A Scalable and Energy Efficient GPU Thread Map for Standard m-Simplex Domains

Cristóbal A. Navarro\(^a\), Felipe A. Quezada\(^b\), Benjamin Bustos\(^b\), Nancy Hitschfeld\(^b\), Rolando Kindelan\(^b\)

\(^a\)Instituto de Informática, Universidad Austral de Chile.
\(^b\)Computer Science Department, University of Chile.

Abstract

This work proposes a new GPU thread map for standard m-simplex domains, that scales its speedup with dimension and is energy efficient compared to other state of the art approaches. The main contributions of this work are the formulation of the new block-space map \(H : \mathbb{Z}^m \mapsto \mathbb{Z}^m\) which is analyzed in terms of resource usage, and its experimental evaluation in terms of speedup over a bounding box approach and energy efficiency as elements per second per Watt. Results from the analysis show that \(H\) has a potential speedup of up to \(2^x\) and \(6^x\) for 2 and 3-simplices, respectively. Experimental evaluation shows that \(H\) is competitive for 2-simplices, reaching \(1.2^x \sim 2.0^x\) of speedup for different tests, which is on par with the fastest state of the art approaches. For 3-simplices \(H\) reaches up to \(1.3^x \sim 6.0^x\) of speedup making it the fastest of all. The extension of \(H\) to higher dimensional m-simplices is feasible and has a potential speedup that scales as \(m!\) given a proper selection of parameters \(r, \beta\) which are the scaling and replication factors, respectively. In terms of energy consumption, although \(H\) is among the highest in power consumption, it compensates by its short duration, making it one of the most energy efficient approaches. Lastly, further improvements with Tensor and Ray Tracing Cores are analyzed, giving insights to leverage each one of them. The results obtained in this work show that \(H\) is a scalable and energy efficient map that can contribute to the efficiency of GPU applications when they need to process standard m-simplex domains, such as Cellular Automata or PDE simulations.

Keywords: GPU Computing, Thread Mapping, Scalable, Simplex Domains, Energy Efficiency, GPU Resource Improvement

1. Introduction

GPU computing has become a well established research field in the last years [1 2 3] thanks to the increasing performance of GPU hardware and the existence of a general purpose programming model, being CUDA [4] and OpenCL [5] the most known implementations of such model. Notorious advancements have been made to better understand how data-parallel algorithms can be more efficient on the GPU architecture. Some of these advancements include the improvement of thread access patterns for coalesced memory access [6], reduction of thread branching [7 8], better use of the \(L_1\) programmable cache [9], fast stencil computations [10], efficient reduction algorithms [11 12], multi-GPU acceleration [13 14 15], and most recently the mapping of different computational patterns to tensor cores [16 12 17 18] as well as ray tracing cores [19 20 21]. One performance aspect of GPU computing that is also being researched in the last years is the study and design of efficient GPU thread maps for data-parallel domains with complex geometries [18 22 23 24]. Several problems exhibit a non-boxed shape data domain in which data-parallel GPU computation must be mapped. Examples of such problems are simplex domains, which exist when doing computation on half-triangular matrices, or when doing a cellular automata simulation on a discrete voxel-based tetrahedron. In such cases mapping the GPU threads efficiently on the simplex is not straightforward for the programmer.

This work proposes a new map \(H : \mathbb{Z}^m \mapsto \mathbb{Z}^m\) for mapping GPU threads onto m-simplex domains efficiently. The map assigns thread-blocks in parallel space to unique locations in the discrete orthogonal m-simplex using \(O(1)\) arithmetic operations. A dedicated analysis is devoted to the special cases of 2-simplex and 3-simplex domains, where new efficient maps are proposed and described, based on a binary self-similar organization of sub-orthotopes. The approach can offer a parallel space improvement of \(\sim 2^x\) and \(\sim 6^x\) for \(m = 2\) and \(m = 3\), respectively, that results in a potential performance improvement given that the maps costs \(O(1)\) arithmetic operations and no \(m\)-th roots are required. Experimental results support the theoretical bounds, showing that \(H\) is competitive in 2-simplices, the fastest one in 3-simplices, and energy efficient. For the higher dimensional case, an analysis shows that building an efficient set of self-similar orthotopes is possible with a potential speedup of \(m!\), given optimal values are found for the scaling and replication factors, \(r\) and \(\beta\) respectively.

This work improves from a previous conference publication [25] by including an improved formulation and a full experimental evaluation using more comparison approaches, additional tests and several GPUs. Also, it extends the previous work by including a complete evaluation of energy efficiency for all approaches, as well as new insights for leveraging Tensor and Ray Tracing cores in modern GPU architectures.
2. The Problem of Mapping GPU Threads onto Simplices

The GPU programming model offers four constructs that allow the execution of highly parallel algorithms: (1) thread, (2) block, (3) grid and (4) kernel. Threads are the smallest work elements and they are in charge of executing the instructions of a GPU kernel, which is the main parallel program written by the programmer. A block is an intermediate structure that contains a set of threads organized as an Euclidean box. Blocks provide fast shared memory access as well as local synchronization for all of its threads. The grid is the largest construct of all three and it keeps all blocks together spatially organized for the execution of a GPU kernel. These three constructs are illustrated in Figure 1 and play an important role when mapping processing resources to a data-parallel domain. In 2022 the CUDA programming model introduced the thread-block cluster, an optional construct that fills the gap in between a block and the grid. It can be used for leveraging memory locality among blocks.

![Figure 1: GPU constructs using the CUDA naming scheme.](image)

For every GPU computation there is a stage where threads, which live in grid space, get mapped to different locations in data space. This mapping is defined as

$$f(x) : \mathbb{Z}^k \rightarrow \mathbb{Z}^m$$  \hspace{1cm} (1)

where \( k \)-dimensional locations \( x = (x_1, x_2, \ldots, x_k) \), in grid space, map to \( m \)-dimensional locations \( y = (y_1, y_2, \ldots, y_m) \) in data space, typically as an injective or bijective function. GPU parallel spaces can be formally described as orthotopes \( \Pi^m_n \) in \( n = 1, 2, 3 \) dimensions that contain discrete elements organized in blocks. Each element corresponds to a thread that must be assigned to the data domain.

A known approach for mapping threads in GPUs is to build a bounding-box (BB) of volume \( l_1 \times l_2 \times \cdots \times l_k \) where \( l_i \) is the length of the data space in the \( i \)-th dimension. Without loss of generality, we can assume a regular geometry to express the bounding box volume as \( n^k \) where \( n \) is the needed length to cover the entire data domain in all dimensions. Using the map

$$f(x) = x$$  \hspace{1cm} (2)

it is possible to obtain the corresponding location of every thread in data space. Such map is highly convenient, simple and efficient for the class of problems where the data space is an orthotope as well, such as vectors, tables, images, matrices or other arbitrary box-shaped data domains. However, this approach is no longer convenient neither efficient once data domains start taking other shapes.

There is a class of highly parallelizable problems where data organizes in the form of a standard simplex. Computations such as the Euclidean distance matrix (EDM) \([26,27,28]\), collision detection \([29]\), adjacency matrices \([30]\), cellular automata simulation on triangular domains \([31]\), matrix inversion \([32]\) and even the \( n \)-body problem \([33,34,35]\), among others, follow the shape of a standard 2-simplex. Here, such simplex is denoted as \( \Delta^2_n \), with \( n \) its length along one dimension, and its working space is \( V(\Delta^2_n) = n(n+1)/2 \in O(n^2) \). The problem is that for this class of problems the default bounding-box (BB) GPU mapping approach turns out to be inefficient because the orthotope of parallel space, denoted \( \Pi^2_n \), produces \( n^2 \) threads, where \( n(n-1)/2 \in O(n^2) \) are unused, as shown in Figure 2.

![Figure 2: The bounding-box approach almost doubles the number of necessary threads when acting on \( \Delta^2_n \).](image)

The problem is not restricted just to two-dimensional cases. Simulation applications that act on voxelized or cell-like regular tetrahedrons, such as PDEs or Cellular Automata, also employ a standard 3-simplex. In the 3-simplex class, the interaction space is \( V(\Delta^3_n) = n(n+1)/2 + n/6 \in O(n^3) \) elements, organized as a discrete orthogonal tetrahedron. Once again, the default bounding-box (BB) approach turns out to be inefficient as it generates a parallel space \( V(\Pi^3_n) \) with \( O(n^3) \) unnecessary threads (see Figure 3).

![Figure 3: The bounding-box approach mapped onto \( \Delta^3_n \) generates a cubical amount of unnecessary threads.](image)

The 2-simplex and 3-simplex problems recently described are two specific cases of the problem of mapping an \( m \)-orthotope to an standard \( m \)-simplex while matching their volumes, at least asymptotically. The standard \( m \)-simplex, is an \( m \)-dimensional polytope where its facets define a convex hull, with one vertex having all of its adjacent facets orthogonal one to each other. This work focuses on the discrete voxelized/cell-like version, denoted \( \Delta^m_n \), which is composed of unitary volumetric elements with location \( x = (x_1, x_2, \ldots, x_m) \) that satisfy

$$\Delta^m_n \equiv \{ x \in \mathbb{Z}^m | 0 \leq x_i \leq n \land x_1 + x_2 + \ldots x_m \leq n \}.$$  \hspace{1cm} (3)
where the absolute Manhattan distance from any element $x$ to the orthogonal corner of the $m$-simplex cannot be greater than $n$. Figure 4 illustrates the geometry of discrete orthogonal simplices for $m = 1, 2, 3$.

The expression for the volume of an $m$-simplex corresponds to the simplicial polytopic numbers, which are defined as

$$V(\Delta_n^m) = \frac{(n + m - 1)!}{m!} = \frac{n(n + 1)(n + 2)...(n + m - 1)}{m!}$$

where an induction [36] can prove its formula, based on the fact that the volume of $\Delta_n^{m+1}$ is the sum of the volumes of $n$ stacked $m$-simplices of lengths $[1, 2, 3, ..., n]$, i.e.,

$$V(\Delta_{n+1}^{m+1}) = \sum_{i=1}^{n} V(\Delta_n^m)$$

and when combined with the properties of sums of binomial coefficients, leads to formula [4].

The parallel space efficiency challenge arises when trying to process an $m$-simplex domain with GPU computing, as its parallel programming model can only manage orthotopes of threads, namely $\Pi^m$, and not arbitrary shapes as simplices. Although the bounding-box approach will work and is simple to implement, its main problem is that in the limit $n \to \infty$, the fraction of extra parallel space of $V(\Pi_n^m)$ that lies outside of the $m$-simplex approaches to

$$\alpha(\Pi, \Delta_n^m) = \lim_{n \to \infty} \frac{V(\Pi_n^m)}{V(\Delta_n^m)} - 1 = m! - 1$$

thus being inefficient in terms of parallel space as all the extra thread-blocks must be discarded at run-time through conditional or arithmetic filters. The implication is a performance penalty at the mapping stage of any kernel execution. Moreover, these extra thread-blocks can produce a negative impact on the performance of multiple concurrent kernels as they could make use of hardware resources that could be potentially available to other kernels.

One solution, used in the past years, is to enumerate all threads or thread-blocks of a given orthotope in sequence, and then use this enumeration as an indexing scheme. Such approach allows to formulate a map [22, 23, 24] of the form $\lambda : \mathbb{Z}^1 \mapsto \mathbb{Z}^m$ with $V(\Pi_n^m) = O(V(\Delta_n^m))$. Although $\lambda(\omega)$ can be computable by elementary and arithmetic functions, it also requires the computation of $m$-th order roots as part of the solution of an $m$-th order equation, which limits the range of problem size $n$ due to potential numerical error in the mapping of thread coordinates. Moreover, the method is restricted to $m \leq 4$ as no general analytical solutions are available for polynomials of $m > 4$. Finding a different kind of map, free of these problems, would improve the state of the art regarding GPU thread maps for standard simplex domains.

The limitations described can be overcomed, in great part, by taking advantage of the dimensionality available in the parallel space, which was an aspect not exploited by the enumeration principle proposed in the past [24]. Although parallel GPU spaces cannot have a geometry different from an orthotope, they still are topologically equivalent to an $m$-simplex and a self-similar set can match the desired domain. Therefore, finding an homeomorphism of the form $\mathcal{H} : \mathbb{Z}^m \mapsto \mathbb{Z}^m$ would produce a map, efficient in parallel space and with zero dimensional distance which would free it from the computation of expensive $m$-th roots.

### 3. Related Work

One of the earliest works in mapping the GPU efficiently onto a 2-simplex is by Jung et. al. [37] in 2008, whom proposed a rectangular box strategy (RB) for accessing and storing a triangular matrix (upper or lower) more efficiently on GPU. Data structures become practically half the size with respect to classical methods based on the full matrix. The strategy was originally intended to modify the data space (i.e., the matrix), however one can apply the same concept to the thread space. One disadvantage of RB is that it only works on 2-simplices, i.e., applying the idea at higher dimensions produces unnecessary threads.

Ries et. al. contributed with a parallel GPU method for the triangular matrix inversion [32]. The authors identified that the parallel space indeed can be improved by using a recursive partition (REC) of the grid, based on a divide and conquer strategy. The approach takes $O(n^3)$ time by doing a balanced partition of the structure, from the orthogonal point to the diagonal. Q. Avril et. al. proposed a GPU mapping function for collision detection based on the properties a upper-triangular map [29]. The map is a thread-space function $u(x) \mapsto (a, b)$, where $\alpha$ is the linear index of a thread and the pair $(a, b)$ is a unique two-dimensional coordinate in the upper triangular matrix. Since the map works in thread space, the map is accurate only in the range $n \in [0, 3000]$ for a simplex of side length $n$.

Navarro et. al. proposed a block-space map, denoted $\mathcal{H}(\omega)$, for 2-simplices and 3-simplices [22, 23, 24], based 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 and 3-simplices. By being a block-space map, the authors report correct mapping coordinates up to problems of $n = 62900$ for the 2-simplex case and up to $n = 1546$ for the 3-simplex case. Beyond these problem sizes, the map requires 64-bit floating point square roots (FP64) which penalize the speedup.

This work explores a new concept of map, denoted $\mathcal{H}(\omega)$, that costs $O(1)$ time and is as fast as $\lambda(\omega)$ and RB methods,
while being free of numerical precision dimensional limitations. The proposed map $\mathcal{H}(\omega)$ preserves locality at block-level and only uses integer arithmetic and bit-shift operations for its computation, making it an exact map by design.

4. Formulation of $\mathcal{H}(\omega)$

Note: for practical purposes, the concept of standard simplex will be just referred to as simplex.

The formulation of $\mathcal{H}(\omega)$ focuses on the special cases $m = 2, 3$, where the mapping can also be supported by illustrations, although it is not limited to just those dimensions, as explained later in Section 6. Also, the analysis first assumes $n$ as a power of two, i.e., $n = 2^k$ with $k \in \mathbb{Z}_+$, and later in sub-section 4.2 expands to the general case of $n \in \mathbb{Z}_+$. The case when $m = 1$ is not considered in the analysis as both the orthotope and simplex match in geometry thus no special map is required.

For the rest of the manuscript, the concept of an efficient map will be based on the following definition:

**Definition 4.1.** An efficient map is a function $\mathcal{H}(\omega) : \mathbb{Z}_+^k \mapsto \mathbb{Z}_+^n$ that can map any thread coordinate $\omega_{i_1,i_2,...,i_k}$ from a parallel space $\Pi^k$ of volume $V(\Pi^k) = V(\Delta_2^n) = o(n^n)$, onto a unique location in $\Delta_2^n$, using $O(1)$ arithmetic or bitwise operations, excluding explicit transcendental operations and roots.

The following sub-sections formulate maps for $\Delta_2^2$, $\Delta_2^n$ and analyze their efficiency according to Definition 4.1.

4.1. Mapping to 2-Simplices

Given a 2-simplex, denoted as $\Delta_2^n$, with $n$ being its side length, its space $V(\Delta_2^n)$ is given by the triangular numbers

$$V(\Delta_2^n) = \frac{n(n+1)}{2}. \quad (7)$$

Each location of the 2-simplex is a pair $(a, b)$, where $a + b \leq n$ must be satisfied. The goal is to find an orthotope $\Pi^2$ with an asymptotic volume of $O(V(\Delta_2^n))$ and an efficient map $\mathcal{H}(\omega)$, that with a constant number of arithmetic operations, can assign threads of $\Pi^2$ to unique locations in $\Delta_2^n$.

**Lemma 4.1.** Given a standard 2-simplex $\Delta_2^n$, there exists a set $S_{2}^{\pi}$ of self-similar regular orthotopes where $V(S_{2}^{\pi}) = V(\Delta_2^n)$.

**Proof.** Let $S_{2}^{\pi}$ be a set of self-similar regular orthotopes where its total volume is defined by the recurrence

$$V(S_{2}^{\pi}) = \left(\frac{n}{2}\right)^2 + 2V(S_{2}^{\pi/2}), \quad (8)$$

where the largest regular orthotope of $S_{2}^{\pi}$ has dimensions $\frac{n}{2} \times \frac{n}{2}$ and the initial condition is $V(S_{2}^{\pi/2}) = 0$. The expansion of Eq. (8) produces the geometric series

$$V(S_{2}^{\pi}) = 2^0\left(\frac{n}{2}\right)^2 + 2^1\left(\frac{n}{2}\right)^2 + \cdots + 2^{\log_2(n)-1}\left(\frac{n}{2^{\log_2(n)}}\right)^2 = \frac{n^2}{2} \sum_{i=1}^{\log_2(n)} \left(\frac{1}{2^{i}}\right), \quad (9)$$

where its reduction via $\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1}$, results in

$$V(S_{2}^{\pi}) = \frac{n^2}{2} \left(1 + \sum_{i=0}^{\log_2(n)} \left(\frac{1}{2^{i}}\right)\right) = \frac{n^2}{2} \left(1 + \frac{(1/2)^{\log_2(n)+1} - 1}{1/2 - 1}\right) = \frac{n(n-1)}{2} = \Delta_2^n. \quad (10)$$

Given that its possible to represent a 2-simplex as a set of regular orthotopes, it is now necessary to group them as a grid because of the GPU programming model.

**Lemma 4.2.** A set $S_{2}^{\pi}$ can be organized into a single irregular super-orthotope $\Pi_2^{\pi}$ of dimensions $\frac{n}{2} \times n - 1$.

**Proof.** In the expansion of formula (9), each term can be thought as the contribution of a sub-set of $2^{n-1}$ regular orthotopes of equal size, each one of dimensions $\frac{n}{2} \times \frac{n}{2}$, where $i$ is the $i$-th level of recursion. When considering whole sub-sets, we have that its space is $2^{n-1}$ $\frac{n}{2} \times \frac{n}{2}$, therefore it is possible to stack these sub-sets to form a super-orthotope $\Pi_2^{\pi}$ of $n(2n-1)$ where its height is the result of the geometric series $\sum_{i=1}^{\log_2(n)} \frac{n}{2^i} = n - 1$.

Figure 5 left, illustrates the super-orthotope and how all sub-sets stack into different layers.

**Theorem 4.3.** Given a discrete orthogonal 2-simplex $\Delta_2^n$, there exists a map $\mathcal{H} : \mathbb{Z}_+^{2^n} \mapsto \mathbb{Z}_+^{2^n}$ from $\Pi_2^{\pi}$ to $\Delta_2^n$ that is efficient by Definition 4.1.

**Proof.** The proof proceeds by construction. Let $S_{2}^{\pi}$ be a set of self-similar orthotopes which by Lemma 4.1 matches $\Delta_2^n$ in space. By Lemma 4.2, all regular orthotopes of $S_{2}^{\pi}$ can be organized to form an irregular super-orthotope $\Pi_2^{\pi}$, which corresponds to a thread-block, i.e., not a thread but a group of threads. Lemma 4.2 also establishes that sub-orthotopes share
the same stack height level only if they are equal in size. Such property allows to compute the stack height level \( b \) of any discrete element of \( \Pi^2 \) as
\[
b = 2^{\left\lfloor \log_2 \omega_i \right\rfloor}
\] (14)
as well as the following parameter
\[
q = \left\lfloor \frac{\omega_i}{b} \right\rfloor
\] (15)
which is the index of the orthotope for which the pair \( \omega \) belongs to, at level \( b \), from left to right. Finally, arithmetic operations between \( \omega, b, q \) produce the map
\[
\mathcal{H}(\omega) = (\omega_i + qb, \omega_i + 2qb)
\] (16)
which is efficient by Definition 4.1 as \( V(\Pi^2_0) = V(\Delta^2_{n-1}) + o(n^2) \) satisfies using \( O(1) \) arithmetic operations excluding roots. □

The map \( \mathcal{H}(\omega) \) from Eq. (16) is a considerable improvement over previous proposed maps based on the enumeration principle as it does not use square roots. Regarding the number of sequential steps, it is also more efficient than the recursive map by Ries et al which was graphically similar but required \( O(\log_2(n)) \) time. Additionally, since blocks have a constant size of \( \rho^2 \ll n \) (with \( \rho \) the number of threads in each dimension of the block), any extra number of threads only occurs at the diagonals or lower boundary, which is asymptotically \( \leq 2n\rho^2 \in o(n^2) \).

The computation of \( \mathcal{H}(\omega) \) requires a constant number of arithmetic operations and two transcendental functions in \( \mathbb{Z}_+ \); \( \{\log_2(\omega_i)\} \) and \( 2^{\left\lfloor \log_2(\omega_i) \right\rfloor} \). Function \( \{\log_2(y)\} \) can be computed fast on GPU by using the relation
\[
\{\log_2(\omega_i)\} = b - clz(\omega_i)
\] (17)
where \( b \) is the number of bits of the word and \( clz \) counts the number of leading zero bits of \( y \), which is an efficient hardware-level instruction available in GPUs. The exponential \( 2^{\left\lfloor \log_2(\omega_i) \right\rfloor} \) is computed as
\[
2^{\left\lfloor \log_2(\omega_i) \right\rfloor} = 2 << (b - clz(\omega_i)).
\] (18)

Considering that the two transcendental functions can be computed using bit-level operations, and that the parameters are reused by registers, it is expected that the parallel space improvement from \( O(n^2) \) to \( O(n) \) unnecessary threads can indeed result in a significant performance improvement, which for the case of 2-simplices can be up to \( 2 \times \) [22] [24]. Moreover, since no square roots are required, \( \mathcal{H}(\omega) \) has the potential to be faster and more precise than previous mapping techniques based on the enumeration principle [29] [22] [24].

4.2. Extending \( \mathcal{H}(\omega) \) to the general case \( n \in \mathbb{Z}_+ \)

The analysis of \( \mathcal{H}(\omega) \) has assumed problems with sizes of the form \( n = 2^i \). For general \( n \), we present three extensions, with the third being the chosen and most efficient one.

1. **Approach \( n \) from above**: build a single orthotope \( \Pi^2_0 \), where \( n’ = 2^{\left\lfloor \log_2(n) \right\rfloor} \) and filter out the threads outside the domain. This approach keeps simplicity at the cost of adding extra threads. The main disadvantage is for values of \( n = 2^k + c \) when \( c \) is a small number, i.e., \( c \ll 2^{k+1} - 2^k \), as it can generate up to \( 4x \) the number of necessary threads.

2. **Approach \( n \) from below**: apply a set of super-orthotopes \( \Pi^2_1, \Pi^2_2, \Pi^2_3, \ldots \), where \( n_i = \log_2(n - \sum_{k=1}^{i-1} n_k) \) for \( i \geq 2 \), \( n_1 = \left\lfloor \log_2(n) \right\rfloor \), plus a set of additional maps for the sub-regions that remain un-mapped below each simplex, at each level. This approach, while not adding extra threads, adds complexity in the algorithm and sequential steps.

3. **Concurrent Trapezoids**: This approach combines the ideas of the other two; it extends the mapped geometry to a trapezoid instead of a simplex, and assumes kernel concurrency, which is a feature that exists in actual GPU architectures. The idea is to launch, concurrently, a set of special orthotopes that map onto trapezoids in the data domain. The set of concurrent trapezoids follows the principle of approaching \( n \) from below, except for the smallest one that approaches the remaining size from above. This last trapezoid triggers when \( 2^{\left\lfloor \log_2(n) \right\rfloor} - n \in T \), with \( T \) an arbitrary threshold value that serves as a mechanism to limit the size of the trapezoids set.

The **Concurrent Trapezoids** approach is the most efficient one in terms of map efficiency because of the following three features: (1) it works concurrently, keeping the cost of \( O(1) \) time unchanged, (2) the set of trapezoids will be small as the triggering condition for the last trapezoid has a high probability to occur at an early stage (only on rare occasions will produce a set of \( \log_2(n) \) elements which is the worst case) when using an adequate threshold, and (3) a small modification in the expression of \( \mathcal{H}(\omega) \) makes the map reusable even when the mapped domain is now a trapezoid instead of a simplex. Figure 6 illustrates the idea behind the concurrent trapezoids approach.

![Figure 6: The concurrent trapezoids approach for a simplex of \( n = 27 \) with threshold of \( T = 1 \).](image-url)

Under the **Concurrent Trapezoids** approach, \( \mathcal{H}(\omega) \) receives three extra parameters that are common for all threads; (1) an
offset $\delta(x, y)$ to specify the starting coordinate of each trapezoid map, (2) the orthotope height of the simplex region with the $B_1$ block included, namely $h_1$, and (3) the height of $B_2$, namely $h_2$. With these three extra parameters, all threads can map to their corresponding locations in the whole trapezoid. The expression for $\mathcal{H}(\omega)$ acting on trapezoids is now:

$$\mathcal{H}(\omega) = (\delta_x + \omega_x + qb + kG_x, \delta_y + \omega_y - kh_2 + 2qb)$$  \hspace{1cm} (19)

where $k = (h_1 - \omega_x) >> 31$ is a 1|0 signed 32-bit integer mask that is zero when threads are in the simplex or $B_1$ region (in thread space), otherwise is the value $k = 1$ to move the $B_2$ region. A reduction via geometric series produces $\delta(x, y)$

$$V \left( \frac{n}{2} \right) = \frac{n(n + 1)(n + 2)}{6}. \hspace{1cm} (20)$$

The formulation extends from the 2D one, now packing a set of 3-orthotopes in a self-similar organization.

**Lemma 4.4.** Given a discrete orthogonal 3-simplex $\Delta^3_n$, a set $S^3_n$ of self-similar regular orthotopes produces a space of $V(S^3_n) = V(\Delta^3_{n-1})$.

**Proof.** Let $S^3_n$ be a set of self-similar regular orthotopes where its total volume is defined by the recurrence

$$V(S^3_n) = \left( \frac{n}{2} \right)^3 + 2V(S^3_{n/2}) = \frac{n^3}{2} \sum_{i=1}^{\log_2(n)} \left( \frac{1}{4} \right)^i. \hspace{1cm} (21)$$

A reduction via geometric series produces

$$V(S^3_n) = \frac{n^3 - n}{6} = V(\Delta^3_{n-1}). \hspace{1cm} (22)$$

The diagonal plane is not considered, thus $V(S^3_n) = V(\Delta^3_{n-1})$. The way how $S^3_n$ (red cubes) is placed onto $\Delta^3_n$ (gray region) is illustrated in Figure 7.

Although the extra red regions can be re-used by mapping them onto their upper tetrahedron-like regions, recursively, packing the entire set into a super-orthotope of regular base $\frac{n}{2} \times \frac{n}{2}$ (which is the idea behind $\mathcal{H}$ in 2-simplices) will introduce a fraction of extra thread-space.

**Lemma 4.5.** Given a set $S^3_n$ with a half-size sub-division scheme, packing $S^3_n$ into a super-orthotope of regular base $\frac{n}{2} \times \frac{n}{2}$ introduces a fraction of extra thread space.

**Proof.** The largest sub-orthotope of the set $S^3_n$ has dimensions $\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2}$, therefore two of the dimensions of the super-orthotope $\Pi^3_{a,b,c}$ will necessarily be $a = \frac{n}{2}, b = \frac{n}{2}$ acting as the regular base. The third dimension will stack the rest of the sub-sets of regular orthotopes. At recursion level $k$, $S^3_n$ provides $2^k$ regular orthotopes of size $\left( \frac{n}{2} \right)^3$, increasing the third dimension of $\Pi^3_{a,b,c}$ by $\frac{n}{2}$. This extra space induced on $\Pi^3_{a,b,c}$ by the height increase is a stack of size $\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2}$. When $k > 1$ the elements provided by $S^3_n$ are not sufficient to fill the $k$-th stack, i.e.,

$$\sum_{i=1}^{2^{k-1}} \left( \frac{n}{2} \right)^3 = \frac{n^3}{2^{2k-1}} \leq \frac{n^3}{2^{2k+2}}. \hspace{1cm} (23)$$

Therefore, the super-orthotope $\Pi^3_{a,b,c}$ containing $S^3_n$ will necessarily have empty spaces at each stack level.

By Lemma 4.5, $\mathcal{H}(\omega)$ will not be efficient by Definition 4.1 as it introduces an extra space of $O(n^m)$. Nonetheless, such a map may still be useful in practice as long as the extra space is a small fraction of $\Delta^3$. A convenient way to pack the sub-orthotopes in thread space is to place the first major cube below and begin the stacking process horizontally from level $k = 2$. Such an approach produces a super-orthotope $\Pi^3_{a,b,c}$ of dimensions $\left( \frac{n}{2} \right) \times \left( \frac{n}{2} \right) \times \frac{3(n-1)}{4}$ for $a, b, c$, respectively (see Figure 8 left).

![Figure 8: A practical packing scheme to map $S^3_n$ onto $\Delta^3$.

While the packing approach is not perfectly efficient, in the infinite limit of $n$, the extra fraction of space of $\Pi^3$ over $\Delta^3$, defined as $\alpha(\Pi^3, \Delta^3)$, is

$$\alpha(\Pi^3, \Delta^3) = \lim_{n \to \infty} \frac{V(\Pi^3)}{V(\Delta^3)} - 1 = \lim_{n \to \infty} \frac{3n^2(n-1)}{16} - 1 = \frac{1}{8} \hspace{1cm} (24)$$
which is 12.5% of $V(\Delta_3^1)$. Such amount of thread space constitutes a small fraction compared to a bounding-box strategy that surrounds the tetrahedron and generates practically 600% of the space of $\Delta_3^1$. Therefore, while $\Pi_3^1$ is not an efficient super-orthotope, there is still potential performance improvement that can be exploited by GPUs assuming a time efficient $O(1)$ cost map $\mathcal{H}(\omega)$ can exist.

**Theorem 4.6.** For any standard discrete 3-simplex $\Delta_3^n$, there exists a time efficient map $\mathcal{H} : \mathbb{Z}_n^3 \rightarrow \mathbb{Z}_{n+1}^3$ from $\Pi_{n/2,n/2,\ldots,n/2}^1$ to $\Delta_3^1$ that requires only $O(1)$ arithmetic or transcendental operations.

**Proof.** Analogous to Theorem (4.3), the origin of $\Pi_{n,b,c}^3$ is assumed at the upper-left corner (see Figure 8) and the origin of $\Delta_3^1$ at the bottom right corner, with the axes aligned to its orthogonal sides. By Lemma 4.4 and 4.5, there exists a super-orthotope that contains the set $\Sigma_{p,q}$ with each sub-set of equally sized regular orthotopes organized by stack levels, except for the first sub-orthotope that is treated as a special case for being displaced from the stacking pattern.

Let $h(\omega)$ be an auxiliary block-space mapping of the single sub-orthotope of size $(n/2)^3$ onto the center of the tetrahedron, defined as

$$h(\omega) = \omega + (0, \frac{n}{2}, 0).$$

(25)

For the $b,q$ parameters, they keep the definitions from Theorem 4.3 as the mapping layout allows projection onto the plane with an additional consideration for the extra spaces that fall out of $\Delta_3^1$ (the red spaces from Figure 7), for which the position is transposed in $x,y$ and complemented in $z$. With these settings, the combination of the parameters $\omega, q, b$ and the auxiliary function $h(\omega)$ allow the formulation of

$$\mathcal{H}(\omega) = \left\{ \begin{array}{ll}
h(\omega), & \omega \in cube_2^3 \land \omega \in \Delta_3^{n+1}, \\
(\omega_3 + q, \omega, + 2q, \omega, - n/2), & \omega \in \Delta_3^{n+1}, \\
(b_1 + 2q) - \omega_3, 2b(1 + q) - \omega_2, 2b - \omega_2 + \frac{n}{2}, & \omega \notin \Delta_3^{n+1}, \\
(26) & \omega \notin \Delta_3^{n+1},
\end{array} \right.$$

which is only time efficient, i.e., $T(h(\omega)) + T(\mathcal{H}(\omega)) = O(1)$.

It is possible to make $\mathcal{H}(\omega)$ fully efficient, i.e., to produce 0% of extra volume and satisfy condition $V(\Pi_3^1) = V(\Delta_3^1) = o(n^m)$ of Definition 4.1 by including concurrent GPU kernel executions in the analysis. Such approach would allow to create multiple parallel spaces, one for each stack level of equally sized sub-orthotopes, without needing to pack them into a single super-orthotope. The number of simultaneous parallel spaces would be limited by the maximum number of concurrent kernels supported by the GPU, which as of 2022 is up to 128 concurrent kernels. In practice, such limit is sufficiently large for many applications as the number of stack levels is in the order of $O(\log_2(n))$. Furthermore, the map works in block-space which acts as an extra factor for reaching larger problem sizes. The next Section presents experimental performance results for $\mathcal{H}(\omega)$ using 2-simplex and 3-simplex tests, running on different GPUs. The results are compared to state of the art approaches.

5. Experimental Results

The proposed map $\mathcal{H}$, as well as state of the art approaches, are measured and compared against the bounding box approach. This gives a total of four approaches to be compared:

- **RB**: Rectangular box [37]
- **L**: Lambda map [3, 24]
- **DP**: Dynamic Parallelism
- **H**: New proposed map

The DP approach is based on CUDA’s Dynamic Parallelism. All four approaches are illustrated in Figure 9.

5.1. Experimental Design

Four different computational patterns are tested on 2-simplex and 3-simplex domains. These are listed in Table 1.

| Test | 2-simplex | 3-simplex |
|------|-----------|-----------|
| 1) [MAP] Only map | | * |
| 2) [ACCUM] + Accumulation | * | * |
| 3) EDM Euclidean Distance Matrix | * | |
| 4) [CA] Cellular Automaton | * | |

These tests serve to know what is the impact of application work in the overall speedup through an improved mapping. The first test, MAP, is a GPU kernel with no application work, i.e., just the thread mapping stage. It uses CUDA volatile variables in order to prevent the compiler to optimize the mapping stage as it is not used afterwards. This kernel should reflect the theoretical speedup as it is mostly mapping work. The ACCUM test is a kernel where each mapped thread adds +1 to its corresponding data-element in the simplex domain. It is a memory-bound kernel as it does 1 read, 1 write and only 1 addition on each data element. The EDM test computes the Euclidean distance matrix, that is the distance $d(p_i,p_j)$ at each data-element of the simplex with location $i,j$. This test is a medium intensity application with more arithmetic work than the accumulation test, as it computes a square root for each data-element. Lastly, the cellular automaton computes John Conway’s game of life [31] on a simplex structure, with periodic boundary conditions for the 2-simplex, and free boundary conditions (fixed dead cells on the boundary) for the 3-simplex case. This test is a more intense memory-bound application which serves as a case where improvements to the mapping stage could have a lesser impact on performance.

The average running time as well as the power consumption are measured during kernel execution. Sufficient repeats are employed for each test in order to reach a statistical standard error of less than 1%. Then, plots of speedup (over the bounding-box approach) and energy efficiency are computed from these values.

The hardware used for all tests is detailed in Table 5.1. The TITAN GPUs share the same system while the A100 GPU is from a DGX A100 node of the Patagón supercomputer of Austral University of Chile [38].
5.2. Speedup and Energy Efficiency for 2-simplices

Figure 10 presents the speedup of each mapping technique over the bounding box approach, under different tests (columns) and GPUs (rows). In the MAP test (first column), $\mathcal{H}$ and RB reach the theoretical speedup of 2x over the Bounding box approach, for all GPUs. With the TITAN RTX, the $\lambda$ map performance gets reduced to 0.5x of speedup once $n \geq 32154$. The reason is because that GPU has very few FP64 units, and at that problem size (with block size 16 x 16 which is the fastest) the map needs to switch from FP32 to FP64 precision in the computation of a square root, otherwise it generates incorrect coordinates. The DP approach also fails to reach the maximum speedup for all GPUs. Instead it reaches up to 1.25x of speedup in the best case using the A100 GPU. In the ACCUM test (second column), the maximum speedup is $\sim 1.25x$, with $\mathcal{H}$ within the fastest maps. The RB map is also within the fastest, and DP becomes competitive as well once the problem size is large enough. In the case of $\lambda$, it stays with the fastest maps as long as the GPU has sufficient FP64 units, otherwise it underperforms with ~ 0.4 of speedup as in the TITAN RTX. For the EDM2D test (third column), speedup is unstable at small problem sizes, but stabilizes at medium to high values. Here, both $\mathcal{H}$ and RB stay competitive through all the range of $n$, followed by $\lambda$ (except when running in the TITAN RTX GPU and DP). For the CA2D test (fourth column), the maximum speedup is 1.25x and is achieved by all approaches using the A100 GPU. On the TITAN RTX GPU the fastest approaches are $\mathcal{H}$ and RB, followed by DP and lastly $\lambda$. It is worth noticing that in this case the performance of $\lambda$ did not drop because the fastest version used thread blocks of size 32 x 32, allowing the FP32 square root to be precise for the whole range. In the TITAN V GPU all approaches reach a speedup of $\sim 1.15x$ for large $n$.

Figure 11 shows the power consumption in the case of the A100 GPU. As a global observation, the duration of all curves are in the same proportion of the speedups reported above. For the MAP test the peak power is near $\sim 100W$, reached by all approaches except for DP which reaches a peak slightly above 90W. For the ACCUM test, $\mathcal{H}$ has an intermediate consumption of up to $\sim 180W$, while $\lambda$ is the most power consuming one with a peak of $\sim 210W$. In the EDM2D test, $\lambda$ reaches the Max Thermal Design Power (TDP) of the A100 GPU, which is 400W. Here $\mathcal{H}$ is positioned in the middle again, being less consuming than RB, and slightly more than DP. The reference Bounding approach oscillates between 270W and 300W. In the last test, the CA2D, $\mathcal{H}$ and RB reach the same peak power consumption while DP is the less consuming one.

Figure 12 presents the energy efficiency of all approaches running on the A100 GPU. The bar charts represent the energy efficiency using the metric of number of elements per second (EPS) per Watt (EPS/W), while the numerical labels at the top of each bar show the total energy used for that particular run. In terms of energy efficiency, which is in log-Y scale, $\mathcal{H}$ stays on average as one of the most energy efficient approaches, shared by RB and DP. On the other hand, the $\lambda$ approach is the less energy efficient one.
5.3. Speedup and Energy Efficiency for 3-simplices

In the case of 3-simplices, the RB approach is discarded because it cannot be extended efficiently to 3D, as \( n/2 \times n/2 \times n \) orthotope is double the necessary space and the arithmetic operations involved would require further research as it will be more complex. The map \( \lambda \) is also discarded, because by definition it extends to another 3-simplex where \( i < j < k \) [24], and not the standard one where \( i + j + k \leq n \) satisfies. Therefore 3-simplex tests consider \( \mathcal{H} \) and two possible approaches using Dynamic Parallelism (DP). The first one is the default extension of the DP approach for 2-simplices, while the second one implements the hinge idea of \( \mathcal{H} \) as in Figure 7 now using DP.

Figure 13 presents the speedup of \( \mathcal{H} \) and the DP approaches over BB, using different 3-simplex tests and GPUs. In the MAP test (first column) \( \mathcal{H} \) approaches to \( \sim 6 \times \) of speedup, matching the theoretical expected value. For the DP variants, the \( DP_{\text{large}} \) version exhibits a higher speedup than the vanilla variant. On the ACCUM test (second column) \( \mathcal{H} \) is the fastest map again, but this time the speedup reaches up to \( \sim 3 \times \). The DP variants...
show similar performance for the TITAN GPUs, and favors the hinge variant in the A100 GPU. Lastly, on the CA3D test (third column), $\mathcal{H}$ reaches a maximum speedup near $\sim 1.5x$ followed by the DP approaches.

Figure 13 presents the power consumption time series for all tests running on the A100 GPU. It is worth noticing that the power consumption behavior for the MAP test puts $DP$ and $\mathcal{H}$ as the approaches with less power peak. On the ACCUM and CA3D test $\mathcal{H}$ increases its peak power consumption surpassing the other approaches, but is the first one in making the power consumption curve to go down. Figure 13 presents the energy efficiency of all approaches running on the A100 GPU. This plots show more easily which map is the most energy efficient in terms of elements per second per Watt. Overall $\mathcal{H}$ stands as the most energy efficient one along with $DP_{\text{hinged}}$.

6. Considerations for extending $\mathcal{H}$ to higher dimensions

The formulation of $\mathcal{H}$ for the 2-simplex and 3-simplex followed specific designs for their corresponding dimensions. Although the map takes constant time in both cases, it is worth noticing that for the 3-simplex it was necessary to introduce 12% of extra parallel volume in order to fit the set $S_n^m$ on $\Pi_n$ as a single-pass map, unless multiple parallel spaces are assumed in the model. When generalizing the approach to $m$-simplices, it is important to first verify if $V(S_n^m) = V(\Delta_n^m) + o(n^m)$ satisfies or not, and if not then analyze how much extra space is introduced. In the general case, the volume of a set $S_n^m$ of self-similar regular orthotopes is

$$V(S_n^m) = (rn)^m + \beta V(S_{rn}^m) \left( \frac{\log_{10}(n)}{m} \right)$$

where $r$ is the scaling factor and $\beta$ the arity or multiplicity of the recursion. Applying the geometric series, the expression becomes

$$V(S_n^m) = (rn)^m \left( \frac{(rn)^\beta \log_{10}(n) - 1}{\beta r^m - 1} \right)$$

One can verify that setting $r = 1/2$ and $\beta = 2$ leads to equations 13 and 22 for $m = 2, 3$, respectively.
The parallel space efficiency of a self-similar set \( S^m_n \) of orthotopes with parameters \( r = 1/2 \) and \( \beta = 2 \) is efficient for \( m = 2, 3 \) and factorially inefficient in \( m \) for \( m \geq 4 \).

Proof. For the general case of \( m \), the self-similar strategy of using \( r = 1/2, \beta = 2 \) produces a fraction of extra volume of

\[
\alpha(S, \Delta^m_n) = \lim_{m \to \infty} \frac{m^m - m^{m-1}}{(m-1)^{m-1}} - 1 = \frac{m!}{2^m} - 2 = O(m!)
\]

The efficient maps \( \mathcal{H}(\omega) \) proposed for \( \Delta^2_n \) and \( \Delta^3_n \) in fact take advantage of the early behavior of the extra space function, which remains near zero. For \( m \geq 4 \) the factorial behavior of the extra volume dominates the cost. \( \square \)

For example, when \( m = 4 \) the volume is

\[
V(S^m_n) = \frac{n^4 - n}{14} > \frac{(n-1)n(n+1)(n+2)}{24}
\]

and \( \alpha(S^m_n, \Delta^m_n) \) approaches \( 5/7 \) of \( \Delta^m_n \). For \( m = 5 \) and \( m = 7 \) it approaches to \( 3x \) and \( 39x \) the volume of \( \Delta^m_n \), and so on.

Theorem 6.2. Finding an efficient self-similar set \( S^m_n \) for \( \Delta^m_n \) is an optimization problem.

Proof. By Lemma 6.1 the \( r = 1/2, \beta = 2 \) approach is only efficient for \( m = 2, m = 3 \). For all the rest of the cases, finding an efficient set requires to find parameters \( r, \beta \) such that it minimizes the target function \( V(S^m_n) - V(\Delta^m_n) \). The constraints are \( \beta > 1, 1/r > \beta \) as well as \( 1/r, \beta \in \mathbb{Z}^+ \).

Additional considerations include that \( \beta^{\log_{1/(\beta)}(m)} \) should not grow too fast as it has an impact on what is the initial \( n_0 \) for which \( V(S^m_n) \geq V(\Delta^m_n) \) holds. As an example, a value of \( r = 1/(m^{1/m}) \) produces the required \( m! \), making \( \beta \) a free parameter to be adjusted with \( \beta \geq 2 \). Choosing \( \beta = 2 \) provides a set \( S^m_n \) that covers \( \Delta^m_n \) from a certain \( n \geq n_0 \), where \( n_0 \) is a value that increases with \( m \). It is possible to bring \( n_0 \) closer to the origin by increasing \( \beta \), however the extra volume increases as well, presenting a trade-off. What is interesting is that from \( n \geq n_0 \), the parallel space is practically \( m! \) times more efficient than a bounding box approach, presenting a great potential for transforming this space improvement into a performance one.

7. Insights for Leveraging Tensor/Ray-Tracing Cores

Since 2018, GPU architectures have begun introducing two new types of ASICs (Application Specific Integrated Circuits) onto the chip, i) Tensor cores (TC) and ii) Ray-tracing cores (RTC) to further speedup Deep Learning and Real-time graphics applications, respectively. Leveraging the multiple TCs and RTCs that sit on a modern GPU depends on the application and how well it can reformulate its computation in terms of TC and RTC operations.

7.1. Leveraging Tensor Core Units

The tensor core exposes the matrix multiply accumulate (MMA) operation: \( D_{\text{non}} = A_{\text{max}} \times B_{\text{max}} + C_{\text{max}} \) with different \( m \times n \times k \) formats\(^3\) to choose from depending on the numerical precision: \( 16 \times 16 \times 16 \) or \( 8 \times 32 \times 16 \) for FP16, \( 16 \times 16 \times 8 \) for TF32, or \( 8 \times 8 \times 4 \) for FP64, among others. Each MMA operation requires a dedicated warp of threads (32 threads), therefore parallelism granularity is at the warp level. Adapting \( \mathcal{H} \) to TCs requires the reformulation of its computation, first by proposing a matrix layout to place its variables/ constants and second by producing the desired computation from the fused multiply add (FMA) operations that take place in each row-column product

\(^3\)Internally, the GPU schedules how the operation is decomposed into even smaller matrices, such as \( 4 \times 4 \), to be processed by each TC.
of the MMA. In the case of $\mathcal{H}$, given the map is at block-level, an efficient adaptation should try to obtain several block coordinates for each MMA executed. An efficient adaptation of $\mathcal{H}$ to TCS would employ a format where the product $A \times B$ acts on non-square matrices, as the FMA operations are relatively short in length. The TF32 format $16 \times 16 \times 8$ can be an attractive solution if more precision is required, or the FP16 format $8 \times 32 \times 16$ to increase the number of simultaneous block coordinates, although at less precision. In any case, the layout is to place the constants of Eq. (16) or (19) as rows in $A$, and the $x,y$ block-related inputs for $\mathcal{H}$ as pairs of columns in $B$. Eq. (32) expresses the mentioned TC layout acting for two blocks (distinguished by the super-indices in $B$) in the case of 2-simplices, under a format of $m \times n \times k = 4 \times 4 \times 2$ just for simplicity.

$$D = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \\ b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} \begin{bmatrix} b_1^1 & b_1^2 \\ b_2^1 & b_2^2 \\ b_3^1 & b_3^2 \\ b_4^1 & b_4^2 \end{bmatrix} + \begin{bmatrix} c_{x_1} & c_{y_1} & 0 & 0 \\ c_{x_2} & c_{y_2} & 0 & 0 \\ 0 & 0 & c_{x_3} & c_{y_3} \\ 0 & 0 & c_{x_4} & c_{y_4} \end{bmatrix} \tag{32}$$

The are four constants $a_1, a_2, a_3, a_4$, and the product $A \times B$ generates a block coordinate for each block twice. This is to allow threads to add their local coordinate on top of it, through the addition of matrix $C$. The resulting matrix is $D$ which contains the definitive positions per-thread to access the simplex domain. It is worth mentioning that this multi-block approach can be further improved by the use of the recent thread block cluster construct, which allows to group several CUDA blocks and synchronize them to share data, such as matrix fragments.

### 7.2. Leveraging Ray Tracing Cores

Ray Tracing (RT) cores are hardware-implemented pipelines that accelerate the process of querying the collision of a ray with a set of geometric primitives, such as triangles in 3D space. Internally, RT cores employ a hardware-implemented bounding volume hierarchy (BVH) to handle its irregular memory access patterns on the BVH more efficiently. Although the standard application of RT cores is to accelerate the generation of photo-realistic computer graphics from a 3D scene, it is still possible to use the ray-triangle intersection query as a fast tool to solve other kinds of problems, such as thread mapping.

In the case of thread mapping to a simplex domain, RT cores can contribute with a fast hardware accelerated framework that works similar to Dynamic Parallelism, to exploit a non-boxed data domain during execution. Moreover, given that RT cores work in a fully dynamic context, it opens the possibility of mapping threads to fully dynamic changing domains, dropping the static restriction assumed in $\mathcal{H}$ and making it a general purpose mapping. A possible idea to leverage RT cores for simplex domains is to load the simplex as a voxel type geometry in 3D space, and launch waves of rays at different levels of thickness. This would generate a dynamic discovery process where the first waves of rays do a coarse-grained exploration, useful for discarding the 3D space with no simplex data. The process would gradually refine rays until they become voxel grained and access the data elements. The only limitation is that the hardware BVH was designed for 3D space.

### 8. Discussion and Conclusions

This work presented a new map, $\mathcal{H}$, for mapping blocks of threads onto $m$-simplex domains. The map was entirely formulated for the 2-simplex and 3-simplex cases, obtaining theoretical upper bounds on speedup and thread usage efficiency over a standard bounding-box approach. It was also experimentally evaluated and compared to other state of the art approaches, showing that $\mathcal{H}$ is competitive with respect to the other alternative approaches, reaching up to $2x$ of speedup just for the mapping stage, and between $1.1x \sim 1.3x$ of speedup under different tests. For 3-simplices it was overall the fastest map, approaching the theoretical $6x$ for the mapping stage, and between $1.5x \sim 3.2x$ under different tests. When comparing $\mathcal{H}$ with the state of the art approaches, we note that it is the only map that does exhibits a favorable performance and numerical precision in both 2D and 3D cases. Moreover, $\mathcal{H}$ is potentially extendable to higher dimensions offering up to $m!x$ more efficiency. This generalization to $m$-simplices presents a challenge though, as obtaining an optimal set of orthotopes with minimal extra volume becomes an optimization problem where the scaling and replication parameters, $r, \beta$ respectively, produce a trade-off between extra space and the starting value $n_0$ from which the mapping can take place. Knowing what parameters are the optimal for building a self-similar set of orthotopes for any $m$-simplex, as well as find general methods for packing $S^n_m$ into super-orthotope are indeed interesting questions that deserve further study.

In terms of power consumption, $\mathcal{H}$ showed that although it achieves high power consumption during execution, it compensates by having a shorter duration, leading to reach among the highest values of elements per second per Watt ($\text{EPS/W}$). This makes $\mathcal{H}$ energy efficient, as for the same amount of energy it can process more elements than other approaches.

This work also analyzed the possibility of leveraging both Tensor Cores (TC) and Ray Tracing Cores (RTC) that are otherwise unused during a regular CUDA application. A preliminary analysis shows that it is indeed feasible to implement mappings using these special purpose cores. In fact, TC can serve to speedup $\mathcal{H}$ directly, while RTC units show that it can offer a dynamic mapping approach more general than $\mathcal{H}$. Future research can work on implementing such approaches considering the modern 2022+ GPU architectures, and measure what is the empirical acceleration that these special purpose cores can provide to the thread mapping process.

As a conclusion, this work has shown that $\mathcal{H}$ is an scalable and energy efficient solution that can be of great interest researchers working on applications that need to execute long simulations on simplex structures, such as PDEs or Cellular Automata.

### Acknowledgment

This project was supported by FONDECYT grant #1221357, the Temporal research group and the Patagón Supercomputer from Universidad Austral de Chile.
References

[1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, Gpu computing, Proceedings of the IEEE 96 (5) (2008) 879–899.
[2] J. Nickolls, W. J. Dally, The gpu computing era, IEEE Micro 30 (2) (2010) 56–69.
[3] 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.
[4] Nvidia Corporation, Nvidia CUDA C Programming Guide (2016).
[5] Khronos OpenCL Working Group, The OpenCL Specification, version 1.0.29 (8 December 2008).
[6] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, X. Shen, Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu SIGPLAN Not. 48 (8) (2013) 57–68. doi:10.1145/2517327.2442523
[7] W. W. Fung, I. Sham, G. Yuan, T. M. Aamodt, Dynamic warp formation and scheduling for efficient gpu control flow, in: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) 2007, 2007, pp. 407–420. doi:10.1109/Micro.2007.30
[8] L. V. Lucas Vespa, Unraveling the divergence of gpu threads, in: 2018 International Conference on Computational Science and Computational Intelligence (CSCI), 2018, pp. 1398–1403. doi:10.1109/CSCI46756.2018.00270
[9] X. Mei, X. Chu, Dissecting gpu memory hierarchy through microbenchmarking, IEEE Transactions on Parallel and Distributed Systems 28 (1) (2017) 72–86. doi:10.1109/TPDS.2016.2549523
[10] J. Holewinski, L.-N. Pouchet, P. Sadayappan, High-performance code generation for stencil computations on gpu architectures in: Proceedings of the 26th ACM International Conference on Supercomputing, ICS ’12, Association for Computing Machinery, New York, NY, USA, 2012, p. 311–320. doi:10.1145/2304576.2304619
[11] https://doi.org/10.1145/2304576.2304619
[12] https://doi.org/10.1145/2304576.2304619
[13] M. Harris, et al., Optimizing parallel reduction in cuda, Nvidia developer technology 2 (4) (2007) 70.
[14] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, R. Vega, Gpu tensor cores for fast arithmetic reductions, IEEE Transactions on Parallel and Distributed Systems 32 (1) (2020) 72–84.
[15] E. Herrmann, B. Raffin, F. Faure, T. Gautier, J. Allard, Multi-gpu and multi-cpu parallelization for interactive physics simulations, in: European Conference on Parallel Processing, Springer, 2010, pp. 235–246.
[16] O. Yadan, K. Adams, Y. Taigman, M. Ranzato, Multi-gpu training of convnets, arXiv preprint arXiv:1312.5853 (2013).
[17] J. A. Stuart, J. D. Owens, Multi-gpu mapreduce on gpu clusters, in: 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE, 2011, pp. 1068–1079.
[18] A. Soma, X. Cheng, E. D’azevedo, K. Won, S. Tomov, Optimizing the fast fourier transform using mixed precision on tensor core hardware, in: 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCCW), IEEE, 2018, pp. 3–7.
[19] F. A. Quezada, C. A. Navarro, N. Hitschfeld, B. Bustos, Squeeze: Efficient compact fractals for tensor core gpus, Future Generation Computer Systems 135 (2022) 10–19.
[20] C. A. Navarro, F. A. Quezada, N. Hitschfeld, R. Vega, B. Bustos, Efficient gpu thread mapping on embedded 2d fractals, Future Generation Computer Systems 113 (2020) 158–169.
[21] Y. Zhu, Ruan: accelerating neighbor search using hardware ray tracing, in: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022, pp. 76–79.
[22] S. Zellmann, M. Weier, I. Wald, Accelerating force-directed graph drawing with rt cores, in: 2020 IEEE Visualization Conference (VIS), IEEE, 2020, pp. 96–100.
[23] N. Morrical, I. Wald, W. Usher, V. Pascucci, Accelerating unstructured mesh point location with rt cores, IEEE Transactions on Visualization and Computer Graphics (2020).
[24] 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, 6th IEEE International Symposium on Cybersecurity and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC-SS/ICESS 2014, Paris, France, August 20-22, 2014, 2014, pp. 375–382. doi:10.1109/HPC.2014.64
[25] 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, Valparaso, Chile, October 10-14, 2016, 2016.
[26] C. A. Navarro, M. Vernier, B. Bustos, N. Hitschfeld, Competitiveness of a non-linear block-space gpu thread map for simplex domains, IEEE Transactions on Parallel and Distributed Systems 29 (12) (2018) 2728–2741. doi:10.1109/TPDS.2018.2849705
[27] C. A. Navarro, B. Bustos, N. Hitschfeld, Analysis of a self-similar gpu thread map for data-parallel m-simplex domains, in: 2019 International Conference on High Performance Computing & Simulation (HPCS), 2019, pp. 1002–1010. doi:10.1109/HPCSS.2019.9188081
[28] D. Man, K. Uda, H. Ueyama, Y. Ito, K. Nakano, Implementations of parallel computation of euclidean distance map in multicores processors and gpus, in: Networking and Computing (ICNC), 2010 First International Conference on, 2010, pp. 120–127. doi:10.1109/IC-NC.2010.26
[29] Q. Li, V. Kecman, R. Salman, A chunking method for euclidean distance matrix calculation on large dataset using multi-gpu, in: Proceedings of the 2010 Ninth International Conference on Machine Learning and Applications, ICMLA ’10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 208–213.
[30] D. Man, K. Uda, Y. Ito, K. Nakano, A gpu implementation of computing euclidean distance map with efficient memory access, in: Proceedings of the 2012 Second International Conference on Networking and Computing, ICNC ‘12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 68–76.
[31] Q. Avril, V. Gouranton, B. Arnoldi, Fast collision culling in large-scale environments using gpu mapping function, in: EGPGV, 2012, pp. 71–80.
[32] J. Kepner, J. Gilbert, Graph Algorithms in the Language of Linear Algebra, Software, Environments, Tools, Society for Industrial and Applied Mathematics, 2011.
[33] M. Gardner, The fantastic combinations of John Conway’s new solitaire game “life”, Scientific American 223 (1970) 120–123.
[34] 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, SC ’09, ACM, New York, NY, USA, 2009, pp. 9:1–9:10.
[35] R. Yokota, L. A. Barba, Fast n-body simulations on GPUs, CoRR abs/1108.5815 (2011).
[36] J. Bédorf, E. Gaburov, S. Portegies Zwart, A sparse octree gravitational n-body code that runs entirely on the GPU processor J. Comput. Phys. 231 (7) (2012) 2825–2839. doi:10.1016/j.jcp.2011.12.024
[37] https://doi.org/10.1016/j.jcp.2011.12.024
[38] L. Ivanov, The n-body problem throughout the computer science curriculum, J. Comput. Sci. Coll. 22 (6) (2007) 43–52.
[39] https://doi.org/10.1109/TIFS.2017.2710571
[40] J. Costello, On the number of points in regular discrete simplex (corresp.), IEEE Transactions on Information Theory 17 (2) (1971) 211–212. doi:10.1109/TIT.1971.1054599
[41] J. H. Jung, D. P. O’Leary, Exploiting structure of symmetric or triangular matrices on a gpu, Tech. rep., University of Maryland (2008).
[42] A. U. of Chile, Patagon supercomputer (2021). https://patagon.uach.cl