Optimal Parallelogram Selection for Hierarchical Tiling

XING ZHOU, MARÍA J. GARZARÁN, and DAVID A. PADUA,
University of Illinois at Urbana-Champaign

Loop tiling is an effective optimization to improve performance of multiply nested loops, which are the most time-consuming parts in many programs. Most massively parallel systems today are organized hierarchically, and different levels of the hierarchy differ in the organization of parallelism and the memory models they adopt. To make better use of these machines, it is clear that loop nests should be tiled hierarchically to fit the hierarchical organization of the machine; however, it is not so clear what should be the exact form of these hierarchical tiles. In particular, tile shape selection is of critical importance to expose parallelism of the tiled loop nests. Although loop tiling is a well-known optimization, not much is known about tile shape selection.

In this article, we study tile shape selection when the shapes are any type of parallelograms and introduce a model to relate the tile shape of the hierarchy to the execution time. Using this model, we implement a system that automatically finds the tile shapes that minimize the execution time in a hierarchical system. Our experimental results show that in several cases, the tiles automatically selected by our system outperform the most intuitive tiling schemes usually adopted by programmers because of their simplicity.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Hierarchical tiling, tile shape, compiler optimizations

ACM Reference Format:
Xing Zhou, María J. Garzarán, and David A. Padua. 2014. Optimal parallelogram selection for hierarchical tiling. ACM Trans. Architec. Code Optim. 11, 4, Article 58 (December 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2687414

1. INTRODUCTION

Most highly parallel systems are organized hierarchically. Often, different levels in the hierarchy have different organizations and follow different memory models. For example, in a computer cluster, the nodes connected by a network form the first level of the hierarchy. At the next level, each node is typically a shared-memory multiprocessor, which could be connected to one or more GPGPUs. The processors within the GPGPU form a third level of the hierarchy.

Hierarchy-aware optimizations are usually necessary to unleash the potential of hierarchically organized systems. Loop tiling is an effective optimization to improve the performance of multiply nested loops, which are the most time-consuming parts of many programs. Tiling can be applied to improve data locality [Wolf and Lam 1991b; Wolfe 1989a; Abu-Sufah et al. 1981; Ahmed et al. 2000; Song and Li 1999; Coleman and McKinley 1995; Ramanujam and Sadayappan 1991] and expose parallelism [Wolf and Lam 1991a; Wolfe 1989b; Andonov et al. 2001; Högstedt et al. 1999; Wonnacott 2000].

Authors' address: X. Zhou, M. J. Garzarán, and D. A. Padua, Department of Computer Science, University of Illinois at Urbana-Champaign; emails: {zhou53, garzaran, padua}@illinois.edu.

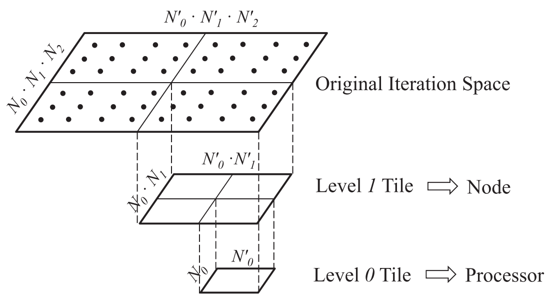
DOI: http://dx.doi.org/10.1145/2687414

```
for(int $i = 0$; $i < N$; $i$++)
  for(int $j = 0$; $j < N'$; $j$++)
    do_compute($i$, $j$);
```

(a) A 2-depth loop nest

```
for(int $i_2 = 0$; $i_2 < N_2$; $i_2$++)
  for(int $j_2 = 0$; $j_2 < N'_2$; $j_2$++)
    for(int $i_1 = i_2 \cdot N_1$; $i_1 < i_2 \cdot N_1 + N_1$; $i_1$++)
      for(int $j_1 = j_2 \cdot N'_1$; $j_1 < j_2 \cdot N'_1 + N'_1$; $j_1$++)
        for(int $i_0 = i_1 \cdot N_0$; $i_0 < i_1 \cdot N_0 + N_0$; $i_0$++)
          for(int $j_0 = j_1 \cdot N'_0$; $j_0 < j_1 \cdot N'_0 + N'_0$; $j_0$++)
            do_compute($i_0$, $j_0$);
```

(b) Hierarchical tiling with rectangle tiles



(c) Match tiling level to hardware hierarchy

Fig. 1. Hierarchical tiling of loop nest.

To make better use of hardware hierarchies, loop nests can also be tiled hierarchically to fit the organization of the target machine. Figure 1(a) shows a doubly nested loop, covering an iteration space $I = \{(i, j) : 0 \leq i < N \wedge 0 \leq j < N'\}$. After applying a two-level tiling, the number of levels of nesting increases to six, as shown in Figure 1(b). The outer two pairs of loops i_k - j_k ($k = 1, 2$) form a subiteration space $I_k = \{(i_k, j_k)\}$ at the corresponding tiling level. Figure 1(c) matches each tiling level to one of the hierarchy levels of a simple cluster with SMP nodes. In top-down order, (1) the iteration space of the loop nest is tiled, and each tile is mapped to a node of the cluster, and (2) on each node, the assigned tile is tiled again. Each of these second-level tiles is assigned to a processor in the node. Hierarchical tiling can also be done bottom-up by partitioning the original iteration space into tiles that are assigned to the lowest level of hardware and then tiles are grouped into larger tiles for higher levels of hardware.

The shape and size of tiles can have a significant impact on locality, intertile communication, and parallelism [Ohta et al. 1995; Xue 1997; Lim et al. 1999]. However, little is known about strategies for the selection of tile shapes. Existing strategies would typically result in loops similar to that shown in Figure 1 where rectangles are used at both levels of tiling. In this article, we extend the study of tile shape selection to the more general shape of parallelograms.

In hierarchical tiling, tile shapes at the different levels interact with each other and together determine execution time. This means that to minimize execution time, it is not sufficient to select the tile shape at each level separately; in some cases, a better tile shape selection is obtained by tackling hierarchical tiling in a global manner. Figure 2 shows an example of different tile shape choices. Arrows represent the dependences between tiles. Dashed lines show the tiles that can be executed in parallel in a wavefront schedule. The numbers stand for the order of the execution of the wavefronts. In the

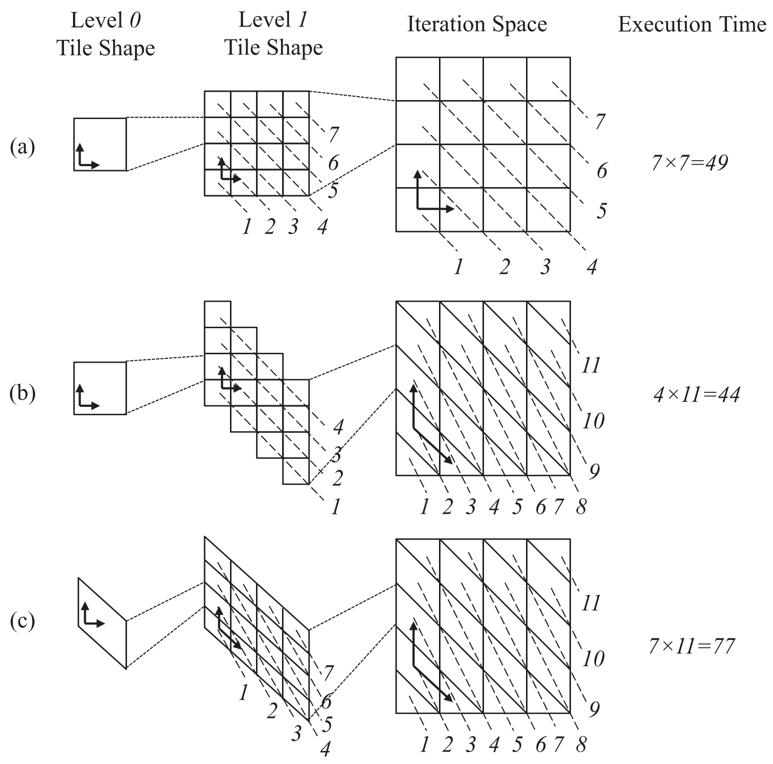


Fig. 2. Different tile shape choices for hierarchical tiling. Arrows are the dependences between tiles. Dashed lines represent the tiles that can be executed in parallel in a wavefront schedule. The numbers stand for the order of the wavefronts.

three cases shown in the figure, the iteration space is constrained by two dependences: $\vec{d}_0 = (1, 0)$ and $\vec{d}_1 = (0, 1)$. There are several possible shapes at each level of tiling. The straightforward choice is to use squares at all levels as in Figure 2(a), where the tiles along the same diagonal line can be executed in parallel. If we assume that the computation time of each tile at the lowest level (level 0) is one unit of time, the total execution time of the schedule in Figure 2(a) is $7 \times 7 = 49$ units of time. However, there are better tile shapes. If we change the level 1 tile shape into a parallelogram, as shown in Figure 2(b), the execution time in terms of units of time becomes $4 \times 11 = 44 < 49$. However, nonrectangular parallelograms are not necessarily always a better solution. If we also choose a parallelogram shape for the level 0 tiling, as shown in Figure 2(c), the total execution time would increase to $7 \times 11 = 77$ units of time (here we assume that the parallelogram tile at level 0 takes the same time to execute as the square tile.).

This article makes the following contributions:

—We present a model that computes execution time as a function of the tile shapes. The model assumes unlimited resources so that tile shape is the only factor that determines execution time. This model is used to guide tile shape selection.
—We show that the problem of optimal tile shape selection for hierarchical tiling is a nonlinear bilevel programming problem.
—We describe the implementation of an automatic system for the selection of tile shapes in a hierarchical system. When the best tile shapes are not rectangular, this system can find good choices in the space of parallelograms.

The rest of the article is organized as follows. Section 2 discusses related work. Section 3 introduces basic concepts and the assumptions used in the rest of the article. Section 4 describes the model that computes ideal execution time in terms of tile shape. Section 5 discusses the implementation of the automatic tile shape selection system to find good tile shapes. Section 6 present our experimental results. Section 7 concludes the article.

2. RELATED WORK

Tiling is a well-studied transformation for the optimization of loop nests [Wolf and Lam 1991a, 1991b; Wolfe 1989a, 1989b; Abu-Sufah et al. 1981; Ahmed et al. 2000; Song and Li 1999; Coleman and McKinley 1995; Ramanujam and Sadayappan 1991; Andonov et al. 2001; Högstedt et al. 1999; Wonnacott 2000; Baskaran et al. 2009; Bondhugula et al. 2008]. It is useful to improve locality and/or parallelism. Most of the research in the area has focused on regular or intuitive tile shapes such as rectangles and regular parallelograms. However, studying more general shapes is important, because as our experiments show, more general shapes of parallelograms have the potential to achieve higher performance than what is possible with the simpler shapes mentioned previously that can be produced with the help of simple transformations such as loop skewing. Notice that although the work by Bondhugula et al. [2008] can also find tiles that belong to this more general class of parallelograms, their approach only considers one level of tiling, whereas our approach takes into account all levels of tiling.

It is well known that tile shape, as well as tile size, can have a significant impact on locality, intertile communication, and parallelism. Xue [1997] presents an approach to find the parallelogram or hyperparallelepiped[1] tile shape that minimizes communication volume between tiles. This work does not consider the impact of tile shape on parallel scheduling, although minimizing communication between tiles can sometimes result in higher parallelism. Högstedt et al. [1999] introduced a model to select the tile shape that minimizes execution time. Since their work only considers a single level of tiling, they show that for a single level of tiling, the best execution time can be determined by solving a linear programming problem. None of these papers discusses the tile shape selection problem in the context of hierarchical tiling.

The idea of taking advantage of the hierarchy of the hardware is not new. Liu et al. [2011] propose a cache hierarchy-aware tile scheduling technique to maximize data reuse. Leung et al. [2010] implemented a C-to-CUDA compiler that performs hierarchical decomposition for GPUs. Hierarchical tiling has been proposed to improve performance in the presence of memory hierarchy and to enhance parallel schedule [Carter et al. 1995; Hartono et al. 2009; Carter et al. 1996; Guo et al. 2006; Bikshandi et al. 2006; Zhou et al. 2012]. In Zhou et al. [2012], we introduced hierarchical overlapped tiling, a transformation that tries to reduce the overhead introduced by overlapped tiling by taking into account the hierarchy of the machine (overlapped tiling reduces communication overhead by introducing redundant computation). Tiles in overlapped and hierarchical overlapped tiling always have the shape of a trapezoid. In addition, to minimize redundant computation, the slope of the side edges of the trapezoid is determined by the slope of the extreme dependence vectors. Thus, the only adjustable parameter that can impact the shape of the trapezoid is the height of the trapezoid. The approach discussed in this article can use any parallelogram shape, as long as it is compatible with the constraints imposed by the dependence vectors. Section 6 compares the execution times obtained by the approach that we propose in this article with those obtained using hierarchical overlapped tiling.

[1]A *3D parallelepiped* is a three-dimensional figure formed by six parallelograms.

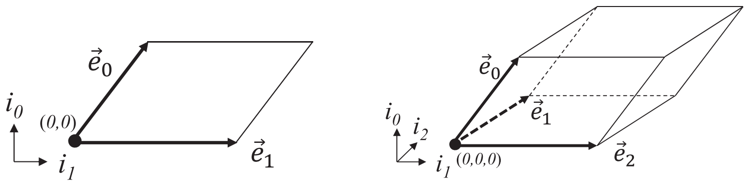
```
for(int $i_0 = lb_0$; $i_0 < ub_0$; $i_0$++)
  for(int $i_1 = lb_1$; $i_1 < ub_1$; $i_1$++)
    ...
      for(int $i_{n-1} = lb_{n-1}$; $i_{n-1} < ub_{n-1}$; $i_{n-1}$++)
        << loop body >>
```

(a) Loop nest with scalar induction variables

```
for($\vec{i} = (i_0, i_1, ..., i_{n-1}) \in I = [lb_0..ub_0] \otimes ... \otimes [lb_{n-1}..ub_{n-1}]$)
        << loop body >>
```

(b) The representation with vector induction variables

Fig. 3. An n-depth loop nest.



(a) Two-dimensional space (b) Three-dimensional space

Fig. 4. The edge vectors (thick arrows) of iteration spaces.

The work of Renganarayana and Rajopadhye [2004] is closest to that presented in this article. Their work presents a model to estimate the overall execution time of loop nests. Their framework determines the optimal tile sizes for hierarchical tiling by solving a convex optimization problem. However, this study only considers hyperrectangular shapes.

3. DEFINITIONS

This section defines the terms used in the rest of the article.

3.1. Iteration Space Representation

The iteration space of the loop nest shown in Figure 3(a) is $I = \{\vec{i}\} \subseteq \mathcal{Z}^n$, where $\vec{i}$ is the vector of loop index values. This loop nest can also be represented as shown in Figure 3(b). To simplify the problem, we restrict the shape of the iteration space to n-dimensional hyperparallelepipeds. We use n edges of the hyperparallelepiped sharing a common vertex to represent the n-dimensional iteration space and choose the common vertex as the origin. Figure 4 gives two examples of the edge vectors that define two- and three-dimensional iteration spaces. Assume that $\vec{e}_k = (e_{k,0}, e_{k,1}, \ldots, e_{k,n-1})$, $k = 0, 1, \ldots, n-1$ are the n edge vectors of the hyperparallelepiped-shaped iteration space I. The *edge matrix* of I is

$$E = \begin{pmatrix} \vec{e}_0 \\ \vec{e}_1 \\ \cdots \\ \vec{e}_{n-1} \end{pmatrix}.$$

We define the function *span*(E) as follows:

$$\begin{aligned} span(E) &= \{\vec{i} : \vec{i} \in \mathcal{Z}^n \mid \exists \vec{a} = (a_0, a_1, \ldots, a_{n-1}), \\ \vec{0}_n &= (0, 0, \ldots, 0) \le \vec{a} \le (1, 1, \ldots, 1) = \vec{1}_n, \vec{i} = \vec{a} \cdot E\}. \end{aligned}$$

```
for ($\vec{i'} = (i'_0, i'_1, ..., i'_{n-1}) \in I'$)
    for ($\vec{j} = (j_0, j_1, ..., j_{n-1}) \in J$)
        << loop body >>
```

Fig. 5. Loop nest after tiling.

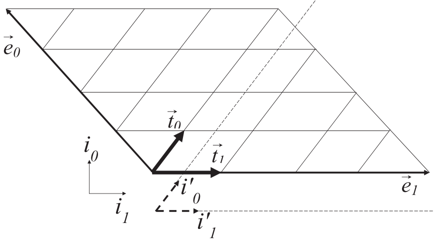


Fig. 6. The effect of tiling transformation.

Clearly, $I = span(E)$, and the total number iterations in I is $|det(E)|$ (because $|det(E)|$ is the volume of the parallelepiped).

3.2. Tiling Transformation Representation

When all dimensions are tiled,[2] tiling is a map from $\mathcal{Z}^n$ to $\mathcal{Z}^{2n}$: $I \rightarrow [I' : J] = \{(\vec{i'} : \vec{j})\}$, in which $\vec{i'} = (i'_0, i'_1, \ldots, i'_{n-1})$ is the index for each tile and $\vec{j} = (j_0, j_1, \ldots, j_{n-1})$ is the iteration index within each tile. We only consider disjoint tiles; thus, for each $\vec{i} \in I$, there is a unique $(\vec{i'} : \vec{j})$ corresponding to it and vice versa.

The loop nest in Figure 3 is transformed by tiling into the form shown in Figure 5. The top n loop nests form a new n-dimensional iteration space I'. We can then tile recursively to produce a hierarchically tiled loop nest. We say that I' is the tiled iteration space of the original iteration space I.

Again, to simplify the problem, we assume that the shape of all nonboundary tiles are the same n-dimensional hyperparallelepiped. Assume that $\vec{t}_k = (t_{k,0}, t_{k,1}, \ldots, t_{k,n-1})$, $k = 0, 1, \ldots, n-1$ are the n edges of the hyperparallelepiped tile; we call the following matrix T *tiling matrix*:

$$T = \begin{pmatrix} \vec{t}_0 \\ \vec{t}_1 \\ \cdots \\ \vec{t}_{n-1} \end{pmatrix}.$$

Each nonboundary tile, J, is equal to $span(T)$ and contains $|det(T)|$ iterations.

To study how I, I', and T are related, we derive an equation involving E, E', and T, where E' is the edge matrix of I' ($I' = span(E')$). If we describe tiling as an affine transformation, the shape of I' is guaranteed to be an n-dimensional hyperparallelepiped because the shape of I is hyperparallelepiped. Since we assume that tiling is an affine transformation, we must be able to find an $n \times n$ matrix A such that

$$E' = E \cdot A.$$

Next, consider the effect of tiling transformation represented by A. As shown in Figure 6, after tiling, under the new coordinates system shown as axis labels i'_0 and i'_1, the vectors $\vec{t}_0$ and $\vec{t}_1$ become $(1, 0)$ and $(0, 1)$, respectively. More generally, we have the

[2]The problem becomes simpler if one or more dimensions are not tiled.

following equation ($\mathbb{1}_n$ denotes the $n \times n$ identity matrix):

$$T \cdot A = \begin{pmatrix} \vec{t}_0 \\ \vec{t}_1 \\ \dots \\ \vec{t}_{n-1} \end{pmatrix} \cdot A = \begin{pmatrix} 1, 0, \dots, 0 \\ 0, 1, \dots, 0 \\ \dots \\ 0, 0, \dots, 1 \end{pmatrix} = \mathbb{1}_n.$$

We conclude that $A = T^{-1}$ and $E' = E \cdot T^{-1}$. We say that T^{-1} represents the affine transformation of the tiling transformation with tile shape T.

3.3. Hierarchical Tiling

As mentioned in Section 3.2, it is possible to recursively tile an iteration space I. Hierarchical tiling applies tiling at each level. We follow a bottom-up approach for hierarchical tiling. First, we use tiling matrix T_0 to tile the original iteration space $I_0 = I$, producing a new iterations space I_1. More generally, on the k-th level of tiling, we apply tiling matrix T_k to tile the iteration space I_k and generate the new iteration space I_{k+1}. Assume that E_k is the edge matrix of I_k, then

$$I_k = span(E_k), \quad E_{k+1} = E_k \cdot T_k^{-1} = E_0 \cdot \prod_{j=0}^{k} T_j^{-1}.$$

3.4. Dependences

Dependences are the partial order requirements that must be enforced between iterations. Two iterations can be scheduled either concurrently or in any sequential order if there are no direct or indirect dependences between them. Otherwise, the iteration at the source of the dependence must finish before the iteration at the destination of the dependence starts.

Dependences can be represented by dependence vectors. A dependence vector $\vec{d} = (d_0, d_1, \dots, d_{n-1})$ indicates that any iteration $\vec{i}$ must finish before iteration $\vec{i} + \vec{d}$. For a loop with n levels of nesting and m dependence vectors $\vec{d}_0, \vec{d}_1, \dots, \vec{d}_{m-1}$, we define the following $m \times n$ matrix D as the *dependence matrix*:

$$D = \begin{pmatrix} \vec{d}_0 \\ \vec{d}_1 \\ \dots \\ \vec{d}_{m-1} \end{pmatrix}.$$

Without loss of generality, we assume that $m \geq n$ and there are n dependence vectors that are linearly independent.[3]

The dependences determine which tiling transformations are valid. In fact, the new loop nest after tiling must preserve the dependences existing in the original loop nest. In this work, we assume that the computation within a tile is *atomic*, which means that intertile communication (resulting from an intertile dependence) only happens before innertile computation starts and/or after innertile computation is finished. This requirement means that it must be possible to topologically sort all tiles, and as a

[3]Otherwise, the iterations of one of the loop would be independent of each other. This loop can be ignored for the purposes of tiling so that the problem can be simplified into one with a lower dimensional iteration space.

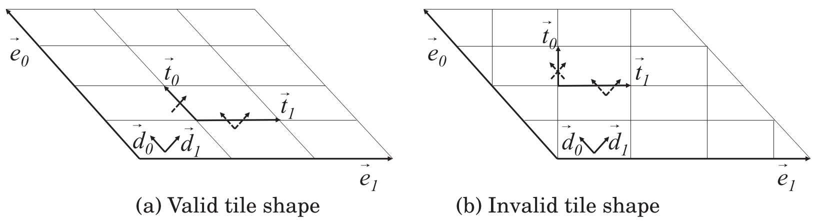


Fig. 7. Valid and invalid tile shapes.

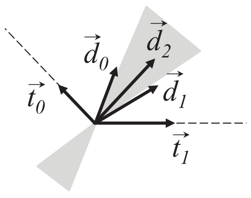


Fig. 8. Constraints of tile shape imposed by dependences.

result, that there can be no cycles in the intertile dependence graph, which means that the hyperplanes defining the tiles must not be crossed by two dependence vectors with different directions.

Figure 7 gives two examples of tile shapes. The tile shape shown in Figure 7(a) is valid, because dependence vectors only cross the hyperplane in one direction. For example, for the tile boundaries that are parallel to $\vec{t}_0$, only dependence vector $\vec{d}_1$ can go across from the left side to the right side; for the tile boundaries that are parallel to $\vec{t}_1$, both dependence vector $\vec{d}_0$ and $\vec{d}_1$ can go across, although always from the lower side to the upper side. This means that for each boundary hyperplane, the tile has either in-bound dependences or out-bound dependences. However, the tile shape in Figure 7(b) is invalid, as there are both in-bound and out-bound dependences crossing the boundary hyperplane of $\vec{t}_0$. This means that the horizontal neighboring tiles are in a dependence cycle.

Figure 8 depicts the restriction that the topological sorting requirement imposes on the dependences. To produce a valid tile shape without a dependence cycle between tiles, the infinite cone spanned by extending the tile edges $\vec{t}_0, \vec{t}_1, \ldots, \vec{t}_{n-1}$ must contain every dependence vector.[4] Thus, for the example in Figure 8, $\vec{t}_0$ and $\vec{t}_1$ must not be in the shaded area. To describe the preceding observation formally, we say that each dependence vector $\vec{d}_k$ must be covered by the cone spanned by the extension cords of $\vec{t}_0$, $\vec{t}_1, \ldots, \vec{t}_{n-1}$,

$$\begin{gathered}\forall \vec{d}_k, \quad \exists \vec{a} = (a_0, a_1, \ldots, a_{n-1}) \geq \vec{0}_n = (0, 0, \ldots, 0) \\ \vec{d}_k = a_0 \cdot \vec{t}_0 + a_1 \cdot \vec{t}_1 + \cdots + a_{n-1} \cdot \vec{t}_{n-1} = \vec{a} \cdot T.\end{gathered} \tag{1}$$

We further require that tiles be large enough so that intertile dependences only exist between adjacent tiles (including diagonal adjacency). This requirement can be expressed as follows:

$$\begin{gathered}\forall \vec{d}_k, \quad \exists \vec{a} = (a_0, a_1, \ldots, a_{n-1}) \leq \vec{1}_n = (1, 1, \ldots, 1) \\ \vec{d}_k = a_0 \cdot \vec{t}_0 + a_1 \cdot \vec{t}_1 + \cdots + a_{n-1} \cdot \vec{t}_{n-1} = \vec{a} \cdot T.\end{gathered} \tag{2}$$

[4]If $\vec{d}_i$ is in the infinite cone spanned by extending the tile edges $\vec{t}_0, \vec{t}_1, \ldots, \vec{t}_{n-1}$, it must be possible to get $\vec{d}_i$ from a linear combination of $\vec{t}_0, \vec{t}_1, \ldots, \vec{t}_{n-1}$.

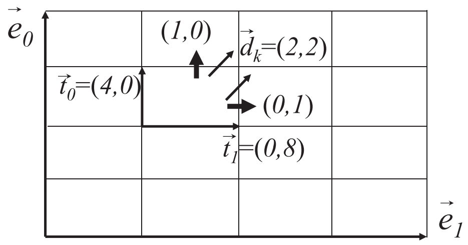


Fig. 9. Intertile dependences (thicker arrows) generated by original dependence vector $\vec{d_k}$.

The preceding two requirements can be merged into the following:

$$\forall \vec{d_k}, \quad \exists \vec{a}, \quad \vec{0} \le \vec{a} \le \vec{1}, \quad \vec{d_k} = \vec{a} \cdot T, \tag{3}$$

which is the constraint imposed by dependences when choosing the tile shape.

When the original iteration space I is tiled by matrix T, each dependence vector $\vec{d_k}$ is also transformed to $\vec{d'}_k$ by the affine transformation represented by T^{-1}. Since T must satisfy the constraints of Equation (3), we have

$$\vec{d'}_k = \vec{d_k} \cdot T^{-1} = \vec{a} \cdot T \cdot T^{-1} = \vec{a}, \quad \vec{0}_n \le \vec{a} \le \vec{1}_n. \tag{4}$$

As shown in Figure 9, a dependence vector $(2, 2)$ would result in three dependence vectors after tiling: $(1, 0)$, $(0, 1)$, and $(1, 1)$. In general, a single dependence vector $\vec{d_k}$ of the original iteration space I generates one or more dependences between tiles in the tiled iteration space I'. The rule is as follows: for $\vec{d'}_k = \vec{a} = (a_0, a_1, \ldots, a_{n-1})$, if $a_j > 0$, there is a dependence $(0, \ldots, 1, \ldots, 0)$ imposed on the j-th dimension in I'. If there are n dependence vectors $\vec{d_{k_0}}, \vec{d_{k_1}}, \ldots, \vec{d_{k_{n-1}}}$ that are linearly independent, the intertile dependences in I' generated by those n dependence vectors must include n orthonormal unit vectors.

From the previous observation, we conclude that D_k, which is the dependence matrix at the k-th level of tiling, must have the following form:

$$D_0 = D,$$
$$D_k = \begin{pmatrix} \vec{d}_{k,0} \\ \vec{d}_{k,1} \\ \cdots \\ \vec{d}_{k,m-1} \end{pmatrix} = \begin{pmatrix} 1, 0, \ldots, 0 \\ 0, 1, \ldots, 0 \\ \cdots \\ 0, 0, \ldots, 1 \\ \cdots \end{pmatrix} = \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix},$$
$$\forall \vec{d}_{k,j}, \quad k \ge 1, 0 \le j < m, \quad \vec{0} \le \vec{d}_{k,j} \le \vec{1}. \tag{5}$$

3.5. Execution Model

Assuming that the sequential execution time only depends on the amount of computation, and that the execution time of each iteration is constant, the sequential execution time of a loop with iteration space I would be

$$Time_s(I) = \hat{t}_c \cdot |I| = t_c \cdot |det(E)|. \tag{6}$$

Here $\hat{t}_c$ is the execution time of each iteration in iteration space I.

The execution time of a parallelized iteration space depends on the schedule of iterations within the iteration space. To not be tied to any specific scheduling scheme, we use the concept of ideal execution time in our analysis. *Ideal execution time* is the minimal execution time of an iteration space I that can be achieved by any valid schedule of iterations. If each iteration in I is viewed as an activity and D represents

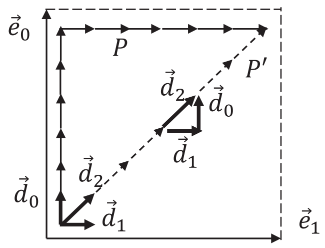


Fig. 10. The dependent paths (P and P') within an iterations space. The length of a dependent path is the number of iterations on the path. If any path contains $\vec{d}_2$ (P'), we can always replace $\vec{d}_2$ with $\vec{d}_1$ and $\vec{d}_0$ on the same position, which results in a longer path. Hence, the longest path (P) must only contain $\vec{d}_0$ and $\vec{d}_1$.

the dependences between activities, the ideal execution time is determined by the critical path in I. To simplify the problem, we assume that there is infinite hardware parallelism available at each level. In practice, this assumption means that the amount of hardware parallelism is larger than the maximum number of iterations that can be executed in parallel. Therefore, the critical path in I is the longest path of dependences between iterations.

Let $L(E, D)$ denote the length of the longest path of dependent iterations in the iteration space I with edge matrix E and dependence matrix D; the ideal execution time of I is

$$Time(I) = \hat{t}_c \cdot L(E, D). \tag{7}$$

Section 4.2 discusses how to calculate $L(E, D)$. In general, $L(E, D)$ depends on every $\vec{d}_k$ in D. However, if the dependence vectors are not linearly independent, we can ignore some rows in D when calculating $L(E, D)$. For example, the iteration space shown in Figure 10 has three dependence vectors $\vec{d}_0$, $\vec{d}_1$, and $\vec{d}_2$, with $\vec{d}_2 = \vec{d}_0 + \vec{d}_1$. P and P' are two dependence paths within the iteration space. If there is any dependence path containing a pair of neighboring iterations with the dependence of $\vec{d}_2$, such as P', we can always replace $\vec{d}_2$ on the path with a combination of $\vec{d}_0$ and $\vec{d}_1$, as shown in Figure 10. This will result in a longer dependence path. As a result, we can conclude that the longest dependence path must only contain $\vec{d}_0$ and $\vec{d}_1$ (P), and other dependence vectors can be ignored[5] when calculating $L(E, D)$.

In particular, if there are n dependence vectors that are orthonormal unit vectors, we can ignore other rows in D and only consider those n dependence vectors, because any other single dependence vector can be replaced by the combination of one or more orthonormal unit vectors, which will result in a longer dependence path. Based on the preceding observation, we have Equation (8):

$$L\left(E, \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix}\right) = L(E, \mathbb{1}_n). \tag{8}$$

According to Equation (5), under the context of hierarchical tiling, the dependence matrix each at each level $D_k = \begin{pmatrix} \mathbb{1}_n \\ * \end{pmatrix}$, $\forall k \geq 1$. Therefore, Equation (8) can be used to simplify further analysis.

[5]Assume that the iterations space is much larger compared to each dependence vector.

4. MODEL

This section describes a model to estimate the execution time as a function of the tile shape. The model is used to direct tile shape selection.

4.1. Problem Statement

The problem of selecting tile shape for hierarchical tiling is defined as identifying the tile shapes defining an l-level hierarchical tiling that minimizes the execution time of the computation defined by giving an n-dimensional hyperparallelepiped-shaped iteration space I and m dependence vectors. The problem reduces to determining a sequence of tiling matrices $T_0, T_1, \ldots, T_{l-1}$.

The goal of this model is to assess the impact on parallelism of general parallelogram tile shapes in hierarchical tiling. This model does not consider data locality and only focuses on the impact of intertile dependences on parallel scheduling. Our model assumes that maximal exposure of parallelism (instead of sophisticated data reuse schemes) is the dominant factor determining overall performance. Thus, the execution time predicted by this model can be inaccurate if the preceding assumption is not true. However, this assumption should be true if the scale of the system or cluster used is large enough compared to the size of the input data. In addition, as indicated in Section 3, the model makes the following assumptions:

(1) The model considers general parallelogram tile shapes—that is, parallelograms for two-dimensional spaces and hyperparallelepipeds for spaces with higher number of dimensions.
(2) At a given level, all nonboundary tiles have the same shape.
(3) Tiling is an affine transformation.
(4) Computation within a tile is atomic.
(5) There is infinite hardware parallelism available at each level.
(6) The model takes the iteration space as input.

We follow a bottom-up approach to do hierarchical tiling. First, tiling matrix T_0 is selected. T_0 tiles the original iteration space $I_0 = I$. Then, each tile is considered as an iteration of a new iteration space, I_1. On the k-th level of tiling, tiling matrix T_k is selected to tile the iteration space I_k and generate the new iteration space I_{k+1}. Let E_k be the edge matrix of the k-th level iteration space I_k (with $E_0 = E$, the edge matrix of I), then

$$E_{k+1} = E_k \cdot T_k^{-1} = E_0 \cdot \prod_{j=0}^{k} T_j^{-1}, \quad 0 \le k < l.$$

The iterations within the bottom-level tiles (the finest grain, level 0) are executed sequentially, so the per-tile execution time can be calculated by Equation (6) (t_c denotes the execution time of each single iteration):

$$Time(T_0) = Time_s(T_0) = t_c \cdot |det(T_0)|. \tag{9}$$

For upper-level tiles (T_k, $1 \le k < l$), each tile at the level immediately below is considered as a single iteration. The per-iteration execution time is $Time(T_{k-1})$. Let D_k denote the dependences at the k-th level with $D_0 = D$ and t_s^k be the synchronization and/or communication overhead associated to each tile at this level; the per-tile execution time is

$$\begin{aligned} Time(T_k) &= \left(Time(T_{k-1}) + t_s^k\right) \cdot L(T_k, D_k) \quad k > 0 \\ &= \left(Time(T_{k-1}) + t_s^k\right) \cdot L(T_k, \mathbb{1}_n). \end{aligned} \tag{10}$$

Then, the total execution time after tiling is

$$\begin{aligned} Time(I) &= Time(E_l) = \left(Time(T_{l-1}) + t_s^l\right) \cdot L(E_l, D_l) \\ &= \left(Time(T_{l-1}) + t_s^l\right) \cdot L(E_l, \mathbb{1}_n). \end{aligned} \tag{11}$$

Thus, the problem of optimal tile shape selection for hierarchical tiling is equivalent to selecting a sequence of tiling matrices $T_0, T_1, \ldots, T_{l-1}$ to minimize the preceding equation. Equation (11) does not include D; however, according to the discussion in Section 3.4, the shape of T_0 must conform the constraints (Equations (1) and (2)) imposed by each dependence vector in D.

4.2. Compute the Longest Dependent Path

As shown in Equation (11), total execution time is a function of the tiling matrices $T_0, T_1, \ldots, T_{l-1}$. To do a quantitative analysis on the relation between tile shape and execution time, and then find the minimum of Equation (11), we must identify the function L first.

4.2.1. Computing $L(T_k, \mathbb{1}_n)$. $L(T_k, \mathbb{1}_n)$ is the length of the longest dependent path with dependence matrix $\mathbb{1}_n$ in tile T_k. The iteration space in tile T_k can be normalized to an n-dimensional hypercube by applying the affine transformation[6] represented by T_k^{-1}. Thus,

$$L(T_k, \mathbb{1}_n) = L\left(\mathbb{1}_n, T_k^{-1}\right).$$

Consider a row in $\mathbb{1}_n, \vec{d}_j$. Since $D_k = \binom{\mathbb{1}_n}{*}$, $\vec{d}_j$ is also a row in D_k. Because T_k determines the tile shape at the k-th level of tiling, T_k must conform to the constraints imposed by each dependence vector in D_k. According to Equations (1) and (2),

$$\exists \vec{a}_j, \quad \vec{0}_n \le \vec{a}_j \le \vec{1}_n, \quad \vec{d}_j = \vec{a}_j \cdot T_k.$$

Then, let D' denotes the dependences within the tile (normalized tile so that the tile is defined by orthogonal edge vectors):

$$D' = \begin{pmatrix} \vec{d'}_0 \\ \vec{d'}_1 \\ \cdots \\ \vec{d'}_{n-1} \end{pmatrix} = \mathbb{1}_n \cdot T_k^{-1} = \begin{pmatrix} \vec{d}_0 \\ \vec{d}_1 \\ \cdots \\ \vec{d}_{n-1} \end{pmatrix} \cdot T_k^{-1} = \begin{pmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \cdots \\ \vec{a}_{n-1} \end{pmatrix}$$

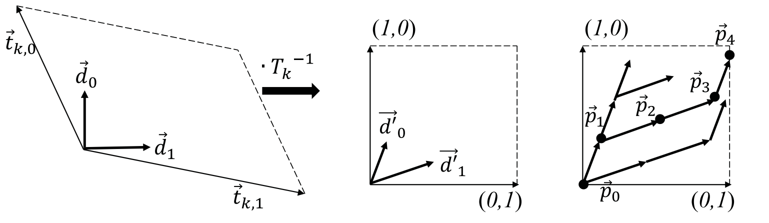
$$\forall \vec{d}_j, \quad \vec{0}_n \le \vec{d'}_j = \vec{a}_j \le \vec{1}_n.$$

In the two-dimensional case, the intuitive explanation of the preceding observation is that the direction of each $\vec{d}_j$ must be in the upward, right, or upper-right direction, as shown in Figure 11(a) and (b). Since the normalized iteration space is a hypercube, the longest path must start from the bottom-left corner, which is the base point $(0, 0, \ldots, 0)$.

To compute $L(T_k, \mathbb{1}_n)$, we must find the longest path P ($\vec{p}_0, \vec{p}_1, \ldots, \vec{p}_{L-1}$, Figure 11(c)) such that

$$\begin{aligned} &\vec{p}_0 = \vec{0} = (0, 0, \ldots, 0), \\ &\forall j = 0, 1, \ldots, L-1, \\ &\qquad \vec{0} = (0, 0, \ldots, 0) \le \vec{p}_j \le (1, 1, \ldots, 1) = \vec{1}, \\ &\qquad \exists \vec{d'}_{r_j}, \vec{p}_{j+1} = \vec{p}_j + \vec{d'}_{r_j}. \end{aligned} \tag{12}$$

[6] To keep the problem in the integer domain, the affine transformation can be revised to $T_k^{-1} \cdot |det(T_k)|$, and the discussion in this section would be still valid.



(a) Original tile (b) Normalized tile (c) Dependent paths

Fig. 11. Determine the length of the longest dependent path within the iteration space of tile T_k. Each $\vec{p}_1$ stands for the coordinate of the point.

L is the length of the dependent path found. Therefore, $L(T_k, \mathbb{1}_n) = max\{L\}$. We have that the last point $\vec{p}_{L-1}$ is defined as

$$\begin{aligned}\vec{p}_{L-1} &= \vec{p}_{L-2} + \vec{d'}_{r_{L-2}} = \sum_{j=0}^{L-2} \vec{d'}_{r_j} \quad 0 \le k_j < n \\ &= c_0 \cdot \vec{d'}_0 + c_1 \cdot \vec{d'}_1 + \cdots + c_{n-1} \cdot \vec{d'}_{n-1} \\ &= \vec{c} \cdot D' \quad \vec{c} = (c_0, c_1, \ldots, c_{n-1}) \ge \vec{0}, \quad \vec{c} \in \mathcal{Z}^n \\ &= \vec{c} \cdot T_k^{-1}.\end{aligned}$$

Because $\vec{0}_n \le \vec{d'}_j \le \vec{1}_n$, if $\vec{0} \le \vec{p}_{L-1} \le \vec{1}$, we also have that $\vec{0} \le \vec{p}_j \le \vec{1}$. And because $\sum_{j=0}^{n-1} c_j$ is the total number of steps of the path, $max\{L\}$ can be found by solving the following linear programming problem $LP(T_k^{-1}, \vec{0}, \vec{1})$:

$$\vec{0} \le \vec{c} \cdot T_k^{-1} \le \vec{1}, \quad \vec{c} \ge \vec{0}, \quad max\left\{\sum_{j=0}^{n-1} c_j\right\}. \tag{13}$$

Therefore, we conclude that $L(T_k, \mathbb{1}_n) = LP(T_k^{-1}, \vec{0}, \vec{1})$.

4.2.2. Computing $L(E_l, \mathbb{1}_n)$. The meaning of $L(E_l, \mathbb{1}_n)$ is the length of the longest dependent path in the iteration space of the highest-level tiling. Similarly, we can apply the affine transformation represented by E_l^{-1} to normalize the iteration space:

$$L(E_l, \mathbb{1}_n) = L\big(\mathbb{1}_n, E_l^{-1}\big).$$

Let D' denote the transformed dependence matrix:

$$D' = \begin{pmatrix} \vec{d'}_0 \\ \vec{d'}_1 \\ \cdots \\ \vec{d'}_{m-1} \end{pmatrix} = \mathbb{1}_n \cdot E_l^{-1} = E_l^{-1}.$$

However, unlike the case when calculating $L(T_k, \mathbb{1}_n)$, there is no guarantee that $\forall \vec{d'}_j, \vec{0} \le \vec{d'}_j \le \vec{1}$. More intuitively, each dependence vector $\vec{d'}_j$ can point in any direction. Therefore, $L(E_l, \mathbb{1}_n)$ cannot be directly solved by the linear programming problem of Equation (13).

Since dependence vectors $\vec{d'}_j$ can point in any direction, the longest dependent path does not necessarily start from origin $(0, 0, \ldots, 0)$ of the hypercube iteration space. Thus, the dependent path P in the iteration space E_l is as defined by Equation (12)

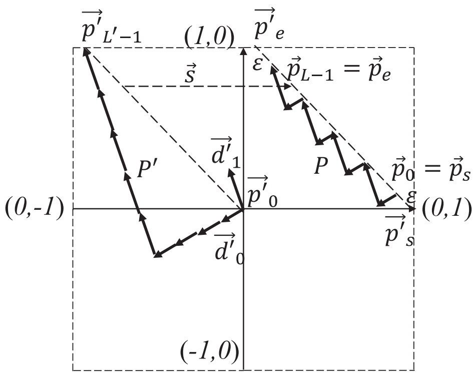


Fig. 12. Construct path P according to a given path P'. The lengths of P and P' are approximately equal.

with the difference that $\vec{p}_0$ is not necessarily equal to $\vec{0}$. To simplify the analysis, we define a related problem that is graphically depicted in Figure 12: find the longest path P', which is a sequence of points $\vec{p'}_0, \vec{p'}_1, \ldots, \vec{p'}_{L'-1}$ such that

$$\begin{aligned}
&\vec{p'}_0 = (0, 0, \ldots, 0) = \vec{0},\\
&-\vec{1} = (-1, -1, \ldots, -1) \le \vec{p'}_{L'-1} \le (1, 1, \ldots, 1) = \vec{1},\\
&\forall j = 0, 1, \ldots, L-2, \quad \exists \vec{d'}_{r_j}, \quad \vec{p'}_{j+1} = \vec{p'}_j + \vec{d'}_{r_j}.
\end{aligned} \tag{14}$$

L' is the length of path P'. P' starts from the origin and only requires that the end point $\vec{p'}_{L'-1}$ is within the hypercube surrounding the origin. $\vec{p'}_{L'-1}$ can be calculated as follows:

$$\begin{aligned}
\vec{p'}_{L'-1} &= \vec{p'}_{L'-2} + \vec{d'}_{r_{L'-2}} = \sum_{j=0}^{L-2} \vec{d'}_{r_j} \quad 0 \le k_j < n\\
&= c_0 \cdot \vec{d'}_0 + c_1 \cdot \vec{d'}_1 + \cdots + c_{n-1} \cdot \vec{d'}_{n-1}\\
&= \vec{c} \cdot D' \quad \vec{c} = (c_0, c_1, \ldots, c_{n-1}) \ge \vec{0}, \quad \vec{c} \in \mathcal{Z}^n\\
&= \vec{c} \cdot E_l^{-1}.
\end{aligned}$$

Therefore, $max\{L'\}$ can be solved through the following linear programming problem $LP(E_l^{-1}, -\vec{1}, \vec{1})$:

$$-\vec{1} \le \vec{c} \cdot E_l^{-1} \le \vec{1}, \quad \vec{c} \ge \vec{0}, \quad max\left\{\sum_{j=0}^{n-1} c_j\right\}. \tag{15}$$

Next, we need to prove that $max\{L'\}$ is approximately equal to $max\{L\}$ so that $LP(E_l^{-1}, -\vec{1}, \vec{1})$ is a good approximation of $L(E_l, \mathbb{1}_n)$. First, given any path P, we can construct a path P' by letting $\vec{p'}_j = \vec{p}_j - \vec{p}_0$. Thus, it is easy to see that $max\{L\} \le max\{L'\}$.

On the other hand, given a path P' with start point $\vec{p'}_0 = \vec{0}$ and end point $\vec{p'}_{L'-1} = (p'_0, p'_1, \ldots, p'_{n-1})$, to find a corresponding path in space $span(\mathbb{1}_n)$ (which is a hypercube), we construct $\vec{s} = (s_0, s_1, \ldots, s_{n-1})$ as follows:

$$\forall j = 0, 1, \ldots, n-1 \qquad s_j = \begin{cases} 1, & p'_j < 0,\\ 0, & else. \end{cases} \tag{16}$$

By shifting with the offset $\vec{s}$, $\vec{p'}_0$ and $\vec{p'}_{L'-1}$ are moved within the space $span(\mathbb{1}_n)$:

$$\begin{aligned}
&\vec{p'}_s = \vec{p'}_0 + \vec{s}, && \vec{0} \le \vec{p'}_s \le \vec{1}, && \vec{p'}_s \in span(\mathbb{1}_n)\\
&\vec{p'}_e = \vec{p'}_{L'-1} + \vec{s}, && \vec{0} \le \vec{p'}_e \le \vec{1}, && \vec{p'}_e \in span(\mathbb{1}_n)\\
&\vec{p'}_e = \vec{p'}_s + \vec{c} \cdot D'.
\end{aligned} \tag{17}$$

We assume that the iteration space is much larger compared to the length of each dependence vector: $\forall \vec{d'}_j, |\vec{d'}_j| \ll 1$. Then we must be able to find two actual iteration points $\vec{p}_s$ and $\vec{p}_e$ close to $\vec{p'}_s$ and $\vec{p'}_e$, respectively:

$$\begin{aligned}
&|\vec{p}_s - \vec{p'}_s| < \varepsilon, && \vec{p}_s \in span(\mathbb{1}_n),\\
&|\vec{p}_e - \vec{p'}_e| < \varepsilon, && \vec{p}_e \in span(\mathbb{1}_n),\\
&\vec{p}_e - \vec{p}_s = b \cdot \vec{c'} \cdot D', && b \in \mathcal{Z}, \vec{c'} \in \mathcal{Z}^n.
\end{aligned} \tag{18}$$

Then we construct a new path P within the hypercubic space $span(\mathbb{1}_n)$ as follows:

$$\begin{aligned}
&C = \sum_{j=0}^{n} c'_j, \quad L = b \cdot C\\
&\vec{p}_0 = \vec{p}_s,\\
&\vec{p}_{j \cdot C + k} = j \cdot \vec{c'} \cdot D' + \vec{d'}_{j_k}, \quad 0 \le j < b, \quad c'_{j_k} \ne 0\\
&\vec{p}_{L-1} = \vec{p}_e.
\end{aligned}$$

The length of path P is $L = b \cdot C$. From Equations (17) and (18), we have

$$\begin{aligned}
&L > L' - \left\lceil \frac{\varepsilon}{min\{|\vec{d'}_j|\}} \right\rceil = L' - \varepsilon',\\
&max\{L\} > max\{L'\} - \varepsilon'.
\end{aligned}$$

The intuition behind the previous discussion is that path P' is decomposed into small repeated pieces and arrange the same number of repeated pieces in a line to construct P. Since the start and end points of P and P' are very close, the length of these two paths should also be close. Figure 12 illustrates this idea.

Since we already proved that $max\{L\} \le max\{L'\}$, we can conclude that $max\{L\} \approx max\{L'\}$, with the error no larger than ε'. As a result, we use the solution of the linear programming problem $LP(E_l^{-1}, \vec{-1}, \vec{1})$ to estimate $L(E_l, \mathbb{1}_n)$:

$$L(E_l, \mathbb{1}_n) = L\left(E_0 \cdot \prod_{k=0}^{l-1} T_k^{-1}, \mathbb{1}_n\right) \approx LP\left(E_l^{-1}, \vec{-1}, \vec{1}\right).$$

4.3. Summary

According to the discussion in preceding two sections, we can transform the problem of computing the execution time of hierarchical tiled loop nests into the following recurrences:

$$\begin{aligned}
Time(I) &= (Time(T_{l-1}) + t_s^l) \cdot L(E_l, \mathbb{1}_n)\\
&\approx \left(Time(T_{l-1}) + t_s^l\right) \cdot LP\left(E_l^{-1}, \vec{-1}, \vec{1}\right)
\end{aligned} \tag{19}$$

$$Time(T_k) = \begin{cases} t_c \cdot |det(T_0)|, & k = 0,\\ (Time\left(T_{k-1}) + t_s^k\right) \cdot LP\left(T_k^{-1}, \vec{0}, \vec{1}\right), & k > 0. \end{cases}$$

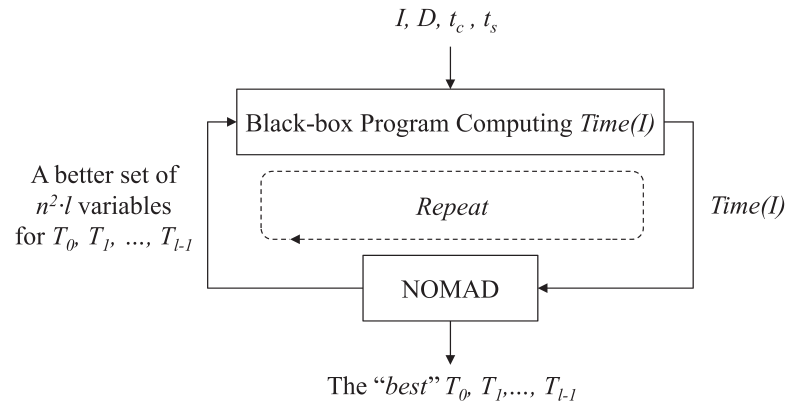

I, D, t_c, t_s
Black-box Program Computing *Time(I)*
A better set of $n^2 \cdot l$ variables for $T_0, T_1, \ldots, T_{l-1}$
Repeat
Time(I)
NOMAD
The "*best*" $T_0, T_1, \ldots, T_{l-1}$

Fig. 13. The process of automatic tile shape selection using NOMAD.

Given the edge matrix E of the original iteration space I, an optimal tile shape for hierarchical tiling is given by the sequence of valid tiling matrices $T_0, T_1, \ldots, T_{l-1}$ that minimize $Time(I)$ in Equation (19). If I is n-dimensional, each tiling matrix T_k contains n^2 elements. In total, there are $n^2 \cdot l$ variables. Thus, given that Equation (19) is a recurrence that contains linear-programming problems terms, we can conclude that the problem of selecting the tile shape that minimizes $Time(I)$ in Equation (19) is a bilevel programming problem. Because $Time(I)$ is nonlinear, it generally cannot be easily solved directly.

Notice that for a single level of tiling, if we assume infinite resources, as our model does, the minimum execution time is determined by the dependence critical path. However, for multilevel tiling, the critical path is a path of dependent tiles. Thus, the critical path at one level of the hierarchy depends on the critical path at the lower level. Because of this, the best tile shape at one level might not be the best one when the next level is also considered. This makes the problem of finding the best tile shapes for hierarchical tiling a bilevel programming problem.

5. AUTOMATIC TILE SHAPE SELECTION

According to the previous discussion, optimal tile shape selection for hierarchical tiling is an multidimensional nonlinear optimization problem without a known analytical solution. We use NOMAD [Le Digabel 2011] to select the tile shapes automatically based on the model described in Section 4. NOMAD is a tool to solve optimization problems defined by an opaque program. We use NOMAD to optimize $Time(I)$, whose computation requires solving $n - 1$ linear programming problems (Equation (19)). We use LP_Solve [Berkelaar et al. n.d.] to solve these linear programming problems. The program accepts the iteration space I, the dependence matrix D, the single execution time t_c, the tiling overhead at each level t_s, and the $n^2 \cdot l$ components of the tiling matrices $T_0, T_1, \ldots, T_{l-1}$, and outputs the result of $Time(I)$. NOMAD implements the Mesh Adaptive Direct Search (MADS) algorithm [Abramson et al. 2009] to repeatedly generate a better set of $n^2 \cdot l$ values as the input to the black-box program by searching on the mesh. NOMAD is not guaranteed to find an optimal solution; it will stop and output the current best solution when no better solution can be found. The process of automatic tile shape selection using NOMAD is shown in Figure 13.

6. EXPERIMENTS

6.1. Experiment Platform

We evaluated the strategy for tile shape selection discussed in this article on Blue Waters. Blue Waters is a powerful supercomputer thath consists of nodes with an NVIDIA Tesla K20X GPU accelerator. The Tesla K20X GPU contains 2,688 CUDA cores. The collection of all nodes and the GPUs within each node naturally form two levels of hardware hierarchy. We only use 256 nodes for the experiments. We see the collection of nodes as the first level and the cores within the GPU as the second. For the purposes of this study, we ignore the intermediate level containing the CPU.

6.2. Benchmarks

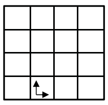
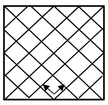
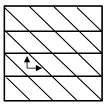
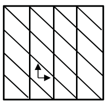
We use seven stencil programs as benchmarks: one-dimensional and two-dimensional *Gauss* and *Jacobi*, *PathFinder*, *Poisson*, and *HotSpot*. *Gauss* and *Jacobi* are the kernels of the Gauss-Seidel and Jacobi finite difference PDE solvers, respectively. *PathFinder* is a dynamic programming algorithm to find a minimum weighted path. *Poisson* is a solver of the Poisson equation, using the Laplace operator [Ames 1977] over a two-dimensional grid with a five-point stencil. *HotSpot* implements a chip temperature estimation model [Huang et al. 2004]. *PathFinder* and one-dimensional *Gauss* and *Jacobi* are one-dimensional stencils that form two-dimensional iteration spaces (time dimension + space dimension), whereas the rest of benchmarks are two-dimensional stencils that form three-dimensional iteration spaces (time dimension + two space dimensions).

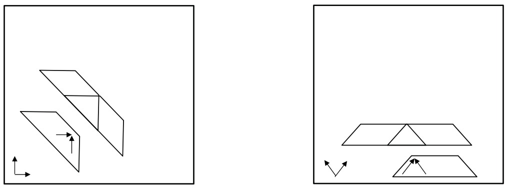
6.3. Tiling Schemes

We implemented the automatic tile shape selection system discussed in Section 5. We also implemented a tool using the Omega Library [Kelly et al. 1995] to transform loop nests onto the selected tiled form. The tool generates OpenCL [Khronos OpenCL Working Group 2008] code for the inner loops and MPI code for the outer loops. In this way, the lower level of tiles will be mapped onto the GPU device of each node, whereas the higher level of tiles will be mapped onto different nodes of the cluster.

We compare the resulting tiled loops with other tiled versions of these loops that (1) make use of common tile shapes or (2) are the result of applying the hierarchical overlapped tiling method introduced in Zhou et al. [2012]. The common tile shapes that we used in the experiments include*Square*, *Diamond*, and *Skewing*. Depending on the direction of skewing, we compare two tile shapes of *Skewing*: *Skewing-1* and *Skewing-2*. These tile shapes either are very regular or can be easily constructed by simple loop transformation such as loop skewing. We can say that these common tiling schemes are the easiest and most natural to use when applying tiling manually. Table I lists the tile shapes for two- and three-dimensional iteration spaces and the corresponding tiling matrices. No tile shape can be used for all cases, because the tiling matrix that represents the tile shape must satisfy Equation (1), and therefore correctness is guaranteed only for some dependence matrices. Table I also shows examples dependence matrices that enable the use of each tile shape. For example, $D_g = \begin{pmatrix} 1, 0 \\ 0, 1 \end{pmatrix}$ is the dependence matrix of *Gauss*, and $D_j = \begin{pmatrix} 1, -1 \\ 1, 1 \end{pmatrix}$ is the matrix of the boundary dependence vectors of *Jacobi*. *Square*, *Skewing-1*, and *Skewing-2* can be applied to iteration spaces with dependence matrix D_g, but *Diamond* cannot. However, only *Diamond* and *Skewing-1* are good for iteration spaces subject to the dependence matrix D_j, whereas the other two tile shapes are not.

Table I. Common Tiling Schemes: *Square, Diamond,* and *Skewing*

	Square	Diamond	Skewing-1	Skewing-2
Tile Shape				
Tiling Matrix	$\begin{pmatrix} x, 0 \\ 0, x \end{pmatrix}$	$\begin{pmatrix} x, -x \\ x, x \end{pmatrix}$	$\begin{pmatrix} x, -x \\ 0, x \end{pmatrix}$	$\begin{pmatrix} x, 0 \\ -x, x \end{pmatrix}$
Good for Dependences	$D_g = \begin{pmatrix} 1, 0 \\ 0, 1 \end{pmatrix}$	$D_j = \begin{pmatrix} 1, -1 \\ 1, 1 \end{pmatrix}$	D_g or D_j	D_g
3D Tiling Matrix	$\begin{pmatrix} x, 0, 0 \\ 0, x, 0 \\ 0, 0, x \end{pmatrix}$	N/A	$\begin{pmatrix} x, -x, -x \\ 0, x, 0, \\ 0, 0, x \end{pmatrix}$	N/A



(a) One-dimensional Gauss (b) One-dimensional Jacobi

Fig. 14. The rotated trapezoid tile for *Gauss* and the normal trapezoid tile for *Jacobi*.

For hierarchical tiling, it is possible to choose different tile shapes for each level of tiling. However, according to Equation (5), for the k-th level of tiling, $k > 0$, the dependence matrix always contains $\mathbb{1}_n = D_g$. This means that *Diamond* can only be applied at the lowest (0-th) level.

For hierarchical overlapped tiling, trapezoid tiles are used at each level. This type of tiling cannot be represented using tiling matrices. To fit the dependence vectors, the trapezoid tile must be "rotated" for *Gauss* as shown in Figure 14(a).

6.4. Platform Parameters

We measured the execution time of a single iteration t_c and the synchronization/communication overhead t_s at each level on the experimental platform. The parameter t_c is application dependent, mainly determined by the number of points of the stencil; t_s depends on the target machine. Our experimental results show that for 3-point, 5-point, and 9-point stencils, t_c is about 1.3 μs, 1.5 μs, and 1.6 μs, respectively. At level 1 of the hardware parallelism (internode), t_s is the communication overhead of sending and receiving MPI messages between nodes. Because level 0 parallelism is mapped onto the GPU accelerator of each node with OpenCL code, at level 0 (intranode) t_s is the overhead of starting an OpenCL kernel and transmitting data between host memory and device memory. Our experimental results show that at level 0, t_s is about 18 μs; at level 1, t_s is about 63 ms for two-dimensional iteration spaces and 124 ms for three-dimensional iteration spaces.[7]

[7]The communication overhead is higher for three-dimensional iteration spaces because tiles in a three-dimensional iteration space have more neighbors than two-dimensional iteration spaces, and thus more messages must be communicated.

Table II. The Common Tiling Schemes Used for Comparison

Benchmark	*Tiling Scheme 1*		*Tiling Scheme 2*		Iteration Space
	Level 0	Level 1	Level 0	Level 1	
1D Gauss	Square	Square	Square	Skewing-1	$2^{24} \times 2^{24}$
1D Jacobi	Diamond	Square	Diamond	Skewing-1	$2^{24} \times 2^{24}$
PathFinder	Diamond	Square	Diamond	Skewing-1	$2^{24} \times 2^{24}$
2D Gauss	Square	Square	Square	Skewing-1	$(2^{16})^3$
2D Jacobi	Skewing-1	Square	Skewing-1	Skewing-1	$(2^{16})^3$
Poisson	Skewing-1	Square	Skewing-1	Skewing-1	$(2^{16})^3$
Hotspot	Skewing-1	Square	Skewing-1	Skewing-1	$(2^{16})^3$

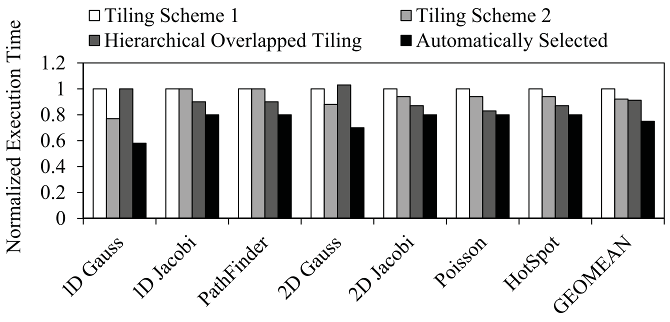


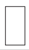
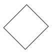
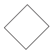
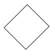
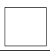
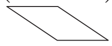
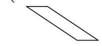
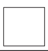
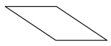
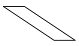
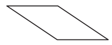
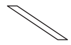
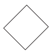
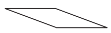
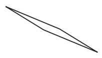
Fig. 15. Execution times of the different schemes normalized to the time of Scheme 1.

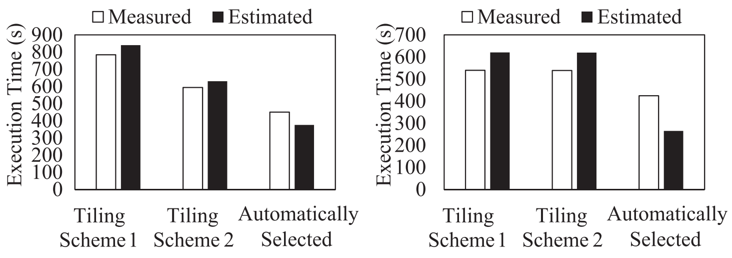
6.5. Performance

In this section, we compare the performance achieved by the code generated by the techniques of this article with code that is built using some combination of the common tiling shapes mentioned previously. For the comparison, we use two common tiling schemes: *Tiling Scheme 1* and *Tiling Scheme 2*, as well as the hierarchical overlapped tiling. As discussed in Section 6.3, a tiling scheme does not work for every problem because of the constraints imposed by dependences. Table II describes the two tiling schemes (*Tiling Scheme 1* and *Tiling Scheme 2*) for each benchmark. The last column of Table II also shows the problem size for each benchmark: for one-dimensional stencil benchmarks, the size of the iteration space is always $2^{24} \times 2^{24}$, whereas for two-dimensional stencil benchmarks, the size of the iteration space is $2^{16} \times 2^{16} \times 2^{16}$.

Figure 15 shows our experimental results. The figure shows the execution time of the different schemes normalized to the execution time of *Tiling Scheme 1*. Execution time includes the overall execution time of the entire application and includes communication and computation. To make a fair comparison, we do an empirical search to determine the best tile size for each tiling scheme. The white bar and grey bar in Figure 15 represent the execution times achieved by *Tiling Scheme 1* and *Tiling Scheme 2* using the optimal value of x's for the tiling matrix at each level. The black bar is the execution time obtained using the strategy described in this article, which automatically generates the tile shapes for a two-level hierarchical tiling. As the figure shows, the codes using the automatically selected tile shapes are always faster than the corresponding codes using the common tiling schemes and the corresponding codes using the hierarchical overlapped tiling method [Zhou et al. 2012]. Notice that hierarchical overlapped tiling does not perform well for *Gauss*, as this approach is generally not appropriate for this type of stencils. On average, the automatically selected tile shapes achieve 34%, 23%, and 20% speedup over *Tiling Scheme 1*,

Table III. Tile Shape Selection for One-Dimensional Gauss and Jacobi

	1D Gauss			1D Jacobi		
	Tiling Scheme 1	Tiling Scheme 2	Auto-Selected	Tiling Scheme 1	Tiling Scheme 2	Auto-Selected
T_0	$\begin{pmatrix} 8, 0 \\ 0, 8 \end{pmatrix}$	$\begin{pmatrix} 8, 0 \\ 0, 8 \end{pmatrix}$	$\begin{pmatrix} 8, 0 \\ 0, 4 \end{pmatrix}$ 	$\begin{pmatrix} 6, -6 \\ 6, 6 \end{pmatrix}$ 	$\begin{pmatrix} 6, -6 \\ 6, 6 \end{pmatrix}$ 	$\begin{pmatrix} 4, -4 \\ 4, 4 \end{pmatrix}$ 
T_1	$\begin{pmatrix} 8K, 0 \\ 0, 8K \end{pmatrix}$ 	$\begin{pmatrix} 8K, -8K \\ 0, 8K \end{pmatrix}$ 	$\begin{pmatrix} 8K, -8K \\ 0, 411 \end{pmatrix}$ 	$\begin{pmatrix} 6144, 0 \\ 0, 6144 \end{pmatrix}$ 	$\begin{pmatrix} 6144, -6144 \\ 0, 6144 \end{pmatrix}$ 	$\begin{pmatrix} 8330, -8330 \\ 0, 632 \end{pmatrix}$ 
$T_1 \cdot T_0$	$\begin{pmatrix} 64K, 0 \\ 0, 64K \end{pmatrix}$	$\begin{pmatrix} 64K, -64K \\ 0, 64K \end{pmatrix}$ 	$\begin{pmatrix} 64K, -64K \\ 0, 1644 \end{pmatrix}$ 	$\begin{pmatrix} 36864, -36864 \\ 36864, 36864 \end{pmatrix}$ 	$\begin{pmatrix} 36864, -73728 \\ 36864, 0 \end{pmatrix}$ 	$\begin{pmatrix} 33320, -35848 \\ 33320, -30792 \end{pmatrix}$ 



(a) One-dimensional Gauss. (b) One-dimensional Jacobi.

Fig. 16. Execution time measured and execution time estimated by the model.

Tiling Scheme 2, and hierarchical overlapped tiling, respectively. In particular, for one-dimensional Gauss, it achieves 43% speedup over *Tiling Scheme 1* and 32% over *Tiling Scheme 2*.

Table III lists the tile shape selection for one-dimensional Gauss and Jacobi.

6.6. Model Accuracy

Finally, we also measured the accuracy of the model discussed in this work. Figure 16 shows, for one-dimensional *Gauss* and *Jacobi*, the execution time measured and the execution time estimated by the model for the different tiling schemes. The figure shows that the discrepancy between the measured and the estimated execution time is fairly small (less than 15%) for all tiling schemes except for the automatically selected shapes for one-dimensional *Jacobi*. Factors that can cause inaccuracy include that t_c and t_s can have small variation for different programs and that hardware parallelism is not unlimited (to to simplify the model, it assumes that hardware parallelism is always sufficient compared to the computation parallelism exposed by the program at each level).

In practice, predicting the absolute value of execution time is not as important as predicting which shape will deliver better execution times. Figure 17 plots the measured and estimated execution time for one-dimensional *Gauss* as the tile shape changes for level 0 and level 1. We use $T_0 = \begin{pmatrix} 8, -x_0 \\ 0, 8 \end{pmatrix}$ and $T_1 = \begin{pmatrix} 8K, -x_1 \\ 0, 8K \end{pmatrix}$ to perform two-level hierarchical tiling for one-dimensional *Gauss*. When $x_0 = 0$ and $x_1 = 0$, the tiling is equivalent to *Tiling Scheme 1*. The x-axis of Figure 17(a) and (b) shows the values of x_0 and x_1. We can see that as the tile shape changes, for either level 0 or level 1, the curves for the measured and estimated execution times have similar trends.

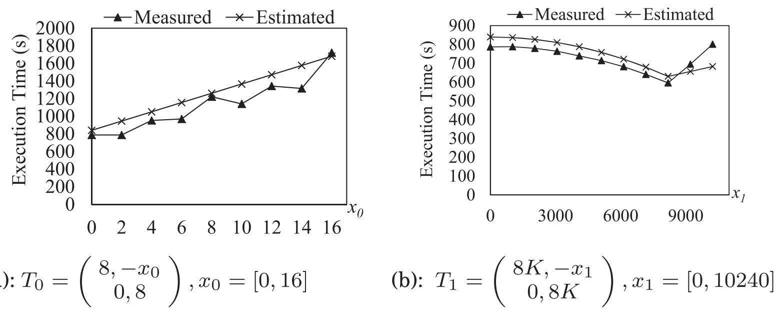


Fig. 17. Measured and estimated execution times as the tile shape changes for one-dimensional Gauss. (a) Level-0 tile shape. (b) Level-1 tile shape.

7. CONCLUSION AND FUTURE WORK

In this article, we build an analytical model to study the relation between the tile shape at each level of hierarchical tiling and the execution time of the tiled loop nest. We show that the problem of optimal tile shape selection for hierarchical tiling is a nonlinear bilevel programming problem. We implement an automatic system that does black-box optimization with our analytical model to automatically select tile shapes for hierarchical tiling. The tile shape selected by the automatic system can be quite different from the intuitive regular shapes. Our experimental results show that irregular tile shapes may have the potential to achieve higher performance over regular, intuitive tiling shapes.

REFERENCES

Mark Abramson, Charles Audet, John Dennis, and Sebastien Digabel. 2009. OrthoMADS: A deterministic MADS instance with orthogonal directions. *SIAM Journal on Optimization* 20, 2, 948–966. DOI:http://dx.doi.org/10.1137/080716980

Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. 1981. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers* 30, 5, 341–356. DOI:http://dx.doi.org/10.1109/TC.1981.1675792

Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. 2000. Tiling imperfectly-nested loop nests. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 31. DOI:http://dx.doi.org/10.1109/SC.2000.10018

William F. Ames. 1977. *Numerical Methods for Partial Differential Equations* (2nd ed.). Academic, San Diego, CA.

Rumen Andonov, Stefan Balev, Sanjay Rajopadhye, and Nicola Yanev. 2001. Optimal semi-oblique tiling. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*. ACM, New York, NY, 153–162. DOI:http://dx.doi.org/10.1145/378580.378619

Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2009. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 219–228. DOI:http://dx.doi.org/10.1145/1504176.1504209

Michel Berkelaar, Kjell Eikland, and Peter Notebaert. n.d. Introduction to lp_solve 5.5.2.0. Retrieved November 17, 2014, from http://lpsolve.sourceforge.net/5.5/.

Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. 2006. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. ACM, New York, NY, 48–57. DOI:http://dx.doi.org/10.1145/1122971.1122981

Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, Laurie Hendren (Ed.). Lecture Notes in Computer Science, Vol. 4959. Springer, 132–146.

Larry Carter, Jeanne Ferrante, and Susan Flynn Hummel. 1995. Hierarchical tiling for improved superscalar performance. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95)*. IEEE, Los Alamitos, CA, 239–245. http://dl.acm.org/citation.cfm?id=645605.662909.

Larry Carter, Jeanne Ferrante, Susan Flynn Hummel, Bowen Alpern, and Kamg-Su Gatlin. 1996. *Hierarchical Tiling: A Methodology for High Performance*. Technical Report.

Stephanie Coleman and Kathryn S. McKinley. 1995. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, NY, 279–290. DOI:http://dx.doi.org/10.1145/207110.207162

Jia Guo, Ganesh Bikshandi, Daniel Hoeflinger, Gheorge Almasi, Basilio Fraguela, Maria J. Garzaran, David Padua, and Christoph von Praun. 2006. Hierarchically tiled arrays for parallelism and locality. In *Proceedings of the 20th International Symposium on Parallel and Distributed Processing (IPDPS'06)*. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639573

Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2009. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, NY, 147–157. DOI:http://dx.doi.org/10.1145/1542275.1542301

Karin Högstedt, Larry Carter, and Jeanne Ferrante. 1999. Selecting tile shape for minimal execution time. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*. ACM, New York, NY, 201–211. DOI:http://dx.doi.org/10.1145/305619.305641

Wei Huang, Mircea R. Stan, Kevin Skadron, Karthik Sankaranarayanan, Shougata Ghosh, and Sivakumar Velusam. 2004. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04)*. New York, NY, 878–883. DOI:http://dx.doi.org/10.1145/996566.996800

Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. 1995. *The Omega Library Interface Guide*. Technical Report.

Khronos OpenCL Working Group. 2008. The OpenCL Specification, Version 1.0.29. Retrieved November 17, 2014, from http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

Sebastien Le Digabel. 2011. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software* 37, 4, 1–15.

Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. 2010. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*. ACM, New York, NY, 51–61. DOI:http://dx.doi.org/10.1145/1735688.1735698

Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99)*. ACM, New York, NY, 228–237. DOI:http://dx.doi.org/10.1145/305138.305197

Jun Liu, Yuanrui Zhang, Wei Ding, and Mahmut Kandemir. 2011. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. 161–170. DOI:http://dx.doi.org/10.1109/CGO.2011.5764684

Hiroshi Ohta, Yasuhiko Saito, Masahiro Kainaga, and Hiroyuki Ono. 1995. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *Proceedings of the 9th International Conference on Supercomputing (ICS'95)*. ACM, New York, NY, 270–279. DOI:http://dx.doi.org/10.1145/224538.224571

Jagannathan Ramanujam and Ponnuswamy Sadayappan. 1991. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. ACM, New York, NY, 111–120. DOI:http://dx.doi.org/10.1145/125826.125893

Lakshminarayanan Renganarayana and Sanjay Rajopadhye. 2004. A geometric programming framework for optimal multi-level tiling. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*. IEEE, Los Alamitos, CA, 18. DOI:http://dx.doi.org/10.1109/SC.2004.3

Yonghong Song and Zhiyuan Li. 1999. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, New York, NY, 215–228. DOI:http://dx.doi.org/10.1145/301618.301668

Michael E. Wolf and Monica S. Lam. 1991a. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4, 452–471. DOI:http://dx.doi.org/10.1109/71.97902

Michael E. Wolf and Monica S. Lam. 1991b. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*. ACM, New York, NY, 30–44. DOI:http://dx.doi.org/10.1145/113445.113449

Michael Wolfe. 1989a. Iteration space tiling for memory hierarchies. In *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 357–361. http://dl.acm.org/citation.cfm?id=645818.669220

Michael Wolfe. 1989b. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing'89)*. ACM, New York, NY, 655–664. DOI:http://dx.doi.org/10.1145/76263.76337

David Wonnacott. 2000. Time skewing for parallel computers. In *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, Vol. 1863. Springer, 477–480. http://dx.doi.org/10.1007/3-540-44905-1_35.

Jingling Xue. 1997. Communication-minimal tiling of uniform dependence loops. In *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, Vol. 1239. Springer, 330–349. http://dx.doi.org/10.1007/BFb0017262.

Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical overlapped tiling. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 207–218. DOI:http://dx.doi.org/10.1145/2259016.2259044

Received June 2014; revised September 2014; accepted October 2014