Contents lists available at ScienceDirect

Future Generation Computer Systems

Efficient GPU thread mapping on embedded 2D fractals

Cristobál A. Navarro [a],[*], Felipe A. Quezada [a], Nancy Hitschfeld [b], Raimundo Vega [a], Benjamin Bustos [c]

[a] *Instituto de Informática, Universidad Austral de Chile, Chile*
[b] *Departamento de Ciencias de la Computación, Universidad de Chile, Chile*
[c] *Millenium Institute Foundational Research on Data, Department of Computer Science, University of Chile, Chile*

ARTICLE INFO

ABSTRACT

This work proposes a new approach for mapping GPU threads onto a family of discrete embedded 2D fractals. A block-space map $\lambda : \mathbb{Z}_\mathbb{E}^2 \mapsto \mathbb{Z}_\mathbb{F}^2$ is proposed, from Euclidean parallel space $\mathbb{E}$ to embedded fractal space $\mathbb{F}$, that maps in $\mathcal{O}(\log_2 \log_2(n))$ time and uses no more than $\mathcal{O}(n^\mathbb{H})$ threads with $\mathbb{H}$ being the Hausdorff dimension of the fractal, making it parallel space efficient. When compared to a bounding-box (BB) approach, $\lambda(\omega)$ offers a sub-exponential improvement in parallel space and a monotonically increasing speedup $n \geq n_0$. The Sierpinski gasket fractal is used as a particular case study and the experimental performance results show that $\lambda(\omega)$ reaches up to $9\times$ of speedup over the bounding-box approach. A tensor-core based implementation of $\lambda(\omega)$ is also proposed for modern GPUs, providing up to $\sim 40\%$ of extra performance. The results obtained in this work show that doing efficient GPU thread mapping on fractal domains can significantly improve the performance of several applications that work with this type of geometry.

1. Introduction

Fractals can be described as self-similar structures [1] where a *similar* [1] geometrical pattern is found at all scales. Several natural phenomena produce fractal patterns that obey a self-similar structure [2], such as plant and tree growth [3,4], terrain formation [5,6], molecular dynamics [7], snowflake crystallization [8], blood vessels [9], morphological features of living organisms [10], among many others, display a fractal design where self-similarity is a relevant feature for modeling its geometrical structure. Computer applications related to these fields may choose to embed the 2D fractal into a discrete Euclidean domain which acts as a bounding-box in memory. Using an embedding space for a fractal helps in achieving an efficient simulation in terms of memory access patterns (*i.e.*, high memory bandwidth), as data-parallel computations (for example, computation of nearest neighbors) can perform aligned memory accesses and exploit the spatial locality within the fractal structure. In other words, memory locations $(x \pm 1, y \pm 1)$ define a neighborhood in both the embedded space and the actual fractal as well (some of the elements of this neighborhood belong to the fractal domain). Fig. 1 illustrates as

an example the H-fractal embedded in a space of $n \times n$ with n being its side length.

Several of the processes performed on a fractal are combinations of *map* and *reduce* operations in different proportions. The *map* component relates to the *per-element* computations while the *reduce* component relates to global results being computed from the parts of the problem (e.g. a global measure from all discrete elements of the fractal). One typical simulation pattern that is a combination of a full *map* with multiple local *reductions* is the tiled nearest-neighbors computation, which operates on all data elements of a fractal and for each one considers a small reduction from its neighborhood data. Such pattern is found in cellular automata transition functions, spin lattice Monte Carlo simulation steps and finite difference method (FDM) time-step computations, among others. Another computational pattern frequently used is the computation of a global measure of the fractal at a given state, which would involve a full *reduction* of the values of all data elements of the fractal passed through a mathematical expression. This second pattern can be found when computing macroscopic measures from microscopic definitions, such as in Spin Lattice models or n-body simulations. Eventually, when the fractal is large enough to the point of containing millions of data elements, a sequential computation can take an excessive amount of time for the practical requirements of a certain field, especially if there are real-time requirements. In these situations GPU computing becomes an attractive tool for accelerating these tasks [11].
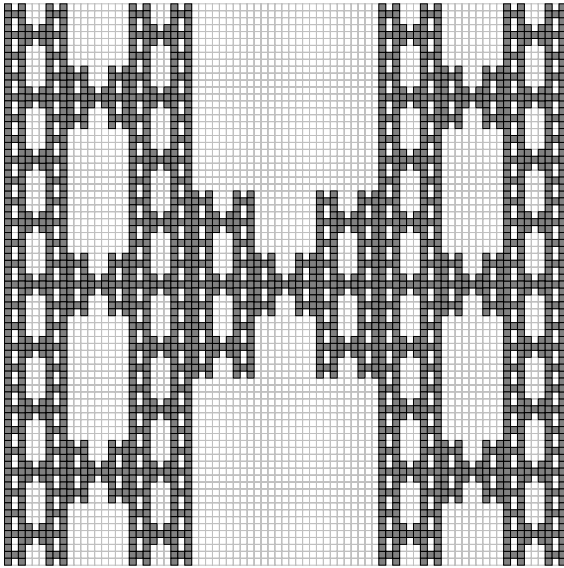

Fig. 1. A H-fractal embedded in a discrete $n \times n$ Euclidean space.

GPU computing has become an important tool for leveraging the performance of several compute demanding applications that contain data-parallel workloads [11]. The two main motivations to use GPU Computing are the (1) high TFLOPS and memory bandwidth, which can be up to an order of magnitude faster than traditional CPU hardware, and (2) the high energy efficiency with respect to traditional CPU based systems. It is important to mention however that exploiting these two features efficiently requires a dedicated algorithm design and implementation as GPUs are more restricted than CPUs in terms of control logic (scheduling, branch prediction, prefetching) and memory access patterns (cache, global memory access). One aspect of GPUs that has been recently studied is achieving efficient GPU thread mapping, an optimization technique that can minimize the number of necessary threads when working with non-trivial data domains. In the GPU programming model, for every GPU computation there is a stage in the pipeline where threads are mapped from parallel work space to data space. A map, defined as $f : \mathbb{Z}^k \to \mathbb{Z}^m$, transforms each k-dimensional point $x = (x_1, x_2, \ldots, x_k)$ in parallel space P^k into a unique m-dimensional point $f(x) = (y_1, y_2, \ldots, y_m)$ in data space D^m. This work uses the notation introduced by previous GPU thread mapping works [12–14], which defines GPU parallel spaces as orthotopes[2] $\Pi^k \in P^k$ in $k = 1, 2, 3$ dimensions. In this work the orthotope of interest is the two-dimensional one, Π^2. A known way of mapping threads to any data-domain is to use the *bounding-box* approach, that builds an orthotope Π^2 sufficiently large to cover the corresponding bounding-box of the data space and threads are mapped using the identity $f(\omega) = \omega$. Such a map is highly convenient and efficient for the class of problems where data space is also defined by an orthotope; such as vectors (Π), tables (Π^2), matrices (Π^2) and box-shaped volumes (Π^3). However, for a discrete embedded 2D fractal, this approach is no longer efficient in terms of parallel space as many threads would fall inside the embedded space but outside the fractal domain, introducing a performance penalty to the execution time when discarding these threads at run-time. For such cases, an efficient orthotope would be one with asymptotically the same number of threads as data elements of

the fractal. Fig. 2 illustrates the unwanted (left to center) and wanted scenarios (right to center) for the case of the Vicsek fractal.

Two research questions arise from this GPU efficiency problem; the first question: *Is there any parallel-space efficient function, namely $\lambda(\omega)$, that can map threads only on the data elements of an embedded 2D fractal?* The second question relates to performance: *Will the parallel-space improvement translate into a significant GPU performance improvement?*

The present work presents theoretical and experimental results that answer these two questions positively. A dedicated analysis is devoted to show that an alternating unrolling strategy allows to define a parallel-space efficient $\lambda(\omega)$ that only requires $\mathcal{O}(n^{\mathbb{H}})$ threads, with $\mathbb{H}$ being the Hausdorff dimension of the fractal. In terms of performance, by taking advantage of intra-block parallelism, $\lambda(\omega)$ becomes computable in $\mathcal{O}(\log_2 \log_2(n))$ time which is fast enough to produce a monotonically increasing speedup with respect to a bounding-box approach, once $n > n_0$ with n_0 being a threshold size different for each fractal. In addition to these results, the $\lambda(\omega)$ map is also adapted to GPU tensor core computation, further increasing its performance by up to 40%. This last result serves as an evidence that GPU tensor cores may be utilized in more ways than what they were initially thought for (deep learning).

This work is an extension and generalization of a previous conference work [13] where preliminary results were presented for a specific fractal. This work generalizes the map for a family of embedded 2D fractals and includes an extensive presentation of experimental results with relevant tests, as well as a new section in which the proposed mapping is further accelerated by encoding the expression into tensor core operations. The rest of the manuscript is organized as follows: Section 2 presents related work on the field, Section 3 characterizes the NBB fractals family, Section 4 formulates the map with new theoretical results, Section 5 describes several approaches for intra-block mapping, Section 6 offers the case study of the Sierpinski gasket with experimental performance measurements and Section 7 concludes the work highlighting the main results, discussing them and describing possible further work within this line of research.

2. Related work

Jung et al. [15] explored the possibilities of improving the GPU mapping on triangular domains, by proposing packed data structures that represent triangular and symmetric matrices with applications to LU and Cholesky decomposition. Their strategy is based on building a *rectangular box* for accessing and storing a triangular matrix (upper or lower). Data structures become practically half the size with respect to classical methods based on the full matrix. The strategy was originally intended for saving memory (*i.e.,* the matrix memory usage), however one can apply the concept analogously to save parallel space.

Ries et al. contributed with a parallel GPU method for the triangular matrix inversion [16]. The authors identified that the parallel space indeed can be improved by using a *recursive partition* of the grid,[3] based on a *divide and conquer* strategy. The mapping approach takes $O(\log_2(n))$ with n being the side of a square matrix.

Navarro, Hitschfeld and Bustos have proposed a block-space map function for 2-simplices[4] and 3-simplices [12,14,17], based

---

[2] A k-orthotope is the generalization of the notion of a rectangle or box, for k dimensions.

[3] A grid is a collection of thread-blocks which are spatially organized and execute asynchronously one from another.

[4] A k-simplex is the generalization of the notion of a triangle to k-dimensions. A 2-simplex corresponds to the triangle while a 3-simplex corresponds to a tetrahedron.
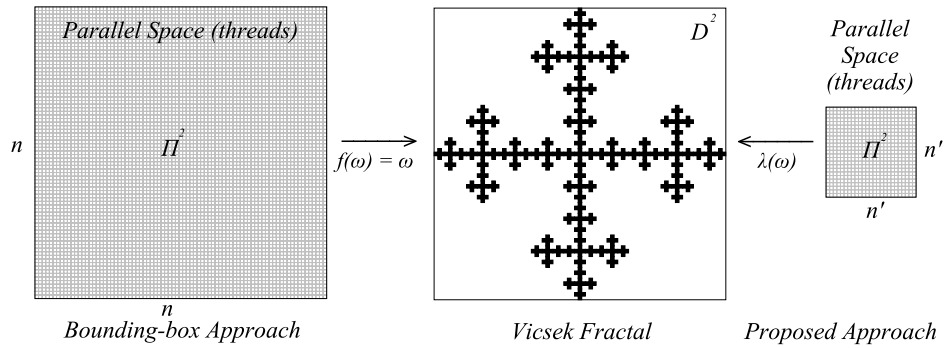
Fig. 2. GPU thread mapping for the Vicsek fractal. The left-to-center mapping illustrates the bounding-box approach and its cost in thread resources, while the right-to-center mapping shows the proposed approach, using significantly less thread resources, but a more elaborate map, namely $\lambda(\omega)$.

on the solution of an m order equation that is formulated from the linear enumeration of the discrete elements. The authors report performance improvement for 2-simplices, and for the 3-simplex case, the mapping technique is extended to the discrete orthogonal tetrahedron, where the parallel space usage can be $6\times$ more efficient. However the authors clarify that it is difficult to translate such space improvement into performance improvement, as the map requires the computation of several square and cubic roots that introduce a significant amount of overhead to the process. From the point of view of data-reorganization, a succinct blocked approach can be combined along with the block-space thread map, producing additional performance benefits with a sacrifice of $o(n^3)$ extra memory.

Exploring the benefits of efficient GPU mapping onto embedded 2D fractals is a relevant yet unexplored topic of research, as its geometry is no longer Euclidean as in the related works. Finding a proper efficient $\lambda(\omega)$ would produce an asymptotic improvement in parallel space and a potential performance improvement that could eventually be exploited.

3. Characterizing discrete embedded 2D fractals

This Section characterizes embedded 2D fractals and defines two Lemmas regarding dimension and space packing, which provide useful insights to formulate an efficient GPU thread map for a family of embedded 2D fractals that share the same construction principle. Also, a block-space mapping strategy is proposed to further improve on the number of map computations performed.

3.1. The Non-overlapping Bottom-up Boxes (NBB) family

Embedded 2D Fractals are discrete non-Euclidean structures that live in $\mathbb{Z}^2$ and are contained inside a tight Euclidean embedding space, *i.e.*, a 2D bounding box. By being discrete structures, these fractals have a lower-bound when scaling down, *i.e.*, a unit of space, but can scale-up infinitely. Because of this, discrete embedded 2D fractals are best described using a bottom-up approach rather than a top-down one. The bottom-up approach consists of defining the fractal as replications of itself from the previous level of scale, with different translations to each replica. An additional restriction is introduced to this type of fractals, which is that the bounding boxes[5] of the replicas cannot overlap in space, that is, a location in the embedding space cannot be occupied by more than one replica. The rest of the manuscript will refer to this kind of fractals as *Non-overlapping Bottom-up Boxes* fractals, or NBB fractals.

Table 1
Example fractals of the NBB family.

Fractal Name	Illustration	NBB Step	Hausdorff Dimension ($\mathcal{H} = \frac{\log(k)}{\log(s)}$)
Sierpinski Gasket			$\frac{\log(3)}{\log(2)} \approx 1.58$
Chandelier (Custom)			$\frac{\log(4)}{\log(3)} \approx 1.26$
H-Fractal			$\frac{\log(7)}{\log(3)} \approx 1.77$
Candy (Custom)			$\frac{\log(12)}{\log(4)} \approx 1.79$
Sierpinski Carpet			$\frac{\log(8)}{\log(3)} \approx 1.89$
X-Fractal (Custom)			$\frac{\log(5)}{\log(3)} \approx 1.46$
Vicsek Fractal			$\frac{\log(5)}{\log(3)} \approx 1.46$
Empty-Bottles (Custom)			$\frac{\log(7)}{\log(3)} \approx 1.77$
Cantor set			$\frac{\log(2)}{\log(3)} \approx 0.63$

Several fractals can be built following this scheme, including the Sierpinski Gasket, Cantor set, Vicsek fractal, H-Fractal, among others. Table 1 presents a list with examples of NBB fractals with their bottom-up building step and their Hausdorff dimension as well.

The notation $\mathcal{F}_n^{k,s}$ is introduced to denote an embedded 2D fractal of the NBB family, where $n \in \mathbb{N}$ is its linear size in one axis, $k \in \mathbb{N}$ the number of self-similar replicas for the next recursive step and $s \in \mathbb{N}$ the scale-up factor between a given scale level and the upcoming one, along each dimension[6] (for example, for the

---

[5] Not to be confused with the bounding-box approach used in GPU computing, which refers to a programming mode that uses the identity $f(x) = x$ to map from parallel space to data space.

[6] It is important to mention that these definitions work for irregular fractals as well, as in the case of the Chandelier fractal, where it can have two definitions; $\mathcal{F}_{n_x}^{4,3}$ or $\mathcal{F}_{n_y}^{4,2}$ along the x and y axis, respectively.

H-fractal $k = 7, s = 3$ and for the Candy fractal $k = 12, s = 4$). The space used by a fractal, denoted as $\mathcal{V}(\mathcal{F}_n^{k,s})$ may be expressed recursively as

$$\mathcal{V}(\mathcal{F}_n^{k,s}) = \sum_{i=1}^{k} \mathcal{V}_i(\mathcal{F}_{sn}^{k,s}) \tag{1}$$

with $\mathcal{V}(\mathcal{F}_1^{k,s}) = 1$ being the limit condition of the recursion. Since k is fixed, and n scales up by factors of s, the volume may expressed as

$$\mathcal{V}(\mathcal{F}_n^{k,s}) = k^r \tag{2}$$

where $r = \log_s(n)$ is defined as the scale level.

3.2. Dimension and packing of NBB fractals

The following Lemma guarantees that the dimensionality of NBB fractals, in their $\mathcal{V}(\mathcal{F}_n^{k,s}) = k^r$ form, is actually fractal.

Lemma 1. *The space occupied by an NBB fractal is in correspondence with its Hausdorff dimension in the scale-up limit.*

Proof. The space occupied by an NBB fractal is $\mathcal{V}(\mathcal{F}_n^{k,s}) = k^r$. Given that $r = \log_s(n)$ and $k^{\log_s(n)} = (s)^{\log_s(k)\log_s(n)}$, the space expression can be rearranged into

$$\mathcal{V}(\mathcal{F}_n^{k,s}) = n^{\log_s(k) = \frac{\log(k)}{\log(s)}} = n^{\mathcal{H}} \tag{3}$$

where the exponent $\mathcal{H}$ is the Hausdorff dimension, *i.e.*, the quotient of the logarithm of the number of replicas and the logarithm of the scaling factor. $\square$

Lemma 1 guarantees that discrete embedded 2D fractals, which have a lower bound in scale and can only grow by scaling up, still exhibit their Hausdorff dimension when $n \mapsto \infty$. Another useful fact is that by having one GPU thread per unitary element of the fractal is already resource-efficient as it would yield a fractal space occupancy in the parallel space as well. The next Lemma relates the geometries of parallel-spaces with fractal domains.

Lemma 2. *A NBB fractal $\mathcal{F}_n^{k,s}$ can pack into a 2-orthotope Π^2 of dimensions $k^{\lceil \frac{r}{2} \rceil} \times k^{\lfloor \frac{r}{2} \rfloor}$ at any scale level r.*

Proof. By induction on r:

- Base case: At scale $r = 0$ the fractal has a space of $\mathcal{V}(\mathcal{F}_n^{k,s}) = 1$ element that packs into a regular 2-orthotope of $1 \times 1 = k^{\lceil \frac{0}{2} \rceil} \times k^{\lfloor \frac{0}{2} \rfloor}$ satisfying $k^{\lceil \frac{r}{2} \rceil} \times k^{\lfloor \frac{r}{2} \rfloor}$.
- Induction step: It is assumed that the orthotope at scale level r is quasi-regular or regular. If r is even, the packing for $r + 1$ will scale by k in the horizontal dimension of the 2-orthotope. If r is odd, the packing for $r + 1$ will scale by k in the vertical dimension of Π^2. Since even and odd must alternate, the dimensions of the packed 2-orthotope Π^2 for $r + 1$ can only be $k \cdot k^{\lceil \frac{r}{2} \rceil} \times k^{\lfloor \frac{r}{2} \rfloor}$ for even r, or $k^{\lceil \frac{r}{2} \rceil} \times k \cdot k^{\lfloor \frac{r}{2} \rfloor}$ for odd r, which is quasi-regular or regular, respectively. $\square$

In GPU Computing the parallel space (grid, blocks, threads) can only be defined as Euclidean boxes in 1D, 2D or 3D, which becomes a constraint when processing NBB fractals. In this context, Lemma 2 gives useful insights on how one could map the Euclidean space onto the Fractal one.

3.3. Changing from thread-space to block-space

An important aspect to consider is at which level the parallel space will be mapped. Two approaches are possible; (1) thread-space mapping and (2) block-space mapping. Let $\lambda(\omega)$ be the map from GPU parallel-space (Euclidean) to embedded 2D fractal space. For the first approach, $\lambda(\omega)$ defines ω as a unique thread location in parallel space. For the second approach, $\lambda(\omega)$ defines ω as a block coordinate in which several threads are contained. The block-space approach has three important advantages over thread-space mapping. First, in block-space, the fractal becomes a coarsened version of the original, requiring fewer elements to be mapped. Second, since the fractal is a simplified version of itself, it is possible to work on higher sizes of n before the CUDA grid maximum dimensions or numerical limits are reached. Third, the block-space approach allows the possibility for threads inside a block to preserve locality, which is essential for doing efficient coalesced memory accesses on GPU global memory.

The next Section formulates $\lambda(\omega)$ with $\omega = (\omega_x, \omega_y)$ being the two-dimensional block coordinate of constant size $|B| = \rho \times \rho$ threads. The change from thread-space to block-space means that blocks are mapped to a simplified version of the fractal of linear size $n_b = n/b$ with $b = \rho$.

4. Formulation of GPU map $\lambda(\omega)$

The function $\lambda : \mathbb{Z}_{\mathbb{E}}^2 \mapsto \mathbb{Z}_{\mathbb{F}}^2$ is introduced as a mapping of block coordinates ω from GPU parallel-space Π^2, which lies in Euclidean space $\mathbb{Z}_{\mathbb{E}}$, onto block coordinates in the embedded fractal space $\mathbb{Z}_{\mathbb{F}}$. The intuition behind the formulation of $\lambda(\omega)$ is an unrolling process applied in parallel to each $\omega \in \Pi^2$ through all the scale levels of the fractal in the scale-down direction until the unit scale limit is reached. At each scale level, different Δ_x, Δ_y offsets are accumulated to form the final $(\lambda_x(\omega), \lambda_y(\omega))$ coordinate in the embedded domain of the fractal.

Theorem 1. *There exists $\lambda(\omega)$ that maps a GPU parallel-space of size $|\Pi^2| = \mathcal{O}(n^{\mathcal{H}})$ to any NBB fractal in $\mathcal{O}(\log_2 \log_2(n_b))$ time using $|B| = \theta(\frac{\log_2(n_b)}{\log_2 \log_2(n_b)})$ threads per block.*

Proof. *By construction*: let $r_b = \log_s(n_b)$ be the block-space scale level of the fractal, Π^2 the 2-orthotope of $k^{\lceil \frac{r_b}{2} \rceil} \times k^{\lfloor \frac{r_b}{2} \rfloor}$ blocks that maps onto the discrete embedded 2D fractal $\mathcal{F}_{n_b}^{k,s}$, with each block having $b \times b$ threads. By Lemma 1, Π^2 is parallel-space efficient in block-space, *i.e.*, $|\Pi^2| = \mathcal{O}(n^{\mathcal{H}})$. A helper index function $\beta_\mu(\omega)$ is defined as

$$\beta_\mu(\omega) = \left(\frac{\omega_x(\mu \mod 2) + \omega_y((\mu + 1) \mod 2)}{k^{\lceil \frac{\mu}{2} \rceil - 1}} \right) \mod k \tag{4}$$

to generate indices in the range $\beta_\mu(\omega) \in [0, k-1]$ that identifies, within scale level $\mu \in [0..r_b]$, which of the k regions of the fractal does block ω belong to. For even μ, $\beta_\mu(\omega)$ acts on ω_x. For odd μ, it acts on ω_y.

An arbitrary numbering is chosen for associating the k blocks of the fractal's NBB step with the k different $\beta_\mu(\omega)$ values.[7] A perfect hash table[8] $H[\ ]$ of size k can be used to map the k values of $\beta_\mu(\omega)$ to (τ_x^μ, τ_y^μ) replica offsets of the form

$$\tau^\mu = H[\beta_\mu(\omega)] = (\tau_x^\mu, \tau_y^\mu), \quad \tau_x^u, \tau_y^u \in [0..s-1]. \tag{5}$$

---

[7] For example, for the Vicsek fractal, where $k = 4$, the regions can follow a numbering of the form top (0), bottom (1), left (2) and right (3). For the Sierpinski gasket, where $k = 3$, a valid numbering could be top (0), bottom (1) and right (2). In the same way, the Candy fractal would require a numbering for its $k = 12$ replicas.

[8] The hash table may be replaced by an arithmetic expression, yielding the same values.

The replica offsets combined with the corresponding fractal replica side length $k^{\mu-1}$, gives the corresponding offset in embedded space

$$\Delta^\mu = (\tau_x^\mu(s)^{\mu-1}, \tau_y^\mu(s)^{\mu-1}) = (\Delta_x^\mu, \Delta_y^\mu) \tag{6}$$

that contributes to the final mapped coordinate. The summation of all partial coordinates produces the map

$$\lambda(\omega) = (\lambda_x(\omega), \lambda_y(\omega)), \tag{7}$$

$$\lambda_x(\omega) = \sum_{\mu=1}^{\log_s(n_b)} \Delta_x^\mu \tag{8}$$

$$\lambda_y(\omega) = \sum_{\mu=1}^{\log_s(n_b)} \Delta_y^\mu \tag{9}$$

which can be computed in $\mathcal{O}(\log_2 \log_2(n))$ time (*i.e.*, $n_b \in \theta(n)$) using a parallel reduction with the threads contained in the ω block. Finally, by Brent's Theorem [18], $|B| = \theta\left(\frac{\log_2(n)}{\log_2 \log_2(n)}\right)$ threads are sufficient for a block of threads to reduce efficiently in parallel. $\square$

Theorem 1 guarantees the existence of an efficient $\lambda(\omega)$ map for any NBB fractal. It is important to mention that the hash table is of fixed size k and the same table is reused at every scale level. In practice this hash table may be defined at compile time as a static resource, or as a GPU shared memory constant array for a whole block of threads. For some fractals it is possible to replace the hash table for an arithmetic hash function that returns the replica offsets directly.

Theorem 2. *Processing a NBB fractal with $\lambda(\omega)$ requires asymptotically less work than using a bounding-box approach.*

Proof. The asymptotic work improvement factor of $\lambda(\omega)$ with respect to the bounding-box approach is the quotient of the costs of mapping all blocks using their corresponding Π^2 structures with the consideration $n_b \in \theta(n)$

$$S_{\lambda(\omega)} = \frac{\mathcal{O}(1)\mathcal{V}(\Pi_{BB}^2)}{\mathcal{O}(\log_2 \log_2(n))\mathcal{V}(\Pi_{\lambda(\omega)}^2)} \tag{10}$$

where Π_{BB}^2 and $\Pi_{\lambda(\omega)}^2$ are the parallel-spaces for the bounding-box and $\lambda(\omega)$ approaches, respectively. The parallel-space of Π_{BB}^2 corresponds to the Euclidean box of $n_b \times n_b$ blocks, and the parallel-space of $\Pi_{\lambda(\omega)}^2$ is $\mathcal{O}(n^\mathcal{H})$ by Lemma 1. Applying the limit $n \to \infty$ gives

$$\lim_{n\to\infty} S_{\lambda(\omega)} = \lim_{n\to\infty} \frac{\frac{\partial}{\partial n}(n^{2-\mathcal{H}})}{\frac{\partial}{\partial n}(\log_2 \log_2(n))} \tag{11}$$

$$= \lim_{n\to\infty} \frac{(2-\mathcal{H})n^{1-\mathcal{H}}}{\frac{1}{n\log_2(n)}} = \infty \quad \square \tag{12}$$

The importance of Theorem 2 is that it guarantees the existence of a fractal size $n > n_0$ where the speedup provided by $\lambda(\omega)$ behaves as a monotonically increasing function. The smaller the Hausdorff dimension of the fractal, the stronger the behavior. The next Section covers the possible approaches to handle intra-block mapping, *i.e.*, how threads inside a block can access individual fractal locations reached by the block-space mapping.

5. Intra-block mapping

Once $\lambda(\omega)$ maps a block ω, all of its threads contained share the same block-space mapped coordinate in embedded space which serves as a reference location for each thread to compute
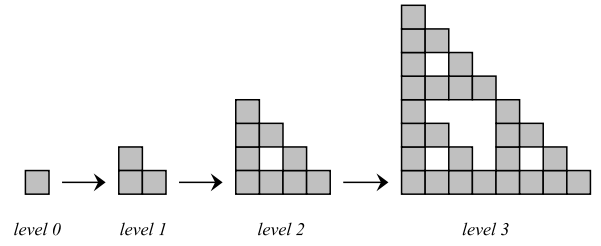

Fig. 3. Bottom-up construction of the discrete Sierpinski gasket.

their individual location in the fractal. This phase of organizing the threads within a block is defined here as *Intra-Block Mapping*, and this Section describes three possible approaches to accomplish it.

5.1. Further unrolling

In this approach threads inside their mapped block may use the same $\lambda(\omega)$, with the same hash table or arithmetic hash function, but this time applied to each thread in local space. By Theorem 1, the *Intra-block map* is still parallel-space efficient and the mapping time becomes $\mathcal{O}(\log_2 \log_2(|B|)) \in \mathcal{O}(1)$ as the size $\rho \times \rho$ of a block is constant.

5.2. Shared lookup table

This second approach is to use a shared lookup table of size $\rho \times \rho = \mathcal{O}(1)$ holding the final offset coordinates for each thread within the same block. Mapping each thread would cost $\mathcal{O}(1)$ memory accesses and the extra memory introduced by the shared table is $\mathcal{O}(\rho \times \rho) \in \mathcal{O}(1)$.

5.3. Bounding sub-boxes

The third approach consists of using the mapped blocks as bounding sub-boxes. This approach introduces a constant number of extra threads in each block, but allows each thread to be mapped just with $f(x) = x$ which costs $\mathcal{O}(1)$. If this method is chosen, then threads require a fast method to know if they belong to the fractal or not.

Regardless of which Intra-block mapping approach is chosen, the final mapping time will not surpass the $\mathcal{O}(\log_2 \log_2(n))$ time, as the blocks have a constant size of threads, regardless of the value of n. Still, it is worth considering the differences in the approaches; *Further Unrolling* introduces a constant cost in mapping time, the *Shared Lookup Table* approach introduces a constant cost in memory and the *Bounding Sub-boxes* introduce a constant in the number of extra threads. Choosing one or another can depend on the specific application, *i.e.*, to avoid competing with the application in the use of memory bandwidth or arithmetic operations.

6. Case study: The Sierpinski gasket

This Section applies the formulations and approaches from Sections 3 and 4, which were generic to all NBB fractals, now for the specific case of the Sierpinski gasket. Experimental performance results are presented for different test cases (involving different compute patterns that are frequently found in discrete simulations), using different fractal sizes of the Sierpinski gasket.

The *Sierpinski Gasket*, illustrated in Fig. 3. was described by Waclaw Sierpinski in 1915.

Being over a century old, this NBB fractal is still relevant as it is object of study in different fields such as the construction of
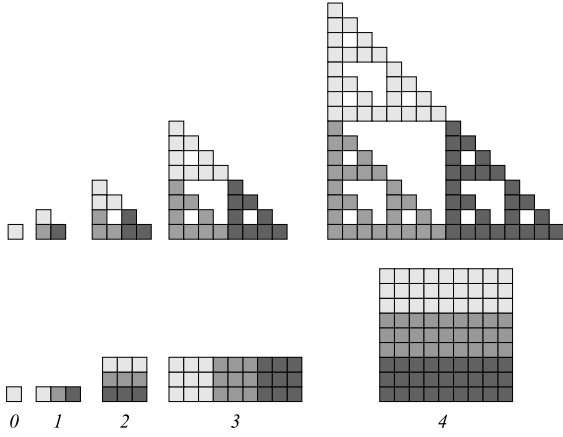
Fig. 4. Each scale of the Sierpinski fractal packs into a 2-orthotope Π^2 of dimensions $3^{\lceil \frac{r}{2} \rceil} \times 3^{\lfloor \frac{r}{2} \rfloor}$.

antennas [19,20], cellular automata [21,22], fractal molecular assembly [23], DNA self-organization [7], self-assembly theory [24, 25] and phase transitions on fractal spin lattices [26–28], among others. The Sierpinski gasket is denoted $\mathcal{F}_n^{3,2}$, where $k = 3$ and $s = 2$.

6.1. Defining $\lambda(\omega)$ for the Sierpinski gasket

The packing process of Lemma 2 describes an unrolling process, in which each block of threads, with coordinate ω in parallel space, accumulates a series of offsets to return a final mapped block coordinate in embedded fractal space. The specific case of the Sierpinski Gasket is illustrated in Fig. 4 where the unrolling principle is visible by the different shades that are in correspondence with the shaded replicas of the fractal.

The helper parameter β_μ, for the case of the Sierpinski gasket, is

$$\beta_\mu(\omega) = \left(\frac{\omega_x(\mu \mod 2) + \omega_y((\mu+1) \mod 2)}{3^{\lceil \frac{\mu}{2} \rceil - 1}} \right) \mod 3. \tag{13}$$

Replica regions are numbered as 0 (top), 1 (middle) and 2 (right) (see Fig. 4, top, for visual reference). The hash table for this fractal is $H[0] = (0, 0)$, $H[1] = [0, 1]$, $H[2] = [1, 1]$ where each pair is the corresponding replica offset. In the case of the Sierpinski gasket, it is also possible to use the following arithmetic hash function

$$h(\beta_\mu) = (\tau_x^\mu, \tau_y^\mu) = (\left\lfloor \frac{\beta_\mu}{2} \right\rfloor, \beta_\mu - \left\lfloor \frac{\beta_\mu}{2} \right\rfloor) \tag{14}$$

as an alternative to the hash table, giving the same replica offsets for each of the x and y directions at scale level μ. The replica offsets are combined with the replica linear sizes to form the offsets in embedded space

$$\Delta^\mu = (\Delta_x^\mu, \Delta_y^\mu) = (\tau_x^\mu 2^{\mu-1}, \tau_y^\mu 2^{\mu-1}) \tag{15}$$

Having defined the required functions and parameters, the $\lambda(\omega)$ map for the Sierpinski gasket becomes

$$\lambda(\omega) = \left(\sum_{\mu=1}^{r_b} \Delta_x^\mu, \sum_{\mu=1}^{r_b} \Delta_y^\mu \right) \tag{16}$$

with $r_b = \log_2(n_b)$. By Theorem 2, $\lambda(\omega)$ is asymptotically faster than a bounding box approach. The theoretical parallel space improvement as well as the speedup are presented in Fig. 5.
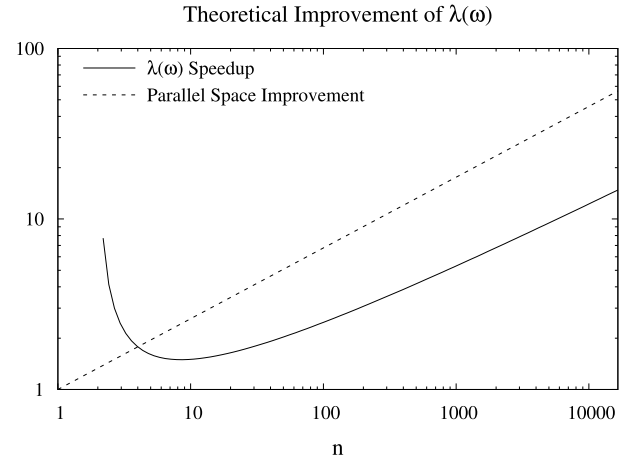

Fig. 5. Theoretical improvement for parallel-space and mapping speedup for the Sierpinski gasket.
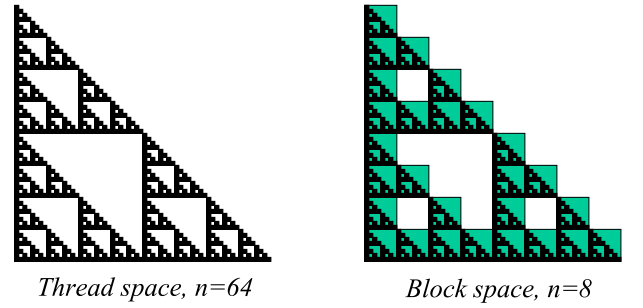

Thread space, n=64   *Block space, n=8*

Fig. 6. In thread-space mapping, threads are directly mapped one-to-one to the elements of the fractal of linear size $n = 64$. In block-space mapping, $|B| = 8 \times 8$ and blocks of threads are mapped onto a simplified version (green) of the fractal of linear size $n_b = 64/8 = 8$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In the plot, one can observe that in theory $\lambda(\omega)$ applied to the Sierpinski gasket produces a monotonically increasing speedup starting from $n \geq n_0 = 10$. The parallel space improvement in the number of threads used has also been included (dashed lines), showing a fixed exponential rate of improvement.

For the intra-block mapping phase, the bounding sub-boxes approach was used. In order to know if a location is part of the fractal, each thread evaluates if t_x & $(b - 1 - t_y) == 0$ is true or false to know if it belongs to the Sierpinski gasket or not, respectively, with & being the bitwise AND operator, b the dimensional block size, and t_x and t_y the thread's coordinate in local space. Fig. 6 illustrates how the block-space map and intra-block mapping are organized compared to a thread-space map.

6.2. Implementation and performance results for the Sierpinski gasket

The case for the Sierpinski gasket was implemented using NVIDIA's CUDA C++ toolkit as a program that performs computations on the data elements of the fractal of side length n (chosen at execution time) using both the bounding-box and $\lambda(\omega)$ approaches. In the case of $\lambda(\omega)$ the x, y arithmetic reductions per-block coordinate w from Eq. (16) are computed using the warp-shuffle parallel reduction, which allows efficient register-level

Table 2
Hardware used for performance tests.

#	Device	Model
	GPU	Titan V, 5120 cuda cores 12 GB
0	CPU	Intel i7-6950X 10-core Broadwell
	RAM	128 GB DDR4 2400 MHz
	GPU	Titan RTX, 4608 cuda cores, 24 GB
1	CPU	Intel i7-6950X 10-core Broadwell
	RAM	128 GB DDR4 2400 MHz

communication among threads within a warp.[9] Experimental benchmarking of GPU thread maps is accompanied with work instructions in the GPU kernel to represent realistic application scenarios.[10] The following three tests were designed, using different workloads:

- Single write (SW): To write a constant value on all the elements of a Sierpinski gasket of scale level r, which is embedded in a $n \times n$ matrix initially filled with zeros.
- Reduction (RD): To perform an arithmetic reduction with all the elements of the Sierpinski gasket.
- Cellular Automata (CA): To perform a Cellular Automaton simulation using a fractal adaptation of Conway's game of life-like rules. This adaptation still uses the Euclidean Moore neighborhood, but only considers as neighbors the cells that belong to the fractal and the cells of the empty embedded space are ignored in the neighborhood counting.

Different fractal sizes were tested in the range $r = 0..16$ (up to $r = 15$ in tests RD and CA due to memory limitations), equivalent to embedding sizes of $n \times n = [\{1 \times 1\}, \ldots, \{65\,536 \times 65\,536\}]$, and using different GPU block sizes in the range $\rho = 1, 2, 4, 8, 16, 32$ in order to find the setting that provides the best performance for both the bounding-box and the $\lambda(\omega)$ approaches. The average performance measures are taken by averaging 100 sub-averages, each one being an average time of 10 consecutive synchronized kernel calls. The standard error for each mean was below 1%. The hardware for performance test is listed in Table 2.

Fig. 7 presents the speedup of $\lambda(\omega)$ over the bounding-box approach, as well as the running times for the two mapping techniques in all three different tests. For values of $n < 2^9$, one can note that only some curves offer speedup. Once $n > 2^9$, the speedup begins to increase for all block-size configurations, reaching the higher values at $n = 2^{16} = 65\,536$, which was the highest problem size that fit in the GPU memory (*i.e.* a fractal embedded in a region of $65\,536 \times 65\,536$). An important aspect to note from the speedup curves is that for the largest possible block size, $|B| = \rho \times \rho = 32 \times 32$, the $\lambda(\omega)$ map runs the tests between $6\times$ to $12\times$ faster than the bounding-box approach. Furthermore, as blocks become smaller in ρ, that improvement increases dramatically, reaching up to $75\times$ of speedup.

The plot of the running times provides further insights on what configuration is the best suited for each mapping technique. By looking at the running times of the small block configurations, one can note that regardless of their high speedup, their running times are the slowest, therefore these block sizes would not be used in practice. For the bounding-box approach the best performance is obtained when the block-size is $|B| = 32 \times 32$. For $\lambda(\omega)$ the best performance is found when using a block of $|B| = 16 \times 16$ threads. If the curves of the best configuration

---

[9] A warp is a group of 32 threads that execute instructions in a lock-step mode and can also communicate their register data among themselves.

[10] Also it may not be clear if the compiler and scheduler optimizes the program, ignoring the mapping instructions, when no writes are performed on memory.

for each implementation are considered, *i.e.*, the ones with bold mark from Fig. 7, right, then the speedup provided by $\lambda(\omega)$ still reaches almost an order of magnitude. The running time using other block sizes are still useful to visualize that as blocks become smaller, the value of n_0 where $\lambda(\omega)$ starts giving monotonically increasing speedup moves closer to the origin, and vice versa. It is important to consider that the GPU, with its current organization and architecture, is not fully utilized when using very small block configurations, leading to an inferior performance than if larger blocks were used. Therefore, in practice large blocks would be utilized and by Theorem 2, beyond $n = 2^{16}$ the speedup would keep increasing in favor of $\lambda(\omega)$.

We believe that these performance results can be useful for the GPU computing community as they show that for any modern programmable GPU, its performance can significantly improve when working with embedded NBB fractals just by employing a different thread map, not changing the rest of the application kernel code at all. In the next Section we describe how it is possible to further accelerate the performance of $\lambda(\omega)$ by adapting it to GPU tensor cores.

6.3. Adapting $\lambda(\omega)$ for accelerated tensor core computation

The Nvidia Volta GPU micro-architecture introduced a specialized hardware component called the Tensor Core. Actual GPUs of year 2018 and beyond can contain up to 640 tensor cores in addition to the regular GPU cores (which perform integer, floating point and read/write operations in parallel). Each tensor core is able to perform matrix-multiply-accumulate (MMA) operations on 4×4 matrices in one GPU clock cycle, which translates into a significant increase of TFLOPS compared to the classic operation mode of the GPU which is through the execution of floating point and integer arithmetic instructions. In order to make use of the Tensor Cores, the programmer must previously divide the problem into sub-problems of 16×16 sub-matrices, called fragments. These fragments are then given to the MMA subroutines which internally sub-divide them into 4×4 fragments to perform the operations in a warp-synchronized manner. Currently, as of 2020, details on how warps map to the fragments, or how tensor core perform the MMA operation are not fully specified by NVIDIA, moreover it is not guaranteed that a tensor-core based computation that works efficient in the Volta architecture (2017), will achieve the same level of performance in the Turing architecture (2019), or vice versa, as there are implementation details that are not exposed to the programmer. What is known is that the potential performance improvement will depend on how well the MMA operation can be exploited. These tensor-core related questions introduce additional motivations for knowing if the theoretical extra TFLOPS provided by a Tensor Core MMA operation, which were originally designed for Linear Algebra and Deep Learning, can be exploited to further speed up the calculation of $\lambda(\omega)$. Recent results support the idea that some computations may adapt well to tensor cores, such as the work of R. Carrasco et al. where they study the potential speedup of computing the traditional arithmetic reduction based on tensor-core MMA operations [29]. From their work, the authors conclude that the new tensor-core based reduction is in theory faster than a CUDA-Core based reduction. In this section we show how the computations for $\lambda(\omega)$ can be adapted as tensor-core MMA operations to calculate the tensor-core version of the map, namely $\lambda_{tc}(\omega)$.

In order speedup the calculation of $\lambda(\omega)$ with tensor cores, its equations must be encoded into a MMA operation in the form $D = A \times B + C$ where A, B, C, D are fragments, and C can be the same as D. There can be several ways to perform this encoding, and this section presents three variants of tensor core adaptation with their performance for the same tests.
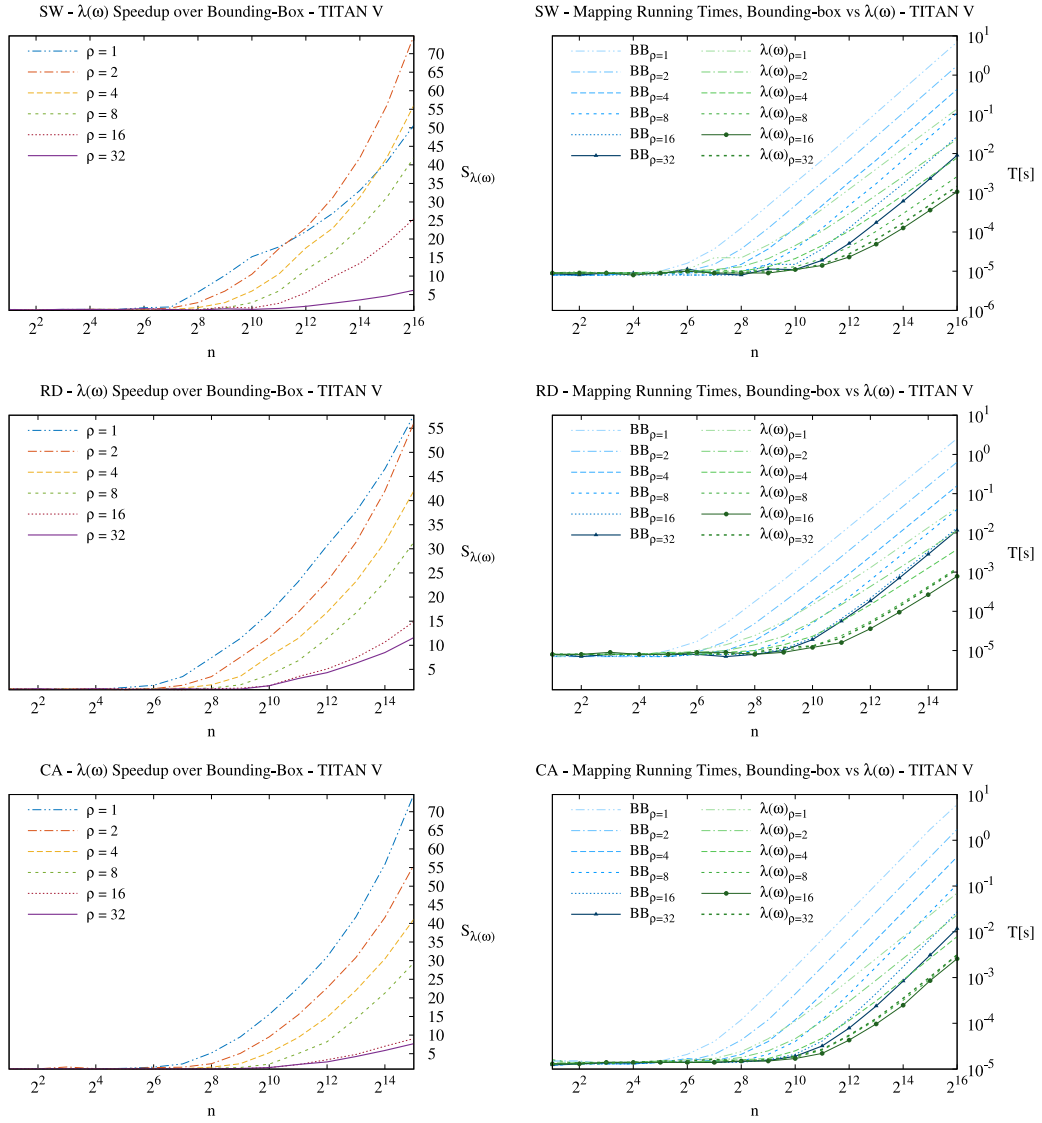
Fig. 7. The left column shows the speedup of $\lambda(\omega)$ with respect to the bounding-box approach at different block-size configurations and on the right column, their absolute running times at different block-size configurations. Each row shows the results of different test being: first row, test 1 simple write. Second row, test 2 reduction. Third row, test 3 Cellular automata.

6.3.1. Variant 1: Simple per-block tensor core operation

This variant employs one MMA computation for each block by exploiting the MMA-like behaviour from Eqs. (8) and (9). This is done by expanding the sum, as well as the Δ_x^μ, Δ_y^μ terms with the expression from Eq. (6), resulting into two sums of products; one to calculate λ_x and the other to calculate λ_y. Each left multiplier from the sum terms is placed as an element of a row of fragment A and each right side of the sum terms is placed as a column of fragment B. Terms are placed in parallel and in the same order to match the corresponding pairements.

The left side of both sums correspond to powers of two, from 0 to $\mu - 1$, and are the same for λ_x and λ_y. Therefore these factors can be encoded only using one row of fragment A and can be re-utilized for both x and y coordinates. The final encoding can be visualized in Fig. 8.

An important technical note is that Fragment B was defined as column major matrix and fragment A as a row major to ease the memory access during the calculations and minimize data divergence. Once the Tensor core operation is done, the results λ_x and λ_y become the first and second elements of the first row of fragment D, respectively.

$$A = \begin{pmatrix} 2^0 & 2^1 & \dots & 2^{\mu-1} \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} B = \begin{pmatrix} \tau_x^1 & \tau_y^1 & 0 & \dots & 0 \\ \tau_x^2 & \tau_y^2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_x^\mu & \tau_y^\mu & 0 & \dots & 0 \end{pmatrix}$$

Fig. 8. Encoding of Variant 1 in a MMA manner. Note that this matrices are dimensions $\mu x \mu$ and when loaded into fragments of 16 × 16, remaining elements are filled with zeros.

6.3.2. Variant 2: Sub-blocked tensor core operation

This variant shares the principle of Variant 1, but expands the idea by subdividing the block of threads into sub-blocks, calculating more block coordinates (one for each sub-block) still using one tensor core MMA operation. Since every fragment has 16 rows and columns, there is the potential of calculating 16 different values; 8 pairs of (λ_x, λ_y). The approach assumes a thread block size large enough, to contain 4 sub-blocks of size $b/2 \times b/2$ threads that are still large enough to produce an efficient computation. These sub-blocks are now treated as independent blocks of threads that are not yet mapped and sit in parallel space
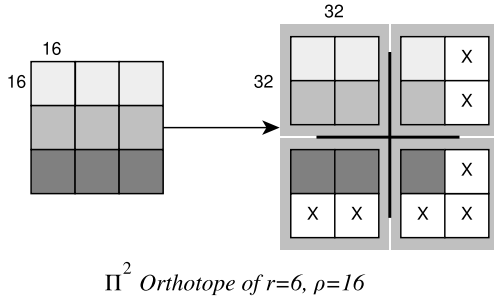
$$\Pi^2 \ Orthotope \ of \ r=6, \ \rho=16$$

Fig. 9. On the left a 2-Orthotope for a Sierpinski gasket of scale level is 6 with a sub-block size of 16 × 16. On the right, the 2-Orthotope when treated with block size of 32 × 32 and then subdivided.

$$A = \begin{pmatrix} 2^0 & 2^1 & \cdots & 2^{\mu-1} \\ 2^0 & 2^1 & \cdots & 2^{\mu-1} \\ \vdots & \vdots & \ddots & \vdots \\ 2^0 & 2^1 & \cdots & 2^{\mu-1} \end{pmatrix} B_x = \begin{pmatrix} \tau_x^1 & \tau_x^1 & \cdots & \tau_x^1 \\ \tau_x^2 & \tau_x^2 & \cdots & \tau_x^2 \\ \vdots & \vdots & \ddots & \vdots \\ \tau_x^\mu & \tau_x^\mu & \cdots & \tau_x^\mu \end{pmatrix}$$

$$C_x = \begin{pmatrix} t_{1,1}^x & t_{1,2}^x & \cdots & t_{1,\mu}^x \\ t_{2,1}^x & t_{2,2}^x & \cdots & t_{2,\mu}^x \\ \vdots & \vdots & \ddots & \vdots \\ t_{\mu,1}^x & t_{\mu,2}^x & \cdots & t_{\mu,\mu}^x \end{pmatrix}$$

Fig. 10. MMA scheme of variant 3. Fragments B_y and C_y share the same representation as their x counterpart. Note that these matrices are dimensions $\mu x \mu$ and when loaded into fragments, the extra elements get filled with zeros.

ready to be mapped with $\lambda(\omega)$. Fig. 9 shows an example using a block size of 32 × 32 subdividing into sub-blocks of 16 × 16.

Once the MMA operation is done, the mapped coordinates for each sub-block are found in the first row of the resulting fragment D. It is worth noticing that this approach introduces some chunks of unused threads which lie outside of the 2-Orthotope generated. The uncolored blocks marked with X in the right side of Fig. 9 represents unused sub-blocks of threads. These extra threads do not introduce a significant cost as they are upper bounded by $O(\sqrt{n^{\mathcal{H}}})$ (i.e, the perimeter of the packed fractal) in comparison to the domain of the 2-Orthotope which is $O(n^{\mathcal{H}})$.

6.3.3. Variant 3: Full A, B and C usage

The two variants described rely only on fragments A and B for its calculation while most of their fields are empty, while fragment C is unused, therefore not taking advantage of the addition operator of the MMA operation. This third variant maintains the same encoding for A and B, but including C in the calculation, filling completely all fields of the 3 fragments. Fig. 10 shows the arrangement of each matrix.

The result of the tensor Core MMA is now a thread coordinate in data space for each thread, whilst in the previous variants, it was a block coordinate in data space for all threads within that block. This means that every thread accesses its corresponding coordinate in fragment D using its parallel space coordinates in a 1:1 mapping. It is implied that this variant necessarily also encodes the intra-block mapping phase into the tensor core operation. Bounding sub-boxes was the mapping used by default. In order for this Variant to work, 2 tensor core MMA operations must be performed per block, one for λ_x coordinate and the other for λ_y of each thread. This variant was developed for block size $\rho = 16$ to fit the 256 λ coordinates of the 256 threads. The $\rho =$

32 version is also possible and would require to sub-divide the block into four sub regions of 16 × 16 performing 8 tensor core operations in total, with the access to data space being contiguous by the sub-regions for they are not treated as independent blocks like in variant 2.

6.3.4. $\lambda_{tc}(\omega)$ results

The tensor core-based methods were tested with the same workloads and hardware as $\lambda(\omega)$ and were compared within their corresponding versions. The speedups of the tensor core variants with respect to the regular $\lambda(\omega)$ are shown in Fig. 11. It is important to review the results of both Volta and Turing architecture, considering that the internal implementation of tensor core operations may give different performances.

Starting with the first test, the single-write (SW), results with the TITAN V show that when $n \geq n_0 = 2^{10}$, speedup curves reach a stable behavior, whereas with $n < n_0$ the speedup curves show unstable behavior oscillating around 1.0 of speedup. In the large scale regime variant 2 gives up to 20% of extra performance over the regular $\lambda(\omega)$ map. With the TITAN RTX, the speedup curves show that again variant 2 increases $\lambda(\omega)$ performance up to a 40%, and variant 1 becomes a usable choice as it gets a speedup above 1.0 in all configurations. In both GPUs, variant 3 results in poor performance when n is greater than n_0 with both $\rho = 16$ and $\rho = 32$ block sizes. In the reduction (RD) test the results were no different from SW, with variant 2 getting the best performance with a 20% and 30% of increased performance in TITAN V and RTX respectively. Variant 1 is only beneficial in TITAN RTX with $\rho = 16$, and variant 3 is inefficient in all block configurations. Lastly, results on the Cellular Automata test (CA) when $n < n_0$ shows the same behaviour as previous tests, but as n grows from n_0 in TITAN V, speedup of all curves start to decrease below 1, the same happens with TITAN RTX except for variant 2 that gets a positive performance boost. The general performance hit observed for all variants may be a consequence of the computation patterns found in the CA simulation.

Summarizing the tensor core results, the variant that showed better results overall was variant 2 with up to a 40% of performance boost over non-tensor core lambda when $n > 2^{10}$. Variant 1 is also better under certain configurations with a 5 ∼ 10% of performance boost. Variant 3 showed slower performance than the regular $\lambda(\omega)$ thus can be discarded.

7. Discussion and conclusions

This work has shown that the $\lambda(\omega)$ map proposed for NBB fractals leads to a significant performance speedup both in theory and in experimental tests with an embedded Sierpinski gasket. The analysis and formulation of $\lambda(\omega)$ has provided three important results in the theoretical aspect; (1) There exists a correspondence between a quasi-regular 2-orthotope and NBB fractals, (2) such correspondence can be computed in just $\mathcal{O}(\log_2 \log_2(n))$ time and (3) the total work required for mapping the 2-orthotope used with $\lambda(\omega)$ is asymptotically smaller than the work generated by the bounding box approach, leading to a monotonically increasing speedup starting from $n \geq n_0$. In particular, Theorem 2 serves as a guarantee that using $\lambda(\omega)$ on any NBB fractal will provide a significant performance speedup starting from a certain fractal scale onward, as the speedup monotonically increases with n.

The experimental performance results confirm the theoretical results, showing monotonically increasing speedup once $n \geq n_0 = 2^9$ and up to 9× of speedup over a bounding-box approach using optimal block-size settings for each approach. Using smaller block-sizes leads to even higher speedups (up to 75×) but slower running times. It is uncommon for GPU applications to use small block sizes, but in case it is required, a significant performance
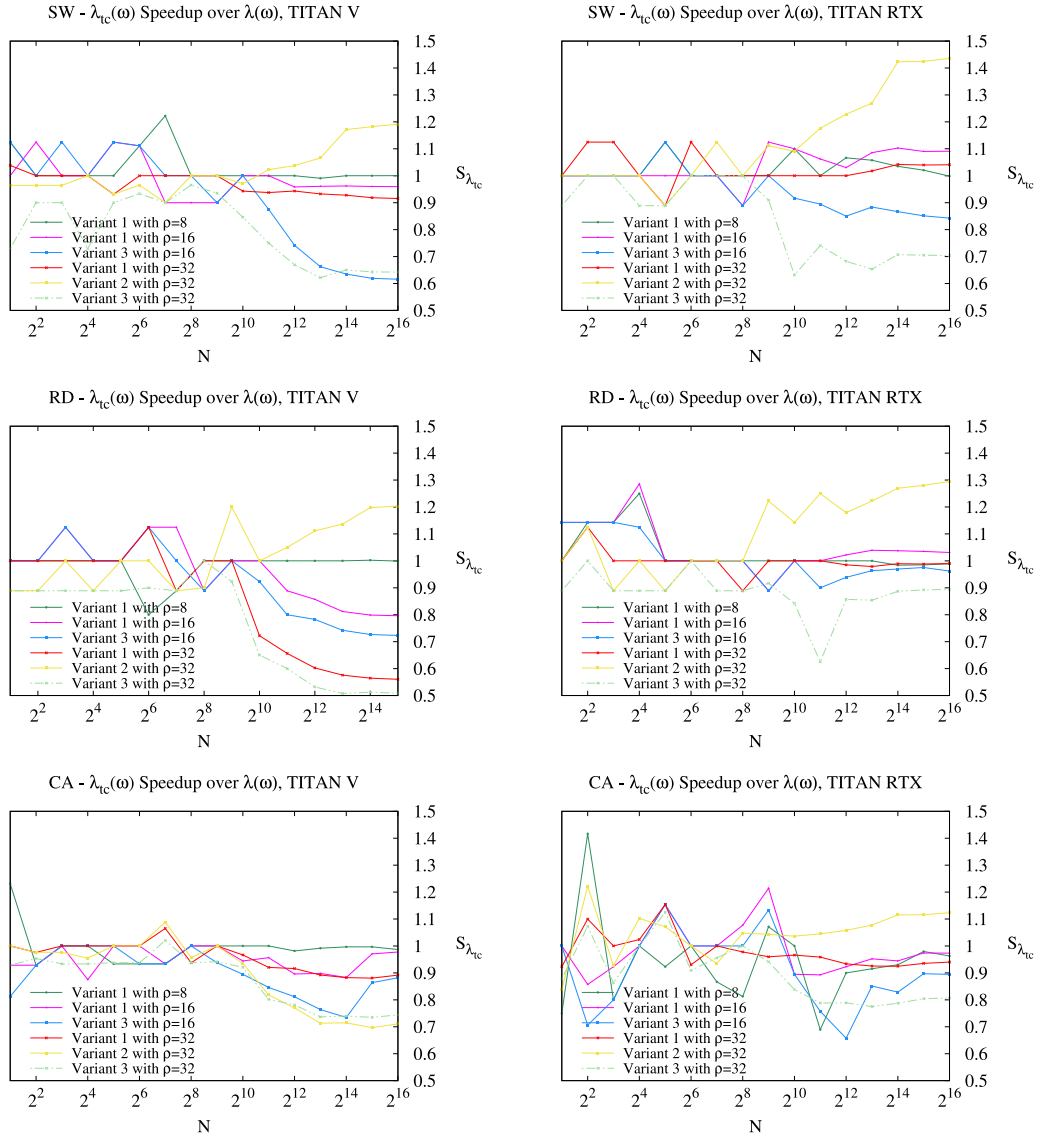
Fig. 11. The graphs shows the speedup of $\lambda_{tc}(\omega)$ with respect to non-tensor core based $\lambda(\omega)$. The left column shows the results with a TITAN V GPU (Volta architecture) and right column results with a TITAN RTX GPU (Turing architecture).

improvement is available with this approach. Still, the exploration of GPU performance under different block-sizes has allowed to understand that small block sizes behave as the theoretical results in a strong way, while the largest block sizes, although still produce monotonically increasing speedup, behave as the theory in a weaker form. The adaptation of $\lambda(\omega)$ to use tensor core computation offered up to $\sim 40\%$ of extra performance. In order to achieve efficient tensor core computation it is important to exploit communication between tensor core fragments and shared memory as most as possible, and in the case of non-Machine Learning tasks, to codify the computation with the least redundant data into the fragments. The closer the task is to linear algebra, the easier this adaptation will be.

It is important to note that the GPU thread map presented in this work, along with the tensor core optimization, can be implemented for any fractal belonging to the NBB family. The implementation of the case study from this work, including the three tests with and without tensor-core adaptations, is available for the community at https://github.com/crinavar/gpumaps.[11]

Future work on this line can follow two paths; (1) further study GPU thread mapping on fractals and extend the NBB family to include other fractals which use rotations in the replicas, such as the Koch curve, and (2) evaluate the performance of compact fractal manipulation in GPU, that is, to pack the data space into an orthotope as the parallel space, and allow operations on the structure without decompressing the fractal, just by using $\lambda(\omega)$ and $\lambda(\omega)^{-1}$ for unrolling and rolling the computations. This last path could allow handling much larger fractals in GPU as the memory used would be in the order of $n^{\mathcal{H}}$. Future research in these directions can provide important insights on the potential benefits of efficient GPU computing for fractal geometry.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

---

[11] Note to the reviewers: the repository will be made open to the community in the published version of this article.

as well as by the Nvidia CUDA Research Center at the Department of Computer Science (DCC) from University of Chile and the Millenium Institute Foundational Research on Data (IMFD), Chile.

References

[1] B.B. Mandelbrot, Fractals, in: Encyclopedia of Statistical Sciences, John Wiley & Sons, Inc., 2004, http://dx.doi.org/10.1002/0471667196.ess0816.

[2] B.B. Mandelbrot, The Fractal Geometry of Nature, Freeman, San Francisco, CA, 1982, URL https://cds.cern.ch/record/98509.

[3] P.E. Oppenheimer, Real time design and animation of fractal plants and trees, SIGGRAPH Comput. Graph. 20 (4) (1986) 55–64, http://dx.doi.org/10.1145/15886.15892, URL http://doi.acm.org/10.1145/15886.15892.

[4] M.W. Palmer, Fractal geometry: a tool for describing spatial patterns of plant communities, Vegetatio 75 (1) (1988) 91–102, http://dx.doi.org/10.1007/BF00044631.

[5] B.T. Milne, Measuring the fractal geometry of landscapes, Appl. Math. Comput. 27 (1) (1988) 67–79, http://dx.doi.org/10.1016/0096-3003(88)90099-9, URL http://www.sciencedirect.com/science/article/pii/0096300388900999.

[6] A.P. Pentland, Fractal-based description of natural scenes, IEEE Trans. Pattern Anal. Mach. Intell. PAMI-6 (6) (1984) 661–674, http://dx.doi.org/10.1109/TPAMI.1984.4767591.

[7] P.W.K. Rothemund, N. Papadakis, E. Winfree, Algorithmic self-assembly of DNA sierpinski triangles, PLoS Biol. 2 (12) (2004) e424, http://dx.doi.org/10.1371/journal.pbio.0020424.

[8] K. He, C.-Y. Xu, L. Zhen, W.-Z. Shao, Fractal growth of single-crystal α-Fe2O3: From dendritic micro-pines to hexagonal micro-snowflakes, Mater. Lett. 62 (4–5) (2008) 739–742, http://dx.doi.org/10.1016/j.matlet.2007.06.082, URL http://www.sciencedirect.com/science/article/pii/S0167577X07006647.

[9] A. Gamba, D. Ambrosi, A. Coniglio, A. de Candia, S. Di Talia, E. Giraudo, G. Serini, L. Preziosi, F. Bussolino, Percolation, morphogenesis, and burgers dynamics in blood vessels formation, Phys. Rev. Lett. 90 (2003) 118101, http://dx.doi.org/10.1103/PhysRevLett.90.118101, URL https://link.aps.org/doi/10.1103/PhysRevLett.90.118101.

[10] E.R. Weibel, Fractal geometry: a design principle for living organisms, Amer. J. Physiol. - Lung Cell. Mol. Physiol. 261 (6) (1991) L361–L369, arXiv:http://ajplung.physiology.org/content/261/6/L361.full.pdf. URL http://ajplung.physiology.org/content/261/6/L361.

[11] C.A. Navarro, N. Hitschfeld-Kahler, L. Mateu, A survey on parallel computing and its applications in data-parallel problems using GPU architectures, Commun. Comput. Phys. 15 (2014) 285–329.

[12] C.A. Navarro, M. Vernier, B. Bustos, N. Hitschfeld, Competitiveness of a non-linear block-space gpu thread map for simplex domains, IEEE Trans. Parallel Distrib. Syst. 29 (12) (2018) 2728–2741.

[13] C.A. Navarro, R. Vega, B. Bustos, N. Hitschfeld, Block-space GPU mapping for embedded sierpiński gasket fractals, in: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017, pp. 427–433, http://dx.doi.org/10.1109/HPCC-SmartCity-DSS.2017.56.

[14] C.A. Navarro, N. Hitschfeld, GPU maps for the space of computation in triangular domain problems, in: 2014 IEEE International Conference on High Performance Computing and Communications, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014, 2014, pp. 375–382, http://dx.doi.org/10.1109/HPCC.2014.64.

[15] J.H. Jung, D.P. O'Leary, Exploiting Structure of Symmetric or Triangular Matrices on a GPU, Tech. rep., University of Maryland, 2008.

[16] F. Ries, T. De Marco, M. Zivieri, R. Guerrieri, Triangular matrix inversion on graphics processing unit, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, in: SC '09, ACM, New York, NY, USA, 2009, pp. 9:1–9:10.

[17] C.A. Navarro, B. Bustos, N. Hitschfeld, Potential benefits of a block-space GPU approach for discrete tetrahedral domains, in: CLEI-2016, XLII Conferencia Latinoamericana de InformáTica, Valparaiso, Chile, October 10-14, 2016, 2016.

[18] R.P. Brent, The parallel evaluation of general arithmetic expressions, J. ACM 21 (2) (1974) 201–206, http://dx.doi.org/10.1145/321812.321815, URL http://doi.acm.org/10.1145/321812.321815.

[19] C.P. Baliarda, C.B. Borau, M.N. Rodero, J.R. Robert, An iterative model for fractal antennas: application to the Sierpinski gasket antenna, IEEE Trans. Antennas and Propagation 48 (5) (2000) 713–719, http://dx.doi.org/10.1109/8.855489.

[20] C. Puente-Baliarda, J. Romeu, R. Pous, A. Cardama, On the behavior of the Sierpinski multiband fractal antenna, IEEE Trans. Antennas and Propagation 46 (4) (1998) 517–524, http://dx.doi.org/10.1109/8.664115.

[21] F. Ohi, Y. Takamatsu, Time-space pattern and periodic property of elementary cellular automata − Sierpinski gasket and partially Sierpinski Gasket −, Japan J. Ind. Appl. Math. 18 (1) (2001) 59, http://dx.doi.org/10.1007/BF03167355.

[22] S. Wolfram, Statistical mechanics of cellular automata, Rev. Modern Phys. 55 (3) (1983) 601–644, http://dx.doi.org/10.1103/RevModPhys.55.601, URL http://link.aps.org/doi/10.1103/RevModPhys.55.601.

[23] J. Shang, W. Yongfeng, M. Chen, et al., Assembling molecular Sierpiński triangle fractals, Nat. Chem. 7 (5) (2015) 389–393, http://dx.doi.org/10.1038/nchem.2211.

[24] D. Doty, Theory of algorithmic self-assembly, Commun. ACM 55 (12) (2012) 78–88, http://dx.doi.org/10.1145/2380656.2380675, URL http://doi.acm.org/10.1145/2380656.2380675.

[25] J.I. Lathrop, J.H. Lutz, S.M. Summers, Strict self-assembly of discrete Sierpinski triangles, Theoret. Comput. Sci. 410 (4) (2009) 384–405, http://dx.doi.org/10.1016/j.tcs.2008.09.062, URL http://www.sciencedirect.com/science/article/pii/S030439750800724X.

[26] Y. Gefen, A. Aharony, Y. Shapir, B.B. Mandelbrot, Phase transitions on fractals. II. Sierpinski gaskets, J. Phys. A: Math. Gen. 17 (2) (1984) 435, URL http://stacks.iop.org/0305-4470/17/i=2/a=028.

[27] Y. Gefen, B.B. Mandelbrot, A. Aharony, Critical phenomena on fractal lattices, Phys. Rev. Lett. 45 (1980) 855–858, http://dx.doi.org/10.1103/PhysRevLett.45.855, URL https://link.aps.org/doi/10.1103/PhysRevLett.45.855.

[28] K.P. Mota, P.M.C. de Oliveira, Monte Carlo simulations for the slow relaxation of crumpled surfaces, Physica A 387 (24) (2008) 6095–6104, http://dx.doi.org/10.1016/j.physa.2008.07.001, URL http://www.sciencedirect.com/science/article/pii/S0378437108006055.

[29] R. Carrasco, R. Vega, C.A. Navarro, Analyzing GPU tensor core potential for fast reductions, in: 2018 37th International Conference of the Chilean Computer Science Society (SCCC), 2018, pp. 1–6, http://dx.doi.org/10.1109/SCCC.2018.8705253.

Cristóbal A. Navarro received the Ph.D. degree in computer science from the University of Chile, in 2015. He is an assistant professor at the Institute of Informatics, Universidad Austral de Chile. During 2015–2018, Cristóbal did a Postdoc research with Prof. Benjamin Bustos. Today, his research interests include GPU computing, computational physics and computer graphics.

Felipe A. Quezada is currently finishing the Bsc degree in computer science as well as the M.Sc. in HPC and data science, both at Universidad Austral de Chile. He is currently working as part of the research team of Dr. Cristóbal Navarro. His interests are mainly focused on HPC, machine learning and computer graphics.

Nancy Hitshfeld received the B.Sc. and M.Sc. degrees in computer science from the University of Chile, in 1984 and 1987, respectively, and the Ph.D. degree in applied sciences (technischen wissenschaften) from the Swiss Federal Institute of Technology (ETH-Zurich), in 1993. During her Ph.D., she worked with the Integrated Systems Institute, developing new discretization algorithms for the simulation of semiconductor devices. Currently, she works as associate professor with the Computer Science Department (DCC), Faculty of Physical and Mathematical Sciences, University of Chile. Her main research interests include geometric modeling, geometric meshes, and parallel algorithms (GPU computing), focused in computational science, and engineering applications.

Raimundo Vega received MS and Ph.D. degree in informatics from the Universidad Politécnica de Madrid, Spain, in 1988 and 1999, respectively. His doctoral thesis proposed a fault tolerant input/output subsystem, which was presented in the annual CLEI reunion in Mexico, the year 2000, obtaining a place among the top 10 best works presented. Nowadays, his investigation area is mainly focused towards Software Engineering and Distributed Systems.

Benjamin Bustos received the doctoral degree in natural sciences from the University of Konstanz, Germany, in 2006. He is an associate professor with the Department of Computer Science, University of Chile. He is head of the PRISMA Research Group, and he is also a researcher with the Millennium Nucleus Center for Semantic Web Research. He leads research projects in the domain of content-based multimedia information retrieval. His research interests include similarity search, 3D object retrieval, multimedia mining, Semantic Web, metric/nonmetric indexing, and pattern recognition.