A methodology for efficient code optimizations and memory management

KELEFOURAS, Vasileios <http://orcid.org/0000-0001-9591-913X> and DJEMAME, Karim

Available from Sheffield Hallam University Research Archive (SHURA) at:
http://shura.shu.ac.uk/19005/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version


Copyright and re-use policy

See http://shura.shu.ac.uk/information.html
A methodology for efficient code optimizations and memory management

Vasilios Kelefouras  
University of Leeds  
v.kelefouras@leeds.ac.uk

Karim Djemame  
University of Leeds  
K.Djemame@leeds.ac.uk

ABSTRACT
The key to optimizing software is the correct choice, order as well parameters of optimizations-transformations, which has remained an open problem in compilation research for decades for various reasons. First, most of the compilation subproblems-transformations are interdependent and thus addressing them separately is not effective. Second, it is very hard to couple the transformation parameters to the processor architecture (e.g., cache size and associativity) and algorithm characteristics (e.g., data reuse); therefore compiler designers and researchers either do not take them into account at all or do it partly. Third, the search space (all different transformation parameters) is very large and thus searching is impractical.

In this paper, the above problems are addressed for data dominant affine loop kernels, delivering significant contributions. A novel methodology is presented that takes as input the underlying architecture details and algorithm characteristics and outputs the near-optimum parameters of six code optimizations in terms of either L1, L2, DDR accesses, execution time or energy consumption. The proposed methodology has been evaluated to both embedded and general purpose processors and for six well known algorithms, achieving high speedup as well energy consumption gain values over gcc compiler, hand written optimized code and Polly.

KEYWORDS
Code optimizations, data cache, register blocking, loop tiling, high performance, energy consumption, data reuse

ACM Reference format:

1 INTRODUCTION
Although significant advances have been made in developing advanced compiler optimization and code transformation frameworks, current compilers cannot compete hand optimized code in terms of performance and energy consumption. Researchers tackle the code optimization problem by using heuristics [12], empirical techniques, iterative compilation techniques [11] and techniques that simultaneously optimize only two transformations, e.g., register allocation and instruction scheduling. The most promising approach is iterative compilation but is extremely expensive in terms of compilation time; therefore researchers and current compilers try to reduce compilation time by using i) both iterative compilation and machine learning compilation techniques [20], ii) both iterative compilation and genetic algorithms [11], iii) heuristics and empirical methods [5], iv) both iterative compilation and statistical techniques, v) exhaustive search [10]. However, by employing these approaches, the remaining search space is still so large that searching is impractical. The end result is that seeking the optimal configuration is impractical even by using modern supercomputers. This is evidenced by the fact that most of the iterative compilation methods use either low compilation time transformations only or high compilation time transformations with partial applicability so as to keep the compilation time in a reasonable level [9] [19] [13]. As a consequence, a very large number of solutions is not tested. This has led compiler researchers to use exploration prediction models focusing on beneficial areas of optimization search space [5]. Our approach differs in three main aspects. First, the transformations are addressed in a theoretical basis; second, together as one problem, and third by taking into account the Hardware (HW) architecture and algorithm characteristics. This way, the search space is reduced by orders of magnitude and as a consequence the quality of the end result is significantly improved.

The main steps of our methodology are as follows. First, we provide an efficient register blocking and loop tiling algorithm; these two algorithms consist of a) loop unroll, scalar replacement, register allocation and b) loop tiling, data array layout, transformations respectively. A unified framework is proposed to orchestrate the aforementioned transformations, together as one problem (as they are interdependent); the transformations are tailored to the target processor architecture details and algorithm characteristics. Second, we make an analysis of how the above transformations affect Execution Time (ET) and Energy consumption (E) and for the first time we provide a theoretical model describing a) the number of L1 data cache (L1dc), L2 cache (L2c) and main memory (MM) accesses and b) the number of arithmetical instructions, as a function of the aforementioned transformation parameters, processor architecture details and algorithm input size: so, we are able to provide the transformation parameters giving a number of memory accesses close to the minimum. Third, taking advantage of this model, we make a first but important step towards correlating ET and E with the aforementioned transformation parameters, processor architecture details and algorithm input size.

The proposed methodology has resulted in five contributions, 1) a single framework addressing the aforementioned transformations theoretically but most importantly as one problem, 2) a new approach applying code optimizations (CO) by taking into account the HW architecture and the application special memory access
2 RELATED WORK

Iterative compilation methods provide the most efficient approach towards the code optimization problem. However, to the best of our knowledge, there is no existing iterative compilation method including all the transformations presented in this paper and all different transformation parameters, because the compilation time becomes too large. Iterative compilation methods use either low compilation time transformations only or high compilation time transformations with partial applicability so as to keep the compilation time in a reasonable level [9] [19]. As a consequence, a very large number of solutions is not tested. In [19], loop tiling is applied with fixed tile sizes. In [9], multiple levels of tiling are applied but with fixed tile sizes. In [13], only loop unroll is applied.

[12] uses an artificial neural network to predict the best transformation (from a given set) should applied. In [5], performance counters are used to determine good compiler optimization settings. In [20], a long-term learning algorithm that determines the best set of heuristics is presented.

The polyhedral model is a flexible and expressive representation for loop transformations. In [17], a fundamental progress in the understanding of polyhedral loop nest optimizations is made. Polly is a high-level loop and data-locality optimizer and optimization infrastructure for LLVM [6]. Pluto, which is used by Polly, is an automatic parallelization tool based on the polyhedral model [4].

There has been significant research on reducing the number of data accesses in memory hierarchy by employing compiler transformations and most commonly loop tiling such as [4] [15]. In [15], a cache hierarchy aware tile scheduling algorithm for multicore architectures is presented.

Code optimizations are also used to reduce energy consumption in software. In [1], a survey about energy reduction methods is given. In [16], several transformation trade-offs are discussed. In [2], a compile-time approach to determine CPU frequency is proposed.

3 PROPOSED METHODOLOGY

In this paper, a novel methodology is presented that takes as input the underlying processor architecture and loop kernel characteristics and outputs the near-optimum parameters of the six aforementioned transformations in terms of either L1/L2/DDR accesses, Execution Time (ET) or Energy consumption (E).

Regarding target applications, this methodology considers affine loop kernels; it considers both perfectly and imperfectly nested loops, where all the array subscripts are linear equations of the iterators (which stands in most cases). This method is also applicable to loop kernels containing SIMD instructions. This method is applicable to all modern single-core and shared cache multi-core CPUs. Regarding shared cache processors, we use the software shared cache partitioning method given in our previous work [8].

No more than \( p \) threads can run in parallel (one to each core), where \( p \) is the number of the processing cores (single threaded codes only).

An abstract representation of our method is illustrated in Fig. 1 and it is further explained in the following Subsections.

3.1 Apply code optimizations

In this Subsection we provide an efficient a) register blocking and b) loop tiling algorithm. The efficient application of loop tiling is not trivial and normally many different implementations are tested, since it a) depends on other transformations (e.g., data layout), b) depends on the target memory architecture and data reuse, c) increases the number of arithmetical instructions. The application of loop tiling for the Register File (RF) is even more complex (register blocking). To our knowledge, no application independent algorithm exists for register blocking; it is a mixture of loop tiling, loop unroll, scalar replacement and register allocation transformations. The above CO are the key to high performance and low energy consumption, especially for data dominant algorithms.

The main steps of the proposed register blocking algorithm are the following:

1. Generate the subscript equations of all arrays
2. Generate the RF inequality (Eq. 1) that provides all the efficient transformation parameters
3. Extract a transformation set from Eq. 1
4. Generate the code

Definition 3.1. Subscript equations which have more than one solution for at least one constant value, are named type2 equations. All others, are named type1 equations.

For example, \((A[2*i + j])\) gives the following type2 equation \((2*i + j = c1)\), while \((A[i][j])\) gives the following type1 equation \((i = c21 \text{ and } j = c22)\).

Each subscript equation defines the memory access pattern of the specific array reference. Obviously, in our methodology type1 and type2 arrays are treated with different policies as they access data in different ways.
The RF inequality (Eq. 1) gives the exact loops that loop unroll is applied to, their unroll factor values and the number of variables/registers allocated for each array. Each subscript equation contributes to the creation of Eq. 1, i.e., equation i gives $A_{ri}$ and specifies its expression. The implementations that do not obey to the extracted inequalities are discarded reducing substantially the search space. The RF inequality is given by

$$n + Sc \leq A_{r1} + Ar_{2} + \ldots + Ar_{n} + Sc \leq FP$$

where $FP$ is the number of the floating point (FP) registers, $Sc$ is the number of FP scalar variables and $n$ is the number of the array references. Without any loss of generality, we assume that the arrays contain FP data only; in this case, the number of integer variables used is always smaller than the number of integer registers. The upper bound of Eq. 1 derives from the fact that if more registers than the available are used, data are spilled to L1dc, increasing the number of L1 accesses. On the other hand, the lower bound value is small because other constraints may be more critical. By using a larger lower bound value, register utilization is increased and therefore the number of L1 accesses is reduced; however, these transformation parameters may conflict to those minimizing the number of MM or L2 accesses, which may be more critical.

The number of variables/registers allocated for every array is given by both Eq. 2 and the three bullets below (the bullet points are given in order to assign variables according to data reuse).

$$Ar_{i} = unr_{i} \cdot 1^{1} \times unr_{2} \cdot 2^{2} \times \ldots \times unr_{n} \cdot n^{n}$$

where the integer $unr_{i}$ are the unroll factor values of the iterators exist in the array’s subscript, e.g., the C[$i$][$j$] array in Fig. 2 gives $(Ar_{C} = 1 \times 4 = 4)$ ($r1 - r4$ variables) as the $(i, j)$ iterator unroll factor values are $(1, 4)$, respectively.

- For the type1 arrays which contain all the loop kernel iterators, only one register is needed ($Ar_{t1} = 1$)
- For the innermost iterator always holds $unr' = 1$
- For the arrays i) containing more than one iterators and one of them is the innermost and ii) all 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 ($Ar_{r1} = 1$)

In the above cases, a different element is accessed in each iteration (no data reuse being achieved) and thus wasting more than one register is not efficient, e.g., in Fig. 2, six registers are used, i.e., $(Ar_{C} = 1 \times 4, Ar_{A} = 1 \times 1, Ar_{B} = 1)$. Note that $(Ar_{B} = 1)$ instead of $(Ar_{B} = 1 \times 4)$ because of the 3rd bullet above (a different element of B is accessed in each $k$ iteration and therefore it is not efficient to waste more than one register).

Let us give an example, first box code in Fig. 2. Eq. 1 gives:

$$3 \leq unr_{1} \times unr_{1} + unr_{1} + unr_{1} \leq FP, unr_{1} \neq 1 \land unr_{1} \neq 1$$

$$3 \leq unr_{1} + 2 \leq FP, unr_{1} = 1$$

$$3 \leq unr_{1} + 2 \leq FP, unr_{1} = 1$$

The 3rd bullet generates 3 branches while the 2nd gives $(unr_{1} = 1)$. The code shown in the second box of Fig. 2 refers to a second branch solution, i.e., $(unr_{1} = 1$ and $unr_{1} = 4)$ and therefore 6 registers are used.

The main steps of the loop tiling algorithm are similar to those of the register blocking algorithm, but a cache inequality (Eq. 4) is generated for each cache memory; each inequality contains the iterators that loop tiling is applied to, the tile sizes and the data array layouts. The implementations that do not obey to the extracted inequalities are automatically discarded by our methodology reducing substantially the search space.

The cache inequality is formulated as:

$$m \leq \lceil \frac{Tile_{ei}}{T_{j}^p} \rceil + \ldots + \lceil \frac{Tile_{ej}}{T_{j}^p} \rceil \leq assoc$$

where $L_{j}$ is the corresponding cache size, $assoc$ is the $L_{j}$ associativity value (e.g., for an 8-way associative cache, $assoc = 8$) and $m$ defines the lower bound of the tile sizes and it equals to the number of arrays in the loop kernel. In the special case where the number of the arrays is larger than the associativity value is not discussed in this paper (normally, $(assoc \geq 8)$). $Tile_{ei}$ gives the tile size of the $i$th array and is given by Eq. 5:

$$Tile_{ei} = T_{1}^p \times T_{2}^p \times T_{3}^p \times type \times s$$

where $type$ is the size of each array’s element in bytes and $T_{j}^p$ are the tile sizes of the iterators existing in the corresponding array subscript. $s$ is an integer and $(s = 1$ or $s = 2$); $s$ defines how many tiles of each array should be allocated in the cache. For the tiles that do not achieve data reuse (a different tile is accessed in each iteration), we assign cache space twice the size of their tiles ($s = 2$ in Eq. 5). This way, not one but two consecutive tiles are allocated into the cache in order for the second accessed tile not to displace another array’s tile. $(\lceil \frac{Tile_{ei}}{T_{j}^p} \rceil)$ is an integer representing the number of $L_{j}$ cache lines with identical cache addresses used for the tile of array 1. Eq. 4 satisfies that the array tiles directed to the same cache subregions do not conflict with each other as the number of cache lines with identical addresses needed for the tiles is not larger than the $(assoc)$ value.

All the tile elements in Eq. 4 must contain consecutive MM locations, e.g., in Fig. 2, none of the tiles does. Otherwise, an extra loop kernel is added for each array, likewise $Atr$ and $Btr$ arrays in Fig. 2; new arrays are created which replace the default ones (extra cost in L/S and arithmetical instructions). There are
some special cases where the arrays do not contain consecutive memory locations but their layouts can remain unchanged in order to avoid the cost of transforming the arrays; in that case, extra cache misses occur and thus a larger error in approximating the number of memory accesses occurs too (Fig. 4-Subsection 4.1). \( T'_I \) is given by one of the following three:

- \( T'_I \) equals to the L1 tile size of i iterator, if tiling for L1 is applied to the i iterator
- \( T'_I \) equals to the unroll factor value of i iterator, if tiling for L1 is not applied to the i iterator and i has a smaller Nesting Level Value (NLV) than the iterator being tiled for L1
- \( T'_I \) equals to the upper loop bound value of i iterator, if tiling for L1 is not applied to the i iterator and i has a larger NLV than the iterator being tiled for L1

Assuming an 8-way 32kbyte L1dc and MMM (Fig. 2), Eq. 4 gives \( (3 \leq \sum \frac{T_{C1}}{T_{MM}} + \sum \frac{T_{C2}}{T_{MM}} + \sum \frac{T_{C3}}{T_{MM}} \leq 8) \). The \((T_{C1}, T_{A1}, T_{B1})\) values of the C-code shown at the right of Fig. 2 are given in the bottom left box (the NLV of \( k \) iterator is 6 while the NLV of \( kk \) is 1), also, floating point values are assumed, 4 bytes each.

In the shared cache case, \( L_1 \) in Eq. 4 is the corresponding shared cache partition size used and each core uses only its assigned shared cache space.

We have implemented an automated C to C tool just for the six studied algorithms, but a general tool can be implemented by using POET [18] tool.

### 3.2 Couple execution behaviour to CO, processor architecture & input size

For all the Eq. 1 and Eq. 4 schedules, we compute the number of L1dc, L2c and MM accesses as well as the number of arithmetical instructions. This problem is theoretically formulated by exploiting the memory architecture details and the special memory access patterns. In particular, one mathematical equation is generated for each memory and for each loop kernel providing the corresponding value. This equation provides the number of memory accesses while the transformation parameters and input size serving as the independent variables of the equation. Loop tiling and loop unroll transformations as well as the input size, are inserted directly to the aforementioned equations while the data layouts, scalar replacement and register allocation transformations as well as the HW architecture, are inserted indirectly (they have been used in order to create Eq.1-Eq.5). This way, we are able to find the solution offering a number of L1dc, L2c or MM accesses close to the minimum.

We are able to approximate the number of memory accesses because no unexpected misses occur in the cache. We assume that the underlying memory architecture consists of separate first level data and instruction caches (modern architectures). In this case, the program code typically fits in L1 instruction cache; thus, it is assumed that the shared cache or unified cache (if any) is dominated by data. For the reminder of this paper, we assume 2 levels of cache, but more/less levels can be used, by slightly changing the following equations.

The equation approximating the number of L1dc accesses follows

\[
L_1.\text{acc} = \sum_{j=1}^{i-\text{arrays}} \sum_{j=1}^{j=M} \left( \frac{\text{up}_P - \text{lo}_W}{T_j} \times \prod_{k=1}^{k=N} \text{up}_R + \text{off}_{set_i} \right) + \text{var}
\]

where \( \text{arrays} \) is the number of arrays, \( M \) is the number of the iterators that control the corresponding array and \( P \) is the number of the iterators that loop unrolls has been applied to (iterators that exist in the subscript of the corresponding array only), e.g., regarding the C array in the code at the right of Fig. 2, the first product of Eq. 6 refers to all the iterators but \( k \) (array reference is outside \( k \) loop) while the second product refers to \( j \) iterator only. \((\text{up}_P, \text{lo}_W)\) give the bound values of the corresponding iterator (normally, they define the algorithm’s input size) and \((T, \text{var})\) refer to the tile size and unroll factor value, respectively.

of \( \text{off}_\text{set} \) gives the number of L1dc of the new loop kernel added in the case the data array layout is transformed. Offset is either \( (\text{off}_{set_i} = 2 \times \text{ArraySize}) \) or \( (\text{off}_{set_i} = 0) \) depending on whether the data layout of the array is changed or not; in the case that the layout is changed, the array has to be loaded and then written again to memory, thus it is \( (\text{off}_{set_i} = 2 \times \text{ArraySize}) \). \((\text{var})\) gives the number of L1 accesses due to the scalar variables; we never use more registers than available and thus the number of RF spills is negligible \( (\text{var} \approx 0) \).

Eq. 6 for the C-code at the right of Fig. 2 gives \( (N^3_3 + N^2_3 + N^3_3) \) \( L_1 \) accesses for \((C, A, B)\) arrays, respectively, and in overall \( (L_1.\text{acc} = N^3_3 + N^2_3 + 3 \times N^2_3) \). Here, the number of L1 accesses strongly depends on the unroll factor value \((N^3_3)\).

The number of L2c accesses is approximated by Eq. 7; at this step, only the new/extra iterators (introduced by loop tiling) must be processed and not the initial iterators exist in the input code.

\[
L_2.\text{Acc.} = \sum_{i=1}^{i=\text{type1}} \text{Type1}.L2.\text{acc.} + \sum_{i=1}^{i=\text{type2}} \text{Type2}.L2.\text{acc.} + \text{code}
\]

where \( \text{type1} \) and \( \text{type2} \) is the number of type1 and type2 arrays, respectively. In this paper, we don’t provide the equations for type2 arrays because of the limited paper size, however, in Section 4, FIR and Gaussian Blur contain type2 arrays. \( \text{code} \) refers to the number of source code accesses and always \((\text{Array}\_\text{acc} \gg \text{code})\) as a) the code size of loop kernels is small and fits in L1 instruction cache, b) we are dealing with data dominant algorithms.

\[
\text{Type1}.L2.\text{acc.} = \text{array}\_\text{size} \times t_i + \text{off}_\text{set}
\]

where \( \text{array}\_\text{size} \) is the size of the array and \( \text{off}_\text{set} \) gives the number of L2 accesses of the new loop kernel added in the case the data array layout is transformed. \( t_i \) gives how many times the corresponding array is accessed from L2 memory and is given by Eq. 9. Regarding the \( \text{off}_\text{set} \) value, when the array size is bigger or comparable to the cache size, then \((\text{off}_{set_i} > 2 \times \text{ArraySize})\). This is because the elements are always loaded in blocks (cache lines) and many lines are loaded more than once (especially in the column-wise case). This is why we use a hand optimized code changing the layout in an efficient way, thus always achieving \((\text{off}_{set_i} \approx 2 \times \text{ArraySize})\).

\[
t_i = \prod_{j=1}^{j=N} (\text{up}_P - \text{lo}_W) \times \prod_{k=1}^{k=N} (\text{up}_P - \text{lo}_W)
\]

where \( N \) is the number of new/extra iterators that a) do not exist in the corresponding array and b) exist above of the iterators of the corresponding array. \( M \) is the number of new/extra iterators that a) do not exist in the array and b) exist between of the iterators of the array, e.g., regarding \((C, A, B)\) arrays in Fig. 2, the iterators referring to the first and second product of Eq. 9 are \((kk, none), (jj, none), (none, ii)\), respectively, giving \((t_k = \frac{N}{kk}), (t_j = \frac{N}{jj}) \) and \((t_j = \frac{N}{II})\), respectively. The first and the second products of Eq. 9 give how
many times the array is accessed due to the iterators exist above the upper new iterator of this array and between the new iterators of this array, respectively.

Eq. 7 for the code of Fig. 2 gives \((L2\text{.acc} = N^3/KK + N^3/JJ + N^3/II + 4 \times N^2)\).

In the case that more than one thread run in parallel under a shared cache, the overall number of cache accesses is extracted by accumulating all the different loop kernel equations.

The number of MM accesses is given by an equation identical to Eq. 7. Moreover, the number of MM accesses because of the type1 arrays is given by an equation identical to Eq. 8 and Eq. 9. However, in Eq. 9, we refer only to the iterators created by applying tiling to the last level cache, e.g., regarding \((C,A,B)\) arrays of MM (Fig. 2), the iterators referring to the first and second product of Eq. 9 are \((kk, none)\), \((none, none)\), \((none, ii)\), respectively, giving \((tC = \frac{N}{KK})\), \((tA = 1)\) and \((tB = \frac{N}{II})\), respectively.

The number of integer and FP instructions is approximated by:

\[
\text{Arith.\_instrs} = \sum_{j=1}^{\text{iterators}} \left[ \frac{\text{up}_{ij} - \text{low}_{ij}}{f_j} \times c_j \right] + \text{offset} \tag{10}
\]

where \(\text{iterators}\) is the total number of iterators and \((\text{up}, \text{low}, T)\) are their corresponding bound values, as in previous equations. \(c_j\) is the number of integer or FP assembly instructions measured inside \(j\) loop (assembly instructions occur between the open and close loop bracket). \(\text{offset}\) is the number of arithmetical instructions of the extra loop kernels added (if the array layouts change).

\((N^3/\text{KK}) \times c_1 + (N^3/(\text{KK} \times \text{II})) \times c_2 + (N^3/(\text{KK} \times \text{JJ})) \times c_3 + (N^3/(\text{KK} \times \text{JJ})) \times c_4 + (N^3/(\text{KK} \times \text{II})) \times c_5 + (N^3/4 \times c_6)\); as it can be observed, the number of arithmetical instructions is strongly affected by a) the number of the loops being tiled (more terms are introduced), b) tile size / unroll factor values of the innermost iterators (here, the unroll factor value of \(j\), i.e., 4, affects the number of instructions at the most). Thus, a larger unroll factor value would be more efficient.

Given that the \(c\) values depend on the target compiler, they cannot be approximated. Thus, we measure the \(c\) values for one transformation set and predict the \(c\) values of the others (where possible), e.g., in Fig. 2, the \(c\) values (assembly instructions) almost remain unchanged by changing the (\(\text{KK, II, JJ}\) values (apart from their maximum and minimum ones because in this case, the number of the loops changes), but not by changing the (\(j\) unroll factor value or the number of the loops being tiled, because the loop body changes and thus more/less assembly instructions are inserted.

In this work, we take advantage of the fact that the \(c\) values almost remain unchanged for different tile sizes, suffice the array layouts remain unchanged and the tile sizes do not take their maximum/minimum values; the \(c\) values are only slightly affected by the compiler, even by using aggressive compilers and high optimization levels. The \(c\) values for different unroll factor values and data layouts are significantly changed and cannot be predicted.

3.3 Performance Models

The aforementioned transformations affect \(P\) and \(E\) in all HW components and thus a different power model is generated for each different processor and MM. An off-line training phase is applied in order to generate the power equations for the target processor and MM (Fig. 3); the custom HW architecture is given as input to the mcpat [14] simulator and a number of simulations takes place for different values of L1,L2,DDR accesses and integer, FP instructions. This way we generate \(P\) equations (Fig. 3); we show that memory power values are linear to their number of memory accesses (Eq. 6-10). Moreover, LoadQ/StoreQ power values are linear to the number of L/S instructions (Eq. 6). The ALU, instruction buffer and instruction decoder power values are linear to the number of ALU instructions and total number of instructions (Eq. 10+6), respectively (Fig. 3). Although this work can be extended to take into account more HW architecture components, in this paper, we approximate \(P\) by using Eq. 11; thus, according to mcpat we do not take into account \(P\) on the renaming unit, instruction cache, RF, TLBs, branch predictor and instruction scheduler.

Figure 3: Power consumption models

Figure 4: Validation of Eq.6-Eq.10 (relative error)
\[
P = P_{L1}(f(L1.\text{acc})) + P_{L2}(f(L2.\text{acc})) + P_{\text{DDR}}(f(\text{DDR.\text{acc}})) + \\
P_{\text{Queue}}(f(L/S.\text{instrs})) + P_{\text{ALU}}(f(\text{ALU.\text{instrs}})) + P_{\text{Instr.\text{buffer}}}(f(\text{instrs})) + P_{\text{Instr.\text{decoder}}}(f(\text{instrs}))
\]

As far as execution time (ET) is concerned, it cannot be approximated by using a mathematical formula; however, we can use current ET models in order to find/predict qualitatively the transformation parameter set giving the fastest binary. Given that all the candidate transformation parameter sets a) refer to the same algorithm, b) the algorithm is static, in Subsection 4.1, we show that we are able to select qualitatively a high quality transformation parameter set, even by using a simple execution time model (Average Memory Access Time (AMAT) [7]). Although more accurate and complex ET models exist like [22], where concurrency in memory hierarchy is taken in account, the aim of this first version is to validate and describe the theoretical background.

3.5 Putting it all together

The proposed methodology is given in Algorithm 1. All the steps have been explained in the previous subsections. All different combinations of loop interchange are generated as it affects Eq. 1-Eq. 10.

In the case that the target metric is not ET or E, but the minimum number of L1 memory accesses, then Algorithm 1 is changed accordingly, i.e., steps (1, 2, 5, 8), (1, 3, 5, 8) or (1, 4, 5, 8) are executed only, respectively. It is important to note that in this case the number of different schedules that have to be further processed by Subsection 3.2 is smaller, i.e., the lower bound values of Eq. 1 and Eq. 4 are no longer needed to be that small. For example, by using a larger lower bound value in Eq. 1, register utilization is increased and therefore the number of L1 accesses is reduced; however, these parameters may conflict to those minimizing the number of MM accesses, which may be more critical. Thus, if the target metric is just the L1dc accesses, there is no need to use such a small Eq. 1 lower bound value. The same holds for L2c and MM too.

Algorithm 1 Proposed Methodology

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>parsing</td>
</tr>
<tr>
<td>2.</td>
<td>apply proposed Register blocking algorithm</td>
</tr>
<tr>
<td>for</td>
<td>(all different RF sets) do</td>
</tr>
<tr>
<td></td>
<td>pick a RF transf. set</td>
</tr>
<tr>
<td>3.</td>
<td>apply loop tiling alg. to L1</td>
</tr>
<tr>
<td>for</td>
<td>(all different L1 sets) do</td>
</tr>
<tr>
<td></td>
<td>pick an L1 transf. set</td>
</tr>
<tr>
<td>4.</td>
<td>apply loop tiling alg. to L2</td>
</tr>
<tr>
<td>for</td>
<td>(all different L2 sets) do</td>
</tr>
<tr>
<td></td>
<td>pick an L2 transf. set</td>
</tr>
<tr>
<td>5a.</td>
<td>generate access equations - Eq. 6-Eq. 9 (all mems)</td>
</tr>
<tr>
<td>5b.</td>
<td>compute the num of accesses in memory hierarchy</td>
</tr>
<tr>
<td>6.</td>
<td>arithmetical instructions</td>
</tr>
<tr>
<td>if</td>
<td>(the num of arith. instrs cannot be predicted for the current set) then</td>
</tr>
<tr>
<td></td>
<td>generate C code (from C to C) for the current set</td>
</tr>
<tr>
<td></td>
<td>generate assembly code - cross compile</td>
</tr>
<tr>
<td></td>
<td>measure the num of FP and integer assembly instrs (get the c values of Eq. 10)</td>
</tr>
<tr>
<td>else</td>
<td>predict the num of arith. instrs (Eq. 10)</td>
</tr>
<tr>
<td>end</td>
<td>if</td>
</tr>
<tr>
<td>7.</td>
<td>compute the ET,P,E values for the current set</td>
</tr>
<tr>
<td>8.</td>
<td>store only the best set depending on the cost function (ET,E,L1,L2,MM)</td>
</tr>
<tr>
<td>end for</td>
<td>apply loop interchange to L2 iterators and go to step4</td>
</tr>
<tr>
<td>end for</td>
<td>apply loop interchange to L1 iterators and go to step3</td>
</tr>
</tbody>
</table>

4 EXPERIMENTAL RESULTS

The experimental results are obtained by using a) the quad-core Intel i7 6700 CPU (CentoS-7 OS) by using both normal C-code and hand
written code with AVX extensions, b) the embedded ARM Cortex-A9 processor on a Zybo Zynq-7000 FPGA platform using Petalinux wri/t_ten code with AVX extensions, b) the embedded ARM Cortex-A9 methodology for efficient code optimizations and memory management Conference’17, July 2017, Washington, DC, USA.

The bench-suite used in this study consists of six well-known data dominant static kernels taken from PolyBench/C benchmark suite version 3.2 [21]. These are: Matrix-Matrix Multiplication (MMM), Matrix-Vector Multiplication (MVM), Gaussian Blur (3 × 3 filter), Finite Impulse Response filter (FIR), a kernel containing mixed vector multiplication and matrix addition (Gemver) and a multiresolution analysis kernel (Diotgen). The kernels are compiled using gcc 4.8.5 and arm-linux-gnuabi-gcc 4.9.2 compilers, for x86 and arm, respectively (‘O3’). The output of our method is compiled with ‘O2’ optimization level in order the compiler to be.

<table>
<thead>
<tr>
<th>MM-M-ZYBO</th>
<th>MM-V-ZYBO</th>
<th>MVM - ZYBO</th>
<th>MVM - i7(AVX)</th>
<th>FIR - i7(AVX)</th>
<th>FIR - ZYBO</th>
<th>Gemver - i7(AVX)</th>
<th>Gemver - ZYBO</th>
<th>Gaussian Blur - ZYBO</th>
<th>Gaussian Blur - i7(AVX)</th>
<th>MMM - i7(AVX)</th>
<th>MMM - ZYBO</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>1.38E+02</td>
<td>5.10E-01</td>
<td>1.20E+02</td>
<td>5.10E-01</td>
<td>1.20E+02</td>
<td>5.10E-01</td>
<td>1.20E+02</td>
<td>5.10E-01</td>
<td>1.20E+02</td>
<td>5.10E-01</td>
<td>1.20E+02</td>
</tr>
<tr>
<td>best ET</td>
<td>1.55E+01</td>
<td>5.00E-01</td>
<td>1.00E+02</td>
<td>5.00E-01</td>
<td>1.00E+02</td>
<td>5.00E-01</td>
<td>1.00E+02</td>
<td>5.00E-01</td>
<td>1.00E+02</td>
<td>5.00E-01</td>
<td>1.00E+02</td>
</tr>
<tr>
<td>best E</td>
<td>1.79E+01</td>
<td>3.80E-01</td>
<td>8.00E+00</td>
<td>3.80E-01</td>
<td>8.00E+00</td>
<td>3.80E-01</td>
<td>8.00E+00</td>
<td>3.80E-01</td>
<td>8.00E+00</td>
<td>3.80E-01</td>
<td>8.00E+00</td>
</tr>
</tbody>
</table>

Table 1: Evaluation over gcc and hand written AVX code

Table 2: Speedup over hand optimized code and Polly
most cases (Table 1). MMM and Dfiogen are the most data dominant kernels and this is why they achieve the highest memory gains and speedup/energy gains on both CPUs. The proposed methodology achieves about (8.5, 30, 2.1, 2.5, 2.3, 3.8) times faster code, for (MMM, diogen, VM, FTR, Gemver, Gaussian Blur), on ZYBO and about (4.4, 3.9), (4, (20, 2.3), (4, 3.8), (1.3, 1.2), (4.8, 1.15) on 17 comparing to the (gcc, hand optimized code), respectively. Regarding energy gains, the proposed methodology achieves about (9.2, 40, 2.5, 2.7, 2.6, 5) times less energy on ZYBO and about (4.6, 4), (20, 2.4), (3.7, 3.6), (1.6, 1.2), (4.8, 1.15) on 17 comparing to the (gcc, hand optimized code), respectively. It is important to note that smaller gain values are achieved on ZY processor because the input C-code contains AVX intrinsics, either directly or indirectly (gcc auto-vectorization). The proposed methodology achieves smaller gain values for SIMD input codes, because hand written AVX-code first, is at a lower level and thus more efficient, second, in many cases it already uses a significant number of the available registers, leaving less space for modifications and third, it is less friendly to register blocking.

Moreover, our methodology is evaluated over hand written optimized code and Polly [6] (Table 2). A large number of experiments has taken place with 10 different unroll factor values and 10 different loop tiling sizes in order to find the best (in Table 2, ‘1 loop’ refers to best loop and best tile size). We have used normal C-code for ZYBO and hand written C-code using AVX intrinsics for 17. As it was expected, hand written optimized code achieves better or equal performance than gcc in all cases and likewise Table 1, our method achieves smaller gain values on the codes using AVX intrinsics. It is important to note that Polly includes other transformations too, which our methodology does not.

5 CONCLUSION AND FUTURE WORK

In this paper, a novel methodology to six of the most popular and important code optimizations is provided for data dominant static algorithms. Instead of applying heuristics and empirical methods we try to understand how software runs on the target HW and how CO affect ET and P. Moreover, we provide a theoretical model correlating the number of memory accesses and arithmetical instructions with CO parameters, HW parameters and input size. To this end, we make a first but important step towards correlating ET and P with CO, HW architecture and input size.

Our future work includes more accurate and complex execution time models such as [22] as well as extending the P model to the remaining HW components. Moreover, it includes more loop transformations such as loop merge and loop distribution and considers nested loops where the array subscripts are not linear equations of the iterators.

ACKNOWLEDGMENTS

This work is partly supported by the European Commission under H2020-ICT-20152 contract 687584 - Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation (TANGO) project.

REFERENCES


