Using a Dynamic Schedule to Increase the Performance of Tiling in Stencil Computations

Michael Freitag
Department of Informatics and Mathematics
University of Passau
D-94032 Passau

***Abstract*—A stencil computation determines the values of points in a grid of some dimensionality by repeatedly evaluating a given function of a grid point and its neighbors. The parallelization and optimization of stencil computations are subject of ongoing research. The most prevalent approach is the subdivision of the iteration domain into smaller pieces, called tiles. We give an overview of a method to increase the performance of one such tiling algorithm further by employing a dynamic schedule for tile processing, improving both load balance and cache efficiency. A set of onedimensional stencil benchmarks exhibits a performance increase of up to 20% in comparison to the Pochoir stencil compiler.**

I. Introduction

Stencil codes are computationally intensive programs commonly found in scientific or engineering applications. They are employed, for example, in the solution of partial differential equations, geometric modeling and image processing [1].

A *stencil* determines the value of a point in a grid of some dimensionality as a function of previous values of this point and its neighbors, which is generally called the *kernel function*. A stencil computation applies the kernel function repeatedly to all points in a grid. While stencil computations are conceptually easy to implement using nested loops (see Figure 1), these implementations frequently exhibit rather poor performance due to insufficient data reuse and parallelism.

```
for (t = 1; t < T; t++) {
  for (i = 0; i < N; i++) {
    B[i] = 0.125 * (A[i-1] - 2 * A[i] + A[i+1]);
  }

  swap(A, B);
}
```

Fig. 1. A loop-based implementation of a onedimensional heat equation stencil.

Fortunately the regular computational structure of stencil computations comes with a number of properties that make them suitable for optimization. Past research has found tiling of the iteration domain to be a key transformation for improving the performance of stencil computations. There are multiple approaches to tiling of stencil codes, including parallelogram tiling [2], diamond tiling [3] and trapezoidal tiling [4]. One of the most efficient tiling schemes is the trapezoidal tiling algorithm employed by the Pochoir stencil compiler [4]. It has been used as a reference to evaluate the performance of other approaches [3], [5]. But, although it already produces highly performant code, this algorithm can be improved further, because its recursive nature induces unnecessary synchronization and dependences between tiles, as is outlined in the next section.

Based on Pochoir's trapezoidal tiling algorithm, we developed a two-stage tiling algorithm which minimizes the amount of dependences and synchronization. The key idea of this algorithm is to decompose the iteration domain into tiles in a separate stage, before actually evaluating the kernel function in the second stage. This allows for otherwise infeasible optimizations to be performed in both stages. In the first stage, a tile dependence graph is generated using a slightly modified version of Pochoir's trapezoidal algorithm. In the second stage, this graph is traversed in parallel following a dynamic schedule while still respecting all dependences between tiles. This is possible with minimal synchronization, and without any barrier synchronization.

The rest of this paper is arranged as follows. Section II identifies the problems with the tiling algorithm employed by Pochoir in more detail. Section III provides an overview of the approach taken to overcome these problems. Section IV presents the results of experimental evaluation, and conclusions are drawn in Section V.

II. Problem Description

The tiling algorithm of Pochoir is based on trapezoidal decompositions [2], [6], which recursively fragment the iteration domain into well-defined tiles of trapezoidal shape (see Figure 2). Let us call these trapezoidal tiles *trapezoids* below [4]. Different types of cuts along the spatial and time dimensions are employed to decompose a trapezoid, each of which requires the size of a trapezoid to meet certain constraints in order to be applicable. Given a trapezoid of adequate size, so-called *spacecuts* and *timecuts* are performed depending on the properties of the trapezoid, resulting in multiple smaller trapezoids that can then be processed recursively [4].

A spacecut is the preferred way of decomposing a trapezoid, as it produces three trapezoids of which two are independent. Thus, these two trapezoids can be processed in parallel, while the third one has to be processed either before or after them (see Figures 2a–2b).

If a spacecut cannot be applied, it is tried to apply a timecut instead. This results in two trapezoids, one of which

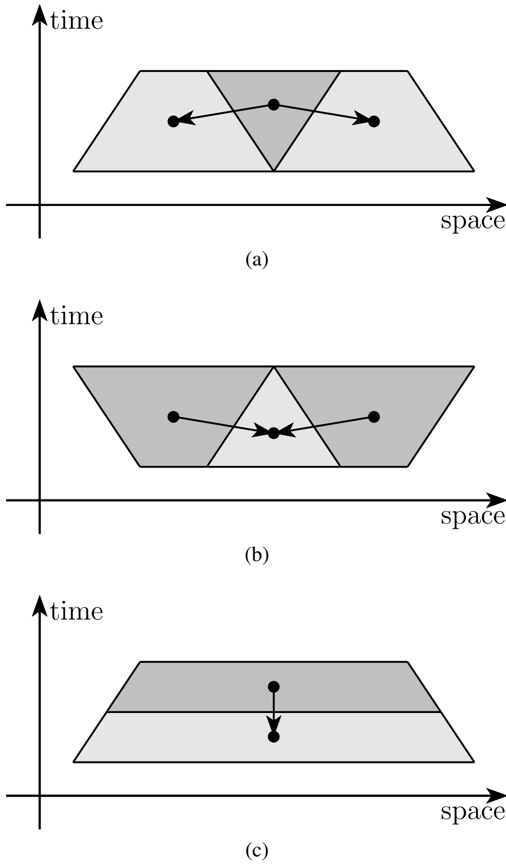

Fig. 2. Example of the resulting subtrapezoids and dependences when applying a spacecut as in (a) and (b) or a timecut as in (c) to a given trapezoid. Trapezoids of the same color can be processed in parallel, and the dark gray trapezoids can only be processed after the light gray trapezoids have been processed.

is dependent on the other. Therefore they have to be processed sequentially (see Figure 2c). If neither a spacecut nor a timecut can be applied, the kernel function is evaluated for all points of the trapezoid to be processed in the last step of the algorithm.

Pochoir does not explicitly keep track of dependences that accumulate while performing space- or timecuts. Instead, in each recursive step, a barrier synchronization is enforced after spawning tasks to handle independent subtrapezoids, thus making sure that all dependences are obeyed before processing the next subtrapezoid. This leads to subtrapezoids inheriting the dependences of their parent trapezoids during spacecuts and timecuts. However, some of these dependences may be redundant, because they do not represent actual flow dependences (see Figure 3). The elimination of these dependences is the starting point for the optimizations we performed.

This behavior is intended in Pochoir's algorithm, as it ensures a processing order that increases cache efficiency greatly [4]. Nevertheless, it gives rise to a number of experimentally verifiable problems. First of all, a notable load imbalance is induced because the processing of some tiles takes longer than that of others. Particularly in boundary regions of the iteration domain, extensive boundary checks and evaluation of boundary functions may increase the processing time of a

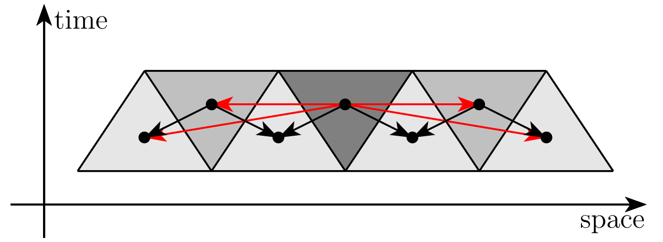

Fig. 3. Example of a trapezoid inheriting its predecessor's dependences. An additional spacecut has been applied to the trapezoids produced by a spacecut as depicted in Figure 2a, leading to the false dependences colored in red.

tile by an order of magnitude. Additionally, data reuse suffers between tiles that are too large to fit into the cache, as such tiles are processed in all before any dependent tile can be processed. Therefore, a large amount of data that could be reused is expelled from the cache before being referenced. Evidence of both of these issues has been found during experimental evaluation and is highlighted in Section IV.

III. Overview of Approach

The algorithm that we developed to overcome these issues is based on the recursive algorithm employed by Pochoir. However, it separates the generation of tiling information and the actual stencil computation into distinct stages. In the first stage, a tile dependence graph is produced which holds information about the shape and dependences of each tile. This graph is used in the second stage to perform the actual stencil computation, by traversing the graph in parallel and evaluating the kernel function on the points contained in each tile, while ensuring that all dependences are respected and as much data as possible is reused.

A. Generation of the Tile Dependence Graph

The tile dependence graph is a directed acyclic graph that holds all information needed to model the decomposition of an iteration domain with a given size. Each node V in the graph represents one trapezoid, while each edge $V \rightarrow W$ indicates a flow dependence between the trapezoids represented by V and W, respectively.

To generate the tile dependence graph, we start with one node representing the entire iteration domain and expand that node by recursively applying spacecuts and timecuts as Pochoir does. In each expansion step, a node is replaced by several nodes representing the subtrapezoids that result from applying a cut to the trapezoid represented by the original node. All dependences of the original node are initially inherited by the replacement nodes. Additional dependences between the replacement nodes are added as needed (see Figures 4a–4b). Due to space limitations, we refrain from a detailed description of Pochoir's trapezoidal algorithm [4].

As opposed to Pochoir, we represent trapezoids as polyhedra using the *Integer Set Library* [7] during the first stage. This enables us to apply a polyhedral dependence analysis in order to eliminate all false dependences that the cutting strategy suggests, immediately after a cut was applied (see Figure 4c).

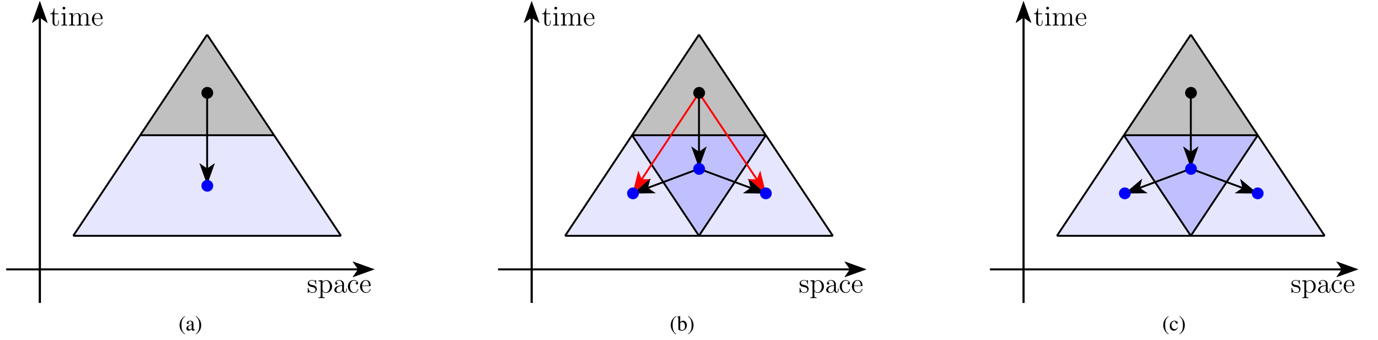

Fig. 4. Example of a recursive expansion of a node in the tile dependence graph by applying a spacecut. The node representing the blue trapezoid in (a) is replaced by nodes representing the blue subtrapezoids in (b). Each node inherits the dependences of the original node in (a), resulting in the redundant red dependences. These are eliminated by a polyhedral dependence analysis, which leads to the final tile dependence graph in (c).

```
1: procedure PROCESS(V)
2:     Evaluate kernel function for all points in V
3:     for all edges V → W do
4:         remove edge V → W
5:         if W has no incoming edges then
6:             Spawn parallel task PROCESS(W)
7:         end if
8:     end for
9: end procedure
```

Fig. 5. Algorithm used to process a node V in the tile dependence graph.

The recursive expansion of the nodes in the tile dependence graph terminates when a tile is smaller than a given threshold value, which can be adjusted to adapt the algorithm to different cache architectures. As boundary checks generally increase the processing time of a tile by a significant amount, a separate threshold value is maintained for boundary tiles. By lowering that threshold, the number of tiles that contain boundary accesses can be reduced further.

B. Traversal of the Tile Dependence Graph

After the tile dependence graph has been generated, it can be used to perform the stencil computation in the second stage for an arbitrary number of times. To do so, we traverse the graph in parallel using the algorithm outlined in Figure 5.

First of all, the kernel function is evaluated on all grid points in the trapezoid represented by a node V (line 2). This step accounts for most of the time needed to process a node. Afterwards, all outgoing edges $V \rightarrow W$ are removed (lines 3-4), and a new parallel task to process W is spawned if W has no more incoming edges, i.e., pending dependences (lines 5-7). Synchronization is needed only while we check whether W has no more incoming edges and spawn a parallel task if so. It is sufficient to synchronize only threads that access the same node, so no barrier synchronization is required.

Our algorithm starts off by spawning tasks for all nodes that have no incoming edges. These nodes represent trapezoids that depend only on the initial conditions of the stencil computation. The algorithm terminates when all nodes in the tile dependence graph have been processed.

C. Further Optimizations

We performed further optimizations on the evaluation of the kernel function itself, similar to those performed by the Pochoir stencil compiler. Code cloning is employed to generate a *boundary clone* and a faster *interior clone* of the kernel function, the latter not performing any boundary checks. Most optimizations target the interior clone, as the boundary region of the iteration domain is small compared to the interior [4].

IV. Empirical Evaluation

We evaluated the performance of our algorithm on a set of onedimensional stencil codes and used Pochoir's tiling algorithm as a reference. Owing to limited space, we present only the results for one selected stencil code in the following, namely the onedimensional heat equation stencil [8] that is shown in Figure 1.

A. Benchmark setup

We used an Intel Core i7-2630QM processor with 4 cores for benchmarks. This CPU has 32 KB L1 cache, and 256 KB L2 cache per core, while all cores share 6 MB L3 cache. All benchmarks use double-precision floating-point operations, leading to a theoretical peak performance of 89.6 GFLOPS. Our implementation was compiled with `g++ -O3`, while Pochoir was compiled with `icpc -O3`, as we were not able to compile Pochoir successfully using the `g++` compiler and vice-versa. Up to now, we have not been able to pinpoint precisely why our code fails to execute properly when compiled with `icpc`. The tile sizes were tuned using a combination of manual search and the Intel Software Autotuning Tool [9] in a limited amount of time.

We gathered information about execution time, cache efficiency and CPU load using the Linux `perf` performance counter subsystem for different problem sizes and on a variable number of cores. We performed benchmarks on grids containing $0.1 \cdot 10^6$ to $1.5 \cdot 10^6$ points over 50000 timesteps using all 4 available CPU cores, as shown in Figures 6a–6c.

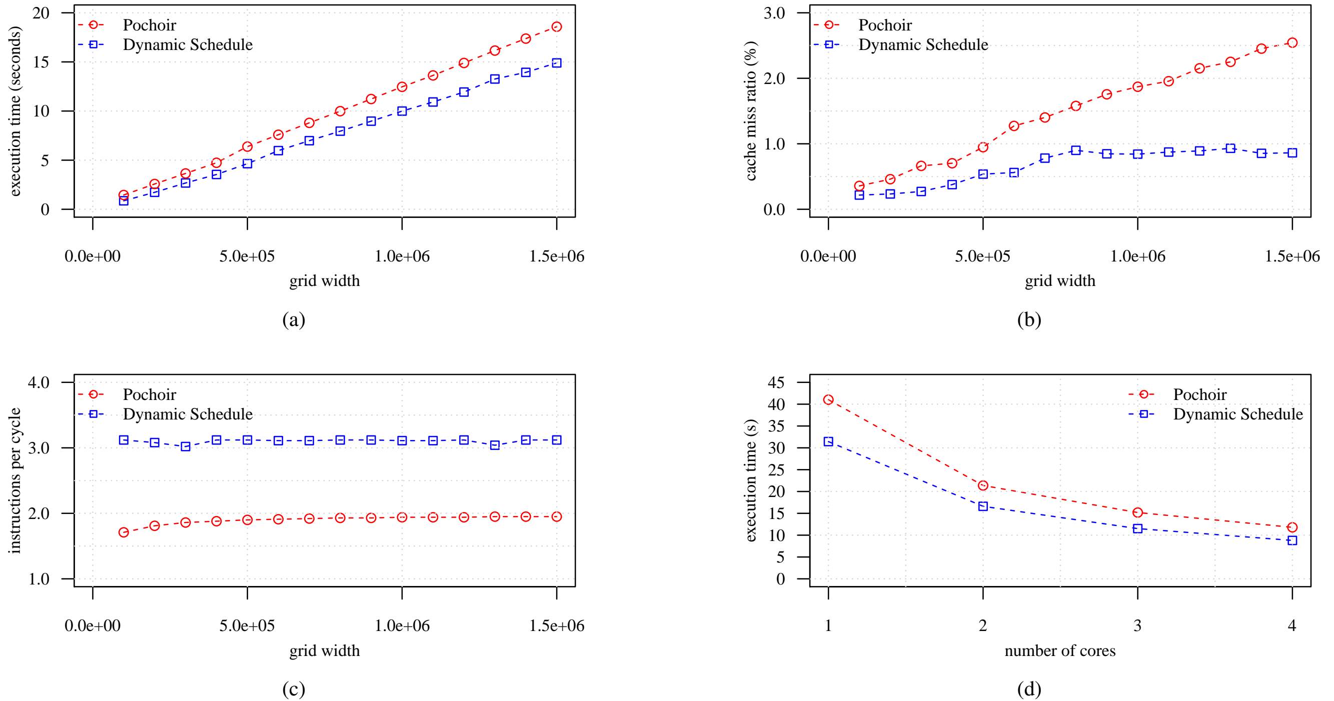

Fig. 6. Benchmark results for a onedimensional heat equation stencil, implemented using our implementation and the Pochoir stencil compiler. (a) shows the execution time in relation to the problem size, i.e. the number of grid points. (b) shows the cache miss ratio, and (c) shows the number of instructions processed per CPU cycle. (d) shows how our implementation and Pochoir scale depending on the number of CPU cores available.

Additionally, we performed benchmarks on a grid containing 10^6 points over 50000 timesteps using 1, 2, 3 and 4 CPU cores, with results as shown in Figure 6d.

B. Benchmark results

We were able to outperform Pochoir on all problem sizes by up to 20%. Figure 6b also shows a notable improvement in cache efficiency for adequate problem sizes. Figure 6c exhibits that, although Pochoir and our implementation perform roughly the same number of instructions, our implementation causes the instruction pipeline to be idle in fewer CPU cycles. This indicates that our implementation induces better load balancing. For other onedimensional stencil codes that we used for benchmarks, our implementation exhibits a similar performance increase in comparison to Pochoir.

V. Conclusion

We gave an overview of how the tiling algorithm of Pochoir can be improved by eliminating redundant dependences and by employing a dynamic schedule for tile processing. A set of onedimensional stencil benchmarks shows a notable improvement of both cache efficiency and CPU load, leading to a significantly shorter execution time. In the future, our implementation will be extended to support different tiling algorithms and stencil codes of higher dimensionality. Related work [3] encourages the assumption that other algorithms can also benefit from employing a dynamic schedule for tile processing.

Acknowledgment

Partial funding was gratefully received from DFG Priority Programme SPP 1648 (SPPEXA), Project ExaStencils (LE 912/15-1).

References

[1] Y. Tang, R. Chowdhury, C.-K. Luk, and C. E. Leiserson, "Coding stencil computations using the Pochoir stencil-specification language," Poster session presented at the 3rd USENIX Workshop on Hot Topics in Parallelism, 2011.

[2] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*. ACM, 2005, pp. 361–366.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, pp. 40:1–40:11.

[4] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir stencil compiler," in *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2011, pp. 117–128.

[5] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th Annual International Conference on Supercomputing (ICS)*. ACM, 2013, pp. 13–24.

[6] M. Frigo and V. Strumpen, "The cache complexity of multithreaded cache oblivious algorithms," in *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2006, pp. 271–280.

[7] "Integer set library homepage." [Online]. Available: http://www.ohloh.net/p/isl

[8] J. F. Epperson, *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, 2007.

[9] "Intel software autotuning tool." [Online]. Available: http://software.intel.com/en-us/articles/intel-software-autotuning-tool/