Compact Multi-Dimensional Kernel Extraction for Register Tiling*

Lakshminarayanan Renganarayana[1] Uday Bondhugula[1] Salem Derisavi[2]
Alexandre E. Eichenberger[1] Kevin O'Brien[1]

[1]IBM T.J. Watson Research Center, Yorktown Heights, New York, USA
[2]IBM Toronto Lab, Ontario, Canada
(lrengan,ubondhug,alexe,caomhin)@us.ibm.com and derisavi@ca.ibm.com

ABSTRACT

To achieve high performance on multi-cores, modern loop optimizers apply long sequences of transformations that produce complex loop structures. Downstream optimizations such as register tiling (unroll-and-jam plus scalar promotion) typically provide a significant performance improvement. Typical register tilers provide this performance improvement only when applied on simple loop structures. They often fail to operate on complex loop structures leaving a significant amount of performance on the table. We present a technique called compact multi-dimensional kernel extraction (COMDEX) which can make register tilers operate on arbitrarily complex loop structures and enable them to provide the performance benefits. COMDEX extracts compact unrollable kernels from complex loops. We show that by using COMDEX as a pre-processing to register tiling we can (i) enable register tiling on complex loop structures and (ii) realize a significant performance improvement on a variety of codes.

1. INTRODUCTION

Register tiling [8, 13] is a well-known technique that increases register locality and the amount of available Instruction Level Parallelism (ILP), while simultaneously decreasing the loop overhead and computational redundancy within the loop body. In its simplest one-dimensional form, this technique reduces to loop unrolling, where a loop body is replicated a fix number of times. Quality of code is generally improved by unrolling as it enables the elimination of redundant computations, the reuse of values via registers, and the increased scheduling flexibility between instances of replicated loop body. Register tiling applies this loop unrolling technique to two or more dimensions of loop structures, and enables higher levels of register locality and ILP. This technique, also known as unroll-and-jam [5, 19] followed by scalar promotion [4, 6], can often result in significant (2x or more) speedups.

*This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

SC09 November 14-20, 2009, Portland, Oregon, USA.

Though register tiling is one of the most profitable loop transformations for improving register locality and instruction level parallelism (ILP) [7, 8, 15], it is often not applied because register tilers usually can handle only simple rectangular loops as inputs. Unfortunately, there is an increasingly large set of loops that have complex (deep imperfectly nested) loop structures. The complexity comes from two sources. First, many applications spend significant amount of time in imperfectly nested loops. Examples are matrix factorizations such as LUD and Cholesky, finite difference methods such as FDTD and ADI, and matrix operations from BLAS level-3. The second source of complexity is from the use of long sequence of loop transformations to optimize locality and coarse-grained parallelism for multi-cores [9, 3]. Though the original source program might have a simpler loop strcuture, the result of these long sequence of transformations is a complex imperfectly nested loop strcuture.

The loops that would benefit from register tiling are often buried deep inside these complex loop nests. Register tilers that attempt to work directly on such complex loop nests often fail to separate out or extract the unrollable loops and end up not applying register tiling at all – as shown in our experimental evaluation (cf. Section 5). This paper proposes a technique that enables register tilers to handle complex inputs and successfully bring their performance benefits to programs optimized for multi-cores. We refer to the task of extracting a loop nest with simple rectangular bounds as *kernel extraction,* and the extracted loop nest, a *kernel.* We propose the use of kernel extraction as an explicit transformation which extracts unrollable kernels and transforms them so that scalar promotion is profitable. Kernel extraction can be viewed as an enabling transformation that makes register tilers more robust against complex code structures (imperfect loop nests, if-conditions, etc.).

To successfully reap the benefits of register tiling, kernel extractors need to have at least the following three features: (i) support for complex imperfect loop nests as input, (ii) compact code size after kernel extraction and (iii) support for coarse-grain parallelism enabling transformations As discussed earlier support for complex loops as inputs is essential. The code size after kernel extraction is becoming increasingly important as accelerators with small, dedicated local-memory are becoming prevalent. The third feature – support for coarse-grain parallelism enabling transformations – is a subtle but important one. At a high level, to achieve scalable parallelization on multi-cores we need coarse-grained parallel units. The transformations (e.g., loop skewing) used for extracting this coarse-grained parallelism require the use of a particular model of loop tiling [3]. We would like the kernel extractor to work with this tiling model so that it can be used on loops optimzied for coarse-grained parallelism. This issue is further discussed in Section 2.3.

	Input loop nest structure	Support for (coarse-grain) parallelism enabling transformations (e.g., skewing of tiles)	Code size after kernel extraction
TLOG [18]	Perfect	No	Compact – minimal
Jimenez et al. [13]	Perfect	Yes	Long – exponential
PrimeTile [10]	Imperfect	No	Long – exponential
COMDEX	Imperfect	Yes	Compact – minimal

Table 1: Comparison of kernel extraction methods. Imperfect loop nests appear in several important scientific applications and support for them is essential. Support for (coarse-grain) parallelism enabling transformations are required for scalable parallelization on multi-cores [3]. Compact code size is preferred in compilation for accelerators with small local stores/I-Cache. COMDEX, the method proposed in this paper, achieves the best in all the desired features.

Table 1 compares our method (COMDEX) with the previous methods for kernel extraction on each of the three features. COMDEX supports all the three features. Whereas none of the previous methods support more than one desirable feature. The limitations of TLOG [18] and PrimeTile [10] are partly due to their intertwining of the extraction of unrollable loop nests with the tiled code generation technique and partly due to their use of unconventional tiling models (see Section 6 for more details). By separating out kernel extraction as an independent transformation and by carefully combining it with conventional tiling models, COMDEX supports transformations of tiles. By using appropriate slices of the iteration space to extract kernels, COMDEX supports imperfect loop nests and compact code size. The code size after kernel extraction in PrimeTile [10] and the scheme of Jimenez et al. [13], grows exponentially with the number of loops tiled. Whereas in COMDEX (as in TLOG [18]), the code size growth is minimal – just another version of the extracted kernel and an if-else condition. The contributions of this paper are listed below.

- A kernel extraction technique, COMDEX, that can support imperfect loop nests, (parallelism enabling) transformations of tiled loops, and produces compact code after kernel extraction. To the best of our knowledge, COMDEX is the first kernel extractor with all these features.
- An implementation of the kernel extractor in the IBM XL compiler, and performance and code size comparison of COMDEX with several other techniques.

The next section provides the background on kernel extraction for register tiling. Section 3 first describes the issues involved in extracting kernels for imperfect loop nests and supporting transformations of tiles. Section 4 introduces the notion of generalized inset describes our kernel extraction algorithm. Section 5 describes the implementation of our technique and evaluates it on a variety of scientific kernels. We outline related work on Section 6 and provide conclusions and directions for future work in Section 7.

2. BACKGROUND: REGISTER TILING AND KERNEL EXTRACTION

Register tiling involves three distinct phases: (i) *tiling* or blocking the loops that needs to be unrolled (ii) *fully unrolling* the loops (iii) applying *scalar promotion* to the unrolled loop body. The unrolling phase consists of unrolling multiple loops and jamming or combining all the unrolled iterations. Such unrolling exposes ILP that can be exploited by the instruction scheduler down the line. The unrolling also exposes opportunities for scalar promotion where invariant array references are promoted to scalars which are then assigned to registers by a register allocator. Together the three phases improve ILP and register locality. A good overview of unroll-and-jam and scalar promotion can be found in [1].

The sufficient legality condition for register tiling of a set of loops is the same as that of regular tiling [23], viz., full permutability of the loops. In this paper, we assume that the loops have been appropriately transformed earlier so that register tiling is legal. For our experiments, we use a scheme that is based on Bondhugula et al. [2, 3] to find an initial compound transformation which optimizes for cache locality / coarse-grained parallelism and makes rectangular tiling valid on the transformed loop nest. Given a loop nest for which register tiling is legal, we address the issues involved in generating the register tiled code, viz., extracting an unrollable kernel from it and transforming it so that scalar replacement can be applied. The rest of this section introduces the concepts involved in register tiling and kernel extraction.

2.1 Full tiles and kernels

Figure 1(a) shows a 2D loop nest typically found in stencil computations. The parallelogram shaped iteration space (for a smaller program size) is shown in Figure 2. Figure 2 also shows two levels of tiling: an outer-level of 4×4 tiling (marked as cache tiles) and inner-level of 2×2 tiling (marked as register tiles). We distinguish here two types of register tiles, viz., *partial* and *full.* Full tiles are those that are completely contained in the iteration space. Partial tiles are partly contained in the iteration space. The origins — lexicographically earliest iteration point — of these tiles are respectively called as *full tile origin* and *partial tile origin*. Since the full tiles are completely contained in the iteration space, the loops that iterate over the points in them can have simple constant bounds which check whether the iterations lie within a given tile. This is the key property that is exploited to extract a kernel.

We call the loops that enumerate tile origins as *tile-loops* and those that enumerate points within a tile as *point-loops.* We would like to fully unroll the 2×2 register tiles to expose ILP and exploit register locality. Unrollers require loops with constant number of iterations, i.e., the difference between the lower and upper bounds of a given loop is a constant. We call a loop nest in which each loop has a constant number of iterations, a *kernel*. We call *kernel extraction* the process of separating out the kernel from a loop nest with complex bounds.

2.2 Kernel extraction using inset

Figure 1(b) shows the tiled-loop nest (generated using HiTLOG [11, 18]). As observed earlier, the bounds of the loops that enumerate the iterations contained in the full tiles have constant number of iterations and are the ones that constitute a kernel – a loop nest that can be directly unrolled. Based on this key observation, we can reduce the problem of extracting the kernel to that of splitting a given set of point-loops into two sets of loops: one enumerating the

```
for (i = 1; i <= Ni; i++)
 for (j = i+1; j <= i+Nj; j++)
   S1(i,j);
```

(a) A 2D iteration space commonly found in stencil computations is shown above.

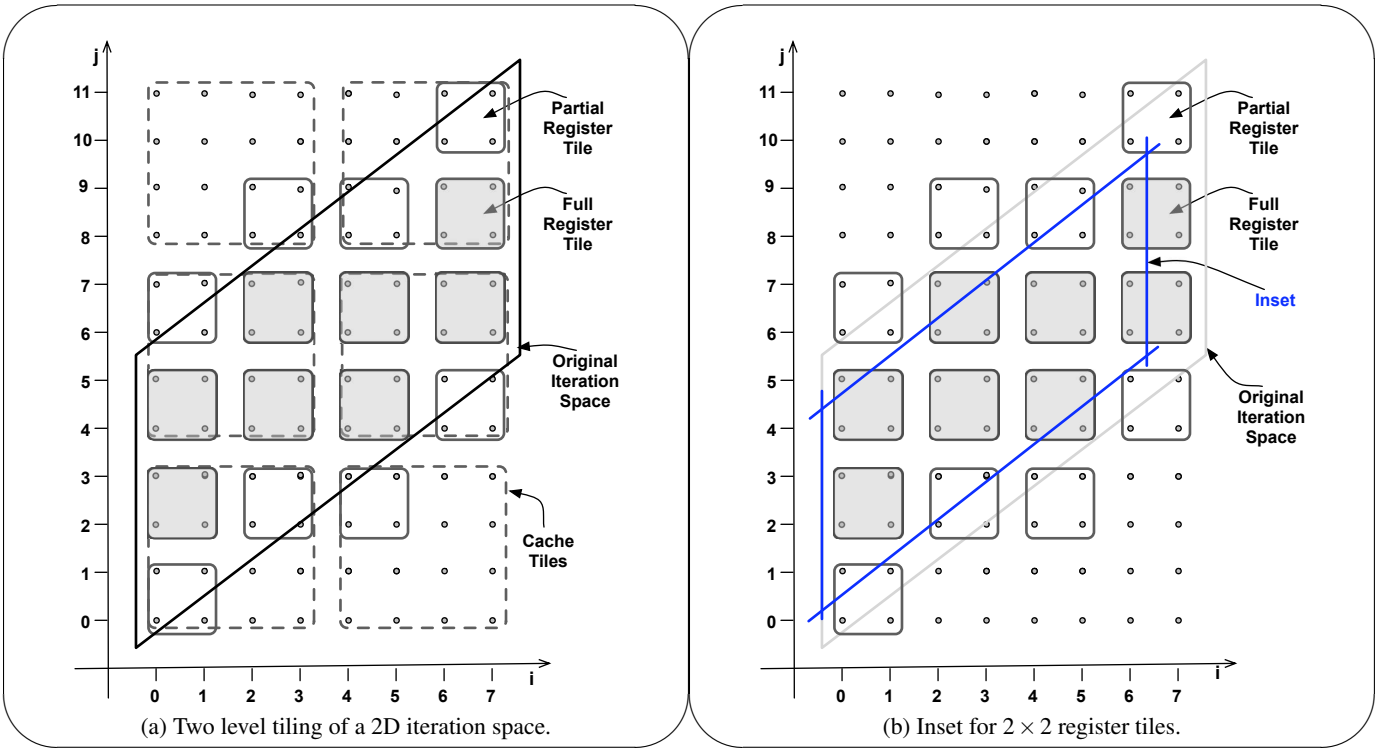
```
iTLB = -Si+2; iTLB = ⌈iTLB/Si⌉*Si;
  for (iT = iTLB; iT <= Ni; iT += Si)
    jTLB = iT-Sj+2; jTLB = ⌈jTLB/Sj⌉*Sj;
    for (jT = jTLB; jT <= iT+Nj+Si-1; jT += Sj)
        for (i= max(iT,1);i<=min(iT+Si-1,Ni);i++)
          for (j= max(jT,i+1);j<=min(jT+Sj-1,i+Nj);j++)
            S1 ;
```

(b) Single level tiled code is shown above. Variables `iTLB` and `jTLB` are used to shift the lower bounds to the nearest tile origins. `Si, Sj` refer to the tile sizes along the i and j directions, respectively.

```
iTLB = -Si+2; iTLB = ⌈iTLB/Si⌉*Si;
  for (iT = iTLB; iT <= Ni; iT += Si)
    jTLB = iT-Sj+2; jTLB = ⌈jTLB/Sj⌉*Sj;
    for (jT = jTLB; jT <= iT+Nj+Si-1; jT += Sj)
      // ----- is (iT,jT) a full tile origin?
      if ( iT-1 >=0 && -iT+Ni-Si+1 >=0 && -iT+jT-Si >=0 && iT-jT+Nj-Sj+1 >=0 ) {
        // ------- Unrollable kernel ---------
        for (i = iT ; i<=iT+Si-1 ; i++ )
          for (j = jT ; j<=jT+Sj-1 ; j++ )
            S1 ;
      } else { // ----- partial tiles --------------
        for (i= max(iT,1);i<=min(iT+Si-1,Ni);i++) {
          for (j= max(jT,i+1);j<=min(jT+Sj-1,i+Nj);j++) {
            S1 ;
      }
```

(c) Code after kernel extraction. Variables `iTLB` and `jTLB` are used to shift the lower bounds to the nearest tile origins. `Si, Sj` refer to the tile sizes along the i and j directions, respectively.

Figure 1: Original and tiled loops (generated using parameterized outset method [18]) are shown in (a) and (b) respectively. We call the loops that enumerate tile origins as tile-loops and those that enumerate points within a tile as point-loops. In (c) note the separation of the kernel with simple rectangular bounds using the inset based condition to check for full tiles.



(a) Two level tiling of a 2D iteration space.

(b) Inset for 2×2 register tiles.

Figure 2: (a) Two level tiling of the 2D iteration space shown in Figure 1. 4×4 cache tiles and 2×2 register tiles are shown. (b) The inset for the 2×2 register tiles is shown. The inset is constructed by shifting in the lines that define upper bounds of the iteration space. Note that the inset contains all the full tile origins and none of the partial tile origins.

iterations in full tiles and the other enumerating iterations in partial tiles. The loops that enumerate the iterations of the full tiles constitute the kernel. In the following discussion we refer to this splitting as the separation of full and partial tiles.

For the case of perfect loop nests, Renganarayanan et al. [18] introduced a polyhedral set called *inset* which can be used to separate the partial and full tiles. The key property of an inset that helps in the separation is that it contains all the full tile origins and none of the partial or empty tile origins. The inset for the 2×2 register tiling is shown in Figure 2(b). It can be efficiently constructed by shifting inside the lines that define the upper bounds of the iteration space. The intuition behind the inset is the following: any point x contained in the inset will be at least $s-1$ points away from the boundary in a given direction, where s is the tile size along the given direction. Hence, if x is a tile origin, the corresponding tile will be completely contained in the iteration space, and hence a full tile.

Figure 1(c) shows the code with the separation of full and partial tiles. The condition in line 6, which checks whether a given tile origin `(iT,jT)` is a full-tile origin, is generated using the inset. Note the simple bounds and the constant number of iterations of the loops in the kernel (lines 8 and 9). If a tile origin corresponds to a full-tile then the kernel is executed. If it is not a full tile origin, then the original set of point-loops with complex bounds are executed. The kernel can be fully unrolled and the array references in the unrolled body can be promoted to scalars. Since full tiles are the majority, a majority of the iterations will enjoy the benefits of unrolling and scalar promotion.

2.3 Support for coarse-grained parallelism enabling transformations

One of the effective ways of extracting coarse-grained parallelism is to first tile the iteration-space and then execute these tiles in parallel [3]. However, inter-tile dependencies might prevent parallel execution of the tiles. In such a case the tiles (i.e., the tile-graph or tiled-loops) are transformed (typically skewed) to expose a wave-front style parallelism between tiles. Such transformations (e.g., skewing) of the tiles and a scheme to generate the transformed tiled-loops are currently known only when the classic (also referred as conventional) tiling model [12, 23] is used. It is not known how to transform the tiles and generate transformed tiled-loops for the unconventional tiling models used by TLOG [18] or PrimeTile [10]. Further, both TLOG and PrimeTile require the use of unconventional tiling models, for all the levels tiling and, more importantly, for the separation of partial and full tiles. Hence the partial/full tile separation scheme proposed by TLOG or PrimeTile cannot be directly applied to the case where we need the classic tiling model to expose coarse-grained parallelism.

The partial vs. full tile separation technique introduced by Renganarayanan et al. [18] is limited to perfect loop nests. In this paper, we develop a generalized inset which can handle imperfect loop nests. Using this generalized inset on slices of the iteration spaces, we are able to extract one or more kernels from a given imperfect loop nest. Our scheme uses the classic tiling model [12, 23] which allows arbitrary transformations (including those required for coarse grain parallelism) of tiled loops. To use the generalized inset with this model, we generate a mapping that maps the tile origins in the conventional model to those required by the inset. The next section introduces the issues involved in extracting

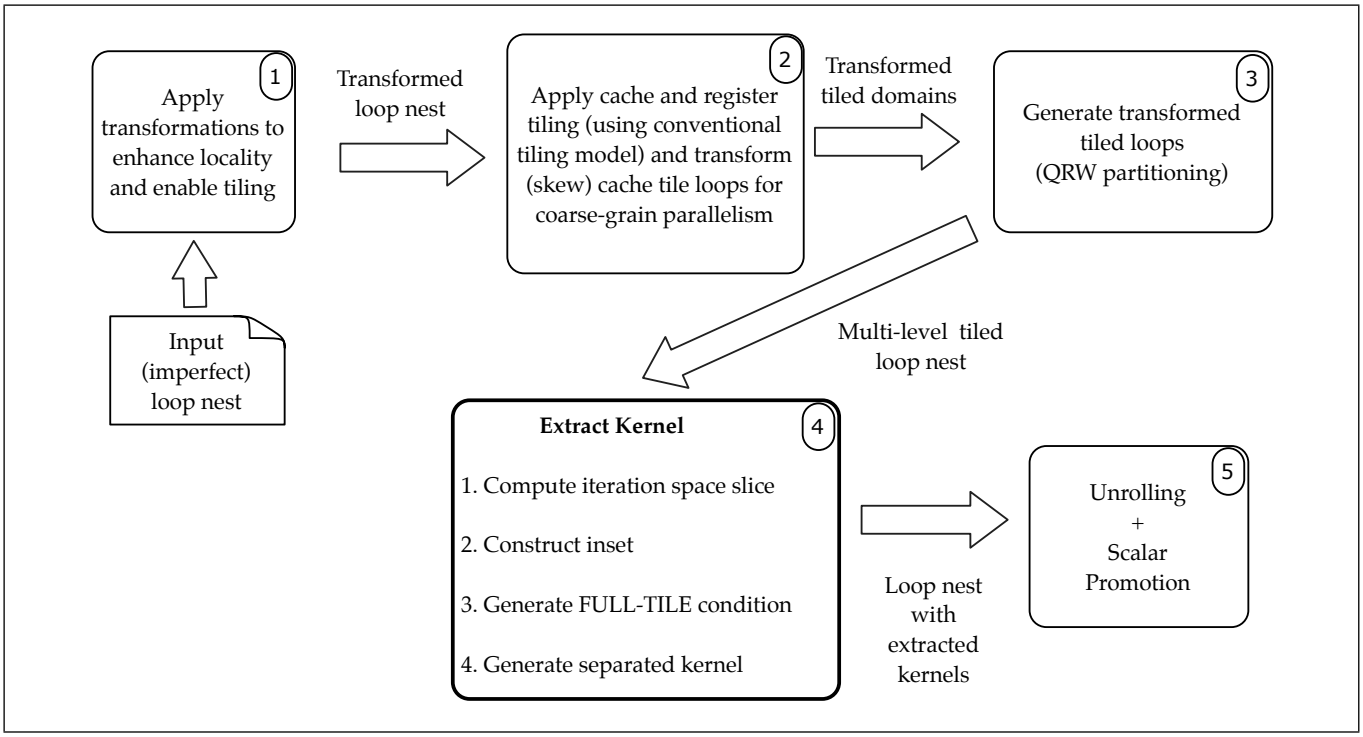


Figure 3: Outline of kernel extraction usage. Note that except for the kernel extraction, the rest are all provided by standard polyhedral loop optimization tools.

kernels from imperfect loop nests using a popular example, viz., imperfect matrix multiplication computation.

3. KERNEL EXTRACTION FROM TILED IMPERFECT LOOP NESTS

We model the kernel extraction as a transformation independent of the other transformations (including tiling) and separate from code generators. It is designed to be used as a pre-processor to the register tiler. This independence from other transformations and support for standard code generators allows us to smoothly integrate COMDEX into a production compiler. A user would typically apply all the parallelism and locality optimizations, including multiple levels of tiling, and then use kernel extraction on the resulting loop structure. After kernel extraction, the user can apply standard unrolling and scalar promotion to the kernel. This typical usage is outlined in Figure 3. Techniques such as TLOG [18] and PrimeTile [10] require that the multi-level tiling and tiled loop generation stages (marked as stages 2 and 3, respectively, in Figure3) be performed by their custom code generators using tiling models that do not support transformations. On the other hand, COMDEX does not have this requirement and can work well with the traditional polyhedral tiling and loop generation model/tools. It is this ability that allows COMDEX to support coarse-grain parallelism enabling transforms such as skewing of tiles.

The input to the kernel extraction algorithm is a multi-level tiled (possibly imperfect) loop nest. The output is another (imperfect) loop nest where the kernel is separated out with an if-else condition as shown in Figure 1(c). The kernel extraction algorithm can be applied to multiple (sub) loop nests in a given imperfect loop nest. In such a case, the resulting loop nest would have multiple kernels separated out. We have found this to be useful in several benchmarks.

Here we use the imperfect matrix multiplication computation as an introductory example. For such a simple computation, typical register tilers use loop distribution to make it perfectly nested and then apply unroll and jam. However, there are several important computations with imperfect loops nests for which loop distribution cannot be used to make them perfectly nested. For such cases we need more powerful techniques, such as the one proposed in this paper, to enable register tiling.

Consider the imperfect matrix multiplication code shown in Figure 4(a). There are two statements, S_1 (initializing elements `C` to 0) and S_2 (computing the result). Let us apply one level of tiling for registers. We first embed the 2D domain of $S1$ into $3D$ with $k = 0$, and then apply tiling to the domains. We can view statement S_1 (due to the embedding of $k = 0$) as executing only in the first iteration of the `k` loop. The tiled domains are then processed by the Quilleré et al. loop generation algorithm [16], which partitions the domains into a disjoint union of domains. The important property of this partitioning is that the resulting set of domains are disjoint, and for every combination of statements we can identify a unique domain that contains its iterations. We refer to the Quilleré et al. loop generation algorithm [16] as *QRW-algorithm* and the corresponding partitioning, with the property mentioned above, as the *QRW-partitioning*.

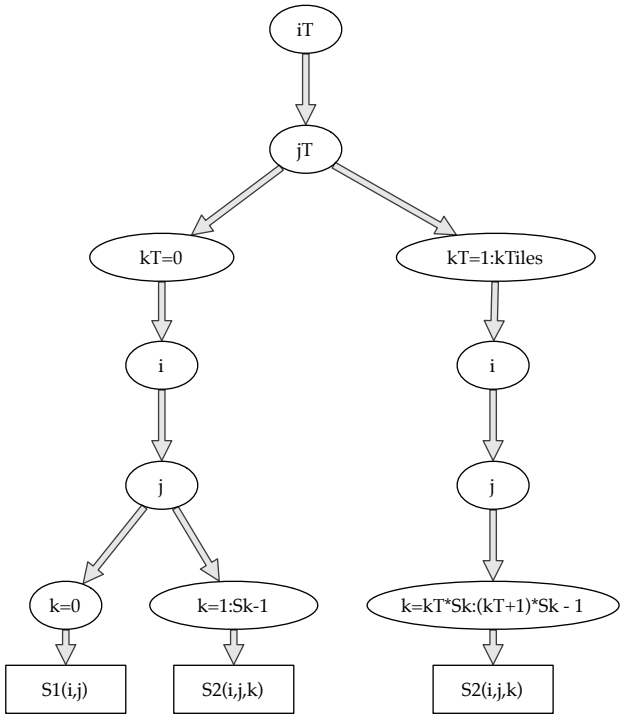
The loop nest resulting from the QRW-algorithm (and partitioning) is shown in Figure 4(b). The resulting loop nest can also be viewed as a tree as shown in Figure 4(c). We use this tree representation as the input to our kernel extraction algorithm. Let us consider extracting the kernel from the point-loops in lines 9, 10, and 11 shown in Figure 4(b) (this corresponds the statement S_2 contained in the rightmost branch of the tree in Figure 4(c)). The kernel extraction involves three important steps, viz., (i) computing the iteration space slice, (ii) constructing the inset, and (iii) generating

```
1  for i = 1 ... N
2   for j = 1 ... N
3     C[i,j] = 0; // S1
4     for k = 1 ... N
5       C[i,j] += A[i,j] * B[j,k]; // S2
```

(a) Pseudo code for imperfect Matrix Matrix Multiplication (MMM)

```
1  for iT, jT
2    for kT = 0
3      for i, j
4        for k = 0
5          S1(i,j)
6        for k = 1:Sk-1
7          S2(i,j,k)
8    for kT = 1:kTiles
9      for i
10       for j
11         for k
12           S2(i,j,k)
```

(b) Pseudo code of imperfect MMM after one level of tiling.



(c) AST after QRW-portioning

Figure 4: Imperfectly nested Matrix Matrix Multiplication.

the full-tile condition. These steps are outlined below.

To start with, we need to identify the set of iterations that are executed by this (lines 9-11 in Figure 4(b)) branch of the tree. We call this the *iteration space slice* of the branch. Note that the execution of the S_1 on the first iteration of the `k` loop results in splitting of the `k` loop and its corresponding tile loop `kT`. This splitting is reflected in the domains of the branches. The key idea is to use this splitting or QRW-partitioning to derive the iteration space slice that correspond to the branch we are interested in (lines 9-11 in Figure 4(b)). We take the domain of the branch and project it on to the point-loop dimensions corresponding to indices `i,j` and `k`. This projection yields the desired iteration space slice.

We use the iteration space slice to compute the inset with the appropriate tile sizes. A formal definition of the concepts of iteration space slice and inset are given in the next section. Once the inset is computed we use it to generate a condition that checks whether a given tile origin is contained in the inset. If it is contained, then it is a full-tile origin. The inset is used to generate an if-else condition as shown in in Figure 1(c) which separates the kernel (full-tile loops with simple bounds) from the partial-tile loops. A formal description of the kernel extraction algorithm and an illustration of it on an example are given in the next section.

4. KERNEL EXTRACTION ALGORITHM

We represent the domain of each statement S_i in the imperfect loop nest by a polyhedron P_i and the domain of the iteration space of the imperfect loop nest by $P = \cup_{i=1}^{n} P_i$, where for $i = 1 \ldots n : P_i$ is the domain of statement S_i. We apply a sequence of statement-wise affine transformations [3] to optimize for cache locality and coarse-grained parallelism. The result of this transformation is a set of transformed domains for each statement. The union of these transformed domains is then processed by the polyhedral loop generator, which uses the QRW-algorithm. Recall that the resulting loops have the QRW-partition property: the resulting set of domains are disjoint, and for every combination of statements we can identify a unique domain that contains its iterations.

We denote this disjoint union of domains, produced from QRW-partitioning by $D = \cup_{j=1}^{x} D_j$ where each domain D_j is a single polyhedron and corresponds to the set of iterations where a unique set of statements are executed. The projection of the each D_j onto the point-loop dimensions gives the slice of the original iteration space which would contain all the full-tile origins. Formally, for any D_j for $j = 1 \ldots x$, the *iteration space slice,* X_j, is defined as $X_j = \text{projectOnto}(D_j, \vec{k})$, where projectOnto() projects the domain D_j onto the dimensions given by $\vec{k}$.

4.1 Generalized inset

We exploit the QRW-partition property to define the generalized inset. The key insight behind our generalized inset is the following: *given a set of statements, there is a unique domain associated with them after QRW-partitioning and the inset of these statements can be directly computed on the iteration space slice induced by the QRW-partition.* In contrast to the inset for perfect loop nests [18], we do not use the original iteration space domain, instead we use the domains induced by the QRW-partitions to compute the inset.

The notion of inset used here is based on Renganarayanan et al. [18]. An iteration space slice X is represented by a polyhedron defined as

$$X = \{\vec{z} \quad | \quad Q\vec{z} \geq (\vec{q} + B\vec{p})\} \tag{1}$$

where, $\vec{z}$ is a $d-$dimensional vector, q is a constant vector of size m, $\vec{p}$ is a vector of size n containing the symbolic parameters of

Algorithm 1 Kernel extraction (COMDEX) algorithm.

Inputs:
`Sin` : an AST representation of a multi-level tiled imperfect loop nest; n: number of loops tiled; `pLoops` : set of point loop numbers; `tLoops` : vector of tile loop numbers; $\vec{s}$: vector of register tile sizes.
Outputs:
`Sout` : an AST with the extracted kernel; `kLoops` : list of kernel loops

1. *Extract domains.* Extract the set of statements enclosed by the `pLoops`. Let D_i be the domain of these statements and D' be the union of these domains.
2. *Compute iteration space slice.* Let D be the projection of the D' on to the n inner most dimensions, which correspond to the original iteration space dimensions. This projection gives us a slice of the original iteration space, that corresponds to the region of iterations executed by the point-loops.
3. *Compute inset.* Compute the inset for D using the tile sizes $\vec{s}$. Let `Inset` denote the compute inset.
4. *Compute mapping.* Scale tile-loop indices (t_i) to map them from tile space numbers to iteration space coordinates. For $i = 1 \ldots n : t_i = t_i \times s_i$.
5. *Generate full-tile condition.* Generate an if condition, using the inset, that checks whether the scaled tile-loop indices belong to the `Inset` or not.
6. *Generate kernel loops.* For $i = 1 \ldots n$: extract the pair of bounds in `pLoops`[i] that has exactly s_i iterations. Construct the kernel loop nest, say `kLoops` with these bounds for point-loops.
7. *Insert if-condition and kernel into AST.* Delete the original point-loops (`pLoops`) from `Sin`. In its place, insert the if-condition with the kernel (`kLoops`) as the true branch and the original point-loops (`pLoops`) as the false branch. Return `kLoops` and the modified AST.

the iteration space, and B is a $m \times n$ matrix. The rectangular tiling related to the kernel extraction is defined by the d dimensional constant vector $\vec{s}$. The generalized inset X_{in} is defined as

$$X_{in} = \{\vec{z} \quad | \quad Q\vec{z} \geq (\vec{q} + B\vec{p}) - Q'\vec{s'}\} \tag{2}$$

where, $\vec{s'} = \vec{s} - \vec{1}$ and Q' is defined as follows

$$Q'_{ij} = \begin{cases} Q_{ij}, & \text{if } Q_{ij} < 0 \\ 0, & \text{if } Q_{ij} \geq 0 \end{cases} . \tag{3}$$

Given an iteration space slice X, the generalized inset corresponding to it can be computed directly based on Eq. 3, in time linear in the number of constraints that define X. The correctness of the generalized inset X_{in} follows from the correctness of the inset [18] for perfect loop nests and the QRW-partition property that the slice X contains all the iterations related to the given set of statements.

4.2 Mapping between tiling models

As discussed earlier, support for transformation of tiled loops is required to enable coarse-grained parallelization. Motivated by this we use the conventional tiling model to tile the loops. In the conventional tiling model [12, 23] the the tile-loops enumerate tile origins in the dense tile space coordinates. On the other hand, the full-tile condition computed using the inset is defined on the original iteration space coordinates. Our scheme generates the tiled loop nest using conventional tiling model (and hence support parallelism enabling transformations) and then maps the tile origins from the dense tile space coordinates to the original iteration space coordinates.

Let $t_1, t_2, \ldots t_n$ be the tile origins in the dense tile space coordinates, enumerated by a set of n tile-loops. The full-tile condition derived from inset is constructed by taking a conjunction of the linear constraints that define the inset. Let $f(i_1, i_2, \ldots, i_n, p_1, \ldots, p_m) \geq 0$ be one such constraint, where $i_k : k = 1 \ldots n$ are the index variables in the iteration space coordinates and $p_k : k = 1 \ldots m$ are the program parameters. The required mapping is achieved by scaling each i_k by the appropriate tile size s_k. That is, each constraint $f(i_1, i_2, \ldots, i_n, p_1, \ldots, p_m) \geq 0$ is transformed to $f(s_1 i_1, s_2 i_2, \ldots, s_n i_n, p_1, \ldots, p_m) \geq 0$. For a full-tile condition, this mapping can be directly computed with the constraints that define the inset and the tile sizes.

4.3 The COMDEX algorithm

The basic COMDEX algorithm for extracting a kernel from an abstract syntax tree (AST) representation of the imperfect loop nest is given in Algorithm 1. The steps of the algorithm and possible extensions are discussed below.

Domain extraction. In this step, we first extract the statements that are enclosed in the given point-loops. These statements could possibly contain if conditions. We compute the domains D_i (or iteration spaces) of these statements and then use their union, D', as the representative domain of all the statements. By using the union we are guaranteed that the full tiles belonging to all the branches in the body of the point-loops will be accounted for.

Iteration space slice computation. In this step, we compute the slice of the original iteration space which corresponds to the iterations that will be executed by the given point-loops. We compute this by projecting the domain D' on to the n inner most dimensions. This projection, D, is then used to compute the inset.

Inset computation. The tile sizes $\vec{s}$ and the domain D are used to compute the inset X_{in} using Eq. 3. Here the inset is computed assuming all the dimensions are tiled. We use degenerate cases of tiling, such as a tile size of 1 or a tile size that is equal to the maximum number of iterations along the particular dimension, to achieve partial tiling. The standard polyhedral code generators remove any extra point/tile loops that result from such degenerate tilings. Hence there are no run-time overheads due to this.

Mapping generation. The inset based condition to check for full tiles expects the tile origins to be in original iteration space coordinates. However, the tile origins enumerated by tile-loops from conventional tiling models are in the (dense) tile space numbers. We need to map these to the iteration space coordinates. This is achieved by scaling each of the tile-loop index by the appropriate tile size.

Full-tile condition generation. Here we exploit the property of the inset that a tile origin corresponds to a full-tile origins if and only if it belongs to the inset. We generate a condition that checks whether a given tile origins belongs to the inset or not, by generating a set of linear conditions on the tile-loop indices. These linear constraints are a direct translation of the constraints that define the inset polyhedron X_{in}. Hence, they can be constructed very efficiently.

Kernel loops generation. In this step, we go through each of the point-loops in `pLoops` and we extract the pair of bounds for each loop with a constant number of iterations that are exactly equal to

the tile size along the corresponding dimension. We then create the kernel loop nest using new loops with the constant iteration bounds.

Kernel insertion. Using the full-tile if-condition constructed in step 5, and the kernel loops generated in the previous step, we build a sub-tree for the if-else block with the full-tile check as the if-condition, and the kernel loops as the true branch and the original point-loops as the false branch. The sub-tree in the input AST that corresponds to the point-loops is replaced by this new if-else block.

4.4 Algorithm walk-through on example

Consider the example imperfect loop structure shown in the left box of Figure 5. Two levels of tiling are shown. The `iCT, jCT,` and `kCT` loops correspond to the cache level tile-loops and the `iRT, jRT, kRT1,` and `kRT2` loops correspond to the register level tile-loops. The `kRT1` and `kRT2` loops create an imperfect loop nest structure and the if condition around statement `S3` further complicates the structure. Let us consider extracting the kernel for the point-loops `i2,j2,` and `k2` that contain the statements `S2` and `S3` and the if condition. The code with the extracted kernel is shown in Figure 5(b).

Our algorithm first extracts the iteration spaces or domains of the statements `S2` and `S3`. We take the union of the two domains. In this case, the if condition makes the domain of `S3` a subset of the domain of `S2`. We project the domain onto the 3 inner-most dimensions (corresponding to the point-loops) to extract the slice of the iteration space that corresponds to the iterations executed by the point-loops `i2,j2,` and `k2.` We construct the inset using this domain and the register tile sizes s_1, s_2 and s_3. We then generate an if condition that checks whether a given tile origin – a (`iRT,jRT,kRT2`) iteration – is a full tile origin, i.e., whether (`iRT,jRT,kRT2`) belongs to the inset. We generate a new set of point-loops for the kernel with constant (tile size) iterations. We then generate an if condition with the kernel loops in the true branch and the original point-loops in the false branch, and replace the original set of point-loops with this new if condition (and the containing loops). This last step of replacement can be seen by observing the changes between the code in lines 9-12 of Figure 5(a) and lines 6-17 of Figure 5(b).

5. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We have implemented the COMDEX kernel extraction technique as a part of the polyhedral optimization system of the IBM XL production compiler. The polyhedral optimization system performs automatic parallelization and locality optimization using a refinement of the scheme proposed by Bondhugula et al. [2, 3]. The optimization scheme uses a script to specify and drive the optimizations — a script is automatically generated for each program and then the compiler applies the optimizations in the script. Such a scheme facilitates automatic and/or user driven tuning of the parameters of the transformations (e.g., tile sizes, unroll factors) and the order in which transformations are applied. We have implemented the kernel extraction technique as a command usable via the scripting interface. This facilitates auto-tuning of the register tile sizes and the order of the inter- and intra-tile loops.

5.1 Evaluation

We evaluate the performance gains due to COMDEX based register tiling by comparing it with the three different schemes, viz., XLSMP, AutoPoly, and PrimeTile. XLSMP denotes the XLC compiler's automatic parallelization option which includes a rich set of traditional loop optimizations, including register tiling. AutoPoly

Name	Description	Loop nest structure	Program size used
STRMM	Product of triangular and square matrices	Perfect nest	3000
TMM	Triangular matrix product	Perfect nest	3000
DSYRK	Symmetric rank k update	Perfect nest	3000
DSYR2K	Symmetric rank $2k$ update	Perfect nest	3000
STRSM	Matrix equation solver	Imperfect nest	3000
SSYMM	Symmetric matrix matrix operation	Imperfect nest	2000
TRISOLV	Multiple triangular solver	Imperfect nest	3000
LUD	Lower Upper Decomposition	Imperfect nest	2000

Table 2: Summary of benchmarks used for evaluation.

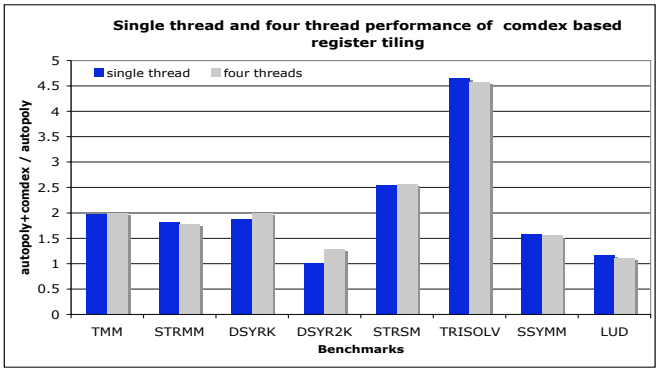


Figure 7: Performance improvement of COMDEX based register tiling. Note that the register tiler was unable to realize this performance improvement without kernel extraction, since the unroller and scalar promotion passes were turned on with the autopoly optimization mode too, but were not applied due to the complexity of the loop structure.

```
1  // iCT,jCT,kCT : cache tile loops
2  // iRT,jRT,kRT : register tile loops
3  // i, j, k : point-loops
4  for iCT, jCT, kCT
5     for iRT, jRT
6       for kRT1 = lb1 ... ub1
7         for i1, j1, k1
8               S1;
9       for kRT2 = lb2 ... ub2
10        for i2, j2, k2
11          S2;
12          if (f(i2,j2,k2)) S3;
```

(a) Example imperfect loop nest structure.

```
1  for iCT, jCT, kCT
2     for iRT, jRT
3       for kRT1 = lb1 ... ub1
4         for i1, j1, k1
5               S1;
6       for kRT2 = lb2 ... ub2
7         if ( FULL-TILE(iRT,jRT,kRT2) ) {
8           for i2 = lb1 ... lb1 + S1
9            for j2 = lb2 ... lb2 + S2
10            for k2 = lb3 ... lb3 + S3
11              S2;
12              if (f(i2,j2,k2)) S3;
13        } else {
14           for i2,j2,k2 //orig point-loops
15             S2;
16             if (f(i,j,k)) S3;
17        }
```

(b) Example after kernel extraction.

Figure 5: Example imperfect loop nest structure before (a) and after kernel extraction (b). Note that (in (b)) the point-loops (`i,j,k`) of the kernel have constant iterations well suited for unrolling. `iCT,jCT,kCT` refer to the cache tile-loops and `iRT, jRT, kRT1, kRT2` refer to register tile-loops. The short hand construct `for iCT, jCT, kCT` is used to represent a sequence of nested loops.

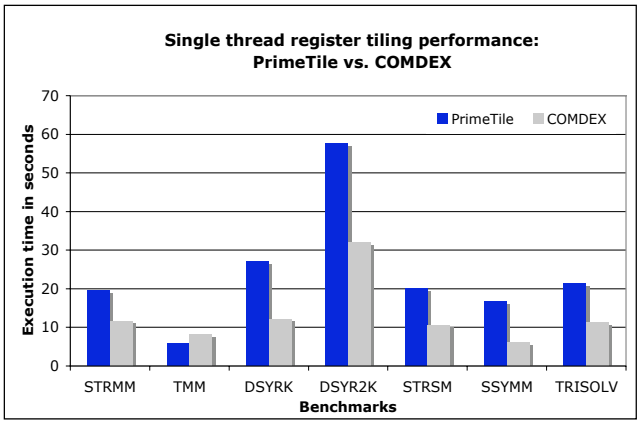


Figure 8: Single thread performance comparison of kernel extraction + register tiling: PrimeTile vs. COMDEX. Note that for some benchmarks COMDEX produced kernels are 2x or more faster PrimeTile.

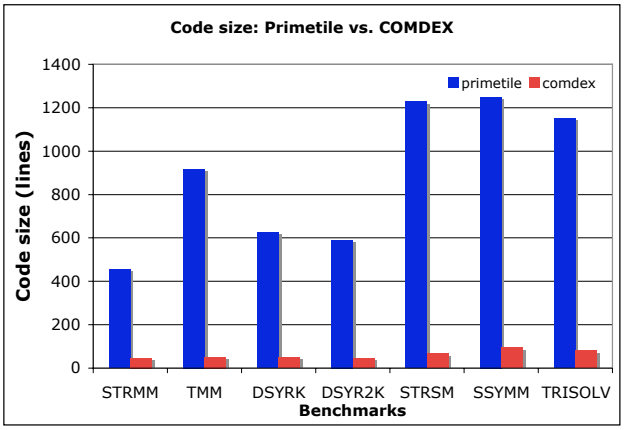


Figure 9: Comparison of the size of the code after kernel extraction: PrimeTile vs. COMDEX. The kernels extracted by COMDEX are a factor of 11x to 19x smaller than those extracted by PrimeTile.

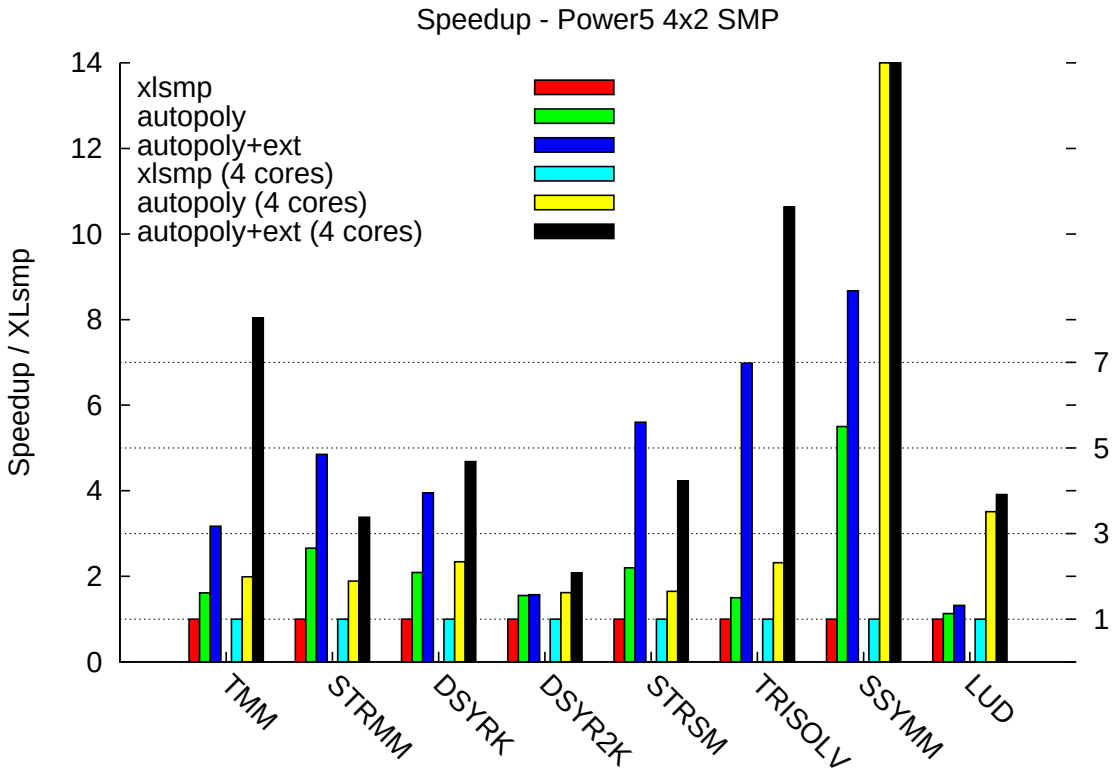


Figure 6: Execution time speedup of autopoly and autopoly+ext over XLsmp is shown for eight kernels. The baseline XLsmp corresponds to the automatic parallelization and optimization of the XL compiler. The autopoly corresponds to polyhedral automatic parallelization system of the XL compiler. The autopoly+ext corresponds to the addition of kernel extraction based register tiling to autopoly.

denotes the polyhedral automatic parallelizer and locality optimizer implemented in the XL compiler framework. PrimeTile [10] is the state-of-the-art kernel extractor for imperfect loop nests. Further, we also evaluate the compactness of the code size after kernel extraction by comparing COMDEX with PrimeTile.

To highlight the importance of transformation of tiles to obtain coarse-grained parallelism, and the benefits of applying register tiling to these coarse-grain parallel loops, we compare the performance of XLSMP, AutoPoly and AutoPoly with COMDEX on a quad core Power5 machine.

For our evaluation, we use a variety of kernels from scientific computations and matrix operations such as BLAS. All the kernels used for experiments have non-rectangular iteration spaces and half of them are perfect loop nests and the rest are imperfect loop nests. Table 2 provides a summary of the benchmarks and their characteristics. All the progams were compiled with `XLC` version `10.1` with `-O3 -qhot -qsmp=auto` option and were executed on a quad core Power5 machine.

Figure 7 shows the speedup obtained purely from register tiling using the COMDEX technique. This graph highlights the amount of performance left on the table without kernel extraction, since, the unrolling and scalar replacement passes were present in the AutoPoly optimization mode too, but were not applied due to the complex loop structure.

Figure 8 compares the single thread performance of register tiling using COMDEX to that of PrimeTile [10]. Note that for 6 out of 7 benchmarks COMDEX out performs PrimeTile — for some kernels COMDEX is 2x or more better than PrimeTile.

Figure 9 compares the code size (in terms of number of lines) after kernel extraction. The code generated by COMDEX is very compact and is around a factor of 11x to 19x smaller than the code generated by PrimeTile. The increase in code size of a kernel extracted by COMDEX is very minimal, in fact, it increases exactly by the number kernel loops (equal to the number of loops tiled) plus two, one for the if condition and another for the else statement. Other than this, the unrolling of the loops will introduce copies of statements equal to the unroll factor. But, this will be incurred by any unroller/register tiler.

The speedup of Autopoly and AutoPoly + COMDEX with respect to XLSMP is shown in Figure 6. The first set of three bars for each benchmark shows the single thread performance and the second set of bars shows the performance for 4 threads. The sin-

gle thread performance for AutoPoly and AutoPoly + COMDEX are normalized with respect to the single thread performance of XLSMP and similarly the 4 thread performance for AutoPoly and AutoPoly + COMDEX are normalized with respect to 4 thread XLSMP performance. Note that the coarse-grain parallelization done by AutoPoly (via skewing of tiles) is more scalable and performs better than the fine-grain parallelization done by XLSMP. This highlights the importance of coarse-grain parallelization using transformation of tiles and performance benefits of regsiter tiling of these coarse-grained parallel loops.

In summary, the experimental results presented in this section show that COMDEX can successfully enable register tiling for complex codes and bring substantial performance improvements. Further, it can generate very compact code and support parallelism enabling transformations, which is shown here to provide scalable parallel performance.

6. RELATED WORK

Several aspects of register tiling have been studied: legality condition [4, 5, 20], efficient generation of register tiled code [20], cost models [20, 24, 17] and empirical searches for selecting unroll factors [22, 14]. Through their pioneering work on multi-level tiling, Carter et al. [7, 8, 15] showed the significant benefits of register tiling in the context of sequential as well as parallel applications.

However, there has been limited work on kernel extraction for register tiling. Some techniques [10, 18, 13], as a part of tiled loop generation, can separate out unrollable kernels. Table 1 compares these techniques with ours. Imperfect loop nests appear in several important scientific applications. Techniques proposed by TLOG [18] and Jimenez et al. [13] are not applicable to them. Further, TLOG and PrimeTile [10] do not support transformations on tiled loops. As discussed in Section 1 and evidenced through the experimental results in Section 5, these transformations are important to achieve scalable parallel performance. The limitations of TLOG [18] and PrimeTile [10] are partly due to their intertwining of the extraction of unrollable loop nests with the tiled code generation technique and partly due to their use of unconventional tiling models.

Vasilache [21] proposes a one-dimensional kernel extractor that works on a given loop and extracts the pair of lower and upper bounds with constant iteration difference. In simple cases, one can repeatedly apply this one dimensional extractor to extract a multi-dimensional kernel. For more complex loop structures, it is not clear how this technique can be automatically applied to multiple dimensions. Further, similar to other dimension by dimension approaches such as PrimeTile, the size of the code after kernel extraction, increases exponentially with the number of loops tiled.

In comparison, COMDEX supports transformations of tiles by separating out kernel extraction as an independent transformation and by carefully combining it with conventional tiling models. Further, it supports imperfect loops nests and compact code size by using appropriate slices of the iteration space to extract kernels.

7. CONCLUSIONS AND FUTURE WORK

Register tilers work well with simple loop structures. However, modern loop optimizers produce complex loop structures which are hard for register tilers to process. In this paper, we have proposed a separate transformation, COMDEX, which extracts simple unrollable kernels from complex loop structures. We have shown that by using COMDEX as a pre-processing to loop unrolling and scalar promotion, we can achieve significant performance improvement even for fairly complex loop structures. Further, we have also shown that COMDEX can support coarse-grain parallelism enabling transformations, which are essential for scalable parallelization. Extensive experimental results demonstrate the advantages of COMDEX over previous techniques in terms of performance as well as code size.

As immediate future work, we plan to integrate the kernel extraction based register tiling with an empirical optimization approach to select kernels to register tile and to select the unroll factors. An empirical selection of the order of tile- and point-loops is another promising direction for locality optimization. Kernel extraction techniques are also applicable in other contexts such as vectorization/SIMDization that benefit from loops with simple bounds.

8. REFERENCES

[1] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[2] Uday Bondhugula, M. Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (ETAPS CC)*, April 2008.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[4] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 53–65, New York, NY, USA, 1990. ACM Press.

[5] Steve Carr and Yiping Guan. Unroll-and-jam using uniformly generated sets. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–357, Washington, DC, USA, 1997. IEEE Computer Society.

[6] Steve Carr and Philip Sweany. An experimental evaluation of scalar replacement on scientific benchmarks. *Software Practice and Experience*, 33(15):1419–1445, 2003.

[7] L. Carter, J. Ferrante, F. Hummel, B. Alpern, and K.S. Gatlin. Hierarchical tiling: A methodology for high performance. Technical Report CS96-508, UCSD, Nov. 1996.

[8] Larry Carter, Jeanne Ferrante, and Susan Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 239–245, Washington, DC, USA, 1995. IEEE Computer Society.

[9] Albert Cohen, Sylvain Girbal, David Parello, M. Sigler, Olivier Temam, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.

[10] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 147–157, New York, NY, USA, 2009. ACM.

[11] HiTLoG: Hierarchical Tiled Loop Generator. Available at: `http://www.cs.colostate.edu/MMAlpha/HiTLoG/`.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[13] Marta Jiménez, José M. Llabería, and Agustín Fernández. Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst.*, 24(4):409–453, 2002.

[14] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):247–270, 2004.

[15] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, June 1998.

[16] Fabien Quilleré;, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal Parallel Programming*, 28(5):469–498, 2000.

[17] Lakshminarayanan Renganarayana, Ramakrishna Upadrasta, and Sanjay Rajopadhye. Optimal ILP and register tiling: Analytical model and optimization framework. In *LCPC 2005: 12th International Workshop on Languages and Compilers for Parallel Computing*. Springer Verlag, 2005.

[18] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *PLDI '07: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 405–414, New York, NY, USA, 2007. ACM Press.

[19] Vivek Sarkar. Optimized unrolling of nested loops. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 153–166, New York, NY, USA, 2000. ACM Press.

[20] Vivek Sarkar. Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 29(5):545–581, 2001.

[21] Nicolas Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Université de Paris-Sud, INRIA Futurs, September 2007.

[22] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[23] Jingling Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[24] K. Yotov, Xiaoming Li, Gang Ren, M. J. S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93:358–386, 2005.