up loop kernels, by overcoming the above problems. This methodology solves four of the major scheduling sub-problems, together as one problem and not separately; these are the sub-problems of finding the schedules with the minimum numbers of i) L1 data cache accesses, ii) L2 data cache accesses, iii) main memory data accesses, iv) addressing instructions. First, the exploration space (possible solutions) is found according to the algorithm's information, e.g. array subscripts. Then, the exploration space is decreased by orders of magnitude, by applying constraint propagation to the software and hardware parameters.

We take the C-code and the memory architecture parameters as input and we automatically produce a new faster C-code; this code cannot be obtained by applying the existing compiler transformations to the original code. The proposed methodology has been evaluated for five well-known algorithms in both general and embedded processors; it is compared with gcc and clang compilers and also with iterative compilation.

Keywords: Data reuse, register allocation, optimization, memory hierarchy, loop tiling, data locality, Diophantine equations

1. Introduction

Regarding data dominant applications (for example linear algebra, image, signal and video processing algorithms), the major performance critical parameters are i) the number of main memory accesses, ii) the number of L3/L2 cache accesses, iii) the number of L1 data cache accesses and iv) the number of executed instructions (we assume that the number of the algorithm instructions cannot be reduced and thus we reduce only the number of addressing instructions). The above compilation/scheduling sub-problems are interdependent and thus they cannot be optimized separately; actually, the refining of one sub-problem causes the degradation of another, e.g. a decrease of the number of L2 data cache accesses will consequently increase the number of L1 data cache accesses. Researchers try to solve this problem by using iterative compilation techniques.

Iterative compilation has five major drawbacks, i) there are memory efficient schedules which cannot be produced by applying the existing compiler transformations, ii) iterative compilation does not use all the existing transformations, including all the different transformation parameters, e.g. unroll factor values and tile sizes, because in this case compilation will last for years, iii) only one level of tiling is applied, which is not efficient, iv) register allocation is applied without taking into account the data reuse; this means that the arrays references are assigned into registers, without taking into account that some are accessed a lot and others do not, v) the data array layouts are not taken into account; we will show that when tiling to multidimensional arrays is applied, the data array layouts must change. These drawbacks are overcome by the proposed methodology.

The proposed methodology finds the exploration space (all possible solutions), neither by applying compiler transformations nor by utilizing the above sub-problems separately. Instead, the exploration space is produced by exploiting the algorithm's information; we create mathematical equations and inequalities, according to the array subscripts, the loops iterators and the loops bounds. These equations (subscript equations), give the data reuse and the production-consumption of the arrays; the memory access pattern of each array reference is given by its subscript equation. Given that the memory access pattern of each array is given by its subscript equation, we claim that all memory efficient solutions (exploration space) can be produced by processing these equations. The subscript equations are processed and a new iteration space is created. Each subscript equation gives either its iterators or even new iterators, to the new iteration space. Then, the exploration space is orders of magnitude decreased by applying constraint propagation to the software and hardware parameters. Regarding the hardware parameters, we produce register file and data cache inequalities, which contain all the (near)-optimum tile sizes; these inequalities contain i) the tiles sizes in elements, ii) the shape of each array's tile. Furthermore, new data array layouts are generated, according to the data cache associativity. All the schedules with different tile sizes and data array layouts, than these the proposed methodology gives, are not considered, decreasing the exploration space.

The major contributions of this paper are: i) the optimization of the above subproblems as one problem and not separately for a wide range of algorithms and computer architectures, ii) the software information and several hardware parameters are fully exploited giving high execution speed solutions and a smaller search space, iii) the proposed methodology, due to the major contribution of number (ii) above, gives a smaller code size and a smaller compilation time, as it does not test a large number of alternative schedules, as the state of the art (SOA) libraries and iterative compilation do.

The experimental results are taken by using a general purpose processor, an embedded processor and Simplescalar simulator [1]. The proposed methodology is evaluated for five well-known data dominant algorithms over two different compilers (speedup from 1.8 up to 18.3) and iterative compilation technique (speedup up to 2.2).

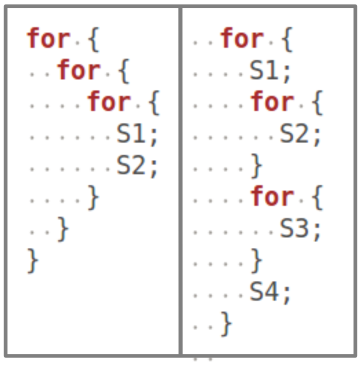
The remainder of this paper is organized as follows. In Section 2, the related work is given. The proposed methodology is given in Section 3 while the experimental results are given in Section 4. Finally, Section 5 is dedicated to conclusions.

2. Related Work

The independent optimization of the back end compiler phases (e.g. transformations, register allocation), leads to inefficient binary code due to the dependencies among them. These dependencies require that all phases should be optimized together as one problem and not separately. Toward this, much research has been done, either to simultaneously optimize only two phases, e.g. register allocation and instruction scheduling [2] [3] or to apply predictive heuristics [4] [5]. Nowadays compilers and related works, apply i) iterative compilation techniques [6] [7] [8] [9], ii) both iterative compilation and machine learning compilation techniques to restrict the configurations' search space and thus to decrease the compilation time [10] [11] [12] [13] [14] [15], iii) iterative optimizations or compiler transformations, by using the Polyhedral model [16] [17] [18] [19], iv) compiler transformations by using heuristics and empirical methods [20]. In iterative compilation, a large number of different versions of the program are generated-executed by applying many compiler transformations, at all different combinations. Iterative compilation requires extremely long compilation times to decrease the exploration space iterative compilation is applied with machine learning compilation techniques. The five major iterative compilation drawbacks are referred to the introduction. The proposed methodology achieves up to 2.1 times lower execution time and an orders of magnitude lower compilation time (Section 4).

The state of the art software libraries, such as ATLAS [21], GotoBLAS2 [22], Eigen [23], Intel_MKL [24], PHiPAC [25], FFTW [26], OpenCV [27] and SPIRAL [28], manage to find a near-optimum binary code for a specific application by using a large exploration space (many different executables are tested and the fastest is picked). Although they achieve high speed, they are application specific and the final schedule is found mostly by using heuristics and empirical techniques. A comparison with the above libraries would be unfair because they use the SIMD (Single Instruction Multiple Data) vector instructions (they support load/store and arithmetical instructions with 128/256-bit data); however, our future work includes the support of SIMD instructions. In [29] [30] [31] [32], we have developed algorithm specific methodologies (we used the SIMD instructions), which produce lower execution time, lower compilation time and lower number of data accesses, than ATLAS [29] [30], FFTW [30] and OpenCV [32]. A comparison between the proposed methodology and [29] [30], is made in Section 4.

Furthermore, many sub-optimum methods exploiting the memory hierarchy have been analyzed in the past, such as [33] [34] [35] [36] [37] [38]. These works apply compiler transformations to the original code (this is not performance efficient). The cache performance optimizations and compiler techniques are presented in [39] and [40]. Finally, regarding data cache miss elimination methods, much research has been done in [41] [42] [43] [44] [45] [46] [47].



```
for {
  for {
    for {
      S1;
      S2;
    }
  }
}
```

```
  for {
    S1;
    for {
      S2;
    }
    for {
      S3;
    }
    S4;
  }
```

Figure 1: Perfectly and imperfectly nested loops are shown at (a) and (b), respectively.

Regarding register allocation problem, many methodologies exist such as [48] [49] [50] [51] [52] [53] [54]. In [48] - [52], data reuse is not taken into account; this means that the array references are assigned into registers, without taking into account that some are accessed a lot and others do not. In [53] and [54], data reuse is taken into account either by greedily assigning the available registers to the data array references or by applying loop unroll transformation to expose reuse and opportunities for maximizing parallelism. In contrast to the proposed methodology, the [48] - [54] address the register allocation problem without taking into account the scheduling problem; instead of finding a good schedule that achieves data reuse and then apply register allocation, they just apply register allocation to the given schedule.

3. Proposed Methodology

The proposed methodology takes C-code and the memory architecture parameters as input, and automatically produces a new faster C-code. The software information-characteristics, i.e. data reuse, production-consumption of intermediate results (when a datum is produced it is directly consumed, e.g. $C[k] = C[k] + ...$), data dependences, array subscript equations, existence of common array references, loop iterators, loop bounds, and the major memory architecture parameters, i.e. number of data cache memories, data cache sizes, data cache associativities, data cache line sizes, register file size, are fully exploited.

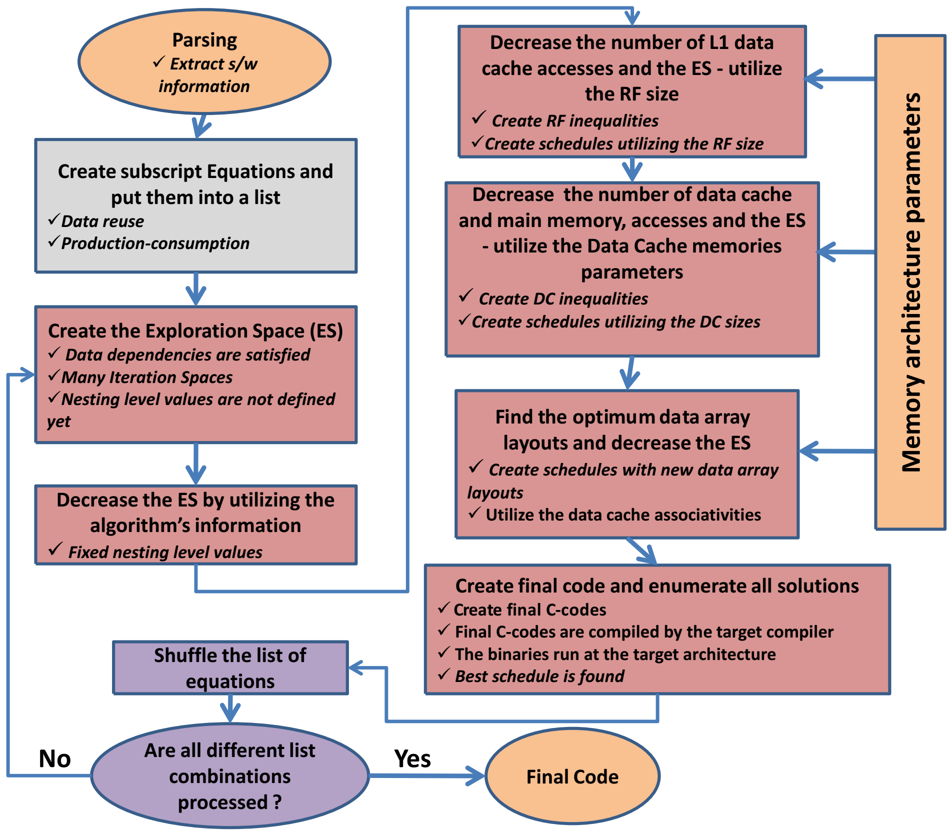

Parsing
✓ Extract s/w information
Create subscript Equations and put them into a list
✓ Data reuse
✓ Production-consumption
Create the Exploration Space (ES)
✓ Data dependencies are satisfied
✓ Many Iteration Spaces
✓ Nesting level values are not defined yet
Decrease the ES by utilizing the algorithm's information
✓ Fixed nesting level values
Decrease the number of L1 data cache accesses and the ES - utilize the RF size
✓ Create RF inequalities
✓ Create schedules utilizing the RF size
Decrease the number of data cache and main memory, accesses and the ES - utilize the Data Cache memories parameters
✓ Create DC inequalities
✓ Create schedules utilizing the DC sizes
Find the optimum data array layouts and decrease the ES
✓ Create schedules with new data array layouts
✓ Utilize the data cache associativities
Memory architecture parameters
Create final code and enumerate all solutions
✓ Create final C-codes
✓ Final C-codes are compiled by the target compiler
✓ The binaries run at the target architecture
✓ Best schedule is found
Shuffle the list of equations
Are all different list combinations processed ?
No
Yes
Final Code

Figure 2: Flow graph of the proposed methodology.

The proposed methodology optimizes source code which contains loops (loop kernels); as it is well known, 90% of the execution time of a computer program is spent executing 10% of the code (also known as the 90/10 law) [55]. We take a loop kernel as input and we produce a new loop kernel which cannot be given by applying the existing transformations to the original code. The methodology optimizes both perfectly and imperfectly nested loops (Fig. 1), which i) no if-condition exists (if they do, current expression is skipped), ii) all the array subscripts are linear equations of the iterators (which in most cases do). Each loop kernel is optimized separately; each loop kernel may contain either perfectly or imperfectly nested loops (Fig. 1).

The proposed methodology is shown in Fig. 2. All the steps are automatic. Firstly, parsing is done; the loops, the loop bounds, the array references,

```
for (i=0; i≤9; i++)
 for (j=0; j ≤9; j++)
  for (k=0; k ≤9; k++)
     ...=A[2*i + j] + B[i][j];
     ...=A[k];
```

$2*i + j=c_1$, $0 \le c_1 \le 27$ (1)

$i=c_{21}$ and $j=c_{22}$, $0 \le c_{21}, c_{22} \le 9$ (2)

$k=c_3$, $0 \le c_3 \le 9$ (3)

$2*i + j - k=c_4$, $-9 \le c_4 \le 27$ (4)

Figure 3: The three first equations contain the separate data reuse of the three array references respectively, while the fourth equation contains the data reuse between the two different references of the array A.

the data dependences, the subscript equations etc, are identified. Then, one mathematical equation is created for each array's subscript and one for each two common array references (e.g. eq.(4) in Fig. 3, it is explained in Subsect. 3.3); each equation defines the memory access pattern of the specific array reference; data reuse is found by these equations. Given that the memory access pattern of each array is given by its subscript equation, we claim that all memory efficient solutions can be produced by processing these equations (Subsect.3.2). After all the equations have been created, all the equations are processed one by one, at all different combinations (Fig. 2), to examine all possible solutions (Subsect.3.2).

For each different combination (e.g. eq.(3), eq.(2), eq.(1) and eq.(4), in Fig. 3), all the equations are processed one by one, creating the new iteration space (Subsect. 3.4); the iteration space is defined by the iterators used and their nesting level values. Each equation inserts its iterators or even new iterators into the new iteration space; for an iteration space to be created all the equations must be fetched.

Afterwards, the exploration space is decreased by applying constraint propagation to the software and hardware parameters. Regarding the software parameters (Subsect.3.3), the exploration space is decreased by fixing the iterators nesting level values (all the schedules having different nesting level values than these the proposed methodology gives, are not considered, decreasing the exploration space). Regarding the hardware parameters, we apply loop tiling for each memory (including the register file) by producing register file and data cache inequalities (Subsect. 3.6 and Subsect. 3.7, respectively), which contain all the (near)-optimum tile sizes. Then, for each schedule has been produced so far, the (near)-optimum data array layouts are found (Subsect. 3.8); the proposed methodology selects both the schedules with the new and the default data array layouts, as by changing the layouts an additional cost is added which may degrade performance. All the schedules with different tile sizes and data array layouts than these the proposed methodology gives, are not considered, decreasing the exploration space. Finally, all these schedules are transformed into C-code, they are compiled by the target compiler and the output binaries are run to the target platform in order to find the one with the best performance (Subsect. 3.9).

The remainder of the proposed methodology has been divided into ten sub-sections. The first subsection contains the basic definitions and notations. The second presents a new loop transformation and the other seven ones explain in more detail the most complex steps of the proposed methodology (Fig 2). Finally the tenth subsection gives an example.

3.1. Definitions and Notations

Definition 1. *Equations which have more than one solutions for at least one constant value, are named type2 equations. All others, are named type1 equations, e.g. eq.(1) and eq.(4) in Fig. 3 are type2 equations, while eq.(2) and eq.(3) are type1 equations.*

Arrays with type2 equations fetch their elements more than once, even if no other/extra iterator exists (the loop kernel contains only the type2 equation iterators), e.g. $2i + j = 7$ holds for several iteration vectors (data reuse); on the other hand, if no extra iterator exists, arrays with type1 equations fetch their elements only once. However, both type1 and type2 arrays may fetch their elements more than once because of the existence of another loop iterator(s) above/between from/of theirs; for example, although eq.(3) in Fig. 3 is of type1, each element of $A[k]$ is fetched 100 times because of the presence of i, j iterators.

To sum up, arrays with type2 equations achieve data reuse at all cases, while arrays with type1 equations achieve data reuse only at the case that extra iterators exist.

The arrays are classified into category-1 and category-2 arrays.

Definition 2. *The arrays whose elements achieve data reuse are classified into category-1 arrays*

Definition 3. *The arrays whose elements do not achieve data reuse or data reuse cannot be exploited, are classified into category-2 arrays*

Definition 4. *The arrays whose subscript equations are of type1 and they contain all the loop kernel iterators (no extra iterator exists), are further classified into Category-2a arrays.*

Statement 1. *The Category-2a arrays fetch their elements just once (there is no data reuse).*

Proof 1. *The subscript equations of these arrays change their values in each iteration vector and thus a different element is fetched in each iteration.*

Definition 5. *The arrays whose subscript equations are not given by a compile time known expression (e.g. they depend on the input data), are further classified into Category-2b arrays.*

Statement 2. *Data reuse of Category-2b arrays cannot be exploited, as the arrays elements are not accessed according to a mathematical formula.*

Definition 6. *If all the iterators of an equation, exist in this equation only and not in another equation, then these iterators are named unique iterators, e.g. the i, j iterators of eq.(1) in Fig. 7 are unique, while the i, j iterators of eq.(1) in Fig. 3 are not.*

3.2. Proposed Loop Transformation

As it has been explained in the previous subsection, arrays with type2 equations fetch their elements more than once (data reuse), even if no extra iterator exists. For example, the iteration vectors ($S = (i, j, k)$) fetching $A[4]$ of eq.(1) in Fig. 3, are more than one (data reuse), i.e. $2 * i + j = 4$ holds for $S1 = (0, 4, X)$, $S2 = (1, 2, X)$ and $S3 = (2, 0, X)$, where X are all the valid k values. In this subsection, we propose a new loop transformation in order to exploit the data reuse of type2 equations. The new transformation treats the type2 equation as a Linear Diophantine Equation (LDE); the solution of an LDE is a mathematical expression which gives the exact iteration vectors that each array's element is fetched only once, e.g. if the proposed transformation is applied on $2 * i + j = c1$ of Fig. 3, each element of $A[2 * i + j]$ is accessed only once.

Statement 3. *The proposed transformation is applied to type2 equations only and gives the minimum number of data accesses of the specific type2 array reference.*

Proof 2. *The minimum number of data accesses is achieved because each array's element is accessed only once.*

The proposed transformation can be applied only if there are no loop carried data dependencies.

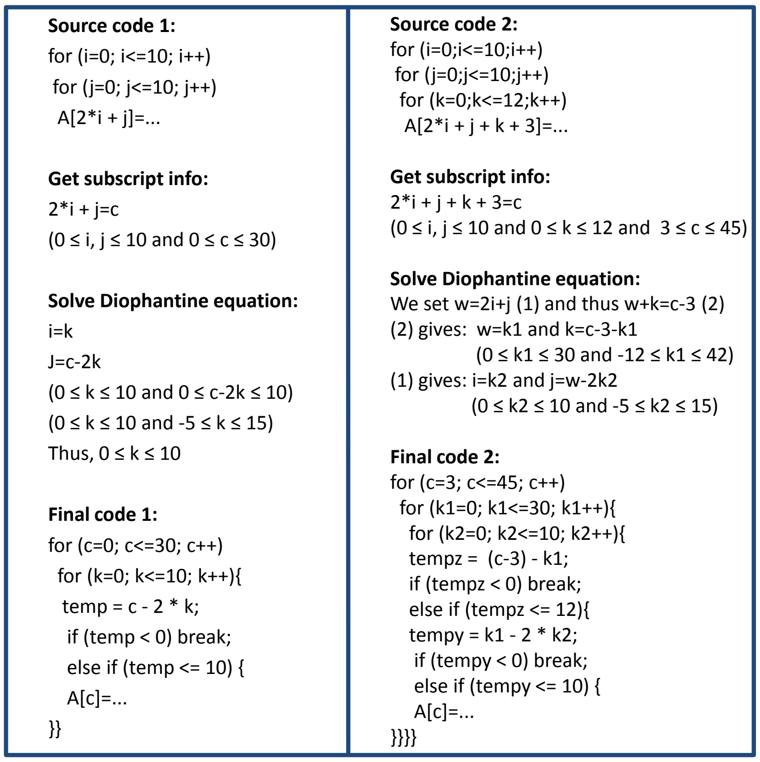
Let us give two examples, Fig. 4. The equation produced by the array subscript, gives all the information needed about the data reuse of array A. This equation is treated as a LDE here; its solution gives the i, j values that c is a constant value, e.g. $c = 10$ for all $0 \leq k \leq 10$ values giving a j value within its bounds (source code 1 in Fig. 4). The solution of the Diophantine equation and the iterator bounds, give a new iteration space which all the array elements are fetched just once; i and j iterators are replaced by k and c iterators. To transform these equations into source code, a new iterator is added into the source code (c iterator) for all the array elements to be accessed in order. The added if-condition statements are necessary. In general, one if-condition statement is needed for each iterator, e.g. at source code 1, if-condition statements for both i and j iterators are needed; however, in most cases it is not necessary to add an if-condition statement for all the iterators because the Diophantine independent variable equals to the iterator, e.g. i=k in source code 1. Thus, the iterator bounds are preserved by the loop bounds and no if-condition statement is needed. The break statements have been inserted to decrease the number of idle iteration vectors.

If the proposed transformation is applied, the number of data accesses is minimized and the data cache lines utilization is increased since all the array's elements are fetched in order and thus from consecutive memory locations. However, the number of arithmetical instructions is increased (extra addressing and branch instructions). The schedule with the minimum number of data accesses does not always provide the best performance, since the number of extra instructions may degrade performance; the best performance depends on the target architecture. This is why type2 equations are treated both as LDE equations and not. The two codes in Fig. 4 are the schedules with the minimum number of data accesses.

3.3. Create Equations

At this step, each array's subscript is transformed into a mathematical equation (Fig. 3).

Statement 4. *Each subscript equation separately, gives the data reuse / production-consumption of the specific array.*



Source code 1:

```
for (i=0; i<=10; i++)
 for (j=0; j<=10; j++)
  A[2*i + j]=...
```

Get subscript info:

2*i + j=c

(0 ≤ i, j ≤ 10 and 0 ≤ c ≤ 30)

Solve Diophantine equation:

i=k

J=c-2k

(0 ≤ k ≤ 10 and 0 ≤ c-2k ≤ 10)

(0 ≤ k ≤ 10 and -5 ≤ k ≤ 15)

Thus, 0 ≤ k ≤ 10

Final code 1:

```
for (c=0; c<=30; c++)
  for (k=0; k<=10; k++){
   temp = c - 2 * k;
    if (temp < 0) break;
    else if (temp <= 10) {
    A[c]=...
}}
```

Source code 2:

```
for (i=0;i<=10;i++)
 for (j=0;j<=10;j++)
  for (k=0;k<=12;k++)
   A[2*i + j + k + 3]=...
```

Get subscript info:

2*i + j + k + 3=c

(0 ≤ i, j ≤ 10 and 0 ≤ k ≤ 12 and 3 ≤ c ≤ 45)

Solve Diophantine equation:

We set w=2i+j (1) and thus w+k=c-3 (2)

(2) gives: w=k1 and k=c-3-k1

(0 ≤ k1 ≤ 30 and -12 ≤ k1 ≤ 42)

(1) gives: i=k2 and j=w-2k2

(0 ≤ k2 ≤ 10 and -5 ≤ k2 ≤ 15)

Final code 2:

```
for (c=3; c<=45; c++)
  for (k1=0; k1<=30; k1++){
    for (k2=0; k2<=10; k2++){
    tempz =  (c-3) - k1;
    if (tempz < 0) break;
    else if (tempz <= 12){
    tempy = k1 - 2 * k2;
     if (tempy < 0) break;
     else if (tempy <= 10) {
     A[c]=...
}}}}
```

Figure 4: Proposed transformation - scheduling with the minimum number of data accesses.

It is obvious that the memory access pattern of each array reference is given by its subscript equation.

Statement 5. *The interaction of two or more equations gives i) the data reuse produced between the common array references (e.g. eq.(4) in Fig. 3) and ii) the interaction among the arrays data, i.e. by fetching one array's element, other array elements are consequently fetched.*

Regarding (i), in the case that there are two array references of the same array, an additional equation is always created to give the iteration vectors that both references access identical elements, e.g in Fig. 3, $A[2]$ is fetched by S1=(0,2,X) and S2=(1,0,X) according to eq.(1) and also $A[2]$ is fetched by S3=(X,X,2) according to eq.(3), where $X = [0, 9]$. Regarding (ii), it is obvious that by fetching one array's element, other array elements are consequently fetched, e.g. by fetching $B(2, 3)$ in Fig. 3, $A(7)$ and $A(0 : 9)$ are fetched because of the first and the second array reference respectively.

Rule 1. *If there are two array references of the same array, an additional equation is created to describe the data reuse between these two references, e.g. eq.(4) in Fig. 3. These equations are further classified into type3 equations.*

Rule 2. *Regarding 2-d arrays, two equations are created and not one because the data array layout has not been found yet, e.g. if* $9 * i + j = c2$ *is taken instead of* $i = c21$ *and* $j = c22$ *for eq.(2) in Fig. 3, then row-wise layout is taken which may not be efficient.*

The Type1, type2 and type3 equations, are treated differently (Subsections 3.3.1- 3.3.3).

3.3.1. Type1 equations

All type1 equations add their iterators into the new iteration space (their nesting level values are found next), e.g. eq.(2) of Fig. 3, gives either $S_1 = (i, j)$ iteration space or $S_2 = (j, i)$.

3.3.2. Type2 equations

Type2 equations are treated in two different ways, i.e. they are treated either as type1 equations or as Linear Diophantine Equations (LDE) (the transformation of Subsect. 3.2 is applied). In the second case, each element of the current equation array, is now accessed only once (optimum data reuse), e.g. if the proposed transformation is applied on $2 * i + j = c1$ of Fig. 3, each element of $A[2 * i + j]$ is accessed only once. Type2 equations are treated in two different ways because the optimum data reuse of one array does not always provide the optimum data reuse or the best performance.

Rule 3. *type1 equations give their iterators into iteration space, while type2 equations give either their iterators (they are treated as type1 equations) or new ones (the proposed transformation is applied, Statement 3), into iteration space.*

3.3.3. Type3 equations

Type3 equations contain the iteration vectors that both two array references fetch the identical elements (data reuse), e.g. $A[2]$ is accessed by $(0, 2, X)$ and $(1, 1, X)$ because of the $A[i + j]$ reference and by $(X, X, 2)$ because of the $A[k]$ reference in Fig.2. This kind of data reuse is fully exploited too, by treating the type3 equations as LDE. Let us give an example, Fig. 5. The equation giving the iteration vectors that both two references fetch the identical array elements, is $i + j - k = c$ when $c = 0$. When $c = 0$ or $k2 + tempy = tempz$ (final code 0 when $c = 0$ or final code 1, Fig. 5), only common elements are loaded. However, there are elements do not loaded in identical iteration vectors, i.e. $c \neq 0$ in final code 0, and thus $i + j - k = c$.

Rule 4. *type3 equations are treated as LDE equations only; the proposed transformation (Statement 3) is applied to type3 equations giving new iterators*

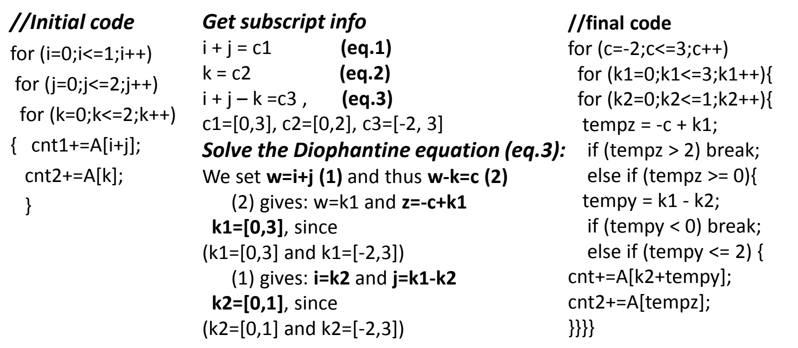
3.4. Find the exploration space

At this step the exploration space is found by processing all subscript equations. Firstly, the iteration space is created.

Statement 6. *The iteration space is created by processing all subscript equations.*

Given that the subscript equations give all the data access patterns and data reuse, we process all subscript equations to find all memory efficient solutions.

To create the iteration space, all equations are processed one by one according to the Rule 3 and Rule 4. The outermost iterators (smallest



```
//Initial code
for (i=0;i<=1;i++)
 for (j=0;j<=2;j++)
  for (k=0;k<=2;k++)
{  cnt1+=A[i+j];
   cnt2+=A[k];
   }
```

```
Get subscript info
i + j = c1          (eq.1)
k = c2              (eq.2)
i + j – k =c3 ,     (eq.3)
c1=[0,3], c2=[0,2], c3=[-2, 3]
Solve the Diophantine equation (eq.3):
We set w=i+j (1) and thus w-k=c (2)
     (2) gives: w=k1 and z=-c+k1
  k1=[0,3], since
(k1=[0,3] and k1=[-2,3])
     (1) gives: i=k2 and j=k1-k2
  k2=[0,1], since
(k2=[0,1] and k2=[-2,3])
```

```
//final code
for (c=-2;c<=3;c++)
  for (k1=0;k1<=3;k1++){
  for (k2=0;k2<=1;k2++){
   tempz = -c + k1;
    if (tempz > 2) break;
    else if (tempz >= 0){
   tempy = k1 - k2;
    if (tempy < 0) break;
    else if (tempy <= 2) {
cnt+=A[k2+tempy];
cnt2+=A[tempz];
}}}}
```

Figure 5: Proposed transformation; the common elements of $A[i+j]$ and $A[k]$ are accessed just once.

nesting level values) of the new loop-kernel are these, whose equation has been processed first, e.g. if eq.(2) of Fig. 3 is processed first, the iteration space is either $S_1 = (i, j)$ or $S_2 = (j, i)$. The iterators with the next larger nesting level values are these, whose equation has been processed second etc, e.g. if eq.(3) of Fig. 3 is processed after eq.(2), the iteration space is either $S_1 = (i, j, k)$ or $S_2 = (j, i, k)$.

Statement 7. *All the equations are processed (according to Rule 3 and Rule 4) one by one, by using all the different combinations, suffice the data dependences are preserved. We process all different combinations in order to examine all memory efficient solutions.*

For a subscript equation, the sooner it is fetched, the better it is treated. This is because i) an array with small nesting level iterator values (its iterators are the upper ones) is fetched less times than one with large ones, e.g. in Fig. 6, 'A' array is fetched only once while 'B' is fetched N^2 times; this is because compilers apply scalar replacement transformation ($A(i, j)$ is replaced by a variable), and ii) for an equation whose iterators have not been assigned into iteration space, we can apply the proposed transformation (Statement 3), decreasing the number of the specific array's accesses.

Let us give an example, Fig. 3. The four equations of Fig. 3 give 7 different iteration spaces, i.e. $S_1 = (i, j, k)$, $S_2 = (j, i, k)$, $S_3 = (c, p, k)$, $S_4 = (k, c, p)$ (where c and p are the new iterators created, if the proposed transformation is applied to eq.(1)), $S_5 = (k, i, j)$, $S_6 = (k, j, i)$ and $S_7 = (c, k1, k2)$ (where $c, k1, k2$ are the new iterators created, if the proposed transformation is applied to eq.(4)).

If there are N different equations, there are up to $N!$ different equation combinations; in practice, the number of different combinations, is smaller than $N!$ because i) identical schedules are produced and ii) data dependences may prohibit some combinations.

Statement 8. *The iteration space created according to statement 7, is further increased and it contains an enormous number of different schedules (exploration space)*

This is because i) loop tiling can be applied for all memories (new iterators are inserted), ii) loop tiling can be applied for all different tile sizes and shapes, iii) the new iterators can take all the different nesting level values, iv) all the array references can be replaced by a different number of variables/registers, v) many different data array layouts can be used.

Statement 9. *The exploration space is decreased by orders of magnitude*

The exploration space is decreased by orders of magnitude because a) only the iteration spaces produced by Statement 7 are considered, b) all the schedules with different number of assigned variables/registers, tile sizes, nesting level values and data array layouts, than these the proposed methodology gives, are not considered.

3.5. Decrease the exploration space by utilizing the algorithm's information

As it has been already mentioned in the previous subsection, by creating the exploration space according to Statement 7, the solutions achieve low data reuse are not examined, decreasing the exploration space; for example, for the Gaussian Blur algorithm, the iteration spaces which are tested and these which are excluded, are shown in Table 2 (Subsection 3.10).

However, the exploration space is decreased even more by utilizing the software characteristics; the nesting level values of the iterators created according to Statement 7, Rule 3 and Rule 4, become fixed (Rules 5, 6). All the schedules having different nesting level values than these the proposed methodology gives, are not considered, decreasing the exploration space even more (Rule 5 and Rule 6).

The exploration space is decreased according to the Rules 5, 6.

Rule 5. *The unique iterators are not interchanged with iterators of another equation.*

Source code:

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
    for (m=0; m<N; m++)
A[i][j] += B[k][m] + k * C[j][m];
```

The equations are:

$i=c_{11}$ & $j=c_{12}$ (1)
$k=c_{21}$ & $m=c_{22}$ (2)
$j=c_{31}$ & $m=c_{32}$ (3)

For equations' order
eq.1, eq.2, eq.3,

eq. (1) gives:
S_1 = (i, j, -, -)

eq. (2) gives:
S_2 = (-, -, k, m)

eq. (3) gives:
S_3 = (i, j, k, m)
S_4 = (j, i, k, m)
S_5 = (i, j, m, k)
S_6 = (j, i, m, k)

Figure 6: An example, eq.(3) gives different iterators nesting levels.

In general, each array is accessed $(q \times r)$ times, where q is the number of the iterations exist above its upper iterator and r is the number of the iterations exist between its upper and lower iterators. It can be easily be proved that if the unique iterators change their nesting level values without satisfying Rule 5, either the $(q \times r)$ value increases or the iterators nesting level values are given by processing another equations' combination (it is not an issue here).

Rule 6. *The nesting level values of the unique iterators are defined according to the target compiler.*

According to Rule 5, the unique iterators of an equation are not interchanged with iterators of another equation. Furthermore, in the case that they are interchanged with each other, the number of load/store and addressing instructions will remain constant; only the number of data cache misses changes. The number of cache misses changes because multi-dimensional arrays, access their elements from no consecutive main memory locations, e.g. if i and j iterators in Fig. 6 are interchanged, A is no further accessed row-wise but column-wise from main memory; however, the data array layout is found next. Apart from reducing the number of data cache misses, there is no use to interchange these iterators. Thus, only these nesting level values accessing the array row-wise are taken for now (C compiler stores the arrays row-wise in main memory), decreasing the number of the data cache misses, e.g. for $A(i, j)$, the i iterator is defined as the outermost one.

3.6. Decrease the number of L1 data cache accesses and the exploration space - utilizing the Register File (RF) size

At this step, the Register File (RF) size and the subscript equations are fully exploited, decreasing the number of L1 data cache accesses and the exploration space. For each iteration space has been created so far, loop tiling for the RF is applied. To utilize the RF size, RF inequalities are produced giving all the (near)-optimum tile sizes. These inequalities contain i) the number of the registers needed for each array reference and for scalar variables, ii) the shape of each array's tile.

The register file inequality is given by:

$$0.8 \times RFs \leq L_{iter} + Var + ws + R_1 + R_2 + ... + R_n \leq 1.2 \times RFs \quad (1)$$

where RFs is the number of the available registers, L_iter is the number of the different iterator references exist in the loop body, Var is the number of scalar variables, ws is the number of the working space registers, i.e. variables for intermediate results and R_i is the number of the variables/registers allocated for the i-th array.

R_i is given by: $R_i = it'_1 \times it'_2 \times ... \times it'_n$, where the integer it'_i are the unroll factor values of the iterators exist in the array's subscript, e.g. for $B(i,j)$ and $C(i,i)$, $R_B = i' \times j'$ (rectangular tile) and $R_C = i'$ (diagonal line tile) respectively, where i' and j' are the unroll factors of i, j iterators.

Rule 7. *Each subscript equation contributes to the creation of ineq.(1), i.e. equation i gives R_i and specifies its expression.*

The iterators are tiled and the new tiled iterators are fully unrolled, according to the RF size, to exploit data reuse; in this way, the registers are reused as many times as the number of the available registers indicate (register utilization).

Let us give an example. In Fig. 7-a, $R_A = i' \times j'$ and $R_B = k'$ variables/registers are allocated for A and B arrays, respectively. If we choose a square tile for array A of size 2×2, i.e. 4 registers for A, and only 1 register for B, then the i, j iterators are tiled and the new tiled iterators are fully unrolled (Fig. 7-b). Then, by assigning the array references into registers, data reuse is achieved (Fig. 7-c). In Fig. 7-a, the (A,B) arrays are accessed $(1, N^2)$ times while in Fig. 7-c $(1, N^2/4)$ times, respectively.

Rule 8. *For each iteration space has been created according to Statement 7, loop tiling for the RF is applied; this means that new iterators are created with loop bounds equal to the tile sizes; the new iterators are fully unrolled and all the array references are replaced by scalar variables according to the RF size and to the subscript equations, achieving data reuse.*

```
for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
   for (k=0;k!=N;k++)
     ... = A[i][j] + B[k];
```

(a)

```
for (i=0;i!=N;i+=2)
 for (j=0;j!=N;j+=2)
   for (k=0;k!=N;k++) {
     ... = A[i][j]     + B[k];
     ... = A[i][j+1]   + B[k];
     ... = A[i+1][j]   + B[k];
     ... = A[i+1][j+1] + B[k];
     }
```

(b)

```
for (i=0;i!=N;i+=2)
 for (j=0;j!=N;j+=2) {
 reg0=A[i][j];   reg1=A[i][j+1];
 reg2=A[i+1][j]; reg3=A[i+1][j+1];
   for (k=0;k!=N;k++) { reg4=B[k];
     ... = reg0 + reg4;
     ... = reg1 + reg4;
     ... = reg2 + reg4;
     ... = reg3 + reg4;
     }    }
```

(c)

Figure 7: An example, tiling for the RF is applied - we assign 4 registers for A (square tile of size 2×2) and 1 for B.

The bound values of the register file inequality (eq.(1)) are not tight because the output code is C-code and during its compilation (translate the C-code into binary code), the compiler may not allocate the exact number of desirable addressing variables into registers. However, if assembly code would be produced instead of C-code, the register utilization would be the optimum.

The number of L_iter and ws registers is found after the allocation of the array elements into variables/registers, because they depend on the number of tiled iterators. The ws value depends on the target compiler and this is why it is found approximately; the bounds of the RF inequality are not tight for this reason too. The goal is to store all the inner loop reused array elements and scalar variables into registers minimizing the number of register spills.

Statement 10. *All schedules satisfying ineq.(1), decrease the number of L1 data cache accesses.*

Proof 3. *The number of L1 data cache accesses is decreased for two reasons. First, loop tiling for the RF is applied. In general, by applying loop tiling for the RF, the number of L1 data cache accesses is decreased, as parts (tiles) of the arrays remain in the RF (data reuse) and therefore they are not fetched many times from L1. Second, loop tiling is applied according to the RF size and to the subscript equations (memory access patterns). The new iterators are fully unrolled and all the array references are replaced by scalar variables; each variable in the loop body corresponds to a register, minimizing the number of register spills. The larger the RF size, the larger the tiles used and the larger the data reuse being achieved.*

Statement 11. *All schedules with different number of assigned variables/registers than these the proposed methodology gives, are not considered, decreasing the exploration space.*

Rule 9. *Each different set of it'_i values satisfying ineq.(1), gives a different schedule. All different it'_i values satisfying ineq.(1) are examined.*

Rule 10. *The R_i values of ineq.(1), are given by Rules 11- 18 .*

Rule 11. *The innermost iterator is never tiled because data reuse is decreased; if it_i is the innermost iterator, then $it'_i = 1$.*

Proof 4. *By tiling the innermost iterator, e.g. iterator k in Fig. 7, the array references-equations which contain it, will change their values in each iteration; this means that i) a different element is accessed in each k iteration and thus a huge number of different registers is needed for these arrays, ii) all these registers are not reused (a different element is accessed in each iteration). Thus, by tiling the innermost iterator, more registers are needed which do not achieve data reuse; this leads to low RF utilization.*

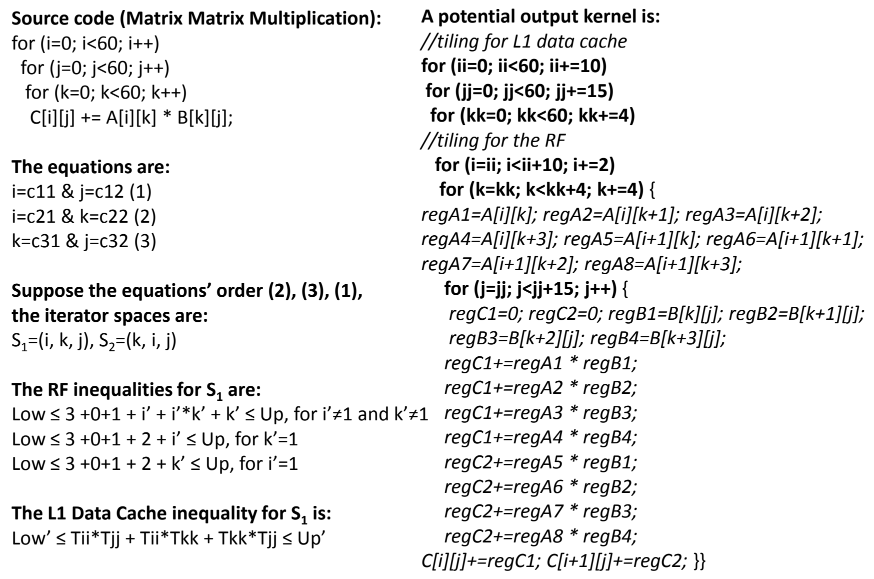
Rule 12. *The type1 array references which contain all the loop kernel iterators, do not achieve data reuse; thus only one register is needed for these arrays, i.e. $R_i = 1$*

Proof 5. *The subscript equations of these arrays change their values in each iteration vector and thus a different element is fetched in each iteration.*

Rule 13. *If the proposed transformation (Statement 3) is applied to eq.(i), then only one register is needed for this array reference, i.e. $R_i = 1$.*

Proof 6. *In this case, the optimum data reuse for this array is achieved since each array's element is fetched just once (all array elements are fetched in-order); thus only one register is needed for this array, e.g. in Fig. 4, $A[c]$ needs only one register.*

Rule 14. *If the proposed transformation (Statement 3) is applied to eq.(i), then the eq.(i) iterators are never tiled and thus $it'_i = 1$. Otherwise, the proposed transformation may be invalid.*



Source code (Matrix Matrix Multiplication):

```
for (i=0; i<60; i++)
 for (j=0; j<60; j++)
  for (k=0; k<60; k++)
   C[i][j] += A[i][k] * B[k][j];
```

The equations are:

```
i=c11 & j=c12 (1)
i=c21 & k=c22 (2)
k=c31 & j=c32 (3)
```

Suppose the equations' order (2), (3), (1), the iterator spaces are:

S_1=(i, k, j), S_2=(k, i, j)

The RF inequalities for S_1 are:

```
Low ≤ 3 +0+1 + i' + i'*k' + k' ≤ Up, for i'≠1 and k'≠1
Low ≤ 3 +0+1 + 2 + i' ≤ Up, for k'=1
Low ≤ 3 +0+1 + 2 + k' ≤ Up, for i'=1
```

The L1 Data Cache inequality for S_1 is:

```
Low' ≤ Tii*Tjj + Tii*Tkk + Tkk*Tjj ≤ Up'
```

A potential output kernel is:

```
//tiling for L1 data cache
for (ii=0; ii<60; ii+=10)
 for (jj=0; jj<60; jj+=15)
  for (kk=0; kk<60; kk+=4)
//tiling for the RF
   for (i=ii; i<ii+10; i+=2)
    for (k=kk; k<kk+4; k+=4) {
regA1=A[i][k]; regA2=A[i][k+1]; regA3=A[i][k+2];
regA4=A[i][k+3]; regA5=A[i+1][k]; regA6=A[i+1][k+1];
regA7=A[i+1][k+2]; regA8=A[i+1][k+3];
     for (j=jj; j<jj+15; j++) {
      regC1=0; regC2=0; regB1=B[k][j]; regB2=B[k+1][j];
      regB3=B[k+2][j]; regB4=B[k+3][j];
     regC1+=regA1 * regB1;
     regC1+=regA2 * regB2;
     regC1+=regA3 * regB3;
     regC1+=regA4 * regB4;
     regC2+=regA5 * regB1;
     regC2+=regA6 * regB2;
     regC2+=regA7 * regB3;
     regC2+=regA8 * regB4;
C[i][j]+=regC1; C[i+1][j]+=regC2; }}
```

Figure 8: An example, Matrix Matrix Multiplication (MMM) algorithm.

Let us give an example, Fig. 3. If the proposed transformation is applied to eq.(1), then the B array's iterators cannot be tiled and thus only one variable/register is needed for B array.

Rule 15. *If there is a type1 array reference i) containing more than one iterators and one of them is the innermost one and ii) all ineq.(1) iterators which do not exist in this array reference have unroll factor values equal to* 1*, then only one register is needed for this array, i.e.* $R_i = 1$*. This gives more than one register file inequalities.*

Proof 7. *When Rule 15 holds, a different array's element is fetched in each iteration vector, as the subscript equation changes its value in each iteration. Thus, no data reuse is achieved and only one register is used. On the contrary, in the case that at least one iterator which do not exist in this array reference is tiled, common array references occur inside the loop body (e.g.* $regC1$ *is reused 3 times in Fig. 8); data reuse is achieved in this case and thus another RF inequality is created.*

Let us give an example (Fig. 8). Suppose the S_1 iteration space whose register file inequalities are shown in Fig. 8. The C array subscript contains i and j iterators. j iterator is the innermost one and thus $i' \times 1$ registers are needed for this array; however, according to Rule 15, C array needs $i' \times 1$ registers if $k' \neq 1$ and 1 register otherwise (if $k' = 1$ then the C array fetches a different element in each iteration vector and thus only one register is needed). The array A needs $i' \times k'$ registers while B array needs k' registers if $i' \neq 1$ and 1 register otherwise. Note that if the i-loop is not tiled ($i' = 0$), the B and C array elements are not reused and there is 1 register for C and 1 register for B (Rule 15). The innermost iterator (j) is not tiled according to the Rule 11 (data reuse is decreased in this case).

Rule 16. *We can decrease the number of it'_i values satisfying ineq.(1), by utilizing the L1 data cache line size. Regarding 1-d arrays, we can select the it'_i values to be either* 1 *or multiples of the L1 cache line size. Regarding multidimensional arrays, we can select the it'_i values which correspond to the x-axis, to be either* 1 *or multiples of the L1 cache line size. The arrays must be written into main memory aligned.*

Moreover, there are cases that data reuse utilization is more complicated as common array elements may be accessed not in each iteration, but in each k iterations, where $k \geq 1$. This holds only for type2 equations (e.g. $ai + bj + c$) where $k = b/a$ is an integer (data reuse is achieved in each k iterations). The proposed methodology exploits data reuse only when $k = 1$ here (Rule 17) as for larger k values, the data reuse is low. For example, at Gaussian Blur algorithm (Subsection 3.10), each time the filter is shifted by one position to the right (mc iterator), 20 elements of in array are reused (reuse between consecutive iterations here, i.e. $k = 1$).

Rule 17. *Arrays with type2 subscript equations which have equal coefficient absolute values (e.g. $ai + bj + c$, where $a == \pm b$) fetch identical elements in consecutive iterations; data reuse is exploited by interchanging the registers values in each iteration. An extra RF inequality is produced for this case.*

Proof 8. *The above arrays access their elements in patterns. As the innermost iterator (let j) changes its value, the elements are accessed in a pattern, i.e. $A[p]$, $A[p+b]$, $A[p+2\times b]$ etc. When the outermost iterator changes its value, this pattern is repeated, shifted by one position to the right ($A[p+b]$, $A[p+2\times b]$, $A[p+3\times b]$ etc), reusing its elements. This holds for equations with more than 2 iterators too.*

Table 1: How tiling affects the number of memories accesses and addressing instructions; the more the ticks are, the less the number of accesses / addressing instructions, are.

	L1	L2	Main Mem	Addr
Tiling for RF	✓✓✓	✓	✓	✓✓✓
Tiling for RF, L1	✓✓	✓✓✓	✓✓	✓✓
Tiling for RF and L2	✓✓	✓	✓✓✓	✓✓
Tiling for RF, L1, L2	✓✓	✓✓	✓✓✓	✓

To exploit data reuse of Rule 17, all the array's registers interchange their values in each iteration, e.g. in (Fig. 10 - a1.2), the $(in0, in1, in2, in3, in4, in5)$ variables interchanging their values in each iteration.

Rule 18. *Regarding very small arrays (e.g. filters in image processing algorithms), it is tested whether their iterators are fully unrolled or not (both solutions are taken).*

To sum up, by applying loop tiling for the RF, as explained above, the numbers of i) load/store instructions (or equivalent the number of L1 data cache accesses) and ii) addressing instructions, are decreased. The number of addressing instructions is decreased for three reasons. Firstly, the array references are replaced by scalar variables and thus the address computations are simplified, e.g. $A(i, j)$ is replaced by reg variable. Secondly, several common subscript expressions which occur by unrolling the loops are replaced by scalar variables, e.g. the array references $array(i + j)$, $array(i + j + 1)$, $array(i + j + 2)$ and $array(i + j + 3)$ are replaced by $array(temp)$, $array(temp+1)$, $array(temp+2)$ and $array(temp+3)$ respectively. Thirdly, loop unroll always decreases the number of addressing instructions.

3.7. Decrease the number of data cache and main memory accesses and the exploration space - utilizing the Data Cache memories parameters

The data cache memories sizes and the subscript equations are fully exploited, decreasing the number of data cache / main memory accesses and the exploration space. To utilize the data cache sizes, a data cache inequality for each data cache is produced, giving all the (near)-optimum tile sizes. Each inequality contains i) the tile size of each array and ii) the shape of each array tile.

The proposed methodology holds for each different cache hierarchy. The number of the levels of tiling for data cache is found by testing; for a 2 levels of data cache architecture, 1 level of tiling (either for L1 or L2 data cache), 2 levels of tiling and no tiling solutions, are applied to all the solutions-schedules that have been produced so far. The optimum number of levels of tiling cannot easily be found since the data locality advantage may be lost by the required insertion of extra load/store and addressing instructions, which degrade performance (Table 1). In table 1, we can see how tiling affects the number of data cache accesses and addressing instructions, e.g. if the performance critical parameter is the number of L1 data cache accesses or the number of addressing instructions, then tiling is not applied for data cache. The separate memories optimization gives a different schedule for each memory and these schedules cannot coexist, as by refining one, degrading another; thus, either a sub-optimum schedule for all the memories or a (near)-optimum schedule only for one memory can be produced. However, if the goal is the minimum number of data accesses for a specific memory (let L_i), loop tiling only for L_{i-1} is applied.

For the reminder of this paper, we refer to architectures having separate L1 data and instruction cache (vast majority of architectures). In this case, the program code always fits in L1 instruction cache since we optimize loop kernels only, whose code size is small; thus, upper level unified/shared caches, if exist, contain only data. On the other hand, if a unified L1 cache exists, memory management becomes very complicated.

Loop tiling is applied to category-1 and category-2a arrays only (Subsect. 3.1). The tiles of Category-1 arrays achieve data reuse and therefore they must definitely fit in data cache. Although category-2a tiles are not reused, they have to fit in data cache to avoid cache conflicts with the category-1 ones; in this way Category-1 tiles remain in data cache. Furthermore, Category-2b arrays cannot be partitioned into tiles as their elements are accessed in a 'random' way; this leads to a large number of data cache conflicts due to the cache modulo effect (especially for large arrays). To eliminate these conflicts, Rule 19 is introduced.

Rule 19. *For all the Category-2b arrays, data cache size which equals to the size of one cache way is granted. In other words, an empty cache line is granted for each different modulo (with respect to the size of the cache) of these arrays memory addresses.*

In this way the reused tiles remain in data cache.

Rule 20. *For each register file inequality solution (schedule) produced so far, loop tiling is applied for all data cache memories and for all valid data cache tile sizes (ineq.(2)). All tile sizes do not satisfy ineq.(2), are not considered, decreasing the exploration space.*

The data cache inequality is given by:

$$0.6 \times L_k \times \frac{(assoc - v)}{assoc} \leq Tile_1 + ... + Tile_n \leq 1.1 \times L_k \times \frac{(assoc - v)}{assoc} \quad (2)$$

where L_k is the k-level data cache size, $assoc$ is the data cache associativity (for an 8-way associative data cache, $assoc = 8$). v value is zero when no Category-2b array exist and one if at least one Category-2b array exists. $Tile_i$ is the tile size of the ith array and $Tile_i = T_1' \times T_2' \times T_n' \times ElementSize$, where T_i' is the unroll factor of the i iterator and $ElementSize$ is the size of each array's element in bytes ($Tile_i$ refers only to Category-1 and Category-2a, array). The tiling inequality of Matrix-Matrix Multiplication algorithm is shown in Fig. 8, e.g. $Tile_C = T_{ii} \times T_{jj}$ where $T_{ii} = 10$ and $T_{jj} = 15$.

Regarding data cache tile sizes, they have to be multiples of the RF tiles sizes. Also, the tile sizes produced by L2 data cache, must be multiples of the tiles sizes produced by L1 data cache and RF (otherwise, a large number of addressing instructions is needed). Thus, the exploration space is further decreased.

The inequality bound values are not tight, i.e. 0.6 and 1.1, because smaller/larger tile sizes which divide exactly the array sizes, may achieve a lower number of addressing instructions (e.g. having an array of 2048 elements and only 800 fit in data cache, a tile size of 512 will achieve a lower number of addressing instructions than this of 800).

Statement 12. *Each different set of T_i' values satisfying ineq.(2) gives a different schedule*

Statement 13. *All schedules satisfying ineq.(2) decrease the number of data cache / main memory accesses.*

Proof 9. *Likewise Statement 10.*

Ineq.(2) gives a large number different tile sizes. However, the curve expressing the performance versus the tile sizes is smooth having a small number of change points. It is well known that such functions can be drastically speedup as it is not required to test all its points, i.e. tile sizes.

```
//MVM kernel
for (i=0; i! 100; i+=5) {
reg1=0; reg2=0; reg3=0;
reg4=0; reg5=0;
 for (j=0; j!=100; j++) {
regB=X[j];
  reg1+=A[i][j] * regB;
  reg2+=A[i+1][j] * regB;
  reg3+=A[i+2][j] * regB;
  reg4+=A[i+3][j] * regB;
  reg5+=A[i+4][j] * regB;
}
Y[i] = reg1;
Y[i+1]= reg2;
Y[i+2]= reg3;
Y[i+3]= reg4;
Y[i+4]= reg5; }
```

```
//write the new array, A'
a=0;
for (i=0; i! 100; i+=5)
 for (j=0; j!=100; j++) {
   A'[a]=A[i][j];
   a++;}

//new MVM kernel
a=-5;
for (i=0; i! 100; i+=5) {
reg1=0; reg2=0; reg3=0;
reg4=0; reg5=0;
 for (j=0; j!=100; j++) {
regB=X[j]; a+=5;
  reg1+=A[a] * regB;
  reg2+=A[a+1] * regB;
  reg3+=A[a+2] * regB;
  reg4+=A[a+3] * regB;
  reg5+=A[a+4] * regB;
}
Y[i] = reg1; Y[i+1]= reg2;
Y[i+2]= reg3; Y[i+3]= reg4;
Y[i+4]= reg5; }
```

Figure 9: Two potential output schedules for Matrix Vector Multiplication (MVM) algorithm. The code shown at the right is produced by changing the data array layout of that shown at the left.

Statement 14. *The nesting level values of the new tiling iterators are found theoretically (no exploration is applied)*

The nesting level values of the new (tiling) iterators are computed. For each different schedule produced by ineq.(2), The proposed methodology computes the total number of data accesses for all the different nesting level values and the best are selected.

The number of each array's accesses is found by:

$DataAccesses = n \times Tile\_size\_in\_elements \times Num\_of\_tiles$, where n is the number of times each tile is fetched and equals to $(q \times r)$, where q is the number of iterations exist above the upper iterator of the array's equation and r is the number of iterations exist between the upper and the lower iterators of the array's equation.

It is important to say that tiling is applied for multiple index variable subscripts too (type2 equations), e.g. code shown in Fig. 4; the c iterator is tiled and all elements are fetched just once and in-order (consecutive memory locations). So far, compilers do not apply tiling in such cases.

3.8. Find the optimum data array layouts and decrease the exploration space

At this step, the (near)-optimum data array layouts are found. In this way, the spatial data reuse is further utilized and the number of data cache misses is further decreased. For each schedule produced so far, the data array layouts change; both the schedules with the new data array layouts and not, are propagated to the next step. Both solutions are taken as by changing the data array layout the additional cost of re-writing the array to main memory may be high and the number of addressing instructions is increased. However, there are several cases that by changing the data array layout, the performance is increased, i.e. a) if the array whose layout is changed is reused several times, e.g. MMM, b) if the data array layout is precomputed, c) if the input data are produced by the current application at run time; in this case, the initialization and the change of the layout, are made together, decreasing the overhead.

By changing the data array layout, the tiles are written in consecutive main memory locations, i.e. just as they are fetched (tile-wise) according to the new schedule, to increase main memory page and data cache line utilization. In this way the number of the data cache misses is highly decreased. The multi-dimensional arrays are transformed into 1-d arrays having tile-wise data layout in main memory, e.g. Fig. 9. To change the data array layouts, the array subscripts are changed-simplified, i.e. for each array reference a new variable is replacing the previous subscript equation and extra addressing instructions are inserted; these addressing instructions increase/decrease the subscripts values.

Rule 21. *If the number of the arrays is larger than the data cache associativity value, all the array tiles are stored into one array interleaved and tile-wise to eliminate the number of data cache misses due to the cache modulo effect. To do this, all the arrays are partitioned into identical number of tiles.*

If the number of the arrays is larger than the data cache associativity value, the number of data cache misses is high even if the sum of the arrays size is smaller than the cache size. This is because at least one cache way contains more than one array's cache lines; this means that tiles do not remain in data cache as they conflict with each other due to the cache modulo effect. To overcome this problem, all array tiles are stored into one array interleaved and tile-wise (all tiles are written in consecutive main memory locations).

In general, compilers and related works apply loop tiling without taking into account the data array layouts. In the case that the arrays are not written tile-wise in main memory, tiling cannot give a small number of data cache misses for multi-dimensional arrays because tiles are comprised by array sub-rows which are written in different main memory locations; this means that the sub-rows will conflict with each other and thus the tiles do not remain in data cache.

3.9. Create final code and enumerate all solutions

At this step, all the solutions-schedules that have been produced so far, are transformed into C-code. These codes are compiled by the target architecture compiler and the binaries run to the target platform to find the fastest one. Each binary is run only once. Given the input size and the input data type (e.g. float, double), we do not use different input sets, as the proposed methodology optimizes applications which are not affected by the data values (see second paragraph of Section 3). It is important to say that if the input size or the input data type changes, the whole procedure is repeated, as the (near)-optimum schedule normally changes.

The number of the binaries tested depends on the application and on the hardware parameters; the number of binaries tested are from 1000 up to 100000 (the application execution time affects the compilation time). However, we can find a solution very close to the best very fast, by testing an orders of magnitude lower number of binaries. This is achieved by selecting only a few sets of different tile sizes for the data caches; performance is not highly affected by changing the data cache tile sizes, suffice they satisfy the proposed inequalities, i.e. tiles fit in the cache.

In the case that the schedule with the minimum number of data accesses for a specific memory is needed, the compilation time is very small as i) the procedure explained in Subsect. 3.7 is applied only for this memory (1 level of tiling) and ii) given that only the minimum number of data accesses is needed (the number of instructions is not taken into account here), it is possible to estimate their number, for each schedule, instead of running the schedules on the target platform.

3.10. Motivation Example (Gaussian Blur)

Let us consider the C code shown in Fig. 11-(e) (Gaussian Blur) and an architecture of one level data cache. The equations produced are: ($r =$

Table 2: Iteration spaces for the Gaussian Blur C-code.

iteration spaces	
Different combinations	Iteration spaces (common spaces occur)
eq.(4)-eq.(5)-eq.(6)	$S_1 = (r, c, mr, mc)$,$S_2 = (c, r, mr, mc)$
eq.(4)-eq.(6)-eq.(5)	S_1,S_2,$S_3 = (r, c, mc, mr)$,$S_4 = (c, r, mc, mr)$
eq.(5)-eq.(4)-eq.(6)	$S_5 = (r, mr, c, mc)$,$S_6 = (r, mr, mc, c)$ $S_7 = (mr, r, c, mc)$,$S_8 = (mr, r, mc, c)$ $S_9 = (c21, k1, c22, k2)$
eq.(5)-eq.(6)-eq.(4)	S_5,S_6,S_7,S_8,S_9
eq.(6)-eq.(4)-eq.(5)	$S_{10} = (mr, mc, r, c)$,$S_{11} = (mr, mc, c, r)$ $S_{12} = (mc, mr, r, c)$,$S_{13} = (mc, mr, c, r)$
eq.(6)-eq.(5)-eq.(4)	S_{10},S_{11},S_{12},S_{13}
Iteration spaces which are excluded	
$P_1 = (r, mc, c, mr)$,$P_2 = (r, mc, mr, c)$	
$P_3 = (mc, r, c, mr)$,$P_4 = (mc, r, mr, c)$	
$P_5 = (c, mr, r, mc)$,$P_6 = (c, mr, mc, r)$	
$P_7 = (mr, c, r, mc)$,$P_8 = (mr, c, mc, r)$	
$P_9 = (c, mc, r, mr)$,$P_{10} = (c, mc, mr, r)$	
$P_{11} = (mc, c, r, mr)$,$P_{12} = (mc, c, mr, r)$	
$P_{13} = (c22, k2, c21, k1)$	

$c11, c = c12)$ (eq.(1)), $(r + mr - 2 = c21, c + mc - 2 = c22)$ (eq.(2)), $(mr = c31, mc = c32)$ (eq.(3)).

The proposed methodology processes the subscript equations one by one, for six different combinations, to create all the iteration spaces (Subsection 3.4 and Subsection 3.5). The iteration spaces created, are shown in Table 2 ($S_1 - S_{13}$). Table 2 also shows the iteration spaces which are excluded ($P_1 - P_{13}$), decreasing the exploration space. The number of iteration spaces is further increased (Statement 8).

For each one of the $S_1 - S_{13}$ iteration spaces, the register file size is utilized. Regarding S_1, four register file inequalities are produced. The first of the four is the following:

$0.8 \times RFs \leq 3 + 2 + 2 + r' \times c' + 1 + mr' \leq 1.2 \times RFs$, if $c \neq 1$ or $r \neq 1$

A potential solution ($r' = 2$,$c' = 2$,$mr' = 1$) of the above RF inequality is shown at the left of Fig. 10. $R_{in} = 1$ (only one register is used for this array) because of the Rule 15 and $mc' = 1$ because of the Rule 11. The first 3 values of the RF inequality correspond to the L_{iter}, Var and ws respectively (they are found after the register assignment); $L_{iter} = 3$ because mr, mc and c iterators exist in the innermost loop, $Var = 2$ because $addr1$ and $addr2$ variables exist and $ws = 2$; the size of the ws depends on the target compiler and it is found approximately. The second inequality is given due to the Rule 15 and is the following:

$0.8 \times RFs \leq 4 + 0 + 2 + 1 + 1 + 1 \leq 1.2 \times RFs$, if $c = 1$ and $r = 1$

$R_{mask} = 1$ because of the Rule 15.

Furthermore, if Rule 17 is applied the data reuse between different iterations (registers $in0 - in5$) is exploited too, giving the following inequality (a potential solution of this inequality is shown at the right of Fig. 10).

$0.8 \times RFs \leq 3 + 1 + 2 + r' \times c' + r' \times mr' \times c' + mr' \leq 1.2 \times RFs$, if $c' \succ 1$

The fourth inequality is produced if the $mask$ array iterators are fully unrolled and all the array references are replaced by their constant values (Rule 18 - the $mask$ array does not further exist). In this case the iteration space is reduced to $S_1 = (r, c)$ and $R_{out} = 1$, $R_{in} = 1$.

To sum up, all the different values satisfying the above inequalities are possible solutions. This is repeated for all the iteration spaces, i.e. $S_1 - S_{13}$.

All the schedules produced so far are further tiled according to the L1 data cache size, Subsect. 3.4 (solutions that utilize both the register file and the data cache and solution that utilize only the register file).

The L1 inequality is:

$0.6 \times L1 \leq T_r \times T_c + (T_r + mr) \times (T_c + mc) + T_{mr} \times T_{mc} \leq 1.1 \times L1$

All the tile sizes are multiples of the corresponding register file tile sizes. The nesting level values of the tiling iterators are computed as explained in Subsect.3.4.

Afterwards, all the schedules produced so far change their data array layouts (we examine the schedules with the new data array layouts and the schedules with the default data array layouts). Regarding Gaussian Blur, it is not performance efficient to change the data array layouts (for the vast majority of architectures). Finally, all the above schedules are compiled by the target compiler and they run at the target platform to find the fastest one.

A1.1)

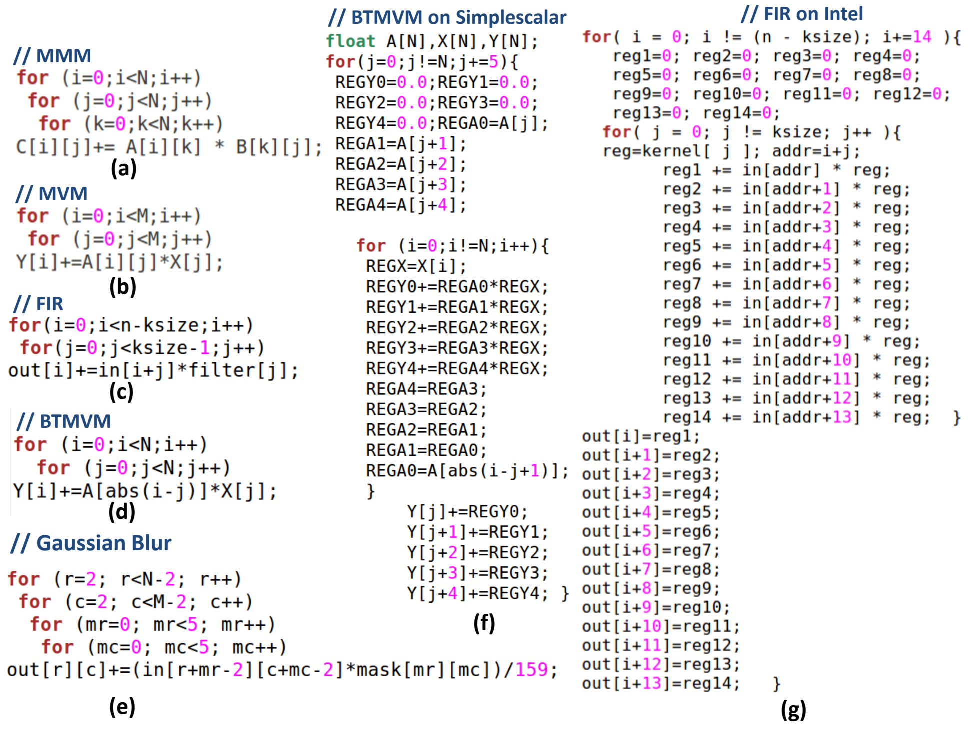
```
for (row = 2; row < N-2; row+=2) {
 for (col = 2; col < M-2; col+=2) {
  out0=0;out1=0;out2=0;out3=0;
     for (mr=0; mr<5; mr++) {addr1=row+mr-2;
     for (mc=0; mc<5; mc++) {
        addr2=col+mc-2;
        reg_mask=mask[mr][mc];

out0 += (in[addr1][addr2] * reg_mask) / 159;
out1 += (in[addr1][addr2+1] * reg_mask) / 159;
out2 += (in[addr1+1][addr2] * reg_mask) / 159;
out3 += (in[addr1+1][addr2+1] * reg_mask) / 159;
                                } }
          out[row][col]=out0;
          out[row][col+1]=out1;
          out[row+1][col]=out2;
          out[row+1][col+1]=out3;
}}
```

A1.2)

```
for (row = 2; row < N-2; row++) {
 for (col = 2; col < M-2; col+=6) {
out0=0;out1=0;out2=0;out3=0;out4=0;out5=0;
   for (mr=0; mr<5; mr++) {
            addr1=row+mr-2;
            in0=in[addr1][col-2];
            in1=in[addr1][col-1];
            in2=in[addr1][col];
            in3=in[addr1][col+1];
            in4=in[addr1][col+2];
    for (mc=0; mc<5; mc++) {
        reg_mask=mask[mr][mc];
         in5=in[addr1][col+3+mc];
            out0 += (in0 * reg_mask) / 159;
            out1 += (in1 * reg_mask) / 159;
            out2 += (in2 * reg_mask) / 159;
            out3 += (in3 * reg_mask) / 159;
            out4 += (in4 * reg_mask) / 159;
            out5 += (in5 * reg_mask) / 159;
    in0=in1; in1=in2; in2=in3; in3=in4; in4=in5;
             } }
         out[row][col]=out0;
         out[row][col+1]=out1;
         out[row][col+2]=out2;
         out[row][col+3]=out3;
         out[row][col+4]=out4;
         out[row][col+5]=out5;
}             }
```

Figure 10: Two potential solutions of the example shown in Subsect. 3.10.



```
// MMM
 for (i=0;i<N;i++)
  for (j=0;j<N;j++)
   for (k=0;k<N;k++)
C[i][j]+= A[i][k] * B[k][j];
```
(a)

```
// MVM
 for (i=0;i<M;i++)
  for (j=0;j<M;j++)
 Y[i]+=A[i][j]*X[j];
```
(b)

```
// FIR
for(i=0;i<n-ksize;i++)
 for(j=0;j<ksize-1;j++)
out[i]+=in[i+j]*filter[j];
```
(c)

```
// BTMVM
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
Y[i]+=A[abs(i-j)]*X[j];
```
(d)

```
// Gaussian Blur

for (r=2; r<N-2; r++)
 for (c=2; c<M-2; c++)
  for (mr=0; mr<5; mr++)
   for (mc=0; mc<5; mc++)
out[r][c]+=(in[r+mr-2][c+mc-2]*mask[mr][mc])/159;
```
(e)

```
// BTMVM on Simplescalar
float A[N],X[N],Y[N];
for(j=0;j!=N;j+=5){
 REGY0=0.0;REGY1=0.0;
 REGY2=0.0;REGY3=0.0;
 REGY4=0.0;REGA0=A[j];
 REGA1=A[j+1];
 REGA2=A[j+2];
 REGA3=A[j+3];
 REGA4=A[j+4];

   for (i=0;i!=N;i++){
    REGX=X[i];
    REGY0+=REGA0*REGX;
    REGY1+=REGA1*REGX;
    REGY2+=REGA2*REGX;
    REGY3+=REGA3*REGX;
    REGY4+=REGA4*REGX;
    REGA4=REGA3;
    REGA3=REGA2;
    REGA2=REGA1;
    REGA1=REGA0;
    REGA0=A[abs(i-j+1)];
   }
       Y[j]+=REGY0;
       Y[j+1]+=REGY1;
       Y[j+2]+=REGY2;
       Y[j+3]+=REGY3;
       Y[j+4]+=REGY4; }
```
(f)

```
// FIR on Intel
for( i = 0; i != (n - ksize); i+=14 ){
   reg1=0; reg2=0; reg3=0; reg4=0;
   reg5=0; reg6=0; reg7=0; reg8=0;
   reg9=0; reg10=0; reg11=0; reg12=0;
   reg13=0; reg14=0;
  for( j = 0; j != ksize; j++ ){
  reg=kernel[ j ]; addr=i+j;
        reg1 += in[addr] * reg;
        reg2 += in[addr+1] * reg;
        reg3 += in[addr+2] * reg;
        reg4 += in[addr+3] * reg;
        reg5 += in[addr+4] * reg;
        reg6 += in[addr+5] * reg;
        reg7 += in[addr+6] * reg;
        reg8 += in[addr+7] * reg;
        reg9 += in[addr+8] * reg;
        reg10 += in[addr+9] * reg;
        reg11 += in[addr+10] * reg;
        reg12 += in[addr+11] * reg;
        reg13 += in[addr+12] * reg;
        reg14 += in[addr+13] * reg;  }
out[i]=reg1;
out[i+1]=reg2;
out[i+2]=reg3;
out[i+3]=reg4;
out[i+4]=reg5;
out[i+5]=reg6;
out[i+6]=reg7;
out[i+7]=reg8;
out[i+8]=reg9;
out[i+9]=reg10;
out[i+10]=reg11;
out[i+11]=reg12;
out[i+12]=reg13;
out[i+13]=reg14;   }
```
(g)

Figure 11: At (a)-(e) the benchmark codes are shown. At (f) and (g), the C-codes produced for BTMVM and FIR, for Simplescalar and Intel i7 3930K are shown, respectively.

4. Experimental Results

4.1. Experimental Setup

The experimental results presented in this section, were carried out on i) a desktop PC with Intel i7 3930K at 3.2 GHz, ii) a Virtex-5 FPGA ML507 Evaluation Platform (SDK 12.4) using PowerPC-440 processor and iii) SimpleScalar simulator [1]. The proposed methodology is compared with i) gcc and clang compilers, ii) iterative compilation technique. In (i), the operating system Ubuntu 14.04 LTS is used and two different compilers, i.e. gcc 4.6.3 and clang 3.0. In (ii), only gcc compiler is used. In (iii), the sslittle-na-sstrix-gcc compiler is used which supports out of order execution. Optimization level -O3 was used at all cases. The proposed methodology is not compared with the SOA libraries such as ATLAS or OpenCV, because they use the SIMD instructions and thus a comparison would be unfair (these libraries are also algorithm specific and the final code is not produced automatically).

The comparison is done for 5 well-known data dominant kernels of linear algebra, image processing and signal processing. These are: Matrix-Matrix Multiplication (MMM), Matrix-Vector Multiplication (MVM), Gaussian Blur (5×5 filter), Finite Impulse Response filter (FIR) and Bisymmetric Toeplitz Matrix-Vector Multiplication (BTMVM). The C-codes of these algorithms are shown in Fig. 11-(a)-(e). These algorithms are mainly selected because they are well known and simple; thus, the reader can easily understand the results (e.g. how tiling for data cache is applied), which are explained in detail.

4.2. Performance Comparison

First, a performance comparison using Intel i7 3930K and PowerPC-440 processors is performed (Fig. 12, Fig. 13, Fig. 14, Fig. 15, Fig. 16). Regarding the performance of the benchmark C-codes (Fig. 11-(a)-(e)) on Intel i7 3930K, gcc produces faster binaries than clang for the MMM (almost 4 times faster) and FIR (from 1.25 up to 1.6 times faster), slightly faster for the Gaussian Blur (1.15 times faster), slightly slower for the BTMVM (1.15 times slower) and approximately equal performance for MVM. Regarding the proposed methodology output C-codes, gcc produces slighlty faster binaries than clang. The speedup values, shown in Fig. 12, Fig. 13, Fig. 14, Fig. 15 and Fig. 16, refer to the ratio of the benchmark time (code in Fig. 11-(a)-(e)) to the proposed methodology time, e.g. the speedup for gcc is

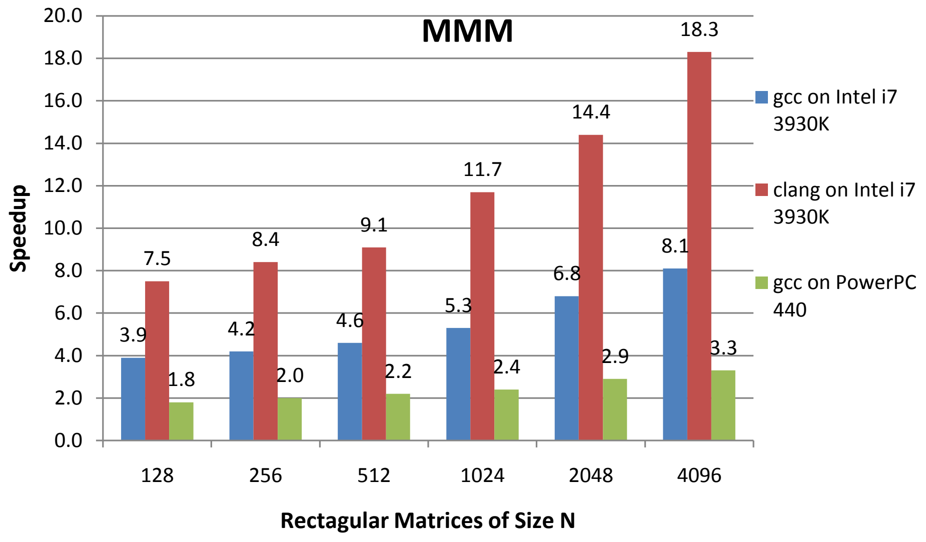


Figure 12: Performance comparison of the proposed methodology and gcc/clang compilers for MMM. The size refer to N of Fig. 11-a.

given by (Unoptimized code in gcc) / (optimized code in gcc), while the speedup for clang by (Unoptimized code in clang) / (optimized code in clang). The proposed methodology achieves higher speedup values on Intel than on PowerPC; PowerPC compiler is more aggressive (it applies more efficient transformations to the benchmark codes), resulting to faster binary code. The speedup values are from 1.8 up to 18 and from 1.9 up to 4.2, for Intel and PowerPC, respectively. As it was expected, at all algorithms, the speedup increases according to the input size; as the memory size increases, the memory management problem becomes more critical. Regarding MMM and Gaussian Blur, the proposed methodology achieves the largest speedup values (Fig. 12, Fig. 13), as i) their arrays are of larger size and ii) these algorithms have more data reuse; memory management has a larger effect in such cases. A more detailed analysis for each algorithm, follows.

Regarding MMM (Fig. 12), the speedup is much higher in clang (from 7.5 up to 18.3) than in gcc (from 3.9 up to 8.1). This is because the benchmark code shown in Fig. 11-(a), is almost four times slower in clang; this means that gcc applies transformations in a more efficient way here. Also, the PowerPC compiler applies efficient transformations to the benchmark code

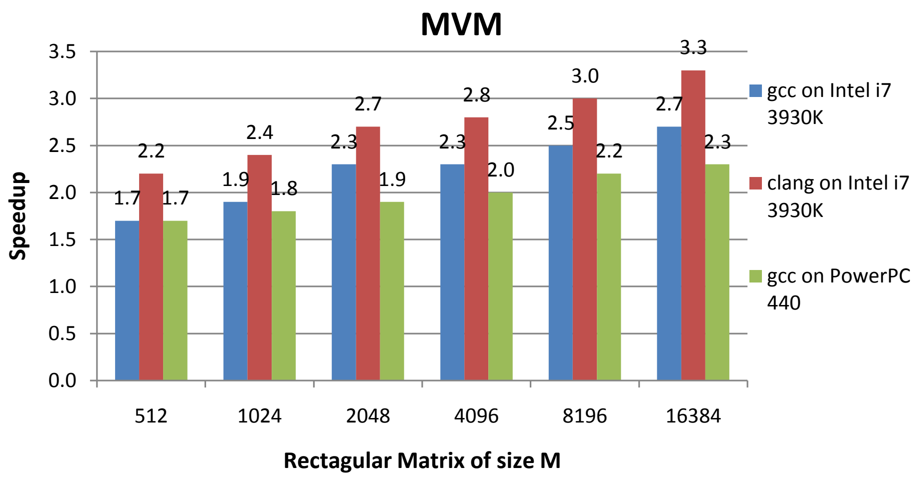


Figure 13: Performance comparison of the proposed methodology and gcc/clang compilers for MVM. The size refer to M of Fig. 11-b.

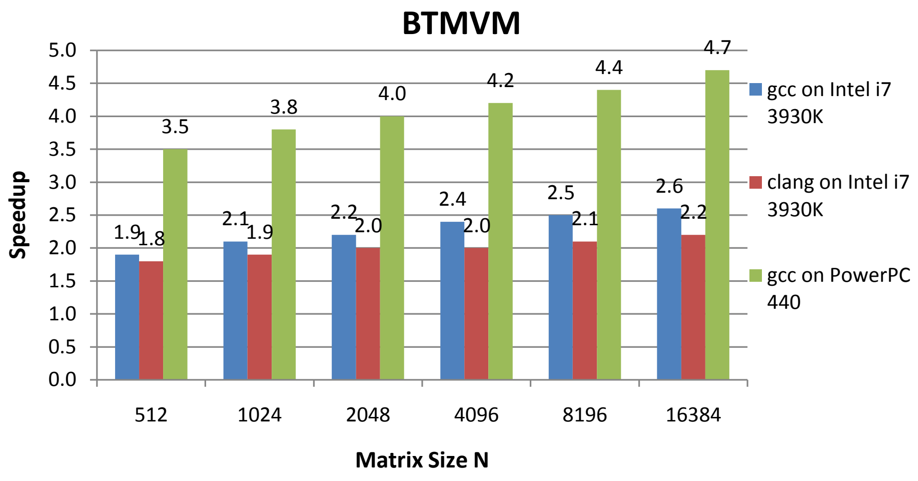


Figure 14: Performance comparison of the proposed methodology and gcc/clang compilers for BTMVM. The size refer to N of Fig. 11-d.

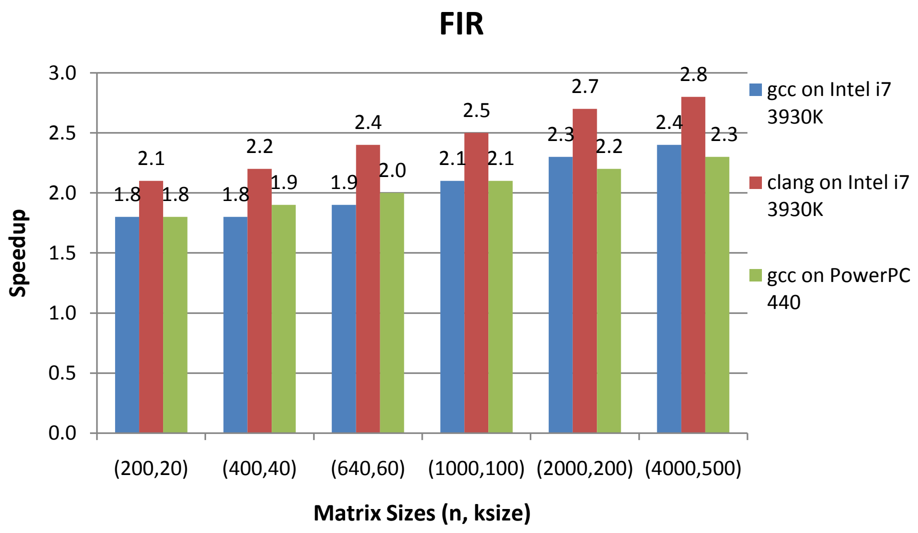


Figure 15: Performance comparison of the proposed methodology and gcc/clang compilers for FIR. The sizes refer to $(n, ksize)$ of Fig. 11-c.

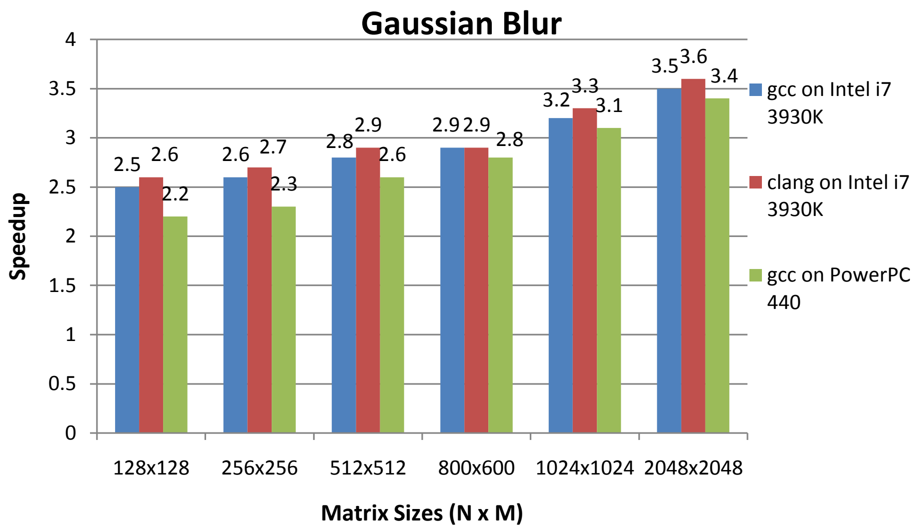


Figure 16: Performance comparison of the proposed methodology and gcc/clang compilers for Gaussian Blur. The sizes refer to (N, M) of Fig. 11-e.

and thus a smaller speedup is achieved (from 1.8 up to 3.3). Regarding the proposed methodology output C-codes, gcc produces about 1.5 times faster binaries. Gcc, clang and all other related tools and libraries, apply loop tiling without taking into account the data array layouts and the memory hierarchy parameters; this leads to a large number of data cache misses (it is explained in the last paragraph of Subsect. 3.8). In the case that $L2 \succ (4 \times N^2)$ (B array fits in L2), the best schedule found applies loop tiling only for L1 data cache as follows. The arrays are partitioned in such a way that two rows of A and $p1$ columns of B fit in L1 (the proposed methodology gives column-wise layout for B); when the first row of A is multiplied by $p1$ columns of B, the next row of A is fetched and it is multiplied by the same columns of B as before etc. For larger array sizes, there is no $p1$ number that two rows of A and $p1$ columns of B fit in L1 and this is why the arrays have to be partitioned even more; in this case, the proposed methodology applies loop tiling for L2 cache too, i.e. 3 rectangular tiles (one for each matrix) have to fit in L2. It is important to say that in this case, sub-rows of A are multiplied by sub-columns of B; however, the sub-rows elements are not written in consecutive main memory locations and this is why the number of data cache misses is highly increased. Thus, when the MMM arrays are partitioned into rectangular tiles, their data layouts must change (A and B arrays), from row-wise to tile-wise (the change of the layouts is included to the execution time).

Regarding MVM (Fig. 13), gcc produces higher quality code than clang. The best schedules produced by the proposed methodology use k registers for Y, 1 for A and 1 for X. The output schedules are similar to that shown in Fig. 13. Changing the data array layout of matrix A is not performance efficient here because i) A array is very large compared to the others (Y,X), ii) it is not reused (each element of A is accessed only once). Regarding Intel processor, tiling for data cache is not performance efficient here since the reused arrays, i.e. Y and X, are small and they fit in data cache in most cases. On the other hand, tiling for data cache is applied for PowerPC processor (large array sizes only). The best tiling solutions for MVM that the proposed methodology gives are as follows. The matrix A is partitioned in rectangular tiles of size $r \times c$ where $c \gg r$ and the array X is partitioned into tiles of size c; the schedule defines that each tile of X is multiplied by all possible tiles of A.

Regarding BTMVM (Fig. 14), the proposed methodology achieves double speed on PowerPC. This is because the $abs()$ function has a larger effect on the performance of PowerPC processor since $abs()$ routine needs more cycles at PowerPC than at Intel. Also, clang produces slightly faster binaries than gcc for the benchmark code here. The C-code produced by the proposed methodology here is similar to that shown at the right of Fig. 11-(f) (more registers for Y and A are used). The proposed methodology assigns A array elements into registers which interchange their values in each iteration (Rule 17 has been applied); thus a smaller number of abs functions is executed increasing performance even more. Loop tiling for L1 is applied only for PowerPC as the array sizes are small in contrast to the large Pentium cache sizes.

Regarding FIR (Fig. 15), the proposed methodology achieves the lowest performance gain, as FIR arrays have very small size and data reuse is small. The proposed methodology output code for clang compiler is shown at the left of Fig. 11-(g) (double precision floating point values). For double precision floating point values, clang and gcc use the XMM registers of Pentium and this is why 15 variables are used for the FIR arrays (Intel contains 16 XMM registers). Loop tiling for L1 is applied only for PowerPC as the array sizes are small in contrast to the large Pentium caches.

Regarding Gaussian Blur (Fig. 16), the speedups are about the same at all cases. The schedules shown in Fig. 10 achieve high execution speed at both processors. The schedules with tile-wise layout for *input* matrix, are not performance efficient because even in this case, data locality cannot be achieved. When tiling for data cache is applied, the best C-codes generated by the proposed methodology, partition the arrays vertically into parts (sub-images of size $M \times p1$).

4.3. Comparison on data accesses and arithmetic instructions

Furthermore, a comparison on the total number of L1 data cache accesses, main memory data accesses and arithmetic instructions is performed, on SimpleScalar simulator (right of Fig. 17). The architecture used here, consists of one level of data and instruction cache. To obtain the L1 data cache accesses gain values, the proposed methodology applies tiling for the RF only. Furthermore, to obtain the main memory data accesses values, the proposed methodology applies tiling for both RF and L1; in this case the L1 data cache size is chosen to be smaller than the arrays data (otherwise loop tiling is useless). It is important to say that SimpleScalar simulator uses a more naive gcc compiler (sslittle-na-sstrix-gcc) in contrast to that of Pentium and PowerPC and thus loop tiling is not applied by gcc. The number of L1

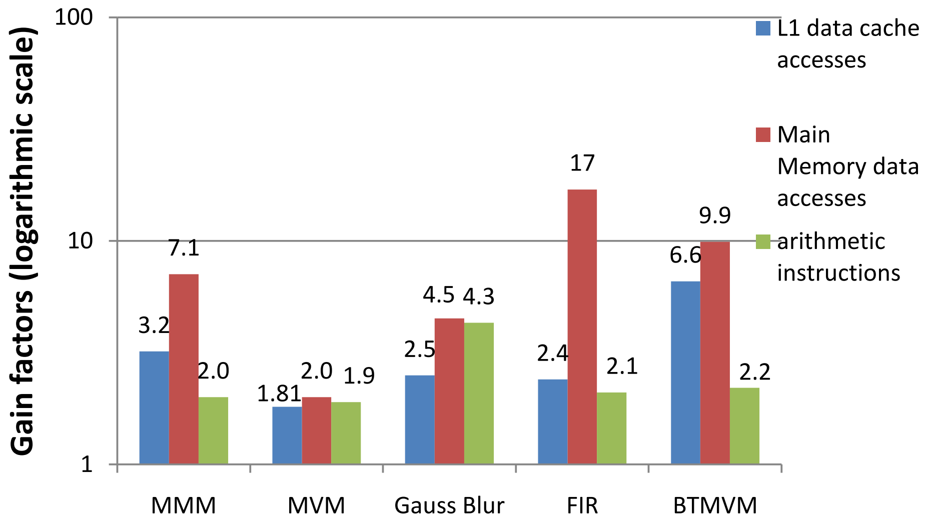


Figure 17: Comparison of the proposed methodology and gcc on SimpleScalar simulator.

and main memory accesses, highly depend on the input size and on the cache size and associativity. The larger the ratio of the array sizes to data cache, the larger the number of data cache accesses is. Furthermore, the smaller the data cache associativity is, the more critical the memory management problem becomes (in Fig. 17, the data cache is four-way associative). The data accesses values shown in Fig. 17, are not the maximum ones, but the average ones (a wide range of array sizes and four-way associative data cache, are taken). It is important to say that for direct-mapped data cache memories, the gain values are much larger than those shown in Fig. 17.

Regarding L1 data cache accesses, the proposed methodology achieves a gain from 1.8 up to 6.6 (Fig. 17). The BTMVM achieves the largest gain because all its three arrays are reused (all arrays achieve data reuse); the C-code produced is that shown at the right of Fig. 11-(f); rule 17 has been applied to exploit the data reuse of A elements. On the other hand, MVM achieves the smallest gain value, because the size of the reused data (Y and X arrays) is small, compared to the total arrays' size (array A is many times larger than Y and X); the code produced by the proposed methodology is similar to that shown at the left of Fig. 9.

Regarding main memory data accesses (Fig. 17), the proposed method-

Table 3: The proposed methodology is compared with iterative compilation. It achieves higher execution speed, in a orders of magnitude smaller compilation time.

	MMM	MVM	Gaussian Blur	FIR	BTMVM
Speedup	2.2	1.04	1.31	1.02	1.45

ology achieves a large gain value for MMM; this is because i) the size of the arrays is many times larger than the L1 size (for medium matrix sizes, e.g. 512×512, the arrays size is 3Mbytes), ii) MMM has a lot of data reuse, iii) the A and B arrays use tile-wize data arrays layouts (the change of the layouts is included to the Simulation). Regarding FIR, if the L1 data cache size is smaller than twice the size of the $filter$ array, loop tiling is necessary. This is because $filter$ array and a part of in array (of the same size) are loaded in each iteration; if L1 is smaller than this value, the $filter$ array does not remain in L1 and it is fetched many times from main memory (this is why a very large gain is achieved). The proposed methodology partitions $filter$ array in such a way that L1 is larger than twice the size of the array (the C-code produced here is that shown at the left of Fig. 11-(g) with an additional loop above the i loop - the j loop is tiled). Regarding MVM, the number of main memory data accesses is small as the size of the reused data (Y and X arrays) is small compared to the total arrays size (array A is many times larger than Y and X); the code produced by the proposed methodology is similar to that shown at the left of Fig. 9. The proposed methodology achieves a large gain value for BTMVM too. The most efficient way of tiling for L1, is the following. The X array is partitioned into tiles and each tile of X is multiplied by the whole A. Regarding Gaussian Blur, the images are partitioned vertically into smaller images.

Regarding the number of arithmetic instructions (Fig. 17), the proposed methodology achieves from 1.9 up to 4.3 less arithmetic instructions. This is because the proposed methodology decreases the number of the addressing instructions (see last paragraph of Subsection 3.6). The largest arithmetic instructions gain value is achieved for Gaussian Blur because Rule 18 has been applied and thus the Gaussian filter elements have been replaced by their values. The other 4 algorithms achieve about the half arithmetic instructions.

4.4. Comparison with Iterative Compilation and other related work

Finally, the proposed methodology is compared with iterative compilation technique on Intel processor. We used almost all the existing compiler transformations, for the above five algorithms; as it was expected, the compiler transformations that affect performance are: loop interchange, loop tiling, loop unroll, scalar replacement, register allocation (via graph coloring). It is important to say that the compilation time is enormous, and to be decreased, only power of 2 input sizes are used. Furthermore, the tile sizes and the unroll factor values are $2, 4, 8, 16, ..., N/2$. Loop tiling and register allocation are by far, the most performance efficient transformations here.

We used the gcc compiler (Table 3) and the C-codes shown in Fig. 11-(a)-(e). Regarding MMM, the proposed methodology achieves double speed for three reasons (Table 3). Firstly, to exploit the memory hierarchy architecture, the proposed methodology applies one level of tiling for each memory, i.e. RF, L1, L2 (for most input sizes). Iterative compilation applies only one level of tiling for each loop, which is not efficient, since MMM has a huge amount of data. Secondly, the proposed methodology changes the data array layouts of A and B from row-wise to tile-wise, as the matrices have been partitioned into smaller ones; iterative compilation does not take into account the data array layouts. Thirdly, in contrast to register allocation via graph coloring, the proposed methodology applies register allocation by taking into account the data reuse and production-consumption. Regarding BTMVM, the proposed methodology achieves a speedup of 1.43 (Table 3). This is because data reuse is fully exploited; the proposed methodology applies Rule 17 here (the C-code produced is similar to that shown at the right of Fig. 11-(f)). By applying Rule 17, the number of data cache accesses and the number of abs functions are decreased, increasing performance. Regarding Gaussian Blur, the proposed methodology achieves a performance gain of 1.28. This is because it applies register allocation more efficient, exploiting data reuse. Regarding MVM and FIR, the proposed methodology and iterative compilation find almost the identical schedules. To sum up, the proposed methodology achieves higher execution speed than iterative compilation in a orders of magnitude smaller compilation time.

In section 2, we explained why the proposed methodology is not compared with the SOA libraries referred to Section 2. However, the proposed methodology is compared with [29] and [30] which are faster than ATLAS library. In [29] and [30] we give a matrix-matrix and a matrix-vector (when the matrix A is regular, Toeplitz and Bisymmetric Toeplitz) multiplication methodology, respectively; we give the schedules achieving high performance for a wide range of hardware architectures. The proposed methodology cannot be compared with [31] because FFT contains nonlinear subscript equations (see second paragraph of Section 3). Also, the proposed methodology is not compared with [32] (Canny algorithm); this is because in [32], the four Canny kernels are optimized together and instead of four, one output loop kernel is produced. The proposed methodology optimizes each loop kernel separately and thus it cannot produce the schedules discussed in [32].

Regarding MVM, the proposed methodology produced all the schedules discussed in [30]. Regarding MMM, [29] uses the SIMD vector instructions and therefore the extra 128-bit XMM registers; thus, we will not make a comparison concerning the schedules utilizing the RF (Subsect.3.6). Except from the schedules utilizing the RF, the proposed methodology produced all the schedules used at [29]. The (near)-optimum data array layouts, data cache tile sizes/shapes and the iterators nesting level values, are produced. Regarding the BTMVM, the proposed methodology produced the exact schedules discussed in Subsection 3.1.1 of [30] but not in Subsection 3.1.2. This is because a) these schedules are not performance efficient in this architecture, b) in Subsection 3.1.2 of [30], BTMVM consists of not one, but of several output loop kernels, with many extra if-conditions; the proposed methodology, in its present form, does not produce multiple output loop kernels and if conditions.

5. Conclusions

We present a new methodology of speeding up loop kernels, for a wide range of algorithms and computer architectures. The major software and hardware parameters are fully exploited, giving better solutions, smaller search space, smaller code size and smaller compilation time, than the SOA libraries and iterative compilation techniques.

6. Aknowledgments

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference

Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

References

[1] T. Austin, E. Larson, D. Ernst, Simplescalar: An infrastructure for computer system modeling, Computer 35 (2002) 59–67. doi:10.1109/2.982917.
URL `http://dl.acm.org/citation.cfm?id=619072.621910`

[2] S. S. Pinter, Register allocation with instruction scheduling: a new approach, J. Prog. Lang. 4 (1) (1996) 21–38.

[3] G. Shobaki, M. Shawabkeh, N. E. A. Rmaileh, Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach, ACM Trans. Archit. Code Optim. 10 (3) (2008) 14:1–14:31. doi:10.1145/2512432.
URL `http://doi.acm.org/10.1145/2512432`

[4] D. F. Bacon, S. L. Graham, Oliver, J. Sharp, Compiler transformations for high-performance computing, ACM Computing Surveys 26 (1994) 345–420.

[5] E. Granston, A. Holler, Automatic recommendation of compiler options, in: In Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO), 2001, pp. 14–23.

[6] S. Triantafyllis, M. Vachharajani, N. Vachharajani, D. I. August, Compiler optimization-space exploration, in: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 204–215.
URL `http://dl.acm.org/citation.cfm?id=776261.776284`

[7] K. D. Cooper, D. Subramanian, L. Torczon, Adaptive optimizing compilers for the 21st century, Journal of Supercomputing 23 (2001) 2002.

[8] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, H. A. G. Wijshoff, A feasibility study in iterative compilation, in: Proceedings of

the Second International Symposium on High Performance Computing, ISHPC '99, Springer-Verlag, London, UK, UK, 1999, pp. 121–132.
URL `http://dl.acm.org/citation.cfm?id=646347.690219`

[9] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, J. W. Davidson, Practical exhaustive optimization phase order exploration and evaluation, TACO 6 (1).

[10] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, D. Jones, Fast searches for effective optimization phase sequences, SIGPLAN Not. 39 (6) (2004) 171–182. doi:10.1145/996893.996863.
URL `http://dl.acm.org/citation.cfm?doid=996893.996863`

[11] E. Park, S. Kulkarni, J. Cavazos, An evaluation of different modeling techniques for iterative compilation, in: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11, ACM, New York, NY, USA, 2011, pp. 65–74. doi:10.1145/2038698.2038711.
URL `http://doi.acm.org/10.1145/2038698.2038711`

[12] A. Monsifrot, F. Bodin, R. Quiniou, A machine learning approach to automatic production of compiler heuristics, in: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications, AIMSA '02, Springer-Verlag, London, UK, UK, 2002, pp. 41–50.
URL `http://dl.acm.org/citation.cfm?id=646053.677574`

[13] M. Stephenson, S. Amarasinghe, M. Martin, U.-M. O'Reilly, Meta optimization: improving compiler heuristics with machine learning, SIGPLAN Not. 38 (5) (2003) 77–90. doi:10.1145/780822.781141.
URL `http://doi.acm.org/10.1145/780822.781141`

[14] M. Tartara, S. Crespi Reghizzi, Continuous learning of compiler heuristics, ACM Trans. Archit. Code Optim. 9 (4) (2013) 46:1–46:25. doi:10.1145/2400682.2400705.
URL `http://doi.acm.org/10.1145/2400682.2400705`

[15] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, C. K. I. Williams, Using machine

learning to focus iterative optimization, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 295–305. doi:10.1109/CGO.2006.37.
URL `http://dx.doi.org/10.1109/CGO.2006.37`

[16] L.-N. Pouchet, C. Bastoul, A. Cohen, N. Vasilache, Iterative optimization in the polyhedral model: Part i, one-dimensional time, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 144–156. doi:10.1109/CGO.2007.21.

[17] L.-N. Pouchet, C. Bastoul, A. Cohen, J. Cavazos, Iterative optimization in the polyhedral model: Part ii, multidimensional time, SIGPLAN Not. 43 (6) (2008) 90–100. doi:10.1145/1379022.1375594.
URL `http://doi.acm.org/10.1145/1379022.1375594`

[18] C. Bastoul, Code generation in the polyhedral model is easier than you think, in: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 7–16. doi:10.1109/PACT.2004.11.

[19] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, I. Rosen, Polyhedral-model guided loop-nest auto-vectorization, in: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, PACT '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 327–337. doi:10.1109/PACT.2009.18.

[20] C. Chen, J. Chame, M. Hall, Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy, Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 0 (2005) 111–122. doi:10.1109/CGO.2005.10.

[21] C. W. Antoine, A. Petitet, J. J. Dongarra, Automated empirical optimization of software and the atlas project, Parallel Computing 27.

[22] A. Krivutsenko, Gotoblas - anatomy of a fast matrix multiplication, Tech. rep., Computational Science and Engineering Technical University Munich (2008).

[23] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org (2010).

[24] Intel math kernel library.
URL `http://software.intel.com/en-us/intel-mkl`

[25] J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology, in: Proceedings of the 11th International Conference on Supercomputing, ICS '97, ACM, New York, NY, USA, 1997, pp. 340–347. doi:10.1145/263580.263662.
URL `http://doi.acm.org/10.1145/263580.263662`

[26] M. Frigo, S. G. Johnson, The fastest fourier transform in the west, Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1997).

[27] G. Bradski, The opencv library, Dr. Dobb's Journal of Software Tools.

[28] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, M. Puschel, Spiral in scala: Towards the systematic construction of generators for performance libraries, in: International Conference on Generative Programming: Concepts and Experiences (GPCE), 2013, pp. 125–134.

[29] V. I. Kelefouras, A. Kritikakou, C. Goutis, A matrix–matrix multiplication methodology for single/multi-core architectures using simd, Supercomputing, Springer.

[30] V. I. Kelefouras, A. Kritikakou, K. Siourounis, C. Goutis, A methodology for speeding up mvm for regular, toeplitz and bisymmetric toeplitz matrices, Supercomputing, Springer.

[31] V. I. Kelefouras, G. Athanasiou, N. Alachiotis, H. E. Michail, A. Kritikakou, C. E. Goutis, A methodology for speeding up fast fourier transform focusing on memory architecture utilization., IEEE Transactions on Signal Processing 59 (12) (2011) 6217–6226.

[32] V. I. Kelefouras, A. Kritikakou, C. Goutis, A methodology for speeding up edge and line detection algorithms focusing on memory architecture utilization, Supercomputing, Springer.

[33] S. Wuytack, J.-P. Diguet, F. V. M. Catthoor, H. J. De Man, Formalized methodology for data reuse exploration for low-power hierarchical memory mappings, IEEE Trans. Very Large Scale Integr. Syst. 6 (1998) 529–537. doi:10.1109/92.736124.
URL `http://dl.acm.org/citation.cfm?id=297294.297307`

[34] P. G. Kjeldsberg, F. Catthoor, E. J. Aas, Data dependency size estimation for use in memory optimization, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 22 (2003) 908–921.

[35] T. Van Achteren, R. Lauwereins, F. Catthoor, Systematic data reuse exploration methodology for irregular access patterns, in: Proceedings of the 13th International Symposium on System Synthesis, ISSS '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 115–121. doi:10.1145/501790.501816.

[36] B. Jang, D. Schaa, P. Mistry, D. Kaeli, Exploiting memory access patterns to improve memory performance in data-parallel architectures, IEEE Transactions on Parallel and Distributed Systems 22 (2011) 105–118.

[37] M. Moazeni, A. A. T. Bui, M. Sarrafzadeh, A memory optimization technique for software-managed scratchpad memory in gpus., in: SASP, IEEE Computer Society, 2009, pp. 43–49.
URL `http://dblp.uni-trier.de/db/conf/sasp/sasp2009.html`

[38] Q. Zhao, R. Rabbah, W.-F. Wong, Dynamic memory optimization using pool allocation and prefetching, SIGARCH Comput. Archit. News 33 (2005) 27–32. doi:10.1145/1127577.1127584.

[39] M. Kowarschik, C. Weiß, An overview of cache optimization techniques and cache-aware numerical algorithms, in: Algorithms for Memory Hierarchies Advanced Lectures, volume 2625 of Lecture Notes in Computer Science, Springer, 2003, pp. 213–232.

[40] M. T. Heath, A. Ranade, R. S. Schreiber (Eds.), Algorithms for parallel processing, The IMA volumes in mathematics and its applications, Springer, Institute of Mathematics and Its Applications, University of Minnesota, 1999, proceedings of the Workshop on Algorithms for

Parallel Processing, held Sept. 16-20, 1996, at the IMA.
URL `http://opac.inria.fr/record=b1095528`

[41] T. Sherwood, B. Calder, J. Emer, Reducing cache misses using hardware and software page placement, in: Proceedings of the 13th international conference on Supercomputing, ICS '99, ACM, New York, NY, USA, 1999, pp. 155–164. doi:10.1145/305138.305189.

[42] C. Cacaval, D. A. Padua, Estimating cache misses and locality using stack distances, in: Proceedings of the 17th annual international conference on Supercomputing, ICS '03, ACM, New York, NY, USA, 2003, pp. 150–159. doi:10.1145/782814.782836.

[43] S. Ghosh, M. Martonosi, S. Malik, Cache miss equations: an analytical representation of cache misses, in: Proceedings of the 11th international conference on Supercomputing, ICS '97, ACM, New York, NY, USA, 1997, pp. 317–324. doi:10.1145/263580.263657.

[44] F. Song, S. Moore, J. Dongarra, L2 cache modeling for scientific applications on chip multi-processors, Parallel Processing, International Conference on 0 (2007) 51. doi:10.1109/ICPP.2007.52.

[45] R. A. Hankins, J. M. Patel, Data morphing: an adaptive, cache-conscious storage technique, in: Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '2003, VLDB Endowment, 2003, pp. 417–428.
URL `http://dl.acm.org/citation.cfm?id=1315451.1315488`

[46] M. Franz, T. Kistler, Splitting data objects to increase cache utilization, Tech. rep., Department of Information and Computer Science University of California (1998).

[47] S. Rubin, R. Bodík, T. Chilimbi, An efficient profile-analysis framework for data-layout optimizations, SIGPLAN Not. 37 (2002) 140–153. doi:10.1145/565816.503287.

[48] H. Rong, A. Douillet, G. R. Gao, Register allocation for software pipelined multi-dimensional loops, SIGPLAN Not. 40 (6) (2005) 154–167. doi:10.1145/1064978.1065030.
URL `http://doi.acm.org/10.1145/1064978.1065030`

[49] S. Hack, G. Goos, Optimal register allocation for ssa-form programs in polynomial time, Inf. Process. Lett. 98 (4) (2006) 150–155. doi:10.1016/j.ipl.2006.01.008.
URL `http://dx.doi.org/10.1016/j.ipl.2006.01.008`

[50] S. G. Nagarakatte, R. Govindarajan, Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation, in: Proceedings of the 16th International Conference on Compiler Construction, CC'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 126–140.
URL `http://dl.acm.org/citation.cfm?id=1759937.1759949`

[51] V. Sarkar, R. Barik, Extended linear scan: An alternate foundation for global register allocation, in: Compiler Construction, 16th International Conference, Braga, Portugal, March 26-30, 2007, Proceedings, 2007, pp. 141–155. doi:10.1007/978-3-540-71229-9_10.
URL `http://dx.doi.org/10.1007/978-3-540-71229-9_10`

[52] M. D. Smith, N. Ramsey, G. Holloway, A generalized algorithm for graph-coloring register allocation, SIGPLAN Not. 39 (6) (2004) 277–288. doi:10.1145/996893.996875.
URL `http://doi.acm.org/10.1145/996893.996875`

[53] L. Wang, X. Yang, J. Xue, Y. Deng, X. Yan, T. Tang, Q. H. Nguyen, Optimizing scientific application loops on stream processors, SIGPLAN Not. 43 (7) (2008) 161–170. doi:10.1145/1379023.1375679.
URL `http://doi.acm.org/10.1145/1379023.1375679`

[54] N. Baradaran, P. C. Diniz, A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures, CoRR abs/0710.4702.
URL `http://arxiv.org/abs/0710.4702`

[55] S.-K. Chang, Data Structures and Algorithms, Vol. 13 of Series on Software Engineering and Knowledge Engineering, World Scientific, 2003.