Tile Size Selection Using Hill Climbing

Eric Asare
Computer Science, NYUAD
ea2525@nyu.edu

Advised by: Riyadh Baghdadi, Muhammad Shafique

ABSTRACT

State-of-the-art deep learning neural networks require extensive computations, which can be costly due to the number of CPU and GPU clusters needed for execution. Optimizing such neural networks is crucial, but it is a challenging task that demands considerable time and expertise when done manually. Compilers, especially schedulers or auto schedulers, offer a solution by optimizing code, making computations faster and lighter, even for embedded systems and smartphones. These compilers rely on search space exploration methods to identify the most efficient optimizations amidst various techniques. This paper proposes and implements the hill climbing algorithm as a search method to efficiently explore the vast search space of code optimizations for the MLIR compiler. Our objective is to expedite finding the best tile size parameters for loop tiling optimization in MLIR compiler. We evaluated our method on benchmarks representing the base of most machine learning models. The simplicity of our method makes it highly suitable for integration into production-level MLIR-based compilers.

KEYWORDS

search space exploration, tiling, MLIR compiler, hill climbing algorithm

Reference Format:
Eric Asare. 2023. Tile Size Selection Using Hill Climbing. In *NYUAD Capstone Project 1 Reports, Fall 2023, Abu Dhabi, UAE.* 6 pages.

This report is submitted to NYUAD's capstone repository in fulfillment of NYUAD's Computer Science major graduation requirements.

جامعة نيويورك ابوظبي
NYU ABU DHABI

Capstone Project 1, Fall 2023, Abu Dhabi, UAE

1 INTRODUCTION

Computation-heavy domains such as image processing, deep learning, and scientific and numerical fields demand significant computational resources, including memory, CPU cycles, and network bandwidth. The long-term goal of the compiler community is to enable compilers to automatically optimize high-level code and generate efficient code for the hardware. Optimizations such as parallelization, vectorization, and tiling can accelerate computations in these domains improving efficiency, execution time, and reducing resource requirements. However, automatic optimization is a challenging problem that involves selecting the precise combination of optimizations among millions of potential optimizations and parameters. Another problem is that optimizations yield varying effects based on whether they are applied individually or in conjunction with other optimization techniques, adding complexity to the process. Loop Tiling is a frequently used technique in modern compilers to optimize code execution. It utilizes memory localities such as caches to significantly improve data locality, thereby enhancing overall performance. The search space of parameters for loop tiling is very large. It typically involves navigating through a graph of actions, where each node represents a specific code state, and each edge within the graph corresponds to applying loop tiling transformation rules based on specific parameters.

This paper introduces and implements hill climbing as an efficient method for exploring the extensive scope of loop tiling parameters for code optimization within the MLIR compiler. Our main contribution lies in the development of the search method itself. Our search method finds the best tiling sizes with minimal search time. Our developed search method will be easy to integrate into autoschedulers built on MLIR framework.

2 RELATED WORK

2.1 Automatic Optimization

Automatic optimization in compilers refers to the process where a compiler, which is a software program responsible for translating source code into machine code, autonomously

optimizes the generated code to improve performance, reduce execution time, conserve memory, or enhance other aspects of program execution. This process involves various techniques and algorithms that analyze the code structure, identify potential performance bottlenecks or opportunities for improvement, and apply transformations to the code to generate more efficient machine code.

2.2 MLIR

MLIR (Multi-Level Intermediate Representation) is a versatile infrastructure for developing compiler optimizations, transformations, and code generation. It provides a framework for representing various levels of abstraction in compilers, making it an ideal choice for implementing automatic optimizations in compiler design, including those related to Machine Learning (ML) workloads. MLIR is open-source and contains mainly specifications for IR (Intermediate Representation) and a code toolkit to perform transformations on it [6]. It provides the infrastructure required to execute high-performance machine learning models in TensorFlow and similar ML frameworks because it is extensible, and allows the composition of transformations at different levels. It also supports defining custom abstractions in the IR to best model different domains and supports different dialects.

2.3 Automatic Scheduling

Automatic scheduling in compilers, which involves using polyhedral frameworks advocated by [1] and [3], aims to address the challenge of optimizing code automatically for distributed systems. Tiramisu, a polyhedral compiler recognized for its ability to create fast and portable code [2], takes a different approach. Instead of complete automation, Tiramisu offers a scheduling language with specific commands tailored for various hardware architectures like multi-core CPUs, GPUs, and distributed systems. One reason why Tiramisu performs better than Halide [7], a non-polyhedral compiler with a scheduling language, is because Halide uses intervals to represent iteration spaces instead of the polyhedral model, preventing it from performing complex affine transformations such as iteration space skewing. MLIR stands out because it can leverage the polyhedral representation, along with numerous other representations. This capability allows for a broader range of optimizations, which is why we are specifically introducing our work on MLIR compilers.

2.4 Loop Transformation

In the context of loop transformations, tiling, as discussed in [9], stands out for its ability to harmonize program characteristics, such as locality and parallelism within the execution environment. This process considers crucial factors like memory hierarchy and the number of CPUs. Tiling is also referred to as *blocking* because it strategically ensures that the data to be used remains in the cache for subsequent iterations i.e tiling partitions the iteration space into smaller blocks, focusing on manageable chunks of data that fit into the cache. This approach significantly reduces cache misses and optimizes memory access patterns. Moreover, by segmenting the iteration space into tiles, computations within each tile can function independently or with minimal communication, fostering parallelism and efficient memory utilization.

2.5 Tiling Optimization Studies

Extensive studies on tiling have addressed various challenges. For instance, *Determining the idle time of a tiling* [5] delves into tile shape optimization, focusing on reducing idle time and enhancing performance by selecting an appropriate tile shape. Additionally, there is a considerable focus on optimizing memory hierarchy in tiling strategies. Coleman and McKinley [4] introduce algorithms tailored for choosing problem-specific tile sizes based on cache and cache line sizes. The aim is to mitigate cache misses, which occur when the required computation data exceeds the cache memory capacity. Such misses impact system performance by requiring data retrieval from external memory sources.

2.6 Parameter Selection Problem

Selecting the best-performing tile size before run-time can be challenging, particularly with fixed tile sizes. Parameterized tiling, as discussed in [8], determines the tile size during runtime. What is interesting about both Tiramisu and Halide is that they use classical methods like Beam Search and MCTS (Monte Carlo Tree Search) to explore the search space generated by the compiler. In this paper, we delve into exploring 2D and 3D tile sizes for a given code. Instead of examining all possible candidates and selecting one, we employ a hill climbing algorithm to expedite the search for the best performing tile size, aiming to achieve optimal performance within the shortest possible time.

3 METHODOLOGY

3.1 Background

The MLIR compiler framework contains built-in passes which includes general transformation, affine and linalg dialect passes. Our work is on the linalg dialect and specifically on the parametric tiling transformation.

In general, the feasible transformations for exploration can be represented as a search space graph, as illustrated in Figure 1. Nodes in this graph denote distinct loop states, corresponding to specific points within the search space. For instance, a node may depict a loop with unrolled inner loops or a loop structured with parametric tiling. The edges

in this graph signify transitions between these states, each representing the application of a transformation rule.

The search process entails exploring the graph to identify the optimal set of transformations that minimize execution time or memory usage. Evaluation using a cost model occurs at every node in the graph. This assessment assists in identifying the most suitable transformation from the available options. Once the best candidate transformation is selected, the process advances to the next node in the graph. Not only does the search involve finding the best transformation candidate but also the best parameters for each candidate.

Since our work is to be integrated into an autoscheduler within MLIR, the search method when fully developed for all transformations should take a list of optimizations as input and produces a subset of optimizations aimed at minimizing execution time. Figure 2 illustrates the input and output design.

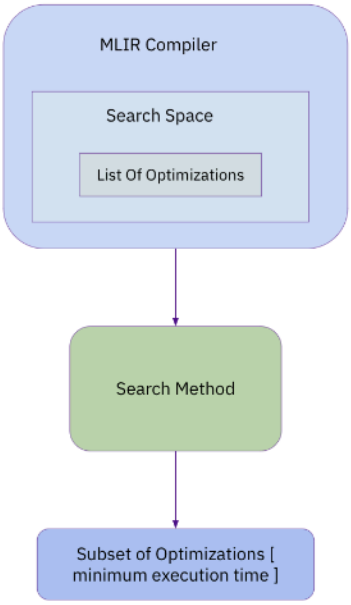

Figure 2: search method architecture

3.2 Choosing Hill Climbing Algorithm

Graph search algorithms include Depth-First Search (DFS), Breadth-First Search (BFS), Beam Search (a variant of BFS), Monte Carlo Tree Search (MCTS), and Hill Climbing. The reason why BFS or DFS is not preferred is that we aim to avoid exploring the entirety of the extensive search space. In the case of Beam Search, the algorithm requires a list of promising solutions at the beginning of the search, but the absence of the most promising solutions at the beginning of the search may lead to overlooking superior alternatives, potentially resulting in sub-optimal or incomplete outcomes. Moreover, due to time constraints, engaging in random simulations, as Monte Carlo does, is not feasible. Therefore, Hill Climbing appears to be the most practical option since it involves iterative steps to enhance solutions by making incremental improvements. We start with an arbitrary solution and then attempt to find a better solution through incremental changes as long as there are improvements, which prevents exploring the entire solution space. When applied to tiling, we change parameters, apply transformations, and check whether it gives a better solution or not.

Additionally, hill climbing is one of the go-to algorithms when computation time is limited, especially in real-time systems. Our objective is to automatically optimize code while still having compilers that run efficiently and take less time to find the best optimizations. Therefore, it defeats the purpose to choose an algorithm that takes a long time to run. Hill climbing is relatively simple and the primary choice for optimization in most AI algorithms. Moreover, hill climbing is an anytime algorithm; it can yield a valid solution even if interrupted before completion. This aspect is particularly crucial when operating within a CPU environment with limited resources.

There are three main problems that could arise from choosing the Hill Climbing algorithm to explore the search space of loop tiling optimization. The first problem involves ridges, which manifest as flat regions within the search space. We address this challenge by expanding our exploration to six options in our 3D search implementation and four options in our 2D search implementation. If no progress is observed compared to the parent node, we halt the search process. The second issue arises when encountering plateaus, where neighboring regions possess similar values with no significant difference. Our algorithm is finely attuned to even microsecond improvements, continuing the search as long as there is any incremental improvement in the performance of the optimization.

The third limitation worth mentioning regarding the Hill Climbing algorithm is its dependence on the initial solution, significantly impacting the quality of the final outcome. In our case, initiating each iteration with a consistent tile size of 32, applied uniformly in both the 2D and 3D tiling implementations, ensures a stable and consistent quality for the resulting solution. Also, local maxima is not a problem because there is no local maximum but multiple global maxima.

3.3 Implementation

Our work was not developed as a standalone search method; rather, it functions as a component within an auto scheduler

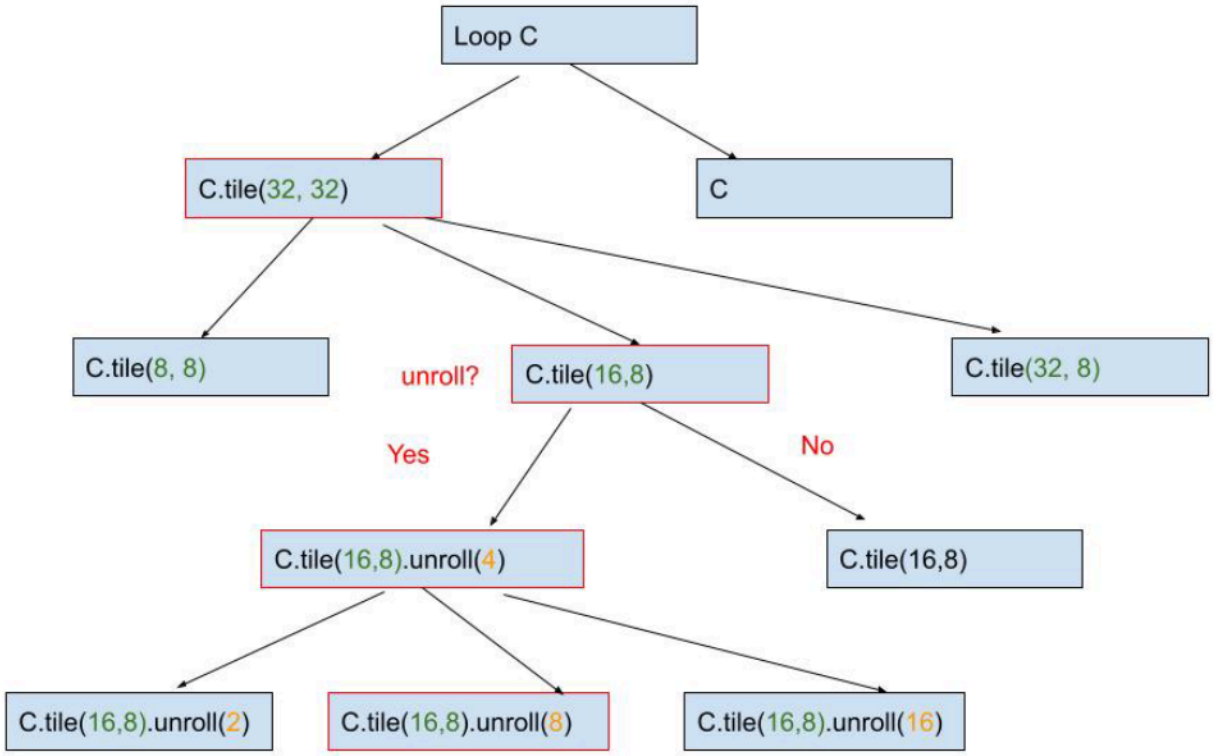

Figure 1: Graph of Optimization Search Space: Nodes Depict Loop States, Edges Signify Transformation Transitions

that integrates with MLIR. The implementation was specifically designed as the API endpoint responsible for providing the auto scheduler with essential parameters to optimize performance through transformations. The primary emphasis of this implementation lies in the loop tiling transformation, particularly within the linear algebra (linalg) dialect of MLIR.

The initial phase of the implementation involved defining the search space. To optimize nested loops, compilers break down the iteration space into smaller tiles or blocks, enhancing data reuse within CPU caches and reducing memory access overhead. The reference to *size 32* specifically denotes the chosen tile size for this optimization technique. It represents loop tiling optimizations where loops are divided into blocks of 32 elements (or iterations).

These tiles can be arranged in different dimensions: either in a flat, 2D configuration, like {32, 32}, or in a three-dimensional, 3D shape like {32, 32, 32}. Within these configurations, there is flexibility in the size of each unit. You can choose different sizes, like 8, 16, 24, 32, 64, or 128 units, allowing for a variety of options at each position in both the 2D and 3D shapes.

Therefore the search space encompasses exploring all possible combinations of these unit sizes within both 3D and 2D arrangements.

The Hill Climbing Algorithm implementation begins by creating an initial state using a tile size of 32, which is then assessed. Using a step size of 8 units, neighboring states are generated and compared repeatedly, continuing until there is a performance improvement or the predefined termination condition is reached. This iterative procedure is outlined in

Algorithm 1. The evaluation examines how the code performs after applying the loop tiling transformation with the specified tile size. The inclusion of the break condition is essential in mitigating ridges and proves beneficial during the search time evaluation process.

Algorithm 1 Hill Climbing Algorithm

```
current_state ← GenerateInitialState()
current_evaluation ← Evaluate(current_state)
while termination_condition not met do
    neighbor ← GenerateNeighbor(current_state +- stepsize)
    neighbor_evaluation ← Evaluate(neighbor)
    if neighbor_evaluation ≥ current_evaluation then
        current_state ← neighbor
        current_evaluation ← neighbor_evaluation
    else
        break
    end if
end while
return current_state, current_evaluation
```

4 EVALUATION

4.1 Method

We wanted to answer two main questions in our evaluation process: 1-Is there a noticeable speed increment when hill climbing is used to search the space? 2-How long does it take to execute for each search? In our evaluation process, we utilized two main criteria. First, we focused on performance, recording the performance for each tile size variation in GFLOPs. Additionally, we measured the search time, which captures the duration from the method's start to end in relation to the complete execution time. This was done for both 2D and 3D tiling. We assessed improved speed across various data sizes and domains. The evaluation was done with 4 benchmarks that form the base of most machine learning models; Matmul, Conv1d, Conv2d, Conv3d. Matmul excels in matrix multiplication for numerical computations. Conv1d manages 1D sequential data in NLP, while Conv2d processes 2D image data for Deep Learning and Computer Vision. Conv3d tackles 3D volumetric data crucial in Medical Imaging and Video Processing.

4.2 Results

Our results affirm that implementing tiling as a loop optimization enhances code performance compared to running untiled code. Across the four benchmarks, our method of finding the best tiling parameters using the hill climbing algorithm outperforms the usage of default parameters. Notably, there is a substantial improvement observed, particularly in conv3d (400X performance improvement) for 3D Tiling and 900X for conv2d in 2D Tiling (refer to Figure 3 and Figure 5). Likewise, our method demonstrates effectiveness in identifying the optimal tile parameters that yield the highest performance enhancement within a shorter timeframe. This is observed in conv2d for 2D Tiling and conv3d for 3D Tiling (refer to Figure 4 and 6 across the benchmarks).

The consistency in observed results across both 2D and 3D Tiling indicates the reliability and uniformity of the findings, showcasing its effectiveness across various tile sizes. Additionally, our method's implementation significantly reduces search time when compared to exhaustive search methods.

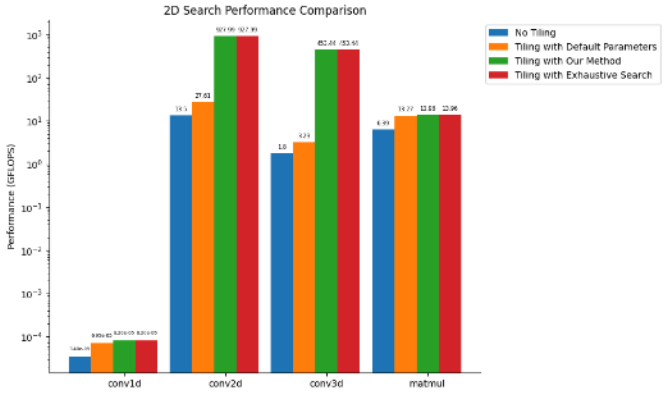

Figure 3: 2D performance comparison for each benchmark, categorized into no tiling, tiling with default parameters, tiling using our method, and tiling using exhaustive search.

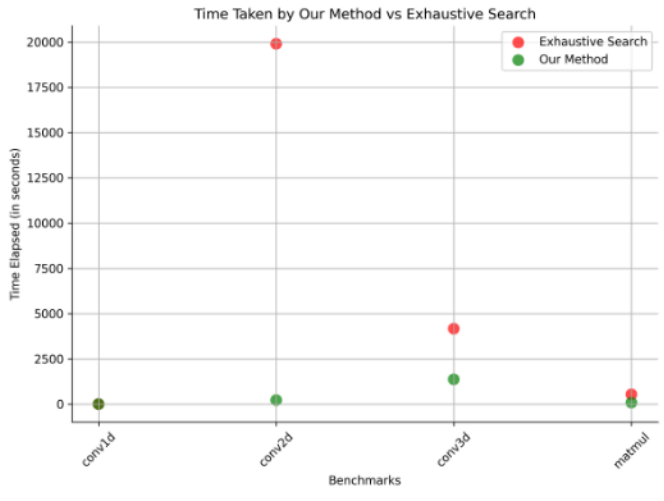

Figure 4: 2D search time comparison for each benchmark, our method against full execution.

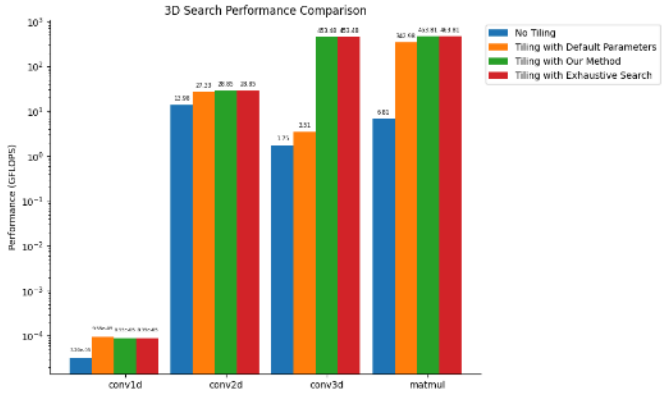

Figure 5: 3D performance comparison for each benchmark, categorized into no tiling, tiling with default parameters, tiling using our method, and tiling using exhaustive search.

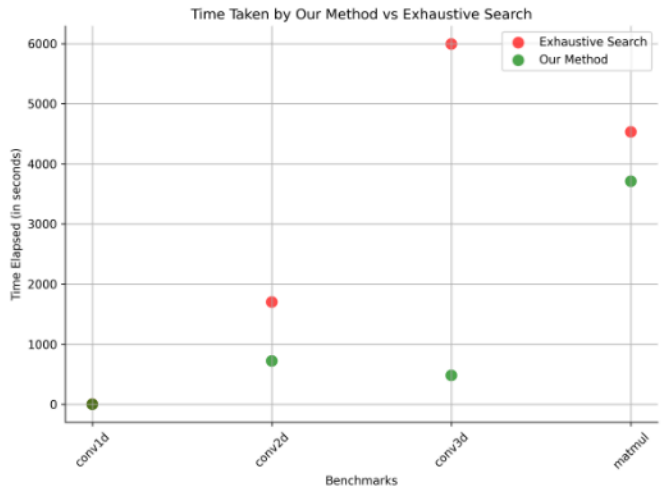

Figure 6: 3D search time comparison for each benchmark, our method against full execution.

5 CONCLUSION

Our goal was to accelerate finding the best tile size parameters in the MLIR compiler. The search space exploration selected was the hill climbing algorithm. The method was implemented and evaluated with 4 machine learning benchmarks. The results are consistent across both 2D and 3D Tiling. Our method has better speed performance and execution time. For future work, the plan involves trying on all optimizations and evaluating on larger benchmarks.

REFERENCES

[1] Saman P. Amarasinghe and Monica S. Lam. 1993. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 126–138. https://doi.org/10.1145/155090.155102

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

[3] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 33, 12 pages. https://doi.org/10.1145/2503210.2503289

[4] 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* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/207110.207162

[5] Karin Högstedt, Larry Carter, and Jeanne Ferrante. 1997. Determining the Idle Time of a Tiling. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 160–173. https://doi.org/10.1145/263699.263716

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[7] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[8] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. 2012. Parameterized Loop Tiling. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 3 (may 2012), 41 pages. https://doi.org/10.1145/2160910.2160912

[9] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449