TOAST: Automatic Tiling for Iterative Stencil Computations on GPUs

Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos, Luís F. W. Góes

Abstract

The stencil pattern is important in many scientific and engineering domains, spurring great interest from researchers and industry. In recent years, various optimizations have been proposed for parallel stencil applications running on Graphics Processor Units (GPUs). In particular, tiling is a technique that can significantly enhance application performance by improving data locality and by reducing the volume of communication between host memory and GPU. In addition, tiling enables stencil applications to process inputs that are larger than the physical GPU memory. However, implementing tiling efficiently is complex, time-consuming, and error-prone. In this paper, we propose TOAST, an automatic tiling mechanism for iterative stencil computations running on GPUs. TOAST has three main benefits: (1) it incorporates an optimization model that seeks to maximize data re-use within tiles, while respecting the amount of dynamically available GPU memory; (2) it offers a virtualized GPU memory for stencil computations, allowing for large input data; (3) it performs optimal tiling transparently to the developer of the parallel stencil application. The current implementation of TOAST augments the PSkel framework with an internal solver based on genetic algorithms. Our experimental results show that TOAST improves the performance of iterative stencil applications by up to $13\times$ compared to their multi-threaded (CPU-based) optimized versions, and up to $48\times$ compared to a naive tiling approach on GPU. The TOAST mechanism is able to automatically achieve a low percentual overhead of data management compared to actual stencil computation.

Citation:

Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos, and Luis F. W. Goes, “TOAST: Automatic tiling for iterative stencil computations on GPUs”, Concurrency and Computation: Practice and Experience, p. e4053, 2017

This preprint version is only for personal use.
The original paper has been published on the journal *Concurrency and Computation: Practice and Experience*. Copyright © John Wiley & Sons Ltd.

1 Introduction

The stencil computational pattern has important applications in many domains, including image processing, cellular automata, and computational fluid dynamics [16]. In particular, parallel stencil computations have been the subject of numerous studies and optimizations targeting both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) [7, 14, 17, 21, 22, 27].

In the context of stencil optimizations for GPUs, an important technique is tiling. Tiling partitions the iteration space into smaller regular blocks, called *tiles*. When tiling stencil computations, neighborhood dependencies inherent to the stencil parallel pattern must be considered before partitioning the data into smaller blocks. One of the main solutions for handling neighborhood dependencies is via overlapped blocks, resulting in redundant data and computation per tile [16, 18].

Tiling can improve parallel stencil applications in at least three ways [29]. First, tiling partitions loop data and computations into tiles, thereby enabling the GPU to handle amounts of input data that exceed the capacity of its internal memory. Second, tiling reorders loop nesting of the stencil, which can improve spatial and temporal locality within the tiles. Third, the partitioning combined with loop reordering potentially reduces the volume of communication between host memory and GPU, which reduces the memory bandwidth requirements for the application.

Despite the potential of tiling to improve application performance, bringing those improvements to fruition entails non-trivial challenges for the application developer. For example, the developer must decide on an optimal tile size within a very large parameter search space [3]. In particular, too large of a tile size may cause a thrashing effect on the GPU memory, whereas too small of a tile size could cause poor data reuse and excessive loop overheads. Moreover, the developer must determine the best ordering of loop nesting to achieve the best locality, which, for iterative applications, also requires determining the maximum number of loop iterations possible within each tile to ensure correctness. Finally, introducing tiling, even into an existing stencil application is complex, time-consuming, and error-prone.

Most previous research on tiling focused on optimizations for CPUs. Specifically, the optimizations promoted concurrency, temporal locality, and/or reduced memory throughput requirements [2, 10, 18, 19, 25, 29]. However, the use of those techniques on GPUs is limited by architectural differences from CPUs. For that reason, recent efforts targeted GPUs [12, 24] and CPU-GPU systems [13]. Nevertheless, they do not simultaneously provide automatic and optimal tiling for parallel stencil computations on GPUs, while supporting large inputs and abstracting tiling for the programmer.

In this paper, we propose TOAST (Transparently-Optimized Automatic Stencil Tiling), a mechanism that performs automatic and transparent tiling of iterative stencil computations running on GPUs. TOAST transparently offers a virtualized GPU global memory for stencil computations, allowing for input data much larger than the available GPU memory. It is based on an optimization model that seeks to maximize data re-use within tiles, while respecting the dynamic amount of available physical GPU memory. To promote data re-use, the optimization model runs as many meaningful iterations as possible on the tile data before moving on to the next tile. As a result, TOAST improves data locality and minimizes the volume of host-to-GPU data transfers. In addition, the optimization runs transparently, thereby enabling the parallel application developer to focus on problem-related implementation aspects.

To evaluate TOAST, we implemented it into the PSkel[1] stencil framework [26]. Specifically, we extended the framework's internals with a generic tiling mechanism and the TOAST optimization solver based on genetic algorithms. Next, we evaluated different versions of four iterative stencil applications: CloudSim [6], Game of Life [11], Jacobi's method [8], and 3D Laplacian [5]. We found that, comparing across multiple tiling configurations, TOAST improved the average GPU application performance by 6×. In addition, compared to a naive tiling GPU approach, TOAST improved performance by up to 48×, while performing within 1% of an oracle. Finally, compared to an optimized multi-threaded implementation running on CPU with 24 threads, TOAST improved the performance of iterative stencil

[1]`http://pskel.github.io`

applications by up to $13\times$.

In summary, our main contributions are: (1) proposing a mathematical optimization model for tiling of stencil computations; (2) implementing the model into a near-optimal tiling mechanism that offers a virtualized GPU global memory for stencil computations, supporting large input data in a transparent manner to developers; (3) augmenting the PSkel framework with the proposed mechanism; and (4) evaluating the mechanism and comparing it against alternative approaches.

We organize the remainder of this paper as follows. Section 2 provides background on the stencil pattern and on tiling, while discussing related work. Section 3 introduces the TOAST mechanism and describes its optimization model. Section 4 presents our evaluation methodology and results, and Section 5, our conclusions.

2 Background and Related Work

In this section, we present background and discuss related work on stencil applications and the tiling technique applied to stencils.

Parallel skeletons and the stencil pattern. A structured parallel program is typically composed of computation and coordination. Computation represents the application's logic and data flow control, whereas coordination manages parallelism and concurrency. Coordination concerns include process and thread synchronization, communication, and load-balancing [23]. Since several parallel applications exhibit common computation and coordination patterns, programmers often model those patterns as parallel skeletons. With skeletons the programmer can focus on designing parallel algorithms rather than worrying about runtime system details, thereby speeding up application development and debugging. Moreover, skeletons add structure to parallel programs, so programmers can build more complex skeletons from simpler ones. Among the several existing patterns of parallel skeletons (e.g., map, reduce, pipeline), the stencil pattern has been used in applications of many important fields, such as quantum physics, weather forecasting, and digital image processing. Due to its importance, many recent efforts in GPU research sought to improve the performance of stencil computations [5,7,16,17,20,22,30,31].

The stencil pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element. In particular, a sliding window (or mask) scans the entire input data set and produces output data using a stencil function. The mask size corresponds to a specific number of neighbors of each element of the input data. The stencil function then performs computations using the mask and the neighbors of each element of the input data to produce a corresponding element in the output data. Finally, the stencil application repeats that process on every element of the input data.

The tiling technique. *Tiling* is commonly used to process large amounts of data on limited hardware resources. The technique dates back to software-based graphics renderers running on CPUs, where input data were subdivided into regular grids (*tiles*). The subdivisions facilitated data handling by processing and storage hardware resources, while exploiting the spatial and temporal locality of the data. Tiling remains relevant even in modern GPU-based architectures, where memory can be scarce in light of applications with large data sizes. In addition, in those systems, the host-to-GPU memory bandwidth is a bottleneck, especially given the data transfer overheads that tiling imposes [16,20].

Tiling on stencil computations. To illustrate and detail how tiling can be applied to stencil computations, we use a formal definition. Let A be a 3D data matrix, with dimensions $\text{dim}(A) = (w, h, d)$, where w, h, and d are, respectively, its width, height, and depth. Using tiles of dimensions (w', h', d') yields $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil \lceil \frac{d}{d'} \rceil$ possible tiles of A. Let $A_{i,j,k}$ be one such tile, where $0 \leq i < \lceil \frac{w}{w'} \rceil$, $0 \leq j < \lceil \frac{h}{h'} \rceil$, and $0 \leq k < \lceil \frac{d}{d'} \rceil$. $A_{i,j,k}$ has offset (iw', jh', kd') relative to the top left corner of A, and $\text{dim}(A_{i,j,k}) =$

$(\min\{w', w-iw'\}, \min\{h', h-jh'\}, \min\{d', d-kd'\})$. The offset is an indexing displacement required for accessing the elements of the tile (see Figure 1).

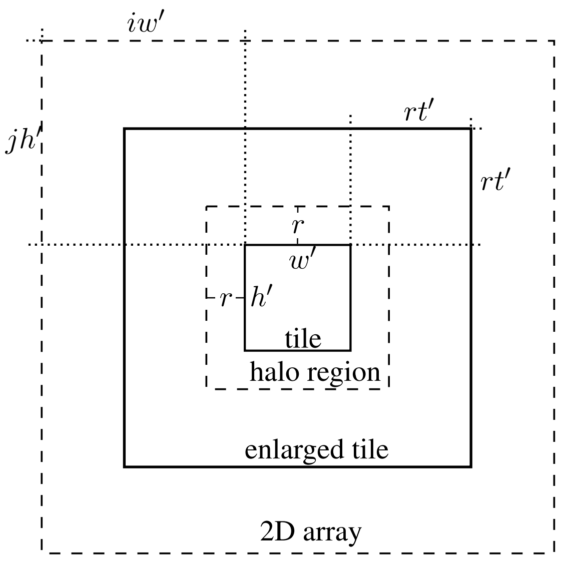


Figure 1: Diagram of 2D tiling. A logical tile (inner solid line) is contained in a 2D array (outer dashed line) with vertical and horizontal offsets given by jh' and iw'. Computing t' consecutive stencil iterations on the tile requires enlarging the logical tile with a ghost zone (area between the inner solid line and the outer solid line), which is comprised of halo regions (the area between the inner solid line and the inner dashed line).

Applying a stencil on A, entails computing an element-wise neighborhood function (mask) containing the displacement of each neighbor of a given central element. Due to neighbor dependence, computing the stencil function along the boundary of a tile requires obtaining values from adjacent tiles. Let r be the range of the neighborhood mask, i.e., r is the most distant displacement required for the neighborhood defined by the mask. The area of range r comprising the neighborhood is often denominated *halo region* [16, 20, 24]. If the stencil function is applied iteratively on A, for t iterations, the neighborhood dependence among tiles limits the number of iterations that can be consecutively computed in a GPU.

To enable performing $t' \in [1, t]$ consecutive iterations for each tile $A_{i,j,k}$, a tile must be enlarged to include a number of adjacent halo regions, collectively called a *ghost zone*. The number of adjacent halo regions that compose the ghost zone is proportional to t'. The overlap across neighboring tiles allows the GPU to generate its halo regions locally for a number of consecutive iterations proportional to the size of the ghost zone. In particular, the enlarged $A_{i,j,k}$ tile transferred to the GPU global memory has enlarged dimensions $\dim^*(A_{i,j,k}) = (\max\{\min\{w'+2rt', w-iw'+rt'\}, 1\}, \max\{\min\{h'+2rt', h-jh'+rt'\}, 1\}, \max\{\min\{d'+2rt', d-kd'+rt'\}, 1\})$ and has offsets $(\max\{iw'-rt', 0\}, \max\{jh'-rt', 0\}, \max\{kd'-rt', 0\},)$, relative to A, as illustrated in Figure 1.

Because of neighborhood dependencies in stencil computations, each iteration on an enlarged tile adds noise to the ghost zone from the border towards the center of the tile. After t' consecutive iterations, only the logical tile region holds correct and non-overlapping values. The larger the value of t', the more redundant computations are required to correctly compute the logical tile region. Thus, sizing the ghost zones poses a trade-off between the cost of redundant computations and the reduction in communication when processing iterative stencil computations on GPUs. Despite the potential benefit of this trapezoidal tiling technique, an improper selection of the ghost zone size may negatively impact the overall performance [24].

Tiling for locality and concurrency. Tiling techniques for iterative stencil computations have been vastly studied in the past. Previous research focused on exploiting temporal locality [10, 19] and/or reducing inter-dependence (and synchronization) across adjacent tiles to improve tile processing throughput [2, 9, 25, 28]. In particular, the approach by Krishnamoorthy *et al.* allows overlap across tiles to reduce synchronization overheads, at the expense of recomputing the overlapped regions across different tiles [18]. Zhou *et al.* leverage hardware hierarchy in multi-core CPUs to reduce the amount of recomputations [34], whereas Henretty *et al.* performs data layout transformation (DLT) to support vectorization and inter-tile concurrency [15].

Inherently, those techniques exploit architectural characteristics of CPUs that do not apply to GPUs. In particular, compared to CPUs, GPUs have large amounts of simpler processing elements that perform poorly in face of diverging control flows [12], but offer much greater computational throughput for arithmetic computations. In addition, GPUs typically feature a global memory and requires explicit or implicit data copying from the main system memory, thereby further hampering already scarce memory throughput. Finally, the amount of global memory in GPUs is severely limited (considerably smaller than CPU memory), thus restricting the execution on GPUs of applications with large input data sizes. To address those differences while improving locality and concurrency in GPUs, recent approaches have been proposed. For example, Verdoolaege *et al.* propose overlapped tiling optimizations for GPUs [33]. Grosser *et al.* propose a hybrid tiling approach that exploits temporal locality, improves thread-level parallelism and avoids thread divergence [12].

Overall, existing CPU and GPU approaches typically entail transforming iteration loops (time dimension) and stencil loops (space dimensions), provided that the stencil data is *enabled*; i.e., available at the memory level at which the processing elements can apply the loop transformations [25]. In CPUs, that level is typically the CPU cache, whereas in GPUs, the level is either the shared or the global memory of the device [13].

In those critical aspects, TOAST deviates from classic tiling approaches in at least three ways. First, TOAST *enables* the processing of input data that can be much larger than the global memory of the GPU, a discussion that is typically not addressed in other techniques. Existing approaches usually implements either tiling optimizations on CPU-GPU hybrid systems, with fixed size tiles and input sizes still limited by the amount of available GPU memory [13], or tiling optimizations on multi-GPU environments, without enabling input sizes that scale beyond the sum of the physical memory capacity of all GPUs [20].

Second, in those approaches, the tile size is known at compile time (e.g., programmer-provided) or found via exhaustive exploration of the search space. Even in cache-oblivious approaches, like [10], the programmer must specify the tile size directly or indirectly (e.g.,by specifying the change in tile start and end points at different timesteps) [15]. Conversely, TOAST abstracts the optimization from the programmer by automatically finding the optimal tile size and ghost zone size.

Finally, instead of performing complex transformations in the time and space loop dimensions, TOAST adopts a simple trapezoidal tile computation shape that leverages the inherent parallelism of GPUs without introducing divergence of control flow. That is possible based on the observation that an optimal configuration of tile and ghost zone sizes that considers the stencil mask size and the dynamically available GPU memory optimizes performance. That approach also avoids the need for complex profiling and dependence analysis.

Automatic tiling. Automatic tiling approaches typically target CPUs and are made available via compilers [33, 34] or frameworks [3, 13]. In particular, Gysi *et al.* [13] propose a framework for automatic tiling optimizations on CPU-GPU hybrid systems. Their approach performs automatic loop tiling and fusion on stencil applications in hybrid CPU-GPU systems, based on a compile-time performance model. However, the approach adopts fixed size tiles, specified manually, and does not support input sizes larger than the amount of available GPU memory. In contrast, Lutz *et al.* [20] addresses the memory capacity limitation of GPUs on multi-GPU environments to a certain extent, by using the GPU memories collectively. However, that approach does not scale beyond the sum of the physical memory capacity of all GPUs. Furthermore, the TOAST mechanism can easily be extended to offer a virtualized GPU memory in a multi-GPU environment.

Optimization models for tiling. Like TOAST, some recent works use optimization models for stencil tiling. For example, Meng and Skadron [24] propose a model for automatically selecting the optimal ghost zone size, while using the shared memory available in GPUs to improve the temporal locality in the GPU memory. Unlike TOAST, their approach performs dynamic profiling and off-line optimization, requiring code annotation. Conversely, TOAST performs fast on-line optimization (considering the dynamically-available GPU memory), transparently to the programmer. In addition, TOAST's optimization model encompasses scenarios where the input is much larger than the available GPU memory.

3 Automatic tiling mechanism

In this section, we describe TOAST, the proposed automatic tiling mechanism for stencil computations on GPUs. TOAST is based on an optimization model that seeks to maximize redundant computations performed on GPU and minimize also the number of host-device data transfers, while respecting the amount of available GPU memory. The current implementation of TOAST solves the optimization problem using a genetic algorithm, selecting near-optimal tiling parameters with a reasonably low overhead.

When the stencil application being processed is larger than the GPU memory, if the user is left with the burden of manually selecting the tiling parameters, the user might either underestimate or overestimate the GPU's memory capacity, leading to bad performance or memory exhaustion, respectively. We have implemented TOAST in PSkel so that tiling can be transparently used whenever an application is excessively large for being processed on a GPU.

3.1 The optimization problem

We propose a mathematical model that describes a heuristic for tiling stencil computations on GPU as an optimization problem. The maximization problem defined has two free parameters, namely, h' and t'. In this paper, we consider that the remaining tiling parameters are fixed in their maximum values, i.e., tile width $w' = w$ and tile depth $d' = d$, so that each tile is contiguously stored on main memory. Contiguous data can be transferred considerably faster between host and device [4].

We derive the objective function from two basic optimization requirements. Given the constraint of available memory, when tiling stencil computations on GPUs, the major goal is to reduce data transfer between host and device, while still maximizing the useful computation performed for each logical tile. Definitions 1 and 2 present those requirements.

Definition 1 **(R_1)** *Maximize the logical tile area, reducing redundant computation.*

$$\max_{h',t'} \frac{wdh'}{wd\min\{h, h' + 2rt'\}}$$

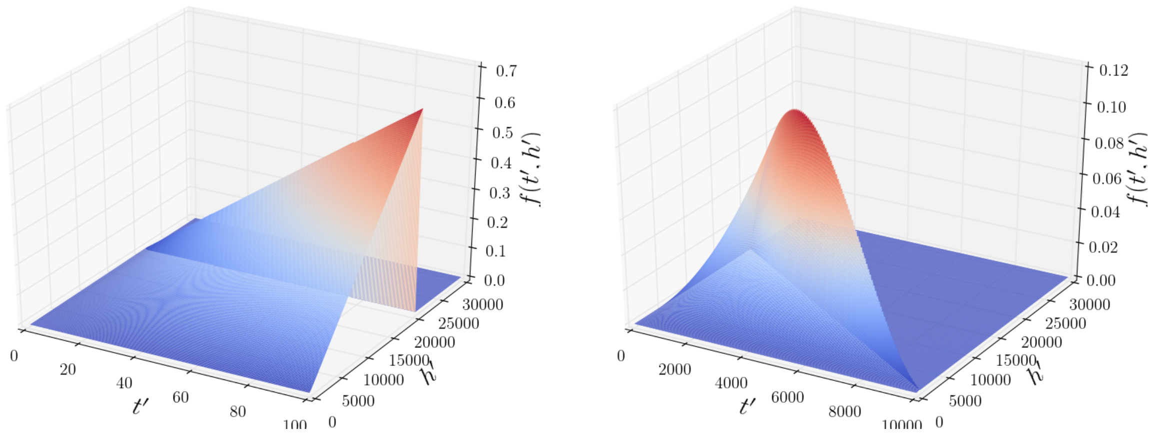
Definition 2 **(R_2)** *Minimize the total number of data transfer between host and device.*

$$\min_{h',t'} \frac{h}{h'}\frac{t}{t'}$$

The requirement R_2 (see Definition 2) can be equivalently defined as a maximization problem. Definition 3 presents the objective function that evaluates both requirements R_1 and R_2.

Definition 3 (Objective function) *Let M_F be the currently available memory on GPU. Let b be the number of bytes required to store in memory each element of the given matrix. If $1 \leq t' \leq t$ and $1 \leq h' \leq h$ are the free parameters of the maximization problem, the objective function is defined as:*

$$\max_{h',t'} f(h',t') := \begin{cases} \frac{h'}{\min\{h,h'+2rt'\}} \frac{h't'}{ht} & \textit{if } 2wdb\min\{h,h'+2rt'\} < M_F; \\ 0 & \textit{otherwise}. \end{cases}$$



(a) Total of 100 iterations.

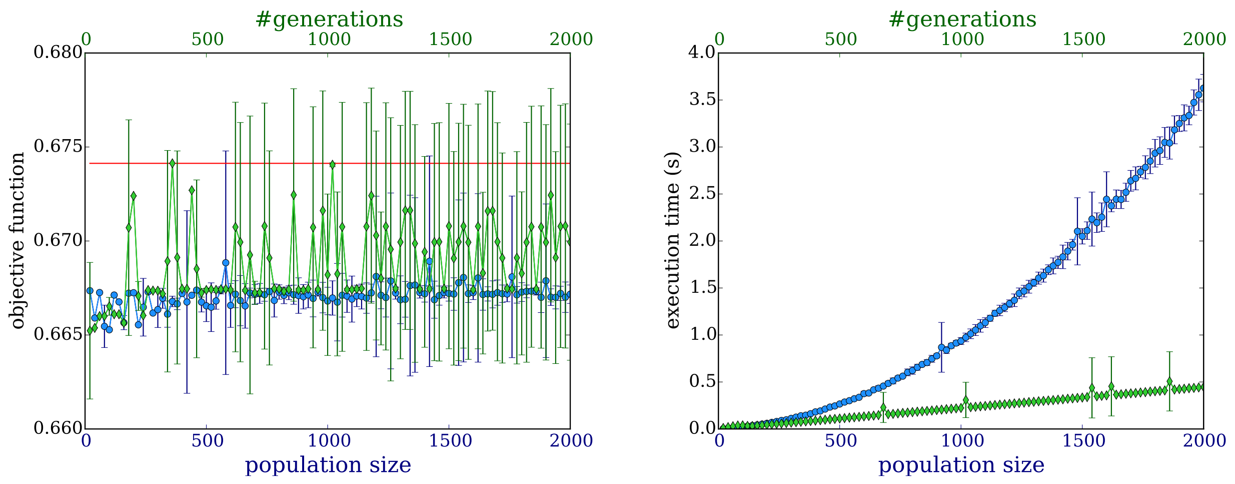
(b) Total of 10000 iterations.

Figure 2: 3D surfaces of the objective function in different scenarios. 2D array with size of 30000×30000, neighborhood mask with range 1, and 4799 MiB of available memory.

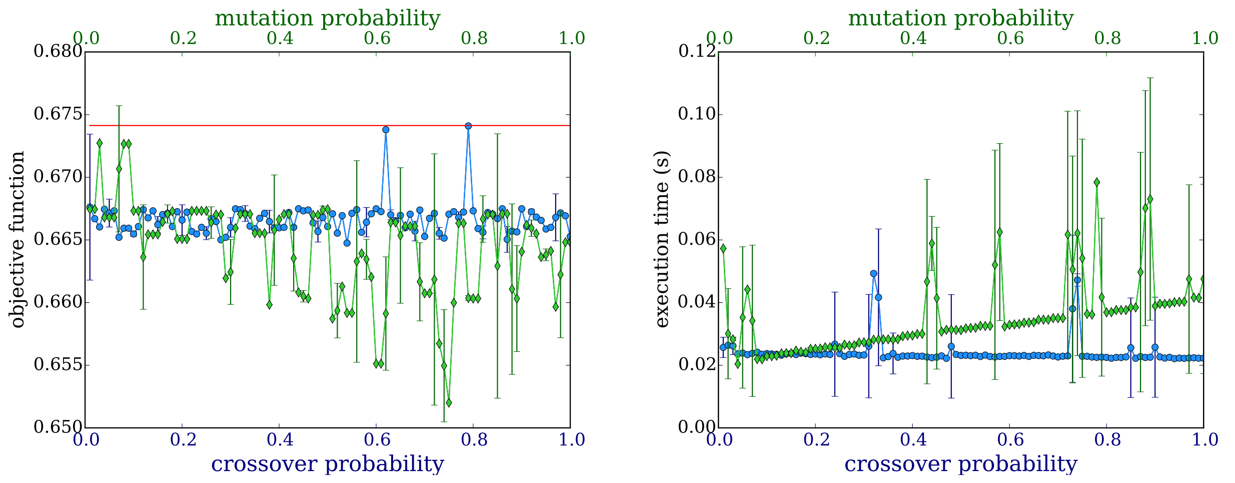
The value for the objective function f is constrained by the available memory on GPU. If each enlarged tile has size, in bytes, $wdb\min\{h, h' + 2rt'\}$, and since both input and output tiles must be stored on memory, then the memory constraint is $2wdb\min\{h, h' + 2rt'\} < M_F$. Note that, if there is enough available memory, the mathematical model is maximal with a single tile that completely covers the input matrix, performing all iterations consecutively on GPU. Figure 2 illustrates the 3D surfaces of the objective function in different scenarios. It is important to notice how the available memory limits both the tile size and the number of consecutive iterations (see Figure 2.b).

3.2 Evolutionary autotuning

The evolutionary autotuning implemented on PSkel solves the maximization problem in Definition 3 using evolutionary computing, namely, genetic algorithm [1]. The genetic algorithm allows for an efficient heuristic that solves the maximization problem with a reasonably low overhead. Figure 3 depicts experimental results of the solution accuracy provided by the genetic algorithm for the objective function, and its execution time. The genetic algorithm is able to generate a solution with an error of less than 1%, under half a second of overhead.



(a) Comparison between the number of generations and the population size.



(b) Comparison between the mutation and crossover probability.

Figure 3: Analysis of the impact among the evolutionary parameters when solving the optimization model using a genetic algorithm (with 95% confidence interval). 2D array with size of 30000×30000, neighborhood mask with range 1, and 100 iterations. When not specified, the default values used are: mutation probability of 10%; crossover probability of 60%; and 100 for both the number of generations and the population size. The horizontal red line represents the maximum value of the objective function.

Figure 3.a shows that, as the number of generations increases, the solution accuracy also tends to increase, in absolute terms, with little penalty on the execution time. However, the population size has little effect on the solution accuracy, while presenting a significant penalty on the execution time. Figure 3.b shows that, as the mutation probability increases, the execution time also increases while the solution accuracy tends to decrease, with the solution error reaching up to 3.5% of the optimal value. The crossover probability shows little effect on both the execution time required for solving the optimization problem, and the solution accuracy.

Therefore, in order to keep the optimization overhead reasonably low, we can parameterize the genetic algorithm with mutation and crossover probability as their default values, as lower values are preferable. Since the population size parameter yields to high overhead penalty with no significant improvement in accuracy, its value can be set as the default value, 100. In addition, the number of generations can be much higher, slightly increasing the solution accuracy, and still resulting in an overhead of less than half a second (about 0.45 seconds on average for 2000 generations).

3.3 The TOAST mechanism runtime

We implemented TOAST mechanism into the PSkel stencil framework [26]. Specifically, we extended the framework's internals with the trapezoidal tiling technique described in Section 2 and the automatic optimization solver based on genetic algorithms. Figure 4 presents a diagram of the TOAST mechanism.

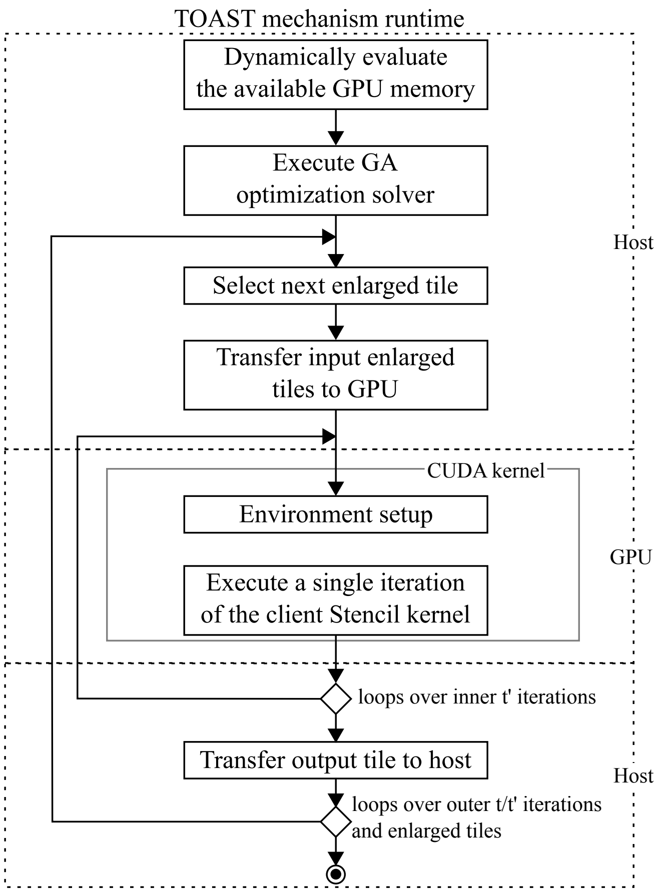

TOAST mechanism runtime
Dynamically evaluate the available GPU memory
Execute GA optimization solver
Host
Select next enlarged tile
Transfer input enlarged tiles to GPU
CUDA kernel
Environment setup
GPU
Execute a single iteration of the client Stencil kernel
loops over inner t' iterations
Transfer output tile to host
Host
loops over outer t/t' iterations and enlarged tiles

Figure 4: A diagram of the TOAST mechanism runtime system.

Because the mechanism is executed during runtime, it is essential to have an optimization solver that adds a reasonably low overhead for finding the optimal tile size and ghost zone size. The genetic algorithm allows for a heuristic that solves the maximization problem in an efficient way.

After we have optimized the tiling parameters, we apply the trapezoidal tiling technique for partitioning the stencil iteration space into overlapped enlarged tiles. The TOAST mechanism, implemented into the stencil framework, favors a partitioning of tiles that are contiguously stored on main memory, as contiguous data are transferred considerably faster between host and device memory [4]. For each enlarged tile, the framework runtime computes each stencil iteration on GPU, with host-side synchronization.

Because the TOAST mechanism is focused on enabling large inputs that otherwise cannot fit completely on the global memory of the GPU, the trapezoidal tiling is performed in such a way that each enlarged tile fills the global memory of the GPU, aiming at maximizing the GPU utilization. For this reason, the TOAST mechanism performs only synchronous communication, with no overlap between communication and computation, the mechanism favors instead the maximization of each enlarged tile. Sending smaller tiles, in order to allow for overlapped communication and computation, would increase the number of tiles and also the number of *outer* tiling iterations.

As illustrated in Code 1, the TOAST mechanism abstracts the optimization from the programmer by transparently and automatically finding the tiling parameters. When the stencil framework dynamically evaluates the available GPU memory, it decides automatically if the data is small enough to completely fit into the available GPU memory or if the TOAST mechanism must be triggered, including the tiling and the automatic optimization. In both cases, the same client stencil kernel is executed, i.e. it has a single application programming interface (API).

4 Experimental Results

4.1 Experimental method

In this section, we describe four iterative stencil applications that we use to evaluate the automatic tiling mechanism implemented in PSkel. The applications cover a wide range of stencil computations, including both 2D and 3D inputs.

GoL (Game of Life). Conway's Game of Life [11] is a cellular automaton that we implement as a matrix in which each cell represents either a living or a dead individual. Over the course of a pre-defined number of iterations (or *generations*), each individual analyzes the state of its neighbors to define its own state in the next iteration. There are four possible neighborhood conditions that define the next state of a cell: (i) a dead cell with exactly three living neighbors becomes alive at the next time step, by *reproduction*; (ii) a living cell with less than two living neighbors dies, due *under-population*; (iii) a living cell with more than three living neighbors dies, due to *overcrowding*; (iv) a living cell with two or three living neighbors or a dead cell with more or less than three neighbors maintains its previous state.

GoL has been used in several studies since its proposal, and it is known to consume significant amounts of computing resources. The GoL computations follow the stencil pattern using the Moore neighborhood mask. In Code 1, we show our implementation of the stencil computations of GoL. For brevity, we do not show the complete code.

```
__parallel__ void stencilKernel(Array2D<int> input,
        Array2D<int> output, Mask2D<int> mask, int i, int j){
  int n=0; // neighbors
  for(int z=0; z<mask.size(); z++)
      n += mask.get(z,input,i,j);
  output(i,j) = (n==3 || (input(i,j)==1 && n==2)) ? 1 : 0;
}
```

Code 1: Stencil kernel for GoL in PSkel.

Laplacian. The Laplace operator (Δ), or Laplacian [5], is used in differential equations that describe many physical phenomena, such as electric potential, heat diffusion, and wave propagation. When applied to a function, the operator denotes the divergence of the function's gradient on Euclidean space (Equation 1), and is equivalent to the sum of the unmixed second partial derivatives at point x_i. Our stencil application implements the second-order finite difference discretization of the Laplacian in 3D space (Equation 2). For each element e in the input matrix, the stencil computes the rate at which the average value of the neighbors deviates e. That computation can be parametrized by the weights α and β.

$$\Delta := \sum_i \frac{\partial^2}{\partial {x_i}^2} \tag{1}$$

$$u'_{i,j,k} = \alpha u_{i,j,k} + \beta(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1}) \tag{2}$$

Jacobi. Jacobi's method [8] is an iterative method for solving matrix equations. The method is iterated until the solution converges. The method is guaranteed to converge if the input matrix is strictly or irreducibly diagonally dominant, i.e., $|u_{i,i}| > \sum_{j \neq i} |u_{i,j}|$, for all i. Equation 3 defines the computation performed at each step of the Jacobi's iterative method for solving 2-dimensional Poisson's elliptical discretized equation. The approximate solution is computed by discretizing the problem in a matrix of $n \times n$ evenly spaced points. Poisson's equation [8] is a partial differential equation of elliptic type largely used in theoretical physics.

$$u'_{i,j} = \frac{u_{i\pm1,j} + u_{i,j\pm1} + h^2 f_{i,j}}{4} \tag{3}$$

At each step, the new value of $u_{i,j}$ is obtained by averaging its neighbors with $h^2 f_{i,j}$, where $h = \frac{1}{n+1}$ and $f_{i,j} = f(ih, jh)$, for a given function f.

CloudSim. Cloud dynamics play a critical role to Earth's climate, general atmospheric circulation, and global water balance, hence being essential to mesoscale meteorology, atmospheric chemistry, and weather forecasting [32]. Clouds are formed from the condensation of water vapors present in the atmosphere. After its formation, a cloud is moved by winds, changing its location and properties, such as temperature, pressure, density, and humidity.

CloudSim is a real stencil application used by a center of climatology. It simulates cloud dynamics based on cellular automaton [6]. CloudSim implements a mathematical model that uses the Von Neumann neighborhood of five cells, each one with two possible states: the presence or absence of a cloud, or a part of a cloud. The model uses three weather properties: the condensed cloud water particles, the temperature, and the winds. The transition rules are based on the thermodynamic principles and weather concepts.

Each cell temperature, $T_{i,j}$, behavior is updated according to the thermodynamic principles which provide a heat transfer among the Von Neumann neighborhood cells. In each iteration k, the whole grid is updated with new cell temperatures as follows

$$T_{i,j}(k+1) = T_{i,j}(k) + \alpha \Delta T(k)$$

where α is a constant that defines the step size of the temperature update, and

$$\Delta T(k) = \frac{T_{i\pm1,j}(k) + T_{i,j\pm1}(k) - 4T_{i,j}(k)}{5}$$

Each cell has a cloud, or part of it, only if the number of condensed cloud water particles is larger than a given threshold. Initially, condensed cloud water particles are randomly inserted into the input grid. The number of condensed cloud water particles in a cell, $W_{i,j}(k)$, at a given iteration k, is defined as a function of its current temperature in the cell, $T_{i,j}(k)$, as follows

$$W_{i,j}(k) = \begin{cases} 0 & \text{if } T_{i,j}(k) > T_C \\ n_{min} + \alpha(T_C - T_{i,j}(k)) & \text{if } T_{i,j}(k) \leq T_C \end{cases}$$

where T_C is the temperature of water condensation at current atmospheric conditions, n_{min} is the minimum number of condensed cloud water particles, and α is the water condensation factor.

Throughout each stencil iteration, the condensed cloud water particles are displaced to a neighborhood in accordance with the wind direction and magnitude. Therefore resulting in a translation or dispersing of the existing clouds.

4.2 Experimental evaluation

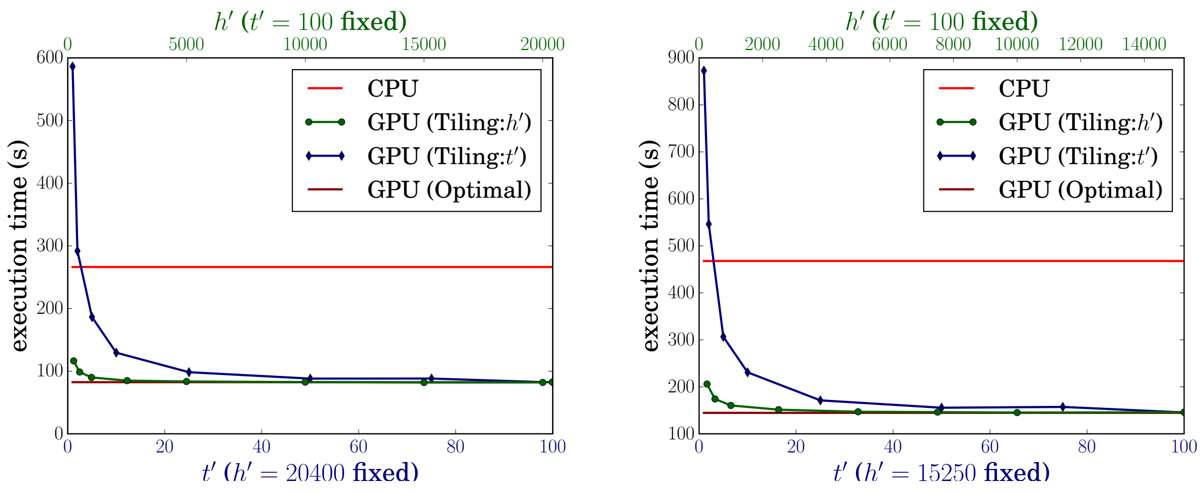
In this section, we present the experimental setup and results. Table 1 presents the settings of the system used throughout the experiments. Each experiment was executed five times and averaged with a confidence interval of 95%. The applications were executed with data sizes that add up to 6.7 GiB and 11.9 GiB, considering the amount of storage required for both input and output arrays.

CPU	
Features	**Platform**
Model	Intel Xeon E5-2620
CPU sockets	2
Cores per socket	6
Threads per core	2
Clock	2.10 GHz
Cache	15360 KiB
GFLOP (peak)	96
Memory size	64 GiB

GPU	
Features	**Platform**
Model	NVIDIA Tesla K20
Threads	2496
Clock	706 MHz
GFLOP (peak)	3520
Memory size	4799 MiB
Memory clock	2600 MHz
Memory bandwidth	208 GiB/s

Table 1: System settings.

Figures 5, 6, 7, and 8 present an experimental analysis of the optimization model in Definition 3. The analysis consists in comparing the optimal tiling given by the optimization model, i.e., the tiling parameters h' and t' that results in the maximum value of the tiling model, against variations of tiling parameters and an optimized CPU execution using all 24 threads available.



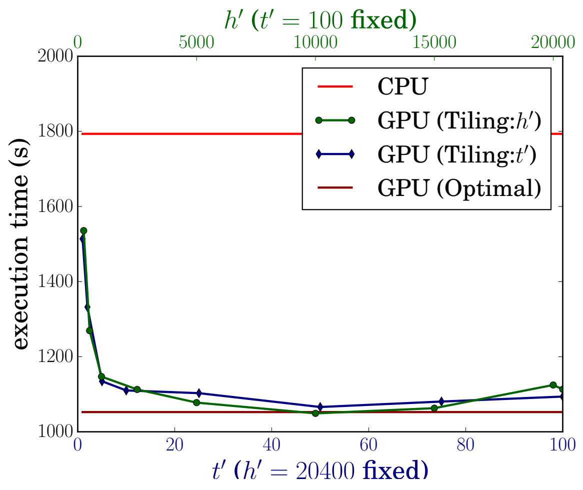
(a) 2D array 30000×30000 (6.7 GiB.)

(b) 2D array 40000×40000 (11.9 GiB.)

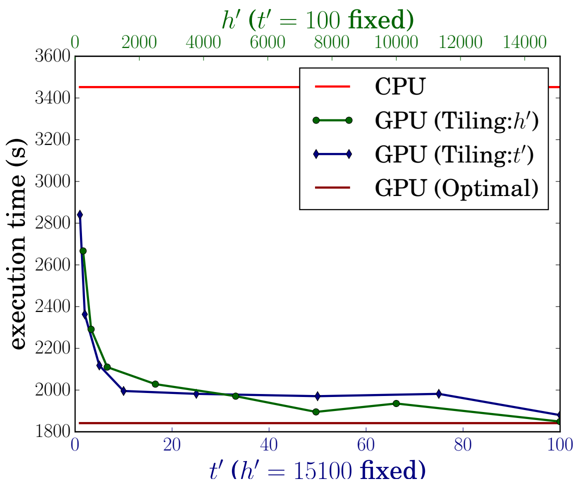
Figure 5: Analysis of the Game of Life application.

As shown in Figure 5, as we increase t' and h', the execution time of GoL in the GPU reduces. A more careful inspection of the generated code revealed that the GoL application involves just integer data manipulation, having 11% of its PTX ISA[2] assembly code (for the stencil kernel) consisting only of branch instructions, while 24% consisting of integer arithmetic instructions (mainly for index manipulation). However, although the GoL application is mostly based on conditional control constructs, for selecting whether a given cell is either alive or dead, inherently leading to *divergence*, it still benefits from the GPU optimized execution, presenting a $3\times$ speedup compared to the optimized CPU execution with 24 threads, as shown in Figure 5.

[2]PTX ISA is a low-level parallel thread execution virtual machine and instruction set architecture offered by NVIDIA GPUs.



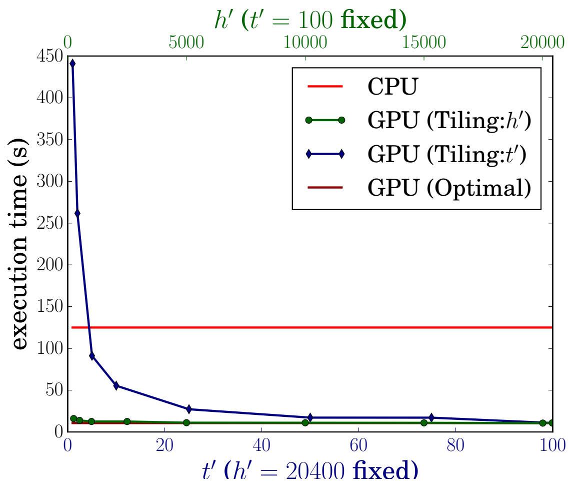
(a) 2D array 30000 × 30000 (6.7 GiB.)



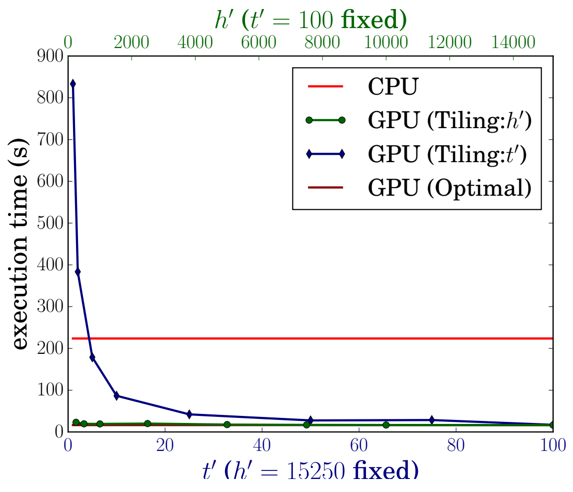
(b) 2D array 40000 × 40000 (11.9 GiB.)

Figure 6: Analysis of the CloudSim application.

CloudSim application also benefits from the GPU implementation, as depicted in Figure 6. Even the most naive tiling approach improves the execution time in 17% compared to the optimized CPU execution, while the optimal tiling improves 3× relative to the same optimized CPU execution time, which represents a difference of 36 minutes. CloudSim is a substantially compute-intensive application, having 42% of its PTX ISA assembly code (for the stencil kernel) consisting only of arithmetic instructions. This application aspect favors the execution on the GPU, which has a processing power peak of 3520 GFLOP compared to only 96 GFLOP from the CPU.



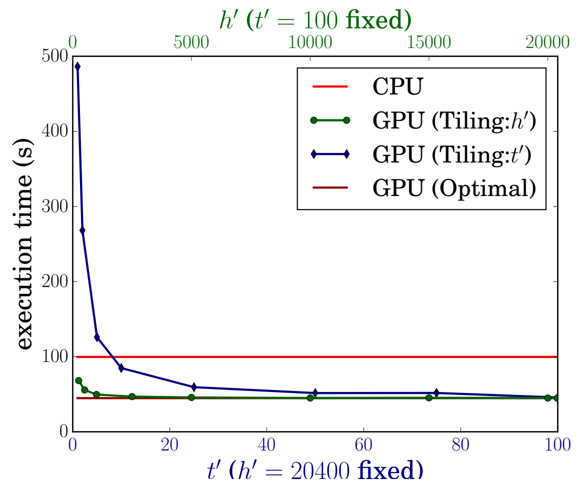
(a) 3D array 200 × 30000 × 150 (6.7 GiB.)



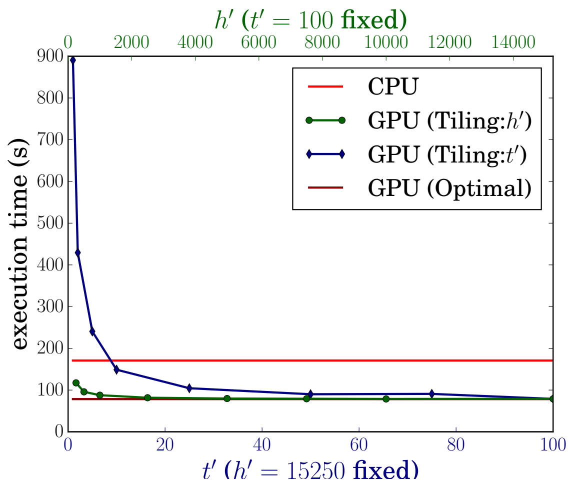
(b) 3D array 200 × 40000 × 200 (11.9 GiB.)

Figure 7: Analysis of the 3D Laplacian application.

Relative to the 3D Laplacian application (see Figure 7), the optimal tiling improves the execution time 13× compared to the optimized CPU execution, and 50× compared to the most naive tiling configuration. The 3D Laplacian is computationally very similar to the Jacobi's method (see Figure 8), where they are the direct implementation of equations 2 and 3, respectively.



(a) 2D array 30000 × 30000 (6.7 GiB.)



(b) 2D array 40000 × 40000 (11.9 GiB.)

Figure 8: Analysis of the Jacobi's method application.

These results show that the applications speedups are proportional to the number of consecutive iterations t' in solutions with optimal or near-optimal tile size. In addition, these results also provide experimental support for the mathematical model proposed in Definition 3, as the optimal tiling parameters produce the best-observed speedups. Notice that poor GPU tiling configuration might lead to poor execution time, possibly even slower than the optimized multi-threaded CPU execution. Thus, instead of leaving the user with the burden of manually selecting the non-trivial tiling parameters, the automatic tiling mechanism is valuable for transparently providing a near-optimal GPU execution.

Figure 9 and Table 2 summarize the analysis of the TOAST mechanism. The execution time for all optimal cases, regarding the optimal tiling configuration, excludes any optimization overhead required for obtaining the tiling parameters. In fact, the evolutionary optimization overhead accounts for most of the difference between the optimial and the TOAST execution time (see Table 2). The average tiling configuration represents average among all tiling configuration executions as illustrated in Figures 5, 6, 7, and 8. In Table 2, the naive tiling configuration represents the worst tiling configuration observed during all the executions. In all those cases, the naive tiling configuration represents the case where a single consecutive iteration is executed on GPU for each tile, i.e., $t' = 1$.

		GPU		
	CPU	Optimal	Average	Naive
GoL	3.04	0.94	1.58	6.68
Jacobi	2.00	0.90	1.91	9.74
Laplacian	11.92	0.93	5.78	44.98
CloudSim	1.70	1.00	1.12	1.46

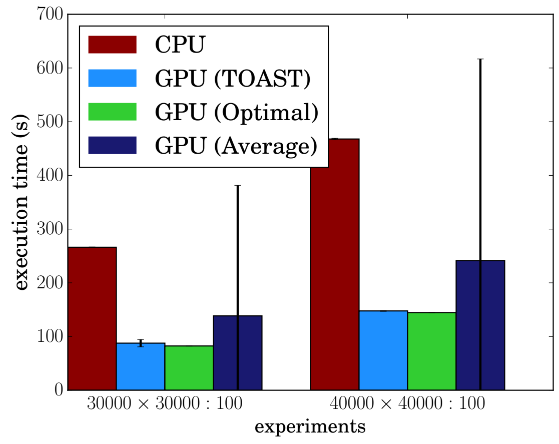
(a) 30000 × 30000 : 100

		GPU		
	CPU	Optimal	Average	Naive
GoL	3.17	0.98	1.63	5.91
Jacobi	2.15	0.98	2.18	11.19
Laplacian	13.04	0.96	6.36	48.57
CloudSim	2.77	0.98	1.76	2.36

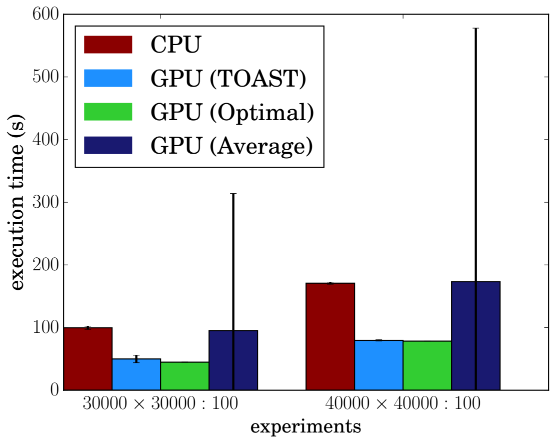
(b) 40000 × 40000 : 100

Table 2: Speedup analysis of the TOAST mechanism running on GPU compared against: (CPU) an optimized CPU execution with 24 threads; (Optimal) an optimal tiling configuration; (Average) an average tiling configuration; and (Naive) a naive tiling configuration running on GPU.

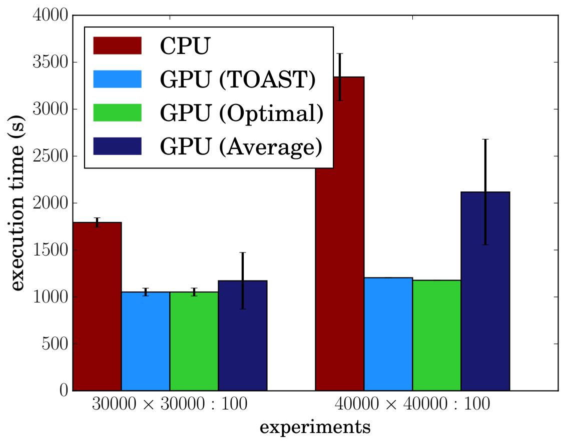
Comparing across multiple tiling configurations, the GPU average highlights the importance for an automatic tiling optimization, as the TOAST mechanism improves in at least 12% compared to the average tiling configuration running on GPU (for the CloudSim application), and up to 6× (for the Laplacian application).



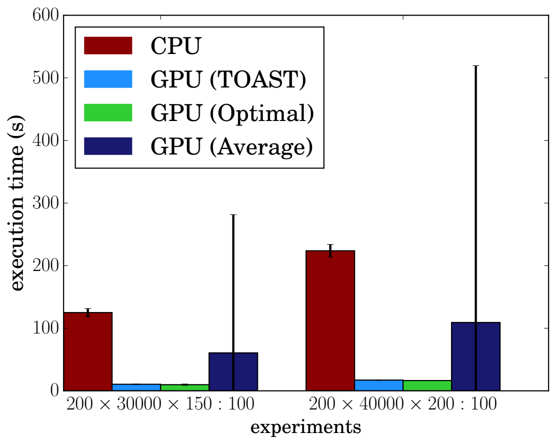
(a) Game of Life.



(b) Jacobi's method.



(c) CloudSim.



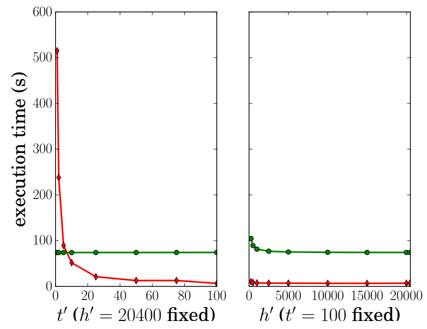
(d) Laplacian.

Figure 9: Analysis of the TOAST mechanism based on the evolutionary autotuning.

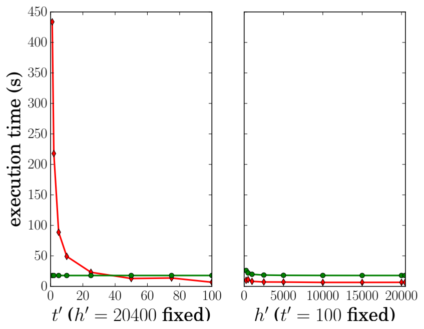
Overhead analysis.

In this section we analyze the impact of managing and transfering data for the TOAST mechanism. For each execution, we measure in isolation the processing time of the actual stencil computation and the time spent with data management, which includes memory allocation, memory copy, and data transfer between host and GPU.

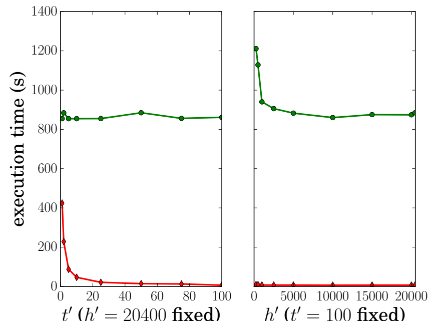
Figures 10 and 11 also illustrate the importance of selecting a good tiling configuration. For low values of t', the parameter for the inner iterations on GPU, the time spent in data management degrades substantially, far exceeding the actual processing time. In the worst case, the time spent in data management can represent up to 99% of the overall execution time (e.g. in the Laplacian application). Similarly, for some computationaly intensive applications, such as the CloudSim application, a poor selection of the h' parameter can also substantially degrade the execution time, highlighting the importance of the trade-off between the cost of redundant computations and the reduction in communication. This trade-off is non-trivial for the average user.



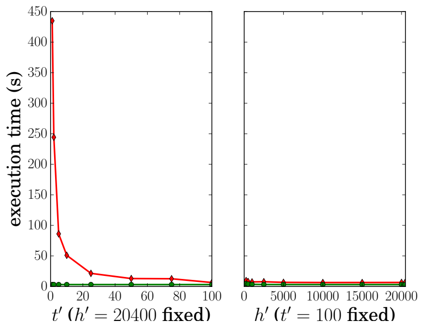
(a) Game of Life (30000 × 30000 : 100).



(b) Jacobi's method (30000 × 30000 : 100).



(c) CloudSim (30000 × 30000 : 100).



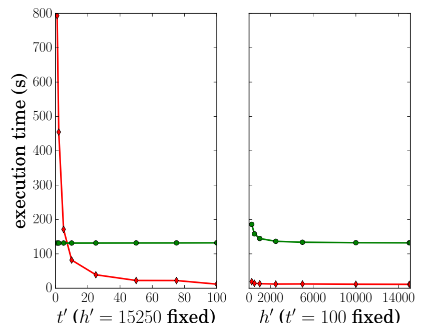
(d) Laplacian (200 × 30000 × 150 : 100).

Figure 10: Overhead analysis of the TOAST mechanism for inputs of size 6.7 GiB. The red line represents data management and the green line represents actual stencil computation.

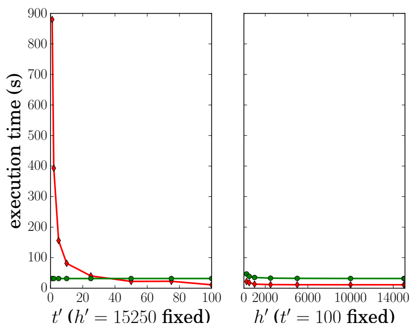
	TOAST		Optimal	
	Data Man.	Computation	Data Man.	Computation
GoL	7.7%	91.0%	8.3%	91.7%
Jacobi	24.8%	70.9%	26.9%	73.1%
Laplacian	58.8%	30.0%	67.5%	32.5%
CloudSim	0.7%	99.2%	0.7%	99.3%

Table 3: Percentual overhead of data management and the percentual of processing time for the actual stencil computation. Data inputs of size 6.7 GiB.

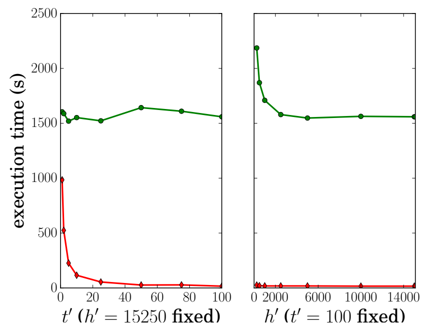
Tables 3 and 4 show the percentual overhead of data management compared to the percentual of processing time for the actual stencil computation. The TOAST mechanism is able to automatically select a good tiling configuration that results in a low overhead of data management. A minimum overhead of data management is unavoidable when processing data larger than the available GPU memory, and the TOAST mechanism is able to reach a good ratio between overhead of data management and useful computation, which is even more significant for computationaly intensive applications.



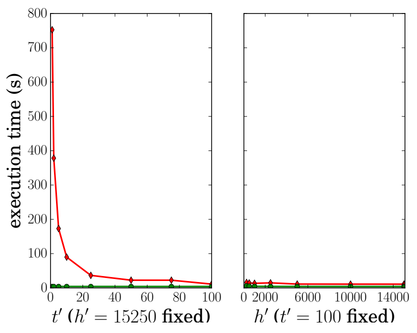
(a) Game of Life ($40000 \times 40000 : 100$).



(b) Jacobi's method ($40000 \times 40000 : 100$).



(c) CloudSim ($40000 \times 40000 : 100$).



(d) Laplacian ($200 \times 40000 \times 200 : 100$).

Figure 11: Overhead analysis of the TOAST mechanism for inputs of size 11.9 GiB. The red line represents data management and the green line represents actual stencil computation.

	TOAST		Optimal	
	Data Man.	Computation	Data Man.	Computation
GoL	8.1%	91.2%	7.9%	92.1%
Jacobi	26.3%	71.6%	26.7%	73.3%
Laplacian	68.6%	25.7%	72.6%	27.4%
CloudSim	0.9%	99.0%	1.0%	99.0%

Table 4: Percentual overhead of data management and the percentual of processing time for the actual stencil computation. Data inputs of size 11.9 GiB.

5 Conclusions and future work

In this paper, we proposed and evaluated TOAST, an automatic and transparent tiling mechanism for stencil computations on GPUs. TOAST transparently offers a virtualized GPU global memory for stencil computations, allowing for input data much larger than the available physical GPU memory. With optimal tiling, TOAST improved the performance of the evaluated stencil applications by $6\times$ on average, comparing across multiple tiling configurations. Compared to a naive tiling approach, TOAST improved performance by up to $48\times$, while performing within 1% of an oracle. Finally, compared to the multi-threaded optimized implementation running on CPU, TOAST improved the performance of iterative stencil applications by up to $13\times$.

The use of tiling for enabling computation on large input data poses an important trade-off between the cost of redundant computations and the reduction in communication. This trade-off is non-trivial for the average user. The TOAST mechanism is able to automatically select a tiling configuration that achieves a low percentual overhead of data management compared to actual stencil computation. This autotuning mechanism abstracts the tiling technique, releasing the programmer from the responsibility of directly manipulating the tiling computation.

Although the evolutionary autotuning mechanism requires an initial overhead for the optimization phase, it is able to select a tiling configuration that results in near optimal execution time. The mathematical model used by the evolutionary autotuning mechanism can be improved to incorporate other aspects of the application or the computing environment, such as energy consumption, the computational intensity of the stencil function, non-contiguous tiling if necessary, etc.

Finally, although in this paper we have used our model to improve stencil tiling on GPUs, the model can also be used to perform tiling on the CPU, for example, by using the size of the last-level cache as the target size parameter. In addition, in a framework like PSkel, which is able to distribute computing across CPU and GPU, our model and tiling mechanism can be used to boost overall performance even further.

References

[1] Thomas Back and H-P Schwefel. Evolutionary computation: An overview. In *IEEE CEC*, pages 20–29, 1996.

[2] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *ACM/IEEE SC*, pages 1–11, 2012.

[3] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *ACM ICS*, pages 197–206, 2015.

[4] B. Betkaoui, D.B. Thomas, and W. Luk. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *IEEE ICFPT*, pages 94–101, 2010.

[5] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IEEE IPDPS*, pages 676–687, 2011.

[6] Alisson Rodrigo da Silva and Maury Meirelles Gouvêa, Jr. Cloud Dynamics Simulation with Cellular Automata. In *SCSC*, pages 278–283, 2010.

[7] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures. In *ACM/IEEE SC*, 2008.

[8] James W Demmel. *Applied numerical linear algebra*. SIAM, 1997.

[9] Michael Freitag. Using a dynamic schedule to increase the performance of tiling in stencil computations. In *IEEE GSC*, pages 45–48, 2014.

[10] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *ACM ICS*, pages 361–366, 2005.

[11] Martin Gardner. Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Scientific American*, 223(3):120–123, 1970.

[12] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *IEEE/ACM CGO*, pages 66–75, 2014.

[13] T. Gysi, T. Grosser, and T. Hoefler. Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *ACM ICS*, pages 177–186, 2015.

[14] Dongni Han, Shixiong Xu, Li Chen, and Lei Huang. PADS: A Pattern-Driven Stencil Compiler-Based Tool for Reuse of Optimizations on GPGPUs. In *IEEE ICPADS*, pages 308–315, 2011.

[15] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A Stencil Compiler for Short-vector SIMD Architectures. In *ACM ICS*, pages 13–24, 2013.

[16] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ACM ICS*, pages 311–320, 2012.

[17] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *IEEE IPDPS*, pages 1–12, 2010.

[18] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM PLDI*, pages 235–244, 2007.

[19] Zhiyuan Li and Yonghong Song. Automatic Tiling of Iterative Stencil Loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, 2004.

[20] Thibaut Lutz, Christian Fensch, and Murray Cole. PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, 2013.

[21] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. Autotuning Stencil-Based Computations on GPUs. In *IEEE CLUSTER*, pages 266–274, 2012.

[22] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *ACM/IEEE SC*, pages 11:1–11:12, 2011.

[23] Michael D. McCool. Structured Parallel Programming With Deterministic Patterns. In *USENIX HotPar*, page 5, 2010.

[24] Jiayuan Meng and Kevin Skadron. A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[25] Daniel Orozco, Elkin Garcia, and Guang Gao. Locality Optimization of Stencil Applications Using Data Dependency Graphs. In *LCPC*, pages 77–91, 2011.

[26] Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. PSkel: A Stencil Programming Framework for CPU-GPU Systems. *Concurrency and Computation: Practice and Experience*, 2015.

[27] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. In *ISCA*, pages 24–35, 2013.

[28] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding Stencil Code Performance on Multicore Architectures. In *ACM CF*, pages 30:1–30:10, 2011.

[29] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. Towards Optimal Multi-level Tiling for Stencil Computations. In *IEEE IPDPS*, pages 1–10, 2007.

[30] Edward Rosten and Tom Drummond. Fusing Points and Lines for High Performance Tracking. In *IEEE ICCV*, pages 1508–1515, 2005.

[31] Edward Rosten and Tom Drummond. Machine Learning for High-Speed Corner Detection. In *ECCV*, pages 430–443, 2006.

[32] Hilding Sundqvist, Erik Berge, and Jón Egill Kristjánsson. Condensation and cloud parameterization studies with a mesoscale numerical weather prediction model. *AMS Monthly Weather Review*, 117(8):1641–1657, 1989.

[33] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, 2013.

[34] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical Overlapped Tiling. In *ACM CGO*, pages 207–218, 2012.