An Autotuning Framework for Scalable Execution of Tiled Code via Iterative Polyhedral Compilation

YUKINORI SATO, Toyohashi University of Technology, Japan
TOMOYA YUKI and TOSHIO ENDO, Tokyo Institute of Technology, Japan

On modern many-core CPUs, performance tuning against complex memory subsystems and scalability for parallelism is mandatory to achieve their potential. In this article, we focus on loop tiling, which plays an important role in performance tuning, and develop a novel framework that analytically models the load balance and empirically autotunes unpredictable cache behaviors through iterative polyhedral compilation using LLVM/Polly. From an evaluation on many-core CPUs, we demonstrate that our autotuner achieves a performance superior to those that use conventional static approaches and well-known autotuning heuristics. Moreover, our autotuner achieves almost the same performance as a brute-force search-based approach.

CCS Concepts: • **General and reference** → **Performance**; • **Hardware** → *Emerging tools and methodologies*; • **Software and its engineering** → *Search-based software engineering*;

Additional Key Words and Phrases: Iterative polyhedral compilation, loop tiling, load balancing, scalability for many-core CPUs

ACM Reference format:
Yukinori Sato, Tomoya Yuki, and Toshio Endo. 2019. An Autotuning Framework for Scalable Execution of Tiled Code via Iterative Polyhedral Compilation. *ACM Trans. Archit. Code Optim.* 15, 4, Article 67 (January 2019), 23 pages.
https://doi.org/10.1145/3293449

1 INTRODUCTION

The rapid spread of many-core processors is a feasible approach for realizing sustainable performance improvements in conjunction with the upcoming semiconductor scaling. Most many-core processors are integrated with emerging new memory technologies such as high-bandwidth 3D stacked memory or storage class non-volatile memory, and these new technologies increase the complexity of memory hierarchies inside shared memory systems. Historically, cache memories implemented in hardware have been used to map an application's memory access patterns into the underlying memory hierarchy. However, cache-based memories are not universal mechanisms.

New Paper, Not an Extension of a Conference Paper.
This work is supported by CREST, Japan Science and Technology Agency, JSPS KAKENHI Grant No. 17K12658, and JST PRESTO Grant No. JPMJPR18M6.
Authors' addresses: Y. Sato, Toyohashi University of Technology, Department of Computer Science and Engineering, 1-1 Hibarigaoka, Tempaku-cho, Toyohashi, 441-8580, Japan; email: yukinori@cs.tut.ac.jp; T. Yuki and T. Endo, Tokyo Institute of Technology, Global Scientific Information and Computing Center, 2-12-1-W8-88, O-okayama, Meguro-ku, Tokyo, 152-8550, Japan; emails: yuki.t.ab@m.titech.ac.jp, endo@is.titech.ac.jp.

1544-3566/2019/01-ART67
https://doi.org/10.1145/3293449

Rather, these approach makes it difficult to fit the wide variety of data locality of applications into such memory subsystems.

To compensate for the weaknesses of the cache-based locality management mechanism for hierarchical memories, expert programmers, especially in the HPC field, often perform manual code refactoring in an attempt to improve the locality of memory references at the software level [27]. Loop transformation [17] is a primary code-tuning technique that has been widely used. Loop transformation is critical for performance but it could not be performed automatically on classical SSA-based compiler optimization passes; in other words, it forced expert programmers to rewrite code by their hands to perform loop transformation.

The polyhedral model is an emerging powerful framework that provides a flexible and expressive representation of loop nests for performance tuning and automatic parallelization [4, 23]. Using optimizers based on the polyhedral model, loop transformation can be automated and can free programmers from the painful code rewriting process. To determine the transformation strategy within a polyhedral optimization process, polyhedral compilers must construct performance models that estimate performance for application-specific memory access patterns. Generally, these performance models are represented as cost functions (or objective functions) in the optimization process.

In polyhedral optimization, loop tiling, which decomposes a loop nest into a set of loop iterations, is a primary type of transformation that allows us to fit the active working set into the underlying memory hierarchy. On one hand, the tile size deeply affects cache behaviors and causes significant changes in the code execution performance. On the other hand, the effective selection of tile size is not straightforward. Hence, tile size selection remains an open problem, and it represents a knob that can be used to tune the underlying performance. However, current typical polyhedral compilers such as Pluto [5] and Polly [11] do not consider tile size parameters in their cost functions. Currently, they use a cost function that minimizes inter-tile communication volume and reuse distances for local execution [5]; thus, they simply set a 32x32x32 tile size as the default to reduce the search space for polyhedral optimization. Previous studies have shown that the default tiling parameter setting enables reasonable performance gains to be achieved on machines with moderate degrees of parallelism. Nevertheless, when targeting the emerging many-core processors, the default setting often negatively affects performance. In other words, tile size is expected to be a primary parameter when applied to a high number of parallel cores with complex memory hierarchies.

Thus, the approaches for exploiting parallelism inside loop nests exposed by the polyhedral model when targeting many-core CPUs should be reconsidered. Applying loop tiling forces the execution to be *strong scaling*, because the chunk of data per processor becomes smaller based on the degree of tiling. Here, strong scaling means that the number of tasks remains constant even when the number of available processors increases. In contrast, in *weak scaling*, the number of tasks increases linearly with the number of processors. Achieving scalability for a strong scaling problem is known to be much more difficult, because we have to consider communication and synchronization among the distributed chunks of data. Therefore, we need to analyze the scalability issues of tiled code on many-core processors with regard to tile size parameters, which reflect features such as the degree of parallelism, the granularity of each task, and task load balancing.

To the best of our knowledge, general analytical models do not exist so far that statically find optimal tile size parameters across a wide range of program and target platforms. One primary reason is that these approaches are often not sustainable when faced with technology advancements, because their input parameters are sometimes sensitive to the hardware generation and often must be refined for new targets. For instance, TurboTiling is the state-of-the-art analytical algorithm for tile size selection, and it can make good use of prefetching to boost the performance [21]. However,

TurboTiling assumes an environment in which the number of threads does not exceed the associativity of the target cache configuration, but this assumption is too optimistic for modern many-core CPUs. Some counterexamples are already in production (e.g., Xeon: 28 threads for a 20-way L3 cache, KNL: 64 threads for 16-way L2 caches or a direct-mapped L3 cache configured at boot time).

In this article, we investigate a novel way to find the optimal tile size parameters. Here, we propose the Polyhedral compilation-based AuTo Tile size optimizer (**PATT**) and attempt to harness an analytical load balancing model to an autotuning framework. We mainly focus on Intel's Xeon Phi (Knight's Landing), which contains 64 physical cores capable of executing up to 256 threads within a CPU. To productively perform loop tiling at compile time, we use Polly together with LLVM tool chains and realize fully automated loop tiling without any modification to source code. Then, we reveal some data locality and scalability management issues in a many-core environment together with the load balancing issue. As parallelism increases, load imbalance considerations become increasingly important for the performance of tiled code, and this aspect is not addressed in the existing tiling research. We believe our findings will be useful for generating high-performance scalable code suitable for upcoming computer systems based on many-core CPUs.

The primary contributions of this article are as follows:

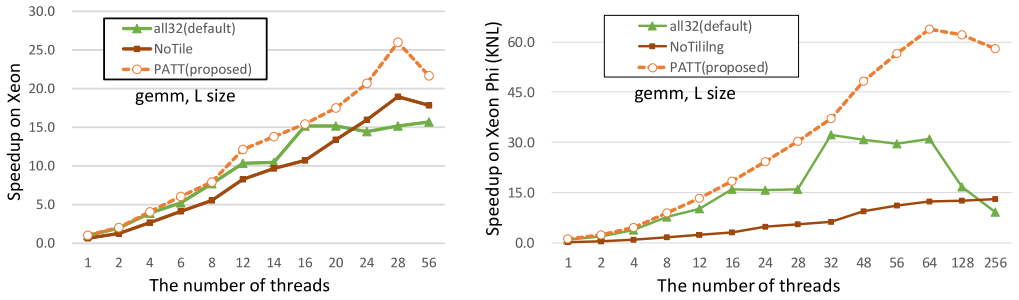
- We show that the scalability of tiled code is strongly influenced by load imbalance issues as the number of concurrent threads increases.
- We propose a special autotuner customized for tile size selection using an iterative compilation framework that analytically adjusts the load balance and empirically autotunes the unpredictable and tricky cache behaviors associated with many-core CPUs.
- We demonstrate that our autotuner achieves superior performance compared to an autotuner that uses typical existing heuristics such as Simulated Annealing (SA) or Nelder-Mead, and our autotuner achieves performance comparable to that of a brute-force approach.
- We show that our proposed fully automated autotuner outperforms Polly's default tiling parameters and the state-of-the-art analytical method.

The rest of this article is organized as follows. In Section 2, we explain our background motivation. Section 3 presents observations on the load balancing issues of tiled code in many-core CPUs and an analytical model. Section 4 presents an autotuning-based mechanism that empirically adjusts loop tile size while considering load balancing. Section 5 presents the experimental framework, the evaluation methodology, and explains the evaluation results. Section 6 provides the related works, and Section 7 concludes this article.

2 MOTIVATION

In this section, we briefly introduce our motivation for this study. Figure 1 shows the speedup ratio when executing gemm code. As a baseline, we use the time for single-threaded execution with all32 (default) tiling. We generate the code using Polly with the following options: For all32, we use Polly's default settings. Here, Polly's default tile size represents a case in which the executable code is generated without any user-specified tile size options. In this situation, Polly internally sets 32 as the tile size parameter for all dimensions. In NoTiling, we turn off the loop tiling. In PATT, we show the results of tile size autotuning proposed in this article. To observe the behavior on many-core processors, we use a machine with dual-Xeon CPUs with 28 physical cores and a Xeon Phi (KNL)-based machine with 64 physical cores. We measure the execution time up to the maximum number of threads supported by hyperthreading. The rest of the detailed experimental configurations are the same as those presented in Section 5.1.

As shown above, loop tiling clearly improves the performance compared to NoTiling under moderate parallelism. However, the speedup of all32 gradually becomes smaller when the number



(a) Speedup on a 28-core machine

(b) Speedup on a 64-core Xeon Phi machine

Fig. 1. Speedup of gemm code compared to its single-threaded execution with all32 (default) tiling.

of threads rises above 16. In particular, we can observe that NoTiling achieves a greater speedup than does all32 in the 28-core machine when the number of threads exceeds 20. For the Xeon Phi, we can observe scalability issues with all32 with more than 16 threads; finally, all32 is slower than the NoTiling performance with 256 threads.

As the degree of parallelism increases, we must consider the tradeoff between the effects of loop tiling and load balancing. This concept is explained as follows. To exploit data reuse in the caches, we need to perform loop tiling aggressively. However, if we set an aggressive tile size, the number of iterations in the outermost loop activated per thread becomes extremely small on many-core processors. This is because the outermost loop iteration space commonly used as a source of loop parallelism is also used as a source of load balancing. Hence, the number of outermost iterations becomes too small to be distributed evenly across the parallel cores when we perform both parallelization and tiling for the outermost loops. This imbalance causes inefficient parallel execution. To find the best tradeoff point between the degree of tiling and load balancing, we propose the PATT framework and attempt to productively resolve this issue by combining an analytical load balancing model with an empirical autotuning method.

Our primary motivation of this study was to realize a fully automated performance tuning mechanism on the current software stack. Then, we focused on Polly as the state-of-the-art program optimizer and parallelizer using an un-optimized serial program as input to productively perform loop tiling at compile time. However, the current implementation of Polly does not support automatic loop parallelization other than the outermost ones. This implementation is reasonable from the perspective that the outermost loop iteration space is a major source of loop parallelism in common uses of OpenMP.[1]

3 ANALYTICAL LOAD BALANCING MODEL

First, we start by building an analytical model that depicts the behaviors of tiled code on many-core processors. Figure 2 shows a representation of the problem space (M, N, P) and how loop tiling is performed to the space. Figure 2(a) depicts a typical space traversal and its corresponding triply nested loop. Figure 2(b) represents a problem space tiled into (TX, TY, TZ) with a sextuply nested

[1]In practice, a situation exists in which parallelization can be applied to loop nests other than the outermost one or where parallelization—including inner dimensions—is far more efficient than parallelizing only the outermost loop. Especially for the latter case, the use of nested parallelism with the OpenMP collapse clause that fuses two outer loops can be useful. However, nested parallelism is not supported yet on the currently available LLVM-Polly and other production-level compilers currently available for evaluation. Therefore, in this article, we assume that all the parallelization sources are from the outermost loop iteration space.

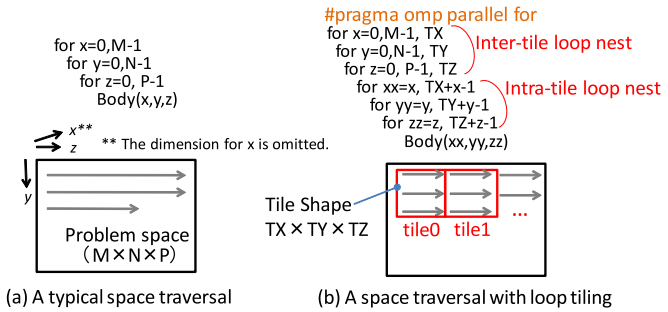


Fig. 2. An outline of loop tiling for a triply nested loop.

loop comprised of the intra-tile loop nest that drives the calculation within a tile and the inter-tile loop nest that traverses different tiles. As discussed in the previous section, we assume that only the outermost loop, which traverses x, is parallelized using OpenMP. Then, the number of iterations of the outermost inter-tile loop becomes M/TX, and these iterations are further parallelized into p threads after loop tiling.

To depict the primary issues behind loop tiling on many-core processors, we calculate N_{IPT}, which is the maximum number of iterations per thread, as a metric for the granularity of parallel execution as follows:

$$N_{IPT} = \left\lceil \frac{M}{TX \cdot p} \right\rceil. \tag{1}$$

Here, we use a ceil function to represent the upper bound among all threads, because some threads execute $\lfloor \frac{M}{TX \cdot p} \rfloor$ or fewer iterations while the maximum threads execute N_{IPT} iterations. The former threads with fewer iterations involve insufficient computation to keep the cores busy. This difference causes a load imbalance across cores.[2]

Then, we attempt to build a performance model that can estimate the execution time of the tiled code. Here, we assume N_{IPT} as the dominant factor for the execution time of inter-tile loops. In addition, we need a model that estimates execution time for the intra-tile loops. Here, the performance of intra-tile parts is sensitive to their memory access patterns, so cache memory plays a critical role in locality management for them. However, building a practical analytical model instead of a time-consuming cache simulation is infeasible, because hierarchical cache memory often behaves in an unpredictable and tricky manner. Because current memory systems constitute a complex memory hierarchy, their behaviors depend not only on simple parameters such as the size of each cache but also a wide range of factors such as associativity, replacement policy, coherence protocol, memory layout, and the actual alignment determined at run time [6, 28]. In fact, cache behavior with regard to system performance is tricky even if we target the regular stencil code or matrix operation kernels. Indeed, highly optimized programs in HPC domains, including these stencil and matrix kernels, still suffer from unpredictable cache conflicts, which are sometimes called cache thrashing [14]. Therefore, we focus on an autotuning-based approach

[2]In this article, we focus on the load balancing issue that appears when execution starts from an unoptimized serial program. To achieve parallelization, Polly internally inserts OpenMP API library functions into the outermost dimensions when the dependencies are resolved. The load balancing issue to be solved here stems from the imbalance in the number of iterations per thread among the CPU cores—especially when the chunk size is very small. Therefore, this imbalance cannot be solved just only dynamic or guided scheduling of OpenMP instead of static scheduling.

that measures the actual performance in the underlying machine and conduct further performance tuning based on findings from the actual cache behaviors.[3]

Hence, we model the intra-tile loop term using only the simple assumption that the execution time of an intra-tile loop is dominated by its tile size TX. Then, $T_{LBmodel}$ can be calculated as follows:

$$\begin{aligned} T_{LBmodel} &= \alpha \cdot N_{IPT} \cdot TX \\ &= \alpha \cdot \left\lceil \frac{M}{TX \cdot p} \right\rceil \cdot TX. \end{aligned} \tag{2}$$

Here, we first calculate the dimensionless value ($N_{IPT} \cdot TX$) and then scale it to the actual execution time with "α," which is a scaling factor applied to each benchmark program to plot to the identical time scale for the actual execution, and it can be calculated simply by normalizing the minimum of the actual execution time with the minimum of the calculated dimensionless values.

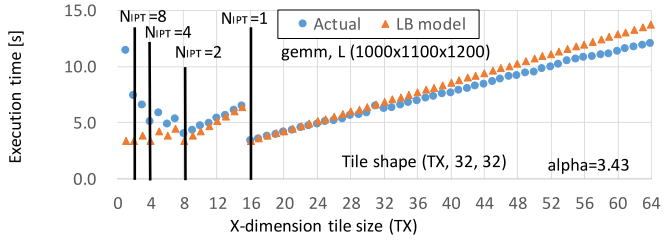
To verify the LB model, we measure the actual execution time of the tiled code.[4] Figure 3(a) shows the actual time and the estimated time ($T_{LBmodel}$) of tiled gemm code with the L dataset on a Xeon Phi with 64 threads. Here, only the x-dimension tile size TX is varied from 1 to 64 while keeping the other tile sizes fixed to 32, that is (TX, 32, 32). We also represent the actual value of the scaling factor, $alpha(\alpha)$. In addition, we calculate the minimum value of TX that satisfies Equation (1) when we set $N_{IPT} = 1, 2, 4, 8$, and plot the obtained pairs of N_{IPT} and TX. We note that the obtained points are observed to be faster than other points in $T_{LBmodel}$ due to the ceil function.

From the result, we observe that the LB model can nicely estimate the execution time of the gemm code except when the size of TX is close to 1. When TX becomes close to 1, the overhead for such a small tile size becomes visible in the actual execution, but is not included in the estimated performance. For other sizes, the effects of load imbalance are nicely modeled with approximate cache behaviors. We can see the points that dramatically improve the execution time from the previous ones. In these points, we observe the ceil function in Equation (1) returns an incremented value compared with inputting previous TX. In the gemm results, $TX = 8$ and $TX = 16$ are good examples of these points, which, respectively, correspond to $N_{IPT} = 2$ and $N_{IPT} = 1$. The workloads of these points are excellently balanced across the parallel resources. In contrast, we suffer from load imbalance when TX is decremented by 1 from these points. When TX is incremented from these points, a gradual performance degradation appears due to the intra-tile execution time factor. Hence, these phenomena essentially imply that the actual performance is primarily modeled by this load-balanced centric approach. This means that the tile size search space can be decomposed into the load balancing term for the outermost loop and the intra-tile cache modeling term across the remaining inner loops.

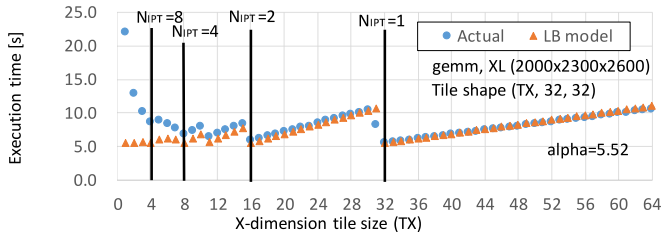
Figure 3(b) shows the results when we change the dataset to XL in gemm. As the results show, we can confirm that the LB model works nicely when we scale the problem size. However, some application programs cannot be modeled as easily as the gemm code discussed above due to tricky cache behaviors found in real systems. As an example, we adopt the atax code from PolyBench, which manipulates 2D arrays using doubly nested loops. Figure 3(c) shows the actual and estimated

[3]Initially, we attempted to investigate the relationships among performance improvements and observed metrics such as cache hit/miss rate and the number of memory accesses. However, during this attempt, we found that the behavior of cache memory is far more complex than we originally assumed. Consequently, we finally decided to discontinue correlating them. Instead, we shifted from our initial completely analytical approach to the autotuning-based approach.

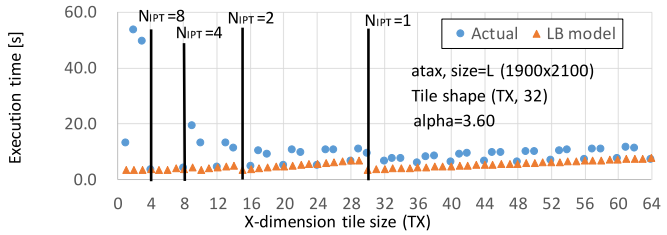
[4]In the LB model, we assume the problem space has been defined before the estimation. This means that the problem space that defines loop bounds needs to be fixed when we estimate the execution time using the LB model. When the problem space is determined at run-time, the LB model cannot be applied for source code level analysis; however, it can still be applied by performing a dynamic analysis of loop bounds at run-time.



(a) gemm (L dataset)



(b) gemm (XL dataset)



(c) atax (L dataset)

Fig. 3. The real and estimated execution times of tiled code and estimations for the LB model on a Xeon Phi.

execution times of the tiled atax code.[5] From the results, we can observe that the actual execution time exhibits periodic variations across TX sizes and that these are mostly different from the ones estimated by the LB model. We consider that this periodic behavior is caused by memory layout and alignment issues and such unpredictable and uncertain behaviors are difficult to model analytically. In addition, we observe that the effects of load imbalance tend to be weaker than those in gemm, which involves a 3D tile size selection space.

Realizing a perfect analytical model that could predict the performance of tiled code, including the effects of caches, would be ideal. However, cache behaviors and the relationship between tile size and performance are far more complex than we could address using an analytical approach. These difficulties motivated us to adopt an autotuning-based approach with the goal of maximizing cache performance through effective tile size selection.

4 AUTOTUNING TILE SIZE SELECTION

In this section, we propose a novel autotuning framework named **PATT** and attempt to address performance optimization transparently, without involving application programmers. Figure 4

[5]Due to some bugs in Polly, it fails to generate executable code for specific TX sizes; therefore, we excluded these from the plot in this figure.

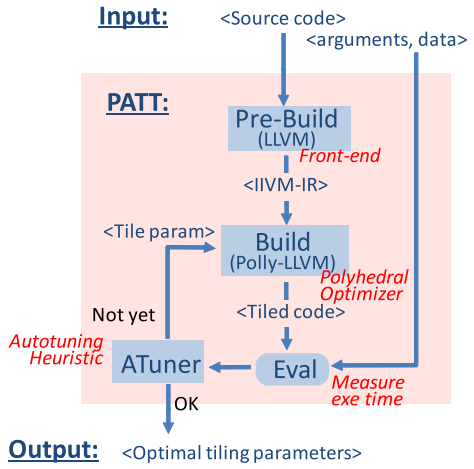


Fig. 4. An overview of the PATT framework.

shows an overview of the PATT framework through iterative polyhedral compilation. Overall, the PATT framework can be seen as a compiler driver written using Python and shell scripts that internally activates individual compiler tools and executable code iteratively. The details are as follows: Using C source code as an input, LLVM-IR is generated by LLVM tool chains in the Pre-Build stage and input to Polly in the Build stage. Polly performs polyhedral optimizations, including loop tiling, and the LLVM back-end generates executable tiled code in the Build stage. In the subsequent Eval stage, the tiled code is executed with its arguments and its execution time is measured. Then, its statistics are fed to the autotuner in the ATuner stage. The heuristics implemented in the autotuner investigate whether the performance can be improved or converged. If room for improvement still exists, then the autotuner outputs the next tile parameters to be evaluated, and reverts back to the Build stage. Otherwise, if the determination is that the obtained performance has converged, the autotuner outputs the optimal tile size.

Here, we build a novel hybrid method that makes use of autotuning together with load balance observations on many-core processors. In Section 3, we observed that tile size search problems can be decomposed into the outermost loop dimension for the load balancing effect and the remaining search space for the cache effect. To further pursue the ultimate load balancing distribution, we couple the analytically obtained TX sizes with autotuning to fine-adjust the final tiling parameter.

Algorithm 1 shows the main part of the load balancing adjustment that picks up the outer loop nest as a load balancing term for parallelization. Here, we start by checking whether the target problem space is larger than 2 dimensions. If so, then we start by decomposing the load balancing term from the whole tile size selection problem.[6] If the problem space is 3D, then we use four prefixed values (1, 2, 4, 8) of N_{IPT} as analytically inferred candidates of tile size (CTX) based on the insight that these are the potential minima of the execution time.[7] Then, considering the case that might occasionally stack to the worst case, which is adjacent to the estimated local minimum

[6]In Algorithm 1, we focus on 3D/2D problem space (3D/2D nested loops) as an example. The underlying concept of Algorithm 1 is the decomposition to 2D search space. Therefore, we believe this can be applied to load balancing for outer loop nests with arbitrary dimensions.

[7]When $N_{IPT} > 8$, TX approaches zero and can hardly be expected to achieve the minimum execution time, as shown in Figure 3. Even when we apply much bigger datasets with larger M, the effects of load imbalancing do not appear; Then, we consider the current implementation that explores $i = 1, 2, 4, 8$ is reasonable and do not miss further opportunities for much larger datasets. Hence, we exclude these regions from the search candidates.

ALGORITHM 1: PATT_ATuner_main

Input: Problem space (M, N, P) or (N, P) for 2D, # threads (p)
  if Problem space is 3D **then**
    for all $i = 1, 2, 4, 8$ **do**
      $CTX_i = \lceil \frac{M}{i \cdot p} \rceil$
      for all $i = CTX_i - 1, CTX_i, CTX_i + 1$ **do**
        $t_i = tile\_eval(i, 32, 32)$
      end for
    end for
    select i such that it minimizes t_i
    $TX = i$
  else if Problem space is 2D **then**
    $TX = NONE$
  end if
  $TY, TZ = Custom\_ATuner\_for\_TSS(N, P)$
Output: Output tile size (TX, TY, TZ)

points, we input the values, including their neighbors (±1), into Procedure `tile_eval()` to improve the robustness to periodic variations in the LB model. If the problem space is 2D, then we bypass the autotuning of TX described above.

Procedure 2 shows how `tile_eval()` works: It comprises the Build stage and the Eval stage in Figure 4. In `build_exe()`, the tiling parameters are input to Polly as a user-specified compiler option (e.g., "`--polly-tile-sizes=x,y,z`"), and then an executable binary that embeds the tile size is generated. Afterward, the true execution time of the tiled code is measured in `measure_exetime()` using the `/usr/bin/time` command.

After all the candidates have been evaluated in Procedure `tile_eval()`, we choose the best TX tile size and pass the remaining tile search space to the following domain-specific autotuning heuristic. Note that when the target problem space is 2D, we can bypass the TX selection phase. This is because the search space will become too small if we decompose the outermost loop for autotuning against the 2D problem space. Therefore, we perform autotuning for the entire 2D space to find much better points.

Here, PATT cyclically repeats the following phases: `tile_eval()`, which includes a code generation phase by Polly, the evaluation phase of tiled code execution, and a subsequent parameter-generation phase represented as ATuner in Figure 4. The ATuner phase is the one orchestrated by Algorithm 1. PATT repeats this cycle as long as the heuristic can generate finer grids. When the recursive re-partitioning process stops, we treat it as having converged.

To search for the optimal tile parameters without the outermost dimension space, we focus on a specially customized autotuning heuristic. Here, we propose a hill-climbing-based search pruning technique and an adaptive grid-based search mechanism. Figure 5 shows the pruning technique inspired by the hill-climbing method. The underlying concept of a hill-climbing algorithm is to

PROCEDURE 2: tile_eval

Input: tile size (i, j, k)
  $build\_exe(i, j, k)$
  $t_{exe} = measure\_exetime(i, j, k)$
Output: t_{exe}

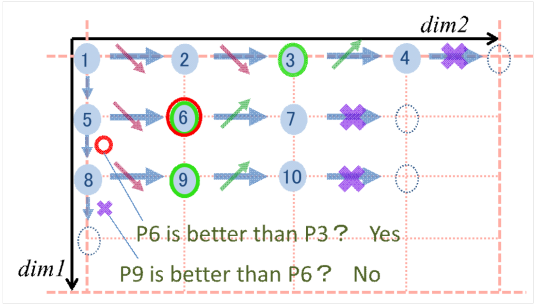


Fig. 5. Pruning evaluation points by hill climbing.

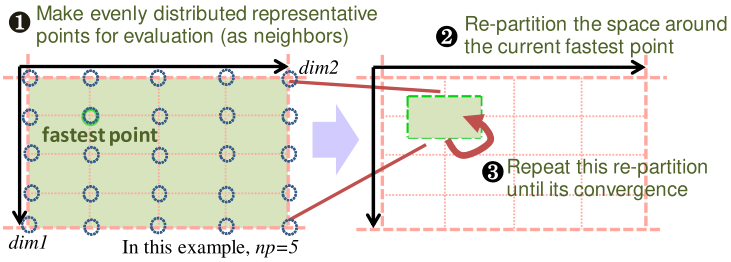


Fig. 6. Adaptive grid-based search mechanism in PATT.

effectively find the lowest value (the global minimum) of the cost function [26]. Here, the value of the cost function corresponds to the execution time of the tiled code, where the location in the state space is defined by the tile size. To reduce the number of search steps, we prune the evaluations after the inflection points. As shown later in the more exhaustive analysis presented in Section 5.2, we observe that the surface of the cost function can mostly be approximated to a convex problem; therefore, the hill-climbing-based search pruning algorithm works well.

Figure 6 shows the adaptive grid-based search mechanism (AGS) implemented in our custom autotuning heuristic. Here, we generate adaptive grids to form representative points for evaluation and formulate the target search regions gradually, from coarse- to fine-grained. Unlike other typical autotuning heuristics, which move search steps forward based on locally adjacent neighbors, ours adaptively sets the neighbors by re-partitioning the space. This helps us to identify the optimal points lying ahead of broad, flat regions.

Procedure 3 shows the details of how our customized autotuning heuristic for effective tile size selection (TSS) works. We start by setting some initial values composed of a start position and an end position within each dimension. Then, we enter the loop that autotunes the tile sizes for dimensions 1 and 2 (dim1, dim2). Here, we calculate the steps to evenly partition the region from the start to the end position and generate representative points based on the AGS mechanism. We set (8, 8) as the initial start and end points and set the number of partitions *np* to 8. Then, we update the fastest time and its tile size within these representative points.

To reduce the number of evaluations (search steps) based on the hill-climbing concept, we prune the search of the remaining successors when the execution time exceeds the previous time. Finally, we check whether our AGS mechanism can generate finer grids. If not, then we exit the AGS loop. Otherwise, we update the start and end positions to narrow the steps among the representative points at the `update_pos()`, and repeat the AGS loop. Here, we focus on the representative point that achieves the best time and re-partition the space by setting the start and end positions for

PROCEDURE 3: Custom_ATuner_for_TSS

Input: Problem size (N, P)
  $pos\_start[] = init_{dim1}, init_{dim2}, pos\_end[] = N, P$
  $np = 8$
  loop {/* Loop for AGS mechanism */}
    $step[] = calc\_step(pos\_start, pos\_end, np)$
    $dim1\_list, dim2\_list = calc\_AGS\_points(pos\_start, step, np)$
    for $q = 0$ to $np - 1$ **do** {/* Body of 2D hill climbing */}
      $j = dim1\_list[q]$
      for $r = 0$ to $np - 1$ **do**
        $k = dim2\_list[r]$
        $t_{jk} = tile\_eval(TX, j, k)$
        if $t_{jk} > t_{current}$ **then**
          break this loop [/* Pruning points for dim2 */]
        end if
        $t_{current} = t_{jk}$; $tile_{current} = (TX, j, r)$
      end for
      if $fastest\_time > t_{current}$ **then**
        $fastest\_time = t_{current}$; $fastest\_tile = tile_{current}$
      else
        break this loop [/* Pruning points for dim1 */]
      end if
    end for
    if all of s in $step[] = 0$ **then**
      return $fastest\_tile$
    end if
    $pos\_start, pos\_end = update\_pos()$
  end loop

the next iteration using $pos\_start = rep - step/(np * 2)$, $pos\_end = rep + step/(np * 2)$, where rep represents the position of the best representative point. If either the updated pos_start or pos_end exceeds its dimensional boundary, then we adopt the boundary point as its updated position. In addition, we assume that a spatial step between neighboring representative points is a multiple of 4 except for the outermost loop tile size, TX.

Note that these pre-defined parameters are determined by the following consideration and our preliminary evaluation of this algorithm: Because Polly performs auto-vectorization (SIMDization) for AVX instructions with a bit width of 256, the most preferable chunk of data is at least four words of 64 bit data. Then, we set the potential tile size parameter to be a multiple of 4 except for the outermost loop to make inner loops interchangeable each other.

5 EVALUATION

5.1 Methodology

In this section, we evaluate our PATT framework on platforms equipped with modern many-core processors. Here, we use a KNL-based machine with 64 cores and a dual-CPU machine with 28 cores. The 64-core machine has an Intel Xeon Phi processor 7210 @ 1.3GHz with 96GB of DDR4 and 16GB of MCDRAM memory. The Xeon Phi is configured in flat mode. The 28-core machine is equipped with two Haswell-generation Intel Xeon E5-2697 v3 @ 2.60GHz CPUs with 128GB DDR4 memory. These machines both run Linux/CentOS 7. In our evaluation, we consider a single

thread per physical core (without hyperthreading) configuration. Moreover, we bind threads to cores using the "taskset -c" command.

To evaluate the advantages of the customized autotuning heuristic used in PATT, we compare the performance of the tiled code with two of the well-known autotuning heuristics: Nelder-Mead and Simulated Annealing (SA). The Nelder-Mead method is a popular numerical method that finds the minimum or maximum of an objective function, and SA is a general randomized algorithm inspired by the metal annealing process. We implement these two heuristics following the approaches used inside the Orio tool [22]. Here, we use Orio's default parameters and coefficients for searching except for the following: We set the all32 tile sizes to their initial values and assume multiples of 4 for tile sizes. In Nelder-Mead, we create an initial simplex using the neighbors of the initial points (all32), whereas the neighbors are set to +4 grids in each dimension. In SA, we set `neighbor_distance=1 and final_temp_ratio=0.005`. In addition to the autotuning heuristics, we evaluate TurboTiling as the state-of-the-art analytical algorithm for tile size selection [21]. We also compare our PATT framework with Polly's default all32 tile size selection as well as with the NoTiling setting.

In this experiment, PATT and the other heuristics are implemented on LLVM 5.0.0 with Clang and Polly (Git version as of 2017/April/24). The use of Polly's polyhedral model is not limited to tiling automation. In our PATT framework, the target program code is automatically parallelized by OpenMP and further optimized by vectorization, loop interchange and other loop transformations.[8] For vectorization, we use the "avx2" option because Polly's current vectorizer does not generate LLVM-IR for 512-bit vector registers.

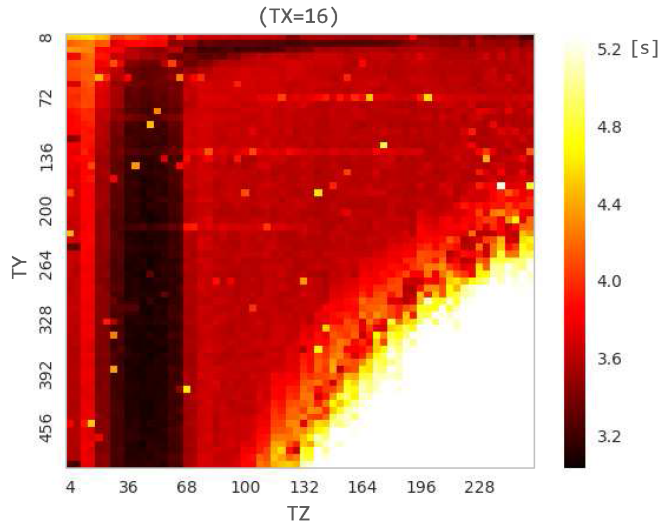
We use PolyBench/C 4.2 in this experiment, which is a benchmark suite composed of 30 numerical computation kernels in various application domains such as linear algebra, image processing, physics simulations, dynamic programming, and statistics [33]. Here, we adopt 13 programs, each of which is automatically tiled successfully by Polly's original implementation. We note that the kernels of the following five programs (atax, gemver, gesummv, mvt, and jacobi-2d[9]) are composed of a doubly nested 2D loop that requires a 2D search space, and the others are composed of a triply nested loop that requires a 3D search space. Here, we use the L and XL datasets with a double precision type.

To increase the robustness and stability for time variations across executions, we modify the original source code to invoke the kernel function multiple times (defined by IT_NUM in source code). Here, we configured IT_NUM so that the single execution time for the multithreaded tiled code is approximately 2 to 4s. We also measured the execution time three times per evaluation in PATT and adopt the median value as representative. Throughout the evaluation, we measure the entire time required for execution, including program initialization and finalization.

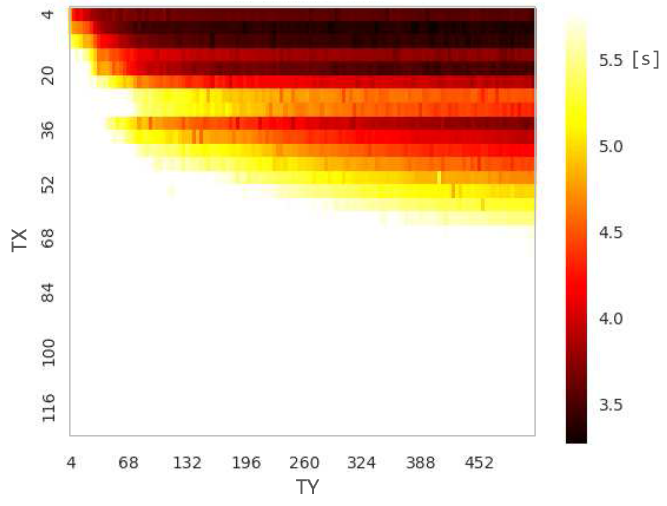
Regarding code generation for each tile size, the native compilation in many-core oriented machines is known to be several times slower than machines with typical server-class fat-core CPUs in the same generation, because most compilation passes are based on single-threaded execution. To avoid this compilation issue, we perform cross-compilation of Xeon Phi code on a fat-core machine. To achieve this, we use the Haswell-based Xeon machine to build executables for Xeon Phi and remotely invoke the executable through ssh.

[8]We run Polly with the following options: "opt -basicaa -polly-process-unprofitable -polly-ignore-aliasing -polly-allow-nonaffine -polly-pattern-matching-based-opts=false -polly-opt-isl -polly-vectorizer=polly -polly-parallel -polly-codegen" and "llc -mattr=+avx2." When we turn off the tiling, we use the "-polly-tiling=false" option. Note that Polly supports code optimizations such as register tiling and multiple levels of cache blocking, but we did not activate them. We believe these factors can be included to PATT's search spaces.

[9]For jacobi-2D, we tiled the doubly nested spatial loop of the 2D problem space by excluding the outermost timestep loop, because the timestep contains dependencies across iterations.



(a) gemm



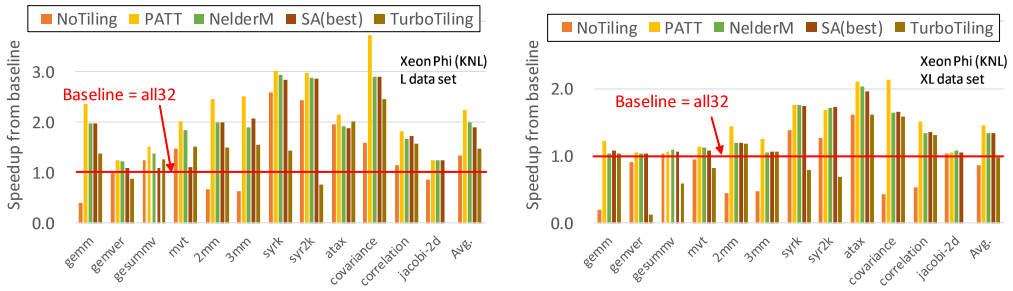
(b) atax

Fig. 7. Visualized heatmap of tiled code performance.

5.2 Effects of Tile Size Selection on Performance

Figure 7 shows heatmaps of the execution time [in seconds] of the tiled code when the locations are defined by tile sizes. To generate these heatmaps, we perform exhaustive parameter searches across 4,000 to 10,000 points. Here, we show two types of programs: the first type requires 3D space searching (gemm) and the second type requires 2D space searching (atax). In gemm, we present a heatmap for $TX = 16$.

From these results, we can see that tile size selection is highly important in enabling the potential system performance. In addition to the heatmaps shown in Figure 7, we generate numerous heatmaps to investigate the surface shapes of the cost functions for tiling with various TX values



(a) L dataset (b) XL dataset

Fig. 8. Speedup of PATT and the other heuristics from all32 executing 64 threads on a Xeon Phi.

across benchmarks. From the results of these preliminary experiments, we concluded that we can commonly approximate them using convex functions. Therefore, we assume that the surface of the cost function can be approximated to a convex problem and that pruning based on the hill-climbing algorithm works well.[10] We also investigate the sensitivity of the outermost tile size to the performance by generating a sequence of heatmaps for a series of TX values. The results confirm that the outermost tile size affects the performance most significantly. In gemm, we observe that the heatmap for $TX = 16$ includes the overall best performance point. For gemm ($TX = 16$), we find that points with the worst execution time of the color map (5.2[s]) are at least 1.6 times slower than those with the fastest execution time (3.2[s]).

We also observe that no straightforward way exists to correlate the heatmap to a particular configuration of cache memory parameters. For gemm, the slanting boundary line near 4.8[s] corresponds to a tile size equal to the L2 cache capacity. However, the fastest point we found is related neither to the L2 capacity nor the L1 capacity. Therefore, this example shows that the relationship between performance and cache behavior can often be tricky and unpredictable.

5.3 Speedup Against the Other Heuristics

Figure 8(a) shows the speedup from the default all32 on the L dataset. Here, we execute 64 threads on a Xeon Phi KNL machine. From these results, we can observe that PATT clearly achieves better performances than do the other heuristics, and it contributes to achieving the performance gains of tiled code. The average speedup of PATT from the baseline all32 is 2.25 on the L dataset. In gemm, PATT achieves a speedup of 2.35 times compared to the baseline, which corresponds to an absolute performance of 147GFLOPS/s. While this performance is several times slower than a typical highly optimized numerical library,[11] PATT can transparently autotune tile size without source code modifications and can be applied to a wide range of programs as a general purpose method. This implies that the autotuning-based approach taken by PATT can productively be applied to a broad range of real applications.

[10]This conclusion is still hypothetical: We do not have complete evidence. Therefore, we validate this concept in Section 5.4 using the theoretical upper-bound implied by a brute-force approach.

[11]PATT's absolute performance could be doubled at maximum if we successfully generated AVX512 code. The long vector length of AVX512 might influence the performance of tiled code, because it affects the number of iterations of the innermost loop in the code. Another inefficiency might stem from the automatic code transformation for parallelization and tiling from serial code. If we evaluate the effects of tile size selection using parallelized source code with adjustable tile parameters, then the absolute score could be considerably higher. In any case, thanks to the robustness of autotuning to adapt to the variabilities in machines and programs, we believe our approach will find an optimum based on the underlying environment.

It has been observed that all32 is sometimes slower than NoTiling (for example, for mvt, syrk, syr2k, atax, and covariance). This slowdown comes from the fact that all32 fails to capitalize on the potential performance of the underlying machine due to load imbalance in many-core processors. In contrast, PATT can prevent such load imbalance and capitalize fully on the advantages of loop tiling by autotuning tricky cache behaviors. Notably, PATT always achieves a higher performance than does NoTiling.

Comparing PATT with Nelder-Mead and SA, we observe that PATT achieves a performance 1.13 times higher than that of Nelder-Mead and 1.18 times higher than SA (best). This result occurs because our special search mechanism, which is customized for tile size selection, outperforms the others when searching for a global optimum value while avoiding local optima. Because Nelder-Mead and SA are both general purpose optimization methods, they have difficulty finding a global optimum located after widespread flat regions, as shown by the heatmaps for tile size selection.[12] In contrast, PATT can find such points due to its adaptive grid-based approach. Here, we note that because SA is very sensitive to random numbers used inside the algorithm, we conducted the evaluation five times for each program using different seeds for the random number generator and show only the best results. We can observe that the variations in execution time between SA (best) and SA (worst) averaged approximately 14%.

Next, we compare PATT with TurboTiling, the state-of-the-art analytical model.[13] To match TurboTiling to the up-to-date many-core machines, we refine it carefully. While TurboTiling originally focused on L3/L2 caches and assumed the last level cache (LLC) implementation in Intel Xeon CPUs, the typical LLC of many-core processors such as XeonPhi (KNL) is L2 and it is shared among two cores. Therefore, we apply the L2-L1 cache parameters for TurboTiling under the assumption that two threads are assigned per L2 cache in 64-thread execution. Because the XeonPhi's cache parameters are beyond the original expectations of the TurboTiling algorithm, the output tile sizes are sometimes too small due to the L1 cache capacity, which causes a deviation from the ideal tile size. From the results, PATT achieves, on average, a performance 1.51 times better than that of TurboTiling. While we have yet to find a unified static model that works ideally in all cases, we can still confirm the advantages of our autotuning-based approach.

For the next evaluation, we compare the differences caused by data size. Figure 8(b) shows the speedup compared to the baseline on the XL dataset. The average PATT speedup on the XL dataset is 1.44 times faster than the baseline; however, this ratio is considerably smaller than when using the L dataset. This result occurs because the load imbalance issue is alleviated as the size of the data becomes larger, and can be explained as follows. When we input a larger dataset, the code execution behavior approaches that of weak scaling, because the per-processor problem size is adequate for parallel execution. Hence, using a larger dataset can contribute to avoiding the scalability issue.

However, we cannot always adjust the problem data size to achieve better scalability. In particular, this is often true when targeting real-world applications where the problem size is typically pre-determined by factors such as the resolution and accuracy of the target dataset. Regardless of the problem size, we must seek a way to achieve scalability for the target. While achieving strong scaling is not a straightforward task on many-core processors, it is quite important for their success. Therefore, we focus on strong scaling in this article and investigate the points that balance

[12]It might be possible to find a global optimum using SA, but that would require much finer parameter tuning. In our preliminary evaluation, we used various parameter sets (other than Orio's default ones) for SA. From the results, to find points closer to the global optimum, SA requires many more search steps than does PATT. This result also highlights the advantages of PATT compared with SA.

[13]In this evaluation, we exclude jacobi-2d, because the parameters for self-temporal reuse (s1, s2) cannot be configured for stencil code by simply using the definition in Reference [21].

Table 1. The Number of Search Steps to Perform Autotuning and the Selected Tile Size (L Dataset, Xeon Phi)

	PATT		NelderM		SA(best)		TurboTiling	
gemm	58	16;1,124;12	24	16;44;48	49	16;40;48	—	4;1,200;2
gemver	50	8;1,428	19	8;96	41	40;80	—	15;2,000
gesummv	28	8;8	15	8;40	21	24;52	—	4;1,300
mvt	48	8;860	16	8;72	17	32;40	—	4;2,000
2mm	52	13;900;12	22	16;32;56	46	16;40;48	—	4;900;4
3mm	56	14;764;12	24	16;36;52	64	16;56;48	—	4;900;4
syrk	84	2;40;460	27	4;20;100	56	4;20;60	—	4;1,000;3
syr2k	71	2;8;680	29	4;16;92	65	4;20;64	—	4;1,000;1
atax	37	8;2,104	14	8;72	45	8;96	—	4;2,100
covariance	58	8;1,136;12	44	4;200;132	75	8;60;40	—	4;1,200;2
correlation	56	9;1,152;12	25	4;76;48	45	8;44;48	—	4;1,200;2
jacobi-2d	43	12;1,300	16	16;48	38	16;24	—	—

the tradeoff between scalability and loop tiling at a fixed data size. From the results, we find that PATT achieves an average speedup of 1.51 times compared with TurboTiling on the L dataset. This implies that PATT is a promising approach to optimizing many-core processors, which require more capability to achieve strong scaling of tiled code. In the case of weak scaling, as shown in the comparison between the L and XL datasets, the load imbalance issue is alleviated to some extent and would be further alleviated if the data size were scaled up.

Table 1 shows the detailed evaluation results of the autotuning heuristics described above. The number of search steps (henceforward denoted as *nSearchSteps*, which indicates how many times the search loop iterates during the autotuning process) is shown in the left part of each element, while the selected tile size (TX, TY, TZ) appears on the right. Because *nSearchSteps* is roughly proportional to the time required to evaluate each heuristic, it functions as one of the metrics for evaluating the autotuning overhead.

From these results, we can observe that PATT requires a greater number of search steps than does Nelder-Mead. In the case of gemm, PATT requires 58 while Nelder-Mead requires only 24. On average, PATT requires 2.3 times as many *nSearchSteps* as does Nelder-Mead on the L dataset. In contrast, the *nSearchSteps* required by SA(best) is 49, which it is almost the same as PATT. SA's result was selected from among five evaluations per program. We note that TurboTiling does not require any searches for online autotuning, because it calculates the tile size at compile time.

Regarding the tile sizes, we can observe that PATT selects much larger tile sizes for several programs. In fact, in some dimensions PATT selects tile sizes that are extremely close to the data domain size. This means that PATT suggests that loop tiling should be turned off for these nests to keep the loop parameters and control loop induction variables similar to the case without tiling. For gemm, PATT selects $TY = 1{,}124$, while the other heuristics select much smaller TY values such as 44 or 40. Because dimensions y and z are interchanged during Polly's optimization pass in most situations, Polly naturally selects larger TY values for vectorization and longer sequential accesses. However, due to their limited global search capabilities, Nelder-Mead and SA converge to small TY values. For TurboTiling, the tile dimensions y and z are assumed to be interchanged during the optimization pass. The results indicate that TY is fixed to the problem size, which means that the innermost loop is left untiled. By contrast, PATT explores much better tile size tuples that can achieve superior performance.

Table 2. Breakdown of the Iterative Polyhedral Compilation Process (Gemm, L Dataset, Xeon Phi)

	Native compilation	Cross-compilation + Offloading
Build [s]	1,086.41	139.65
Trial [s]	384.44	524.38
Other [s]	0.18	0.04

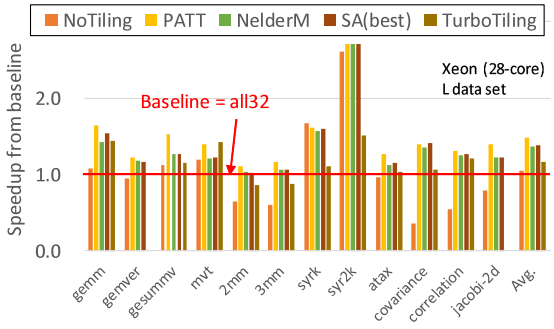


Fig. 9. Speedup of PATT on a 28-core machine.

Table 2 shows the breakdown of the iterative polyhedral compilation time for gemm running on the Xeon Phi with the L dataset. We partition the execution time of PATT into three parts. In the Build part, we measure the time required to generate executable code on Polly/LLVM. In the Trial part, we accumulate the time required for target code executions. In the remote executions, the remote communication time via ssh is included in the Trial part. All the remaining elements of PATT are accumulated in the Other part. From the results, we can observe that the cross-compilation and offloading to Xeon Phi contributes to alleviating the total compilation time issue on the Xeon Phi. We also note that the Other part, which includes the ATuner stage in Figure 4, occupies only a small portion of the total execution time. Some overhead is incurred for offloading in the Trial part.

Next, we investigate the stability and robustness on a different platform. Figure 9 shows the speedup of PATT compared to the baseline all32 tiling on a 28-core Xeon machine using the L dataset.[14] The results show that the average speedup of PATT compared to all32 is 1.64. Since the 28-core Xeon contains only a moderate number of cores compared to the Xeon Phi KNL, the gains achieved by adjusting the balance between loop tiling and load imbalance become smaller. We can also see that PATT's performance is better than that of Nelder-Mead, SA and TurboTiling on this platform. Hence, we can conclude that PATT is a robust and stable approach to loop tiling across various many-core environments.

5.4 Further Performance Assessment of PATT

In this section, we assess the speedup of PATT by comparing its theoretical upper and lower bounds. The brute-force approach (b-force) acts as an upper bound for the potential performance that can be achieved via tile size adjustments. To verify our strategy, which decomposes tile size selection into a load balance term and a cache performance term, we introduce i-Simple, which

[14]Here, we exclude gemver with TurboTiling, because it cannot successfully be executed in this environment.

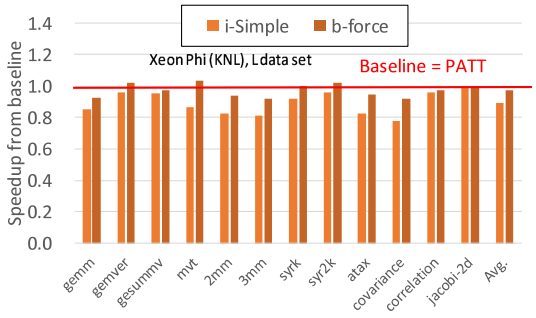


Fig. 10. Comparisons with b-force and i-Simple.

searches only the dimension of outermost loop. Because i-Simple focuses only on load balance adjustments for the outermost inter-tile loop, we can exclude the load balance term from PATT.

We use the following parameters to implement these two approaches. For b-force, we perform an exhaustive search of 1,100 grids for 3D space, where the actual representative points are TX=1, 2, 3, 4, 8, 12, 16, 20, 24, 28, 32 and TY=TZ=4, 8, 12, 16, 20, 24, 28, 32, 64, 128. For i-Simple, we search only the TX dimension; the remaining dimensions are fixed to their default values, i.e., (TX, 32, 32).

Figure 10 shows the speedup of i-Simple and b-force on a Xeon Phi KNL with 64 threads compared to the baseline, which is set to PATT. From these results, we can observe that b-force executes at almost the same level or slightly more slowly than does PATT. On average, b-force achieves 97.3% of PATT's performance. Note that b-force is not always better than PATT. Because we limit the search space of b-force to 1,100 due to the evaluation time restriction, it sometimes fails to find better tile sizes. The original intent of this assessment was to verify whether PATT can find near-optimal parameters that are similar to those of the brute-force approach; therefore, these results validate PATT's performance. In addition, it is worth noting that—even for atax code—PATT tends to find a better tile size than does the b-force. As discussed in Section 3, it is difficult to model tricky cache behaviors solely using the LB model; nevertheless, we can see that our special autotuning heuristic is productively capable of finding near-optimal parameters, even in atax.

We can also see that i-Simple achieves an average of 89.1% of the performance of PATT. These results reveal that load balancing by adjusting the tile size of only the outermost intra-tile loop TX contributes a large portion of the speedup. This result coincides with our assumption that load balancing is a primary factor that determines the performance of tiled code and also implies that the remaining performance gains stem from autotuning for tricky cache behaviors. Additionally, by validating PATT's superiority compared with the other approaches, we have confirmed the hypothesis that the surface of the cost function can be approximated as a convex problem in most of the benchmark programs, datasets, and platforms, because our proposed method discovers near-optimals and achieves better performance than the other methods.

6 RELATED WORKS

Many prior proposals have adopted empirical approaches to perform effective tile size selection, because such approaches can subliminally adapt to various factors based on observations made during actual code execution. Kisuki et al. proposed iterative compilation, in which various versions of a program are built from source code by varying the optimization parameters, and their best version is explored by measuring the actual time for execution [18]. In that study, they applied this technique to determine the optimal tile size and unroll factor simultaneously. In particular, they evaluated several search algorithms, including SA, GA, and random search, and showed that

a straightforward use of these algorithms requires on average 750 to 1,000 search steps to obtain the maximum speedup. In Reference [19], to reduce nSearchSteps, the authors attempted to incorporate static models that capture cache behaviors. The principle idea behind this study [19] is very close to our hybrid approach that harnesses well-modeled analytical characteristics to an autotuning heuristic. However, our approach focuses on load balance modeling as the dominant performance aspect, while theirs focus only on cache behaviors.

Shirako et al. proposed an analytical model that provides upper and lower bounds for practical tile size selection [30]. Using their DL/ML model, they perform an efficient empirical search within a subspace of the full search space. However, their model, which focuses primarily on cache line occupancy by checking array subscripts, is too simple to mimic the tricky cache behaviors that occur during real program execution. In addition, their approach ignores scalability on a highly parallelized CPU.

Bao et al. proposed analytical modeling of cache behavior by focusing on the SCoP of polyhedral programs [3]. Their idea was implemented as the PolyCache tool, which takes a C program and cache parameters as input and outputs the cache miss counts. However, their modeling forces all cache line accesses to be affine. This restriction of PolyCache modeling imposes the use of virtually indexed caches. Modern memory subsystems use physically indexed caches where the virtual-to-physical mapping is determined at runtime, except for L1 caches, as discussed in Reference [28]. Therefore, unpredictable tricky cache behaviors such as intra- or inter-array cache conflicts cannot be modeled by this static modeling approach.

Ranasinghe et al. studied the difference between iteration space tiling and cache-oblivious approaches on modern architectures [25]. They compared cache-oblivious methods that recursively split into smaller tiles through a divide-and-conquer strategy with typical single-level iteration space tiling. Their evaluation showed that these two methods achieve similar speeds (GFLOPS). Because cache-oblivious approaches may also be viewed as tiling, the recursive-based approach might be yet another solution for analytical-based approaches.

Autotuning is an emerging empirical approach to determining the best-performing optimization sequence and parameter values [12]. The concept of autotuning, which usually covers a wide range of topics, can be grouped into the following three categories: self-tuning library generators, compiler-based autotuners, and application-level autotuners. Compiler-based approaches can be transparently applied to a wide range of programs and are good for general purpose tile size selection; therefore, we follow this approach in this study. Note that library- or application-level approaches require intensive manual code optimization.

As a compromise approach to compiler- and application-level autotuning, Hartone et al. introduced an annotation-based empirical tuning system called Orio [13]. They implemented an annotation parser for their own annotation languages to conduct source-to-source translation from the annotated code to the optimized version. However, using this approach, directives must be manually inserted into the original source code. Therefore, Orio is not fully transparent to users: users must learn the details of Orio's directives when applying them to their own code.

Ansel et al. [2] proposed OpenTuner, an application-level framework for building multi-objective program autotuners. OpenTuner includes pre-defined generic search methods such as variants of hill-climbing and Nelder-Mead, and it contributes to freeing application programmers from having to implement a new interface or well-known heuristics for each project. Another contribution of this study is that the authors recognized the importance of domain-specific autotuning for performance optimization. Therefore, the customized approach for addressing the load balancing issue on many-core CPUs presented in this work complements this previous approach.

Allowing tile sizes to be symbolic parameters adjustable at compile time is enormously beneficial to autotuning [15]. However, parametric tiling in its full generality is known to be

non-linear, which breaks the mathematical foundations of the polyhedral model. Hence, most polyhedral compilation tools perform fixed-sized tiling by compiling a program version for each tile size. Then, these programs are iteratively activated to perform autotuning. In Reference [15], mono-parametric tiling was proposed, which formulated the tile sizes as multiples of a single block parameter. While such an approach has not yet been implemented in production-level polyhedral compilers, it could be coupled with our hybrid autotuning approach.

Machine-learning-based iterative compilation is proposed to automatically focus on a search space comprising dozens of transformation or optimization options. Agakov et al. presented a method for building a predictive model that is independent of the search algorithm or search space used [1]. They focused on machine learning techniques when building their predictive models and used a wide range of CPU behaviors in a one-off training/learning phase for each different application program. Yuki et al. presented a machine-learning-based method to synthesize a tile size selection algorithm from profiled program features [34]. Liu et al. also presented a tile size prediction model that used a generalized regression neural network [20]. In these approaches, compiler optimizations and the underlying platform specifications are treated as a black box. Then, each program's execution time is measured across various scenarios, and used to train a predictor that selects the best optimizations for each program. However, due to their black-box view, it is impossible for these approaches to identify the specific key parameters that significantly affect the performance.

Grosser et al. proposed a polyhedral-model-based effective tiling scheme for GPUs and considered a mass of constraints specific to them [10]. To bound the data footprint of tiles to allow all temporary values to be stored in shared memory, they combined hexagonal tiling on an outer spatial dimension with classical tiling along the other dimensions. Prajapati et al. presented an optimal tile size selection mechanism for hybrid hexagonal/classical tiling [24]. They developed an analytical model to predict the execution time of stencil code on a GPU platform and used those values for tile size selection. Their model focuses especially on predicting the tile sizes that yield the highest performance levels (within 20% of the best level) and apply that to autotuning. While they imply that the thread-per-block parameter has a significant impact on performance, they did not model that factor in their proposed method. Moreover, their approaches to load balancing are completely different from those aimed at many-core processors.

Banser et al. proposed an approach that performs an iterative schedule optimization method in the polyhedral model and targets tiling and parallelization [9]. They provided a tool called Polyite that searches an optimal schedule from the legal loop transformation space via a genetic algorithm. The underlying concept is very close to our PATT: Both approaches iteratively perform polyhedral optimization for loop tiling and parallelization. The primary difference is that PATT focuses on iterative compilation and search parameters to obtain the compiler's tile size setting, while their approach searches for an optimum among the legal schedules for a SCoP of the loop. However, the schedule space has many more dimensions than do the tile size parameters; thus, finding a better solution is much more difficult than our simple tile size selection approach. Danner extended Polyite to form a performance prediction function during the schedule search partly by using a machine learning technique [7]. This approach is similar to our approach using the LB model; however, our method explicitly unveils the program's key parameters that affect performance without requiring any black-box components.

Doerfert et al. proposed expression propagation that distributes expressions to compute the intermediate results redundantly across parallel resources as an alternative to communicating their results from a single node on the top of a polyhedral optimization framework [8]. This type of optimization is orthogonal to typical polyhedral optimization approaches, and their evaluation results showed significant potential for performance improvement over the typical ones.

To improve code tuning productivity, a transparent and dynamic autotuning technique is the ultimate solution. Tavaragiri et al. proposed a dynamic autotuning framework that attempts to select the optimal tile size during the production run based on parameterized tiling of code [31]. Sato et al. presented a transparent performance tuning mechanism based on loop tiling that used binary code as its input [29]. Jain et al. proposed a dynamic code-rewriting-based tile size adaptation called ShapeShifter [16] targeted for resource sharing on datacenter servers. While these approaches would dramatically improve the productivity for autotuning tile size selection processes, they do not consider whether the autotuned performance is close to optimal or not. Because these techniques are orthogonal to this work, these two directions may be able to be merged in the future.

For scalability issues on tiling, Wonnacott et al. pointed out that applying loop tiling for particular conditions forces the code to be opposed to the concept of weak scaling [32]. In particular, they focused on tiling for the temporal dimension as a specific condition and discussed its scalability in regions outside weak scaling. In our article, we extend the cases to widespread many-core processors and provide a feasible way to achieve efficient strong scaling by adjusting the key parameters.

7 CONCLUSIONS

In this article, we have proposed an iterative compilation framework that empirically autotunes the loop tile size for many-core CPUs. For many-core CPUs, finding the optimal tile size parameters is becoming increasingly important, because the typical tile selection policy often conflicts with load balancing among cores. We modeled the load balancing issue on many-core CPUs analytically and combined it with autotuning to adapt to the tricky performance behaviors of cache memories.

We have evaluated our special autotuning method for tile size selection using PolyBench. The results showed that our autotuner successfully achieves speedups that are, on average, 2.25 times greater than the default tiling parameters used in Polly and 1.51 times greater than the state-of-the-art analytical method (TurboTiling). We also demonstrated that our approach achieves better performance than SA or Nelder-Mead with reasonable autotuning search overhead. Further, we assessed the performance of our approach compared to a brute-force search-based approach and confirmed that our mechanism can productively achieve nearly the same level of performance as the upper bound.

In future work, we plan to implement a PATT-based tile size selection mechanism on Polly. Because the PATT framework can indicate a reachable upper bound for static tile size selection, PATT can serve as a practical baseline when designing sophisticated tile size selection mechanisms that can be embedded in compilers.

PATT is available at `https://github.com/YukinoriSato/PATT` to share the findings described in this article. We encourage researchers and developers to download PATT as a basis for productive performance tuning.

REFERENCES

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. 295–305.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. 303–316.

[3] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proc. ACM Program. Lang.* 2 (Dec. 2017), 32:1–32:26.

[4] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software and International Conference on Compiler Construction (CC'10/ETAPS'10)*. 283–303.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 101–113.

[6] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically sound performance evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 219–228.

[7] Dominik Karl Danner. 2017. *A Performance Prediction Function Based on the Exploration of a Schedule Search Space in the Polyhedron Model*. Master's thesis. University of Passau.

[8] Johannes Doerfert, Shrey Sharma, and Sebastian Hack. 2018. Polyhedral expression propagation. In *Proceedings of the 27th International Conference on Compiler Construction (CC'18)*. 25–36.

[9] Stefan Ganser, Armin Grösslinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2017. Iterative schedule optimization for parallelization in the polyhedron model. *ACM Trans. Archit. Code Optim.* 14, 3 (Aug. 2017), 23:1–23:26.

[10] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. 66:66–66:75.

[11] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4 (2012), 1–28.

[12] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*.

[13] A. Hartono, B. Norris, and P. Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing*. 1–11.

[14] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2016. Effective padding of multidimensional arrays to avoid cache conflict misses. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI'16)*. 129–144.

[15] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. 2015. *Mono-parametric Tiling Is a Polyhedral Transformation*. Research Report RR-8802. INRIA Grenoble-Rhône-Alpes, CNRS.

[16] Animesh Jain, Michael A. Laurenzano, Lingjia Tang, and Jason Mars. 2016. Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting. In *Proceedings of the International Symposium on Microarchitecture*.

[17] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.

[18] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*. 237–246.

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

[20] Song Liu, Yuanzhen Cui, Qing Jiang, Qian Wang, and Weiguo Wu. 2018. An efficient tile size selection model based on machine learning. *J. Parallel Distrib. Comput.* 121 (2018), 27–41.

[21] Sanyam Mehta, Rajat Garg, Nishad Trivedi, and Pen-Chung Yew. 2016. TurboTiling: Leveraging prefetching to boost performance of tiled codes. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. 38:1–38:12.

[22] Orio. 2008. Retrieved from http://brnorris03.github.io/Orio/.

[23] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. 549–562.

[24] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. 2017. Simple, accurate, analytical time modeling and optimal tile size selection for GPGPU stencils. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. 163–177.

[25] Waruna Ranasinghe, Nirmal Prajapati, Tomofumi Yuki, and Sanjay V. Rajopadhye. 2018. PCOT: Cache oblivious tiling of polyhedral programs. *arXiv preprint arXiv:1802.00166* (2018).

[26] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press.

[27] Nadathur Satish et al. 2015. Can traditional programming bridge the Ninja performance gap for parallel computing applications? *Commun. ACM* 58, 5 (Apr. 2015), 77–86.

[28] Yukinori Sato and Toshio Endo. 2017. An accurate simulator of cache-line conflicts to exploit the underlying cache performance. In *Proceedings of 23rd International European Conference on Parallel and Distributed Computing (Euro-par'17)*. 119–133.

[29] Yukinori Sato, Tomoya Yuki, and Toshio Endo. 2017. ExanaDBT: A dynamic compilation system for transparent polyhedral optimizations at runtime. In *ACM International Conference on Computing Frontiers 2017 (CF'17)*.

[30] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. 101–121.

[31] Sanket Tavarageri, Louis-Noel Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2011. Dynamic selection of tile sizes. In *Proceedings of the 2011 18th International Conference on High Performance Computing (HIPC'11)*. 1–10.

[32] D. G. Wonnacott and M. M. Strout. 2013. On the scalability of loop tiling techniques. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT'13)*.

[33] Tomofumi Yuki and Louis-Noël Pouchet. 2016. *PolyBench 4.2 Document*. Retrieved from https://sourceforge.net/projects/polybench/.

[34] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic creation of tile size selection models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. 190–199.

Received May 2018; revised November 2018; accepted November 2018