

**VOXEL-BASED OFFSETTING AT HIGH
RESOLUTION WITH TUNABLE SPEED AND
PRECISION USING HYBRID DYNAMIC
TREES**

A Dissertation
Presented to
The Academic Faculty

By

Mohammad Moazzem Hossain

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Computer Science



College of Computing
Georgia Institute of Technology
December 2016

Copyright © 2016 by Mohammad Moazzem Hossain

VOXEL-BASED OFFSETTING AT HIGH
RESOLUTION WITH TUNABLE SPEED AND
PRECISION USING HYBRID DYNAMIC
TREES

Approved by:

Dr. Richard W. Vuduc, Advisor
Committee Chair & Associate Professor
School of Computational Science
and Engineering
Georgia Institute of Technology

Dr. Thomas R. Kurfess, Co-advisor
Professor
The George W. Woodruff School of
Mechanical Engineering
Georgia Institute of Technology

Dr. Jarek Rossignac
Professor
School of Interactive Computing
Georgia Institute of Technology

Dr. Thomas M. Tucker
Prinicipal
Tucker Innovations Inc.

Dr. Jeffrey S. Young
College of Computing
Georgia Institute of Technology

Date Approved: 05 October 2016

This PhD dissertation is dedicated to

My Parents

Ayesha & Hafar

My Parents-in-Law

Jesmin & Moaxxem

My Soulmate

Ishita

ACKNOWLEDGMENT

First and foremost, I would like to express my deepest gratitude to my advisor Prof. Richard Vuduc for his continuous support and guidance over the years. He taught me how to communicate ideas more effectively, gave the freedom to focus on challenging yet impactful problems that I felt deeply interested with. Without his supervision, this work would not have been a reality. I'm grateful to my co-advisor Prof. Thomas Kurfess who introduced me to the world of advanced manufacturing. He is one of the most energetic, enthusiastic, and optimistic person I've ever met. He always made me feel as though my work was important emphasizing the over-arching goal that targets bringing the high-performance computing in the forefront of computational fabrication.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Jarek Rossignac, Dr. Jeffrey Young, and Dr. Thomas Tucker for their insightful comments and valuable inputs that not only improved the overall quality of the thesis, but also helped me to broaden the view of my research from various perspectives. Particularly, without the continuous assistance and collaboration from Dr. Thomas Tucker this work would not have been possible. Since the first day of the project, he has been always very supportive feeding with positive motivation to get deeper into the right research direction.

I am grateful to all my colleagues and friends at Georgia Tech who have helped me over these years. I thank my collaborators from the PMRC lab: Dr. Chandra Nath, Roby Lynn, Zhengkai Wu, James Collins, and Brandon Royal. I learned a lot about advanced CNC manufacturing from Brandon and Roby. I thank Dmytro Konobrytskyi who shared the CAD models used in this research. A special thanks to Dean Cooper from Tucker Innovations Inc. who helped to deal with intricacy of some part of the codebase that helped me getting the broader understanding on the

peripheral implementation.

I thank my peers from HPC Garage who helped setting me in the new lab: Jee Whan Choi, Kenneth Czechowski, Marat Dukhan, Piyush Sao, and Jiajia Li. I thank my prior colleagues from the MARS lab: Jen-Cheng Huang, Sungkap Yeo, Tzu-Wei Lin, Lifeng Nai, Dong Hyuk Woo, and Dean Lewis. Throughout the past years, I shared many hard times with Jen-Cheng Huang and Sungkap Yeo, and our mutual encouragements only bolstered the confidence to achieve respective milestones. I would like to thank Prof. Sung Kyu Lim and Dr. Hsien-Hsin Lee for giving me the opportunity to work on the 3D-MAPS project, and also all members of the project, Dong Hyuk Woo, Dean Lewis, Tzu-Wei Lin, Guan hao Shen, Krit Athikulwongse, Michael Healy, Moongon Jung, Dae Hyun Kim, Young-Joon Lee, and Xin Zhao. While designing a 3D-stacked many-core processor from scratch and personally leading the role of software tool-chain development in the first ever 3D processor from academia, I learned a great deal of processor design and optimization from my teammates that directly and indirectly helped me in many aspects.

Most importantly, none of this would have been possible without the love and patience of my family. My immediate family to whom this dissertation is dedicated to, has been a constant source of inspiration and support throughout all these years. I would like to express my heart-felt gratitude to my family. My wife Ishita is an unshakable source of support, understanding, and love. She helped me make it through the occasionally grueling all nighters and other challenging parts of these over last six years. Without her unwavering patience and compromises this dissertation would not be a reality. I am grateful to her for being a part of this journey at the happy and the tough times.

Above all, I am grateful to the One for bestowing me the guidance, intellect, patience, persistence, courage, and essentially everything virtuous that might exist in me.

TABLE OF CONTENTS

ACKNOWLEDGMENT	iv
LIST OF TABLES	ix
LIST OF FIGURES	xi
SUMMARY	xv
CHAPTER 1 INTRODUCTION	1
1.1 Problem Context	1
1.2 The Case for Tunable High-Performance Voxel Offsetting	6
1.3 Research Contributions	11
1.4 Outline	13
CHAPTER 2 LITERATURE REVIEW	16
2.1 Survey of Basic Voxel Data Structures	16
2.1.1 Uniform Grid	16
2.1.2 Two-Level or Tiled Grid	17
2.1.3 Octree and Generalized N^3 -tree	19
2.1.4 Hash Table Representation	23
2.1.5 Hybrid Representation	24
2.2 Compressed Voxel Representation through Geometric Redundancy	26
CHAPTER 3 THE DESIGN, IMPLEMENTATION AND ANALYSIS OF THE HYBRID DYNAMIC TREE	31
3.1 HDT Fundamentals	31
3.1.1 Basic Scheme	31
3.1.2 Building Blocks	33
3.1.3 Design Principles	35
3.1.4 Sturcutre Storage: Memory Pools	38
3.1.5 Storage per Active Voxel	40
3.2 HDT Construction	43
3.2.1 Triangle Mapping	45
3.2.2 HDT Branching	46
3.2.3 Leaf Processing	47
3.3 Experiments and Analysis	49
3.3.1 HDT Benchmarking	49
3.3.2 Storage Comparisons	51
3.3.3 Discussion on HDT Construction	52
3.3.4 Discussion on GPU Speedups	54
3.4 Tunable Hybrid Dynamic Tree	56
3.4.1 Impact of the Root Grid Dimension	56

3.4.2	Impact of the Branching Factor	58
3.4.3	Impact of the Leaf Grid Dimension	60
CHAPTER 4 VOXEL OFFSETTING USING HYBRID DYNAMIC TREE		
		64
4.1	Overview on Offsetting Techniques	64
4.1.1	Exact Offsetting	65
4.1.2	Approximate Offsetting	66
4.2	Convolution and Mathematical Morphology	70
4.2.1	Convolution	70
4.2.2	Mathematical Morphology	71
4.2.3	Dilation and Erosion	72
4.3	HDT Offsetting using Mathematical Morphology	75
4.3.1	Constructing the Skeleton of the Offset HDT	76
4.3.2	Constructing the Offset HDT through Morphological Filtering	79
4.4	Experimental Results and Analysis	83
4.4.1	Dilation Performance Evaluations	83
4.4.2	Erosion Performance Evaluations	88
4.4.3	Computational Complexity of HDT Offsetting	89
4.4.4	Cross-Platform Scalability	89
4.5	Comparison with Prior Studies	92
4.5.1	Multi-Core CPU Offsetting on Distance Field based Representation	92
4.5.2	Comparative Performance Analysis with GPU-Accelerated 3D Convolution	95
CHAPTER 5 TUNABLE VOXEL OFFSETTING WITH HDT PARAMETERS		
		98
5.1	HDT Offsetting with Tunable Root Grid	100
5.1.1	The Impact of the Root Grid on HDT Offsetting	100
5.1.2	HDT Dilations with Variable Root Grid Size	100
5.1.3	Analysis on the CUDA Profiler Statistics	102
5.2	HDT Offsetting with Tunable Node Branching	105
5.2.1	HDT Dilations with Node Branching of 4	106
5.2.2	HDT Dilations with Node Branching of 8	108
5.2.3	Analysis on the CUDA Profiler Statistics	108
5.3	HDT Offsetting with Tunable Leaf Grid	110
5.3.1	Parallelism, Redundant Computation and Storage Trade-Offs	110
5.3.2	HDT Dilations with Leaf Grid of 8^3	111
CHAPTER 6 VOXEL OFFSETTING WITH TUNABLE SPEED AND PRECISION		
		116
6.1	Offsetting Error Measurement	116
6.1.1	Errors in Voxel Offsetting	116
6.1.2	Nearest Neighbor Search in the HDT	118

6.1.3	Evaluations on Voxel Offsetting Error	124
6.2	Speed and Precision Trade-Offs in Voxel Offsetting	126
6.2.1	High-Performance Offsetting with Kernel Decomposition . . .	126
6.2.2	Impact of Kernel Decomposition on Offsetting Complexity .	129
6.2.3	Experimentations on Accuracy and Performance Trade-Offs .	130
CHAPTER 7 MULTI-GPU VOXEL OFFSETTING		135
7.1	Scale-Out Voxel Offsetting on Multiple GPUs	135
7.2	Multi-GPU Implementation of Morphological Filtering	137
7.3	Evaluations on Multi-GPU Voxel Offsetting	142
7.3.1	Impact of Load Distribution Policy	142
7.3.2	Impact of Kernel Execution Alternatives	144
7.3.3	Performance Comparisons between Single and Dual GPU Setups	145
CHAPTER 8 CONCLUSIONS AND FUTURE WORKS		148
8.1	Conclusions	148
8.2	Future Work	151
REFERENCES		153

LIST OF TABLES

Table 3.1	Topology storage per leaf grid for different values of active branching factor.	41
Table 3.2	Geometric statistics of CAD models: Head and Dragon.	50
Table 3.3	Geometric statistics of CAD models: Turbine and Candle Holder.	50
Table 3.4	Total storage per active voxel with different leaf dimensions using the compact storage for cell components.	63
Table 4.1	Dilation results: Head and Dragon.	84
Table 4.2	Dilation results: Turbine and Candle Holder.	85
Table 4.3	Boundary points of the structuring element at different offset distances.	87
Table 4.4	Erosion results: Head, Dragon, Turbine and Candle Holder.	88
Table 4.5	Key architectural specifications of NVIDIA GTX 780Ti and Titan.	91
Table 4.6	Time comparison for dilation of Stanford Buddha [1]	93
Table 4.7	Test configurations for 2% dilation of the diagonal length at different resolutions.	94
Table 4.8	Comparisons of the target problem configurations and respective GPU platforms.	96
Table 5.1	Comparisons of different voxel data representations.	110
Table 5.2	Dilation times for the <i>Head</i> model with different leaf sizes.	112
Table 5.3	Dilation times for the <i>Dragon</i> model with different leaf sizes.	112
Table 5.4	Dilation times for the <i>Turbine</i> model with different leaf sizes.	113
Table 5.5	Dilation times for the <i>Candle Holder</i> model with different leaf sizes.	113
Table 6.1	Boundary points of the structuring element at different offset distances.	129
Table 7.1	Comparative throughput and memory bandwidth of NVIDIA GTX 780Ti and Titan.	142
Table 7.2	Dilation times comparisons for the <i>Head</i> and <i>Dragon</i> models at 2048 ³ resolution.	146

Table 7.3	Dilation times comparisons for the <i>Turbine</i> and <i>Candle Holder</i> models at 2048^3 resolution.	146
-----------	---	-----

LIST OF FIGURES

Figure 1.1 Surface Offsetting in CNC toolpath planning.	7
Figure 2.1 Uniform grid representation.	16
Figure 2.2 Illustration of a tiled grid structure.	18
Figure 2.3 Illustration of a quadtree.	20
Figure 2.4 Boolean operations on two quadtree structures (illustration source: [36]).	20
Figure 2.5 Efficient sparse voxel octree representation [63].	22
Figure 2.6 Voxel hashing data structure by Nießner et al. [81]. Here, the voxel world is conceptually partitioned into an infinite uniform grid. Using the hash function, integer world coordinates are mapped to hash buckets, which store a small array of pointers to regular grid voxel blocks. Each voxel block contains an 8^3 grid of signed distance field (SDF) values. When information for the red block gets added, a collision appears which is resolved by using the second element in the hash bucket.	23
Figure 2.7 High-resolution VDB created by converting polygonal model from <i>How To Train Your Dragon</i> . Images are courtesy of <i>DreamWorks Animation</i>	26
Figure 2.8 Reducing a sparse voxel tree, illustrated using a binary tree, instead of an octree, for clarity. a) The original tree. b) Non-unique leaves are reduced. c) Now, there are non-unique nodes in the level above the leaves. These are reduced, creating non-unique nodes in the level above this. d) The process proceeds until the final directed acyclic graph is deduced (illustration source: [57]).	29
Figure 3.1 Illustration of HDT representation. Each intersecting root cell represents the root of an octree that is adaptively refined until the desired resolution is reached. Each octree (i.e., root cell) in this demonstration represents $(16 \times 2^3)^3$, i.e., 128^3 sub-volume, and hence the entire HDT represents a volumetric resolution of $16^3 \times 128^3$, i.e., 2048^3 . In this example, two small grids and a three-level octree jointly represent a volumetric resolution of 2048^3 that is much shallower than a hierarchy of eleven with a standard octree.	32
Figure 3.2 Compact 192 bits representation of an HDT cell; state (2), depth (30), descendant pointer (64), and cell origin (3×32).	34

Figure 3.3 Distribution of the CUDA threads in the HDT leaf grid space during the HDT construction and volume processing on GPUs.	37
Figure 3.4 Storage implementations for HDT structure on GPU memory. Demonstration shows a sample root cell (C) branching into eight octree cells – all allocated in a contiguous block in the element pool. Two of these first-level octree cells (i.e., C^1 and C^5) get branched into eight octree cells in deeper hierarchy. Out of the eight octants of cell C^1 , four cells (C_1^1 , C_2^1 , C_5^1 and C_7^1) are in <i>boundary</i> state, and each is mapped to a leaf grid in the leaf pool. Similarly, for the cell C^5 three descendants (C_1^5 , C_3^5 and C_5^5) are mapped to respective leaf grids in the leaf pool. Except the purple (branch) and blue (boundary) blocks in the element pool, the rest blocks (green) are either full or empty, and not partitioned.	39
Figure 3.5 Storage analysis of active voxels in HDT. In this example, a balanced octree partitioning (with $K = 4$) for a given root cell is illustrated. Here, a leaf grid is shown as a block of $4 \times 4 \times 4$ voxels instead of general convention (i.e $16 \times 16 \times 16$) for better visualization.	41
Figure 3.6 Demonstration of a sample 3D model in HDT (source: [2]).	43
Figure 3.7 HDT construction on a 2D cross-section of a sample 3D model (source: [2]).	44
Figure 3.8 HDT triangles mapping.	45
Figure 3.9 Hierarchical branching in HDT construction.	46
Figure 3.10 Leaf grid processing in a CUDA thread block.	47
Figure 3.11 Measurements on HDT construction time.	52
Figure 3.12 GPU-speedups of HDT construction at $8192 \times 8192 \times 8192$ resolution.	54
Figure 3.13 Impact of different root dimensions at $8192 \times 8192 \times 8192$ resolution.	57
Figure 3.14 Impact of different branching factors at $8192 \times 8192 \times 8192$ resolution.	59
Figure 3.15 Impact of different leaf dimensions at $8192 \times 8192 \times 8192$ resolution.	61
Figure 4.1 Offsetting a triangle in 2D.	65
Figure 4.2 3×3 morphology kernel.	72
Figure 4.3 Illustration of morphological dilation and erosion [3].	73
Figure 4.4 The case of convolution filtering on a discretized triangle to dilate by one voxel.	74

Figure 4.5 3D Illustration of offsetting. The dilated portion is colored in green, and the original component is colored in red. The surfaces of these two HDTs maintain the given offset distance.	75
Figure 4.6 Offsetting illustration on HDT.	76
Figure 4.7 Dilations of Candle Holder at 2048^3 resolution.	83
Figure 4.8 Erosions of Candle Holder at 2048^3 resolution.	84
Figure 4.9 Dilation times comparison between GTX 780Ti and GTX Titan.	90
Figure 4.10 Surface dilation of a Buddha model (polygonal mesh comprising 1.1 million triangles).	92
Figure 5.1 Impact of different root grid size on 60 voxels dilations at $4096 \times 4096 \times 4096$ resolution.	101
Figure 5.2 Impact of different root grid size on 60 voxels dilations at $2048 \times 2048 \times 2048$ resolution.	102
Figure 5.3 Impact on CUDA warp issue efficiency and global memory transactions with different sizes of root grid at 60 voxels dilations for the Candle Holder model.	103
Figure 5.4 Impact of different branching factors on 40 voxels dilations at $4096 \times 4096 \times 4096$ resolution.	105
Figure 5.5 Impact of different branching factors on 60 voxels dilations at $4096 \times 4096 \times 4096$ resolution.	106
Figure 5.6 Impact of different branching factors on 80 voxels dilations at $4096 \times 4096 \times 4096$ resolution.	107
Figure 5.7 Impact on CUDA warp issue efficiency and global memory transactions with different branching factors at 60 voxels dilations for the Head Model.	109
Figure 6.1 Offsetting error for different dilation distances.	124
Figure 6.2 Offsetting error for different dilation distances.	125
Figure 6.3 Visualization of the impact of filter size in morphological voxel-offsetting.	128
Figure 6.4 Successive offsetting performance at 2048^3 resolution.	131
Figure 6.5 Successive offsetting error.	132
Figure 6.6 Successive offsetting performance at 4096^3 resolution.	134

Figure 7.1 The schematic design of the components in multi-GPU offset implementation.	138
Figure 7.2 Illustration of how the leaf grids in skeletal HDT are processed on a dual GPU setup.	140
Figure 7.3 Impact of load distributions on dual GPU setup.	143
Figure 7.4 Comparisons between single threaded and dual threaded kernel executions on dual GPU setup.	145

SUMMARY

The primary objective of this dissertation is to design and implement a fast and effective computational method of computing offsets for a solid object represented volumetrically. Our approach uses a data structure known as a hybrid dynamic tree (HDT), and the dissertation research contributes the development and analysis of a tunable offset computation algorithm that trade-offs speed, storage efficiency, and geometric accuracy. By “fast” we mean using scalable parallel algorithms that are well-suited to modern massively parallel processors, especially the ubiquitous graphics processing unit (GPU) co-processors available in all modern systems; by “compact” we mean a storage-efficient voxel representation with a low memory footprint; and by “accuracy” we mean the quality of the computed result. Offsetting is a key geometric operation that lies at the heart of various applications, such as toolpath planning in CNC machining and accessible space analysis in robotics, among others. This work is about the application of offsetting in multi-axis CNC toolpath planning, where the offset surface controls the center of the ball-end tool cutters along a collision-free trajectory. At the target resolution of 4096^3 (69 billion voxels in a naive dense representation), we can compute large-scale offsets in minutes, match or beat the number of bits of the representation compared to state-of-the-art alternatives, and experimentally characterize any trade-offs among speed, storage, and accuracy.

In this dissertation, we present a set of practical approaches to efficiently compute the offsets of a voxel model represented in resolutions up to 4096^3 . Using the HDT as the underlying data structure leads naturally to a compact representation. However, the challenge in developing a high-performance offsetting implementation is choosing an optimal configuration of the HDT parameters. These parameters not only govern the memory footprint of the voxelized representation of the solid, but also control the parallel code execution efficiency on GPUs.

This dissertation presents a tunable offsetting method based on mathematical morphology that works on the HDT. Using a convolution-based technique to compute the offsets of a volumetrically represented solid makes the computational complexity tunable, essentially through changing the size of the morphological filters. In general, a smaller kernel results in faster execution due to lower number of neighboring elements in the spherical stencil; however the geometric accuracy scales inversely with the increasing number of morphology operations. Hence, our research analyzes the impact of the decomposition of a large offset distance into a series of offsetting with smaller distances. Such a tuning of the size of the filter kernel further makes our implementation platform-adaptive in that it allows the algorithm to be adjusted depending on the peak performance of the underlying hardware.

Besides trading some geometric accuracy for improved performance, we explore algorithmic speedup through a load-balanced implementation of morphological voxel offsetting across multiple GPUs. Dynamic load-balancing needs estimation of the cost of the entire computation and consideration of the disparity in the peak throughput (often measured in FLOPS) of the underlying GPU devices. In contrast to a boundary-representation, such as a polygonal mesh, voxel-based computation is more amenable to load-balanced execution due to its homogeneous.

While performance and accuracy are the key metrics for the presented offset method, the storage required to represent the HDT at high-resolution is equally important, particularly if the application needs to manipulate multiple HDTs simultaneously in the GPU buffers. This is essential in our target computer-aided manufacturing (CAM) applications, where a sequence of HDTs are managed concurrently in the GPU memory so that the CNC-codes can be ‘played back’ to visualize the impact of multiple iterations of toolpath with respective tool selections. This also allows ‘rolling back’ an offset operation to a prior state in the toolpath planning system, which helps the design of automated manufacturability analysis. Thus, a

final contribution of this dissertation is to explore the HDT configuration space to optimize storage, and to efficiently construct the HDT at extreme resolutions with high speed-ups on GPU.

CHAPTER 1

INTRODUCTION

1.1 Problem Context

In recent years, advancement in 3D printing has liberated digital manufacturing—fabricating arbitrarily complex free-form shapes is no longer constrained by the manufacturing process. While the layer-by-layer *additive* 3D printing has revolutionized the landscape of rapid prototyping, the technology is limited in terms of the materials that can be used, the finishing quality that can be achieved, and relatively slow printing speeds; thus limiting applicability in many areas. By contrast, classical *subtractive* manufacturing techniques, such as CNC milling, turning and casting are complementary in many aspects. Computer Numerically Controlled (CNC) machines can be considered as one of the most important innovations in the manufacturing industry in the 20th century that helped morphing the manual hand-driven products manufacturing process to the contemporary automated production systems. CNC based manufacturing technologies are applicable for a large variety of materials, demonstrate high-quality surface finishes, and result into parts with robust structural properties. However, these subtractive processes are severely limited in terms of the shapes that can be produced.

Conceptually, CNC based subtractive manufacturing can be described as an incremental material removal process from a solid workpiece where a rotating cutter intersects the surface of the stock to wear away material; thus, layers of surface get exfoliated in multiple iterations, and incrementally shaped to the target geometry. A fundamental limitation to realize the fullest capability of multi-axis CNC machines for the fabrication of free-form geometric shapes arises from the conventional approaches of machine program (called *G-code*) generation that control the tool movements. Through computer aided design (CAD) tools an engineer visualizes and designs the

object to be manufactured, and a CNC controller sequences a motor driven machine tool through specific path trajectories to create the desired object. Linking these two is a process where a toolpath is generated through computer-aided manufacturing (CAM) software.

As the subtractive process differs from the additive technique, for manufacturability the shape has to be a *height field* so that each of the machining surfaces is accessible to the tool cutter [44]. Broadly speaking, there are two general approaches adopted to respect the constraint on shapes to be a height field. The shapes either have to be designed in a way so that they constitute height fields, or they have to be divided into multiple components that fulfill the constraint individually. For the majority of the industrial products the former approach is adopted, where the shapes respect the height field constraint. By contrast, rapid prototyping of free-form shapes using CNC requires specialized machining expertise to segment a given design into multiple components, and the toolpath for each segment is generated independently. Besides human expertise, it needs tremendous time investment for tuning the path-planning parameters manually to generate collision-free tool trajectories. Though the state-of-the-art CAM packages have significantly streamlined manufacturing pipelines from the CAD to the CNC stage through improved features to decrease the toolpath planning time, yet there is a steep productivity gap. For complex sculptured parts, the toolpath programming time can exceed the actual machining time by a large margin. Such intensive expert labor involvement leads to significantly higher manufacturing cost for CNC based industrial prototyping.

A key reason behind the lack of automation in the CNC path-planning process is the underlying solid geometry representations used in the CAD/CAM software. Typically, CAD interfaces are represented in explicit or parametric form that generally yields high quality in geometric modeling, but challenges the core computations of tool trajectories generation process, such as, surface offsetting, set union or intersection,

etc., to be completely automated. Furthermore, by the nature of how these de-facto solid representations store the core geometric primitives, they are not readily amenable to parallel editing for accelerated generation of the G-code programs. Thus, to simplify CNC programmability for the rapid prototyping of free-form shapes we need an alternative form of solid representation. To bridge the above productivity gap between these complementary manufacturing technologies, namely, CNC machining and 3D printing, the adoption of a voxel-based discretized solid representation can greatly simplify the CNC toolpath planning to fabricate organic free-form shapes [97, 60].

Using a regularly-sampled discrete data representation, such as 3D grid, the voxel processing becomes embarrassingly parallel and well suited for massively parallel computing hardware, such as graphics-processing unit (GPU). However, a cubic scaling of the storage growth and the computations thereon restrict the resolution achievable with a regular grid structure. In practice, a solid that is manufacturable in a CNC process exhibits ‘sparsity’ in its discretized representation. Informally, a discrete voxel data structure is *sparse* if it consists of relatively few boundary voxels (also called *active* voxels). Moreover, as the number of boundary voxels scales with the surface area and the total number of voxels is determined by the volume, the sparsity, *i.e.*, the ratio of the total voxels to the boundary voxels in a uniform grid scales with increasing resolution. Storage of and computation with the non-boundary voxels can be eliminated by a judicious choice of data structure that stores just the active voxels, plus some additional indexing information to indicate which volumetric spaces, *i.e.*, voxel coordinates have been stored.

Thus, the sparsity in the discretized representation can be leveraged to model the solid using an adaptive discrete data representation. In an irregularly-sampled discrete data representation the volumetric space that is uniform, *i.e.*, either completely interior or exterior to the object, can be represented at coarse resolution, and

only the boundary surface gets modeled at the finest resolution. However, the price of a more compact representation in the sparse format compared to that on dense counterpart is more computational overhead per boundary voxels — overheads in the form of extra instructions and, critically, extra memory references, which are often indirect and have an irregular access pattern. While the overhead of extra instructions, particularly that of homogeneous computations across the threads in a GPU thread block, can be well-mitigated exploiting the ever-increasing parallel computing capabilities on GPUs; indirect and irregular memory accesses reduce the overall performance speedup achievable through massive parallelization.

Although, historically the applications of volumetric modeling have been mostly limited in medical imaging and visual effects productions due to the high computational demand for massive 3D data processing, voxelized form of solid representation has been lately adopted both in additive manufacturing [32] and in CNC machining [97, 60]. However, to adopt a voxel based solid representation for CNC manufacturing, there are two major challenges: (1) *how to compactly store the volume at extreme resolution*, and (2) *how to efficiently construct and edit the volume interactively*. These two objectives are contrasting by nature — storage effective representations are computationally inefficient, and vice versa. A suitable data structure thus needs to blend a computation-efficient representation with a compact data organization such that both of the objectives are balanced. This dissertation research uses such a hybrid voxel representation known as a *hybrid dynamic tree* (HDT) [48, 47, 60] that targets fulfilling both of these requirements simultaneously through maximizing the computation efficiency, and minimizing the memory footprint. The name “hybrid dynamic tree” was first introduced by Konobrytskyi [60], who suggested an adaptive hierarchical voxel representation, but did not implement the precise way our work is built upon.

To push the computational capacity beyond the performance limit of multi-core

CPUs, programmers are increasingly opting for parallel implementation. While coarse-grained parallelization on modern CPUs can distribute the computation across tens of processing cores, a graphics-processing unit (GPU) today is equipped with thousands of computing engines. The increased capabilities and flexibility of recent GPU hardware combined with high level GPU programming languages, such as CUDA and OpenCL have unlocked this supercomputing-scale computational power to a desktop workstation. In the context of CNC toolpath automation, prior works have exploited this massive parallelism to accelerate the application [52, 98].

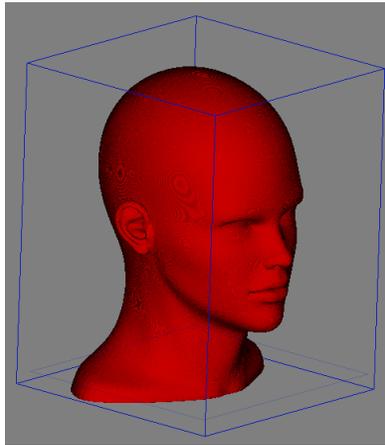
Because the GPU co-processors are optimized for different application areas than general-purpose computation, several important factors need to be considered for developing algorithms and data structures to scale efficiently on these platforms. First and foremost, not all computational problems are suitable for massively parallel computing paradigm. Further, since GPU memory is limited, it is important to maintain the data representation within a strict memory budget. For a specific task, the acceleration on GPU is limited by several factors: (a) the ratio of parallel to sequential part of the algorithm that limits the achievable speedup (formulated by the *Amdahl's Law*), (b) branching diversity, (c) global synchronization requirements, (d) data transfer overhead, (e) the precision in floating point operations (*i.e.*, single- or double-precision), and (f) the ratio of floating point operations to global memory accesses.

This dissertation presents an efficient parallel construction of the HDT data structure, and fast and tunable implementation of voxel offsetting algorithm that heavily leverage the massively parallel computing platform on GPUs to meet the computational demand for accelerated geometric editing of billions of voxels — a case that arises in toolpath planning process with a underlying volumetric representation.

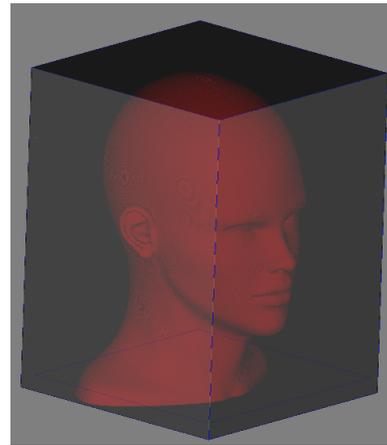
1.2 The Case for Tunable High-Performance Voxel Offsetting

This dissertation deals with the problem of offset generation of free-form objects modeled in the HDT representation that is pertinent to more generic problem, known as offset surface computation. For a given surface, the offset surface is defined as a surface at equal distance from an original surface. It is often used in CNC toolpath planning processes, as the center of a ball-end mill cutting tool may be positioned on the offset surface to move freely without producing overcuts, and yet remaining in contact with the part surface at the outer radius of the ball [60]. By replacing tool contact point trajectory planning with tool center trajectory planning it is possible to eliminate a complicated gouge prevention process and make tool path planning algorithms simpler. Figure 1.1 demonstrates the application of offsetting in CNC toolpath generation process. Here, our target is to make a sample 3D part shown in Fig. 1.1(a). The part is to be manufactured in CNC milling process from a sample cuboid stock, as we see in Fig. 1.1(b). For a given ball-end tool with radius r and “maximum depth of cut” parameter d , first the stock is shrunk by r and the part is expanded by d , as depicted in Figs. 1.1(c) and (d), respectively. Then, a Boolean union operation between these two volumes, shown in Fig. 1.1(e), defines the *contact volume*, i.e., the bounding surface on which the tool center can be positioned without any overcuts and simultaneously respecting the values of r and d . Finally, the resultant stock after the tool paths are applied on the contact volume is presented in Fig. 1.1(f).

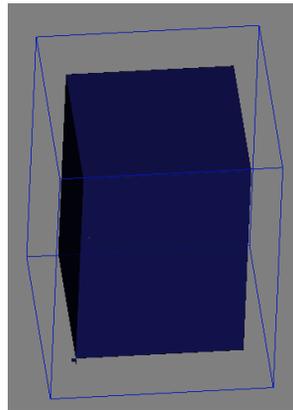
Although the offset surface makes toolpath planning algorithms simpler, finding an offset surface is not a trivial problem for most geometry representations. Special cases such as holes and self-intersections challenge an efficient and robust offset implementation. To construct an offset of HDT represented solids, it can be considered as a composition of geometric primitives, where each primitive defines the offset surface associated with individual voxel on the original surface. Each geometric primitive is delineated by the given offset distance and the center of volumetric space of the voxel,



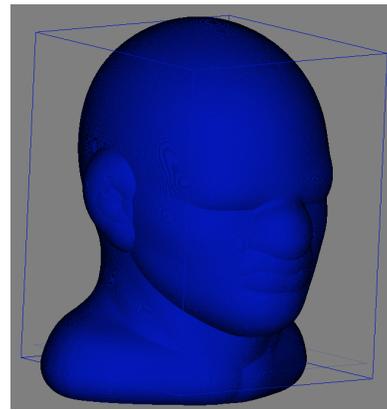
(a) Target part



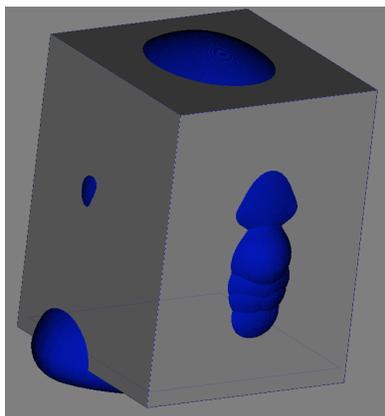
(b) Part shown inside stock



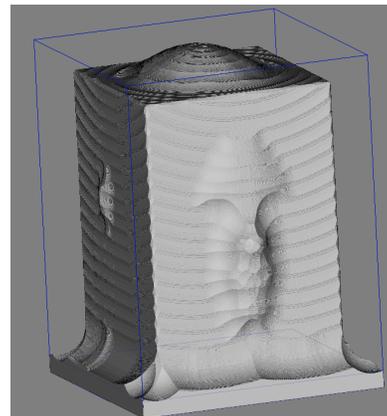
(c) Shrunk stock



(d) Expanded Part



(e) Contact Volume



(f) Stock after toolpass

Figure 1.1: Surface Offsetting in CNC toolpath planning.

which can be computed applying 3D convolutions. Convolution is a fundamental tool in digital image processing that can be understood as a linear filter applied to an input image, parametrized by another image called *kernel*. At each pixel, it outputs a weighted sum of the intensities in the neighborhood where the weights are defined by the kernel.

Algorithms operating on a volumetric representation are generally simple, robust, and insensitive to model complexity by the very nature of its geometry modeling. Furthermore, conceptual designs of such algorithms are fairly comprehensible compared to algorithms operating on boundary representations, such as triangle meshes. However, such design simplicity of complex geometry algorithms comes at the cost of limited spatial resolution in volumetric representation. In most cases, voxels are represented on a regular 3D grid, as such a representation based on uniform indexing is easy to implement. Additionally, most of the volumetric algorithms, such as, convolutional kernels, interpolations, and discretization of differential operators are easier with a uniform sampling scheme [80]. An inherent limitation of uniform sampling is that the memory footprint grows in proportion to the volume of the embedding space. For instance, even with single bit storage per voxel, a uniformly indexed 3D grid at $8192 \times 8192 \times 8192$ resolution requires 64 GB storage.

Large resolutions impose challenging requirements on the computation platform in terms of both performance and memory. To address the ever-increasing performance demand, programmers rely on exposing more and more parallelism for future hardware to exploit. For our target application of 3D convolution on high-resolution voxel representation, superficially, this seems easy: convolution based image processing is enormously data-parallel, which makes it easy to scale on highly parallel hardware, like GPUs. The main challenge is that at the extreme resolutions the size of a uniformly-indexed voxel representation hits the capacity of GPU memory. That requires breaking the data into multiple blocks, and processing the data chunks

sequentially on the accelerators. There are several bottlenecks in this computation methodology: (a) significant communication overhead incurred due to large data transfer over slow PCIe communication channel, (b) reduced locality caused by repeated data allocations and deallocations, (c) redundant computation related to non-boundary (*i.e.*, interior and exterior) voxels processing, and (d) complex *out-of-core* algorithm implementation.

The motivation to accelerate the computation of the convolution based offsetting can be realized by analyzing its computational complexity. With a uniform 3D grid, the computational complexity of convolution is determined by the target resolution and the size of the convolution kernel. In general, for a regular grid of size $N \times N \times N$ and a kernel of size $M \times M \times M$, convolution in the spatial domain takes $\mathcal{O}(N^3 M^3)$ time. As a concrete example, we consider the case of 100 voxel offsetting at 4096^3 resolution. For an offset distance of 100 voxels, the kernel has $(2 \times 100 + 1)^3 \approx 8 \times 10^6$ discrete points to convolve around each input voxel. Hence, the complexity of convolution operation at a resolution of 4096^3 with 100 voxels offsetting translates to an order of $4096 \times 4096 \times 4096 \times (8 \times 10^6) \approx 5.5 \times 10^{17}$ computations. This value is prohibitive to compute voxel offsetting at high resolutions in an interactive application, like in our target CNC toolpath planning operation.

While the convolution in the spatial domain performs an inner product in each sample, in the Fourier domain it can be computed as a simple point-wise multiplication [82]. Due to this convolution property and the complexity of the fast Fourier transform (FFT) [29, 61], 3D convolution can be performed in $\mathcal{O}(N^3 \log N)$ time. Thus, an FFT based convolution computation, often termed as *fast convolution*, is independent of the kernel size, which might be a preferable choice for voxel offsetting with large kernel size. While the Fourier domain is asymptotically faster than the spatial domain, it has limitations as well. First, the data set dimensions have to be a power of two, which is usually not the case for our voxel data. Therefore, one would

have to pad the data that could cause distortion or noise in the filtered output. This gives justification that performing convolution in the spatial domain is also useful in some cases, and particularly for the voxel offsetting in CNC toolpath generation, our research solely focuses on the spatial domain.

Moreover, storage requirement in the Fourier domain is even more critical due to signals representation in floating-points. For better insight, let us again consider the case of offsetting at 4096^3 resolution. To minimize inaccuracy, assume each sample is represented in double-precision floating-point numbers consuming 8 Bytes. Thus, to perform offsetting at a resolution of 4096^3 , general FFT based 3D convolution requires a storage in order of $4096 \times 4096 \times 4096 \times 8$ Bytes = 512 GB — almost *two orders of magnitude* larger than the memory limit of a typical graphics card. Thus, the scope of this research is confined within the spatial domain that leverages the presented hybrid dynamic trees as the underlying voxel representation.

In principle, the efficiency and performance of an application are determined by the algorithm and the hardware architecture that executes the algorithm, but in practice, critically also by the organization of computations and data on that architecture [87]. Particularly, on graphics hardware the organization of computations and data for a given algorithm are constrained by fundamental trade-offs between parallelism, storage and redundant computation. In our context, the traditional approach of voxel representation using a 3D grid unleashes peak parallelism, but the latter two objectives are compromised. Thus, to optimize the storage and redundant computations both the filtering kernel and the voxel data structure should be represented in some sparse arrangement that trades off the achievable peak performance. For instance, to model a spherical filter for convolution operation, the set of discretized boundary points of the sphere can be considered only instead of using a 3D grid. While the number of discrete points in a 3D filter scale in cubic with the offset distance, the size of discretized boundary points on the surface of a sphere only increases quadratically.

Furthermore, using the hybrid dynamic trees as the underlying data structure allows storage-efficient representation and high-resolution volume processing on many-core graphics accelerators. However, with the adoption of underlying sparse data structure the applications often perform just at a fraction of the peak theoretical performance.

This thesis argues and demonstrates that the convolution offsetting of high-resolution HDTs on GPU can be significantly accelerated, and the memory footprint can be simultaneously optimized through careful selection of tunable parameters—both exploiting the configurable sparse voxel data arrangement and leveraging the controllable convolution parameters. In addition, a judicious selection of underlying data representation in the HDT allows seamlessly scaling out the accelerations across multiple GPUs.

1.3 Research Contributions

The main goal of this dissertation is to design and implement a fast and effective computational method of computing offsets for a solid object represented in high-resolution hybrid dynamic tree (HDT). In particular, this dissertation deals with the development and analysis of a tunable offset computation algorithm that trades off speed, storage efficiency, and geometric accuracy. By “fast” we mean using scalable parallel algorithms that are well-suited to modern massively parallel processors, especially the ubiquitous graphics processing unit (GPU) co-processors available in all modern systems; by “compact” we mean a storage-efficient voxel representation with a low memory footprint; and by “accuracy” we mean the quality of the computed result. At our target resolution of 4096^3 , we aim to compute large-scale offsets in minutes, match or beat the number of bits of the representation compared to state-of-the-art alternatives, and experimentally characterize any trade-offs among speed, storage, and accuracy.

Roughly speaking, our contributions can be divided into four parts. First, we

present a parallel algorithm to construct the HDT representation on GPU for a CAD input modeled in triangle mesh. At a modeling resolution of 8192^3 , our GPU-acceleration of the mesh to voxelization process achieves over *two orders of magnitude* speedup compared to single-threaded CPU implementation. Further, we incorporate tunability into the HDT parameters to study the complexity of memory footprint requires in the HDT representation. The developed theoretical storage analysis is validated with rigorous experiments that helps devising optimal parameter selections for storage-compact HDT representation. Additionally, we analyze the impact of tunable HDT configurations on the different phases of HDT construction on GPU’s massively-parallel computing architecture.

Our second contribution is a tunable offsetting method based on 3D convolutions and mathematical morphology that works on the compact hybrid voxel representation. As algorithmic performance depends on many components; such as, layout of voxel organization, the nature of the computation (such as, HDT construction vs. offset computation), and the underlying hardware (like, CPU vs. GPU), our research considers tuning the performance of offsetting operation through the tunable HDT data structure. While using the HDT as the underlying data structure leads naturally to a storage-efficient representation, the challenge in developing a high-performance implementation of an offset algorithm is choosing an optimal configuration of the HDT parameters. These parameters not only govern the memory footprint of the voxelized representation of the solid, but also control the parallel code execution efficiency on GPUs. As diverse HDT configurations affect different steps of the offset computation differently, this research studies the relationships between the tunable parameters and individual phases of the offset algorithm to empirically derive an optimal parameter setup.

The third contribution of this dissertation is to leverage the controllable convolution parameters to devise a fast voxel-based offsetting algorithm with tunable

speed and accuracy. Using a convolution-based offset generation technique makes the computational complexity tunable, essentially through changing the size of the convolution kernel. In general, a smaller kernel results in faster execution due to lower number of neighboring elements in the spherical stencil; however the geometric accuracy scales inversely with the increasing number of convolution operations. Hence, our research analyzes the impact of the decomposition of a large offset distance into a series of offsetting with smaller distances. Further, to analyze the impact of these parameters on the geometric precision of the computed result, we implement a GPU-accelerated error measurement technique. Such a tuning of the size of structuring element as a tuning knob to trade-off performance over precision further makes our implementation platform-adaptive that allows the algorithm to be adjusted depending on the peak performance of the underlying hardware.

To devise even faster parallel offsetting algorithm, our final contribution is to scale-out the offset computation across multiple GPUs in a load-balanced way. We target a dual-GPU platform to demonstrate near-linear scalability in the offset computation. With more and more GPUs integrated on a single computing node, such exploration of algorithmic speedup through load-balanced implementation of offsetting across multiple GPUs emphasizes the high scalability of the HDT’s hybrid data representation. In general, load-balancing needs estimation of the cost of the entire computation and consideration of the disparity in the peak throughput (often measured in FLOPS) of the underlying GPU devices. In contrast to a boundary-representation, such as a polygonal mesh, voxel-based computation is more amenable to load-balanced execution.

1.4 Outline

The contents of this dissertation can be divided into three parts. Chapters 2 and 3 focus on presenting hybrid dynamic trees, and reviewing related data structures.

Chapter 4 uses HDTs as the underlying voxel representation to develop an efficient and parallel offset algorithm. Lastly, Chapters 5, 6 and 7 present some practical approaches to devise high-performance dilation and erosion for large-scale offset distances at high-resolution HDTs. The remainder of this dissertation is organized as below:

- Chapter 2 presents an overview of the basic data structures for discrete voxel based solid representations. Further, some compressed data representations are presented in Chapter 2.
- Chapter 3 describes the fundamentals of the hybrid dynamic tree structures, and presents GPU-parallel algorithms to construct the HDT from triangle mesh input. Further, a detailed analysis on the storage of the HDT structure is presented. Finally, the impact of the tunable parameters on HDT construction process are analyzed in Chapter 3.
- Chapter 4 presents an efficient parallel implementation of offsetting that adopts a 3D convolution based mathematical morphology to expand or contract solids represented in high-resolution HDTs. Coupled with the underlying HDT based sparse voxel representation, parallel morphological filtering demonstrates significant performance acceleration on the many-core GPU hardware as shown in the comparative studies in Chapter 4.
- Chapter 5 analyzes the computational complexity of morphology based offsetting algorithm using the HDT as the underlying data representation, and demonstrates how the offsetting computation can be optimized through careful parameter selections to the tunable HDT parameters.
- Chapter 6 investigates the opportunity of offsetting optimization through tuning of the kernel size parameters of the morphological filters. Further, to analyze

the trade-offs between speed and accuracy, an efficient parallel algorithm to measure the offsetting error is presented in Chapter 6.

- Chapter 7 presents scaled-out acceleration of the offsetting algorithm on a platform with multiple graphics processing units. Also, an analysis on the scalability of the convolution based offsetting using HDT is presented in Chapter 7.
- Chapter ?? discusses our experience with HDTs as the foundational data structure for storage-efficient voxel representation, and analyzes the tunable convolution offsetting algorithm to demonstrate its applicability for interactive CAM applications.

CHAPTER 2

LITERATURE REVIEW

2.1 Survey of Basic Voxel Data Structures

As our research considers the problem of how to efficiently represent and compute on high-resolution volume data, in this section we present a review on the fundamental voxel data structure.

2.1.1 Uniform Grid

To represent a solid object in cubical domain, *uniform grid* is the simplistic data structure, where the axis aligned bounding box of the solid is subdivided into equally sized cells (i.e., voxels) along each of the three main axis X , Y and Z . Voxels discretized on a regular 3D grid is easy to implement because of the uniform indexing. Additionally, most of the volumetric algorithms, such as, convolution kernels, interpolations, discretization of differential operators, are inclined to a uniform sampling scheme [80]. Figure 2.1 shows a uniform voxel grid, along with some Boolean operations operating on two voxel grids to illustrate the simplicity of the underlying algorithms.

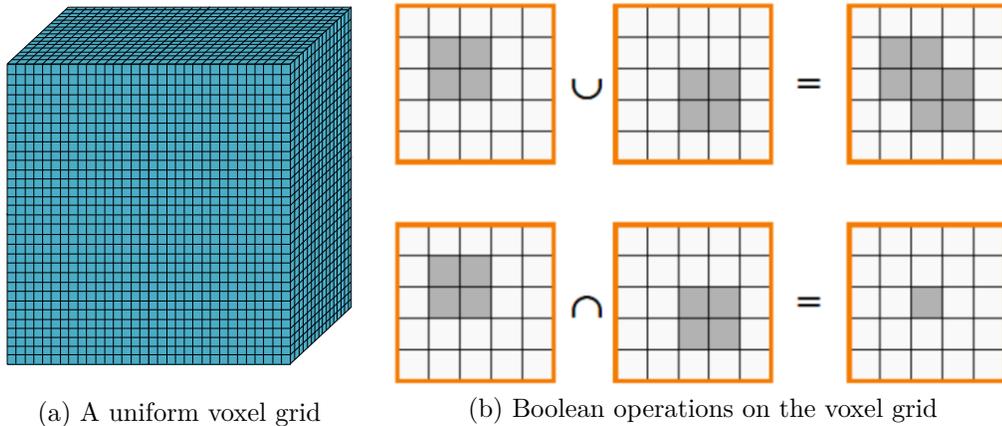


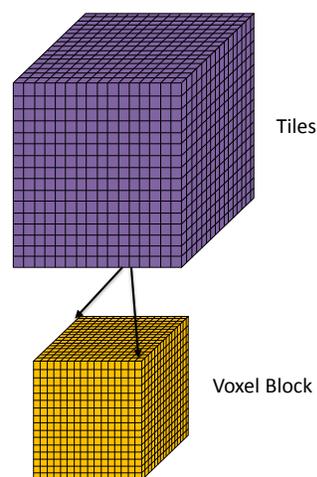
Figure 2.1: Uniform grid representation.

However, with the surface discretized onto a uniformly-spaced grid, the density of the volumetric elements (voxel) increase with a cubic order of the resolution. For instance, to fabricate a 3D part of 1 m^3 volume that is discretized on a uniform grid with a voxel size of $10 \text{ }\mu\text{m}^3$. Even with a data size of 1 bit per voxel, it needs over 100 Terabytes just to store the geometry details of 10^{15} elements. Hence, uniform grid is only suitable if the target resolution is not high, and thus it is not a practical choice for high-precision 3D manufacturing as the discretized output lacks sharp and thin features in geometric modeling with limited volumetric resolution.

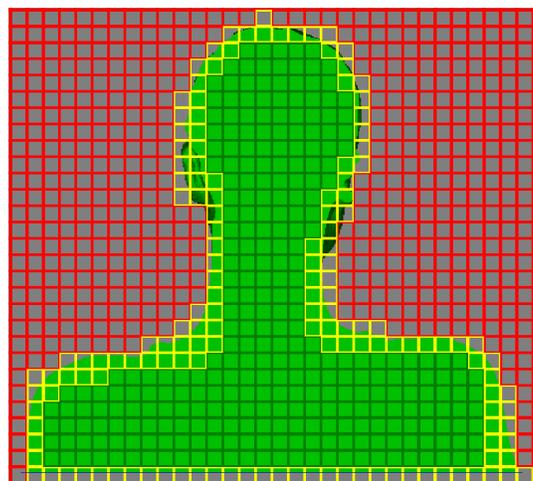
2.1.2 Two-Level or Tiled Grid

To mitigate this high memory overhead with a dense uniform grid structure, an obvious approach is to exploit adaptive refinement in the spatial decomposition. Applying an adaptive refinement to a uniform grid, we have a two-level nested grid structure, termed *tiled grid*, where volume is first divided into uniformly-size cells, also called *blocks*, *bricks*, or *tiles*. Then, each cell is partitioned into a fixed number of voxels if it intersects the target object of interest. Figure 2.2(a) depicts a sample 2-level tiled structure, while in Fig. 2.2(b) a tiled grid is demonstrated on a cross-section of a sample 3D model, in this case a human upper (posterior) body. In Fig. 2.2(b), boundary tiles (yellow) are only further decomposed into voxels, while the outer tiles (red) or the inner blocks (green) are not.

Tiled grid has been used in many production rendering systems, such as, the open-source Field3D [35]. Kalojanov et al. [55] extended their previous method [56] for construction of two-level grids targeting faster ray tracing while keeping the data structure build times comparable to uniform grids. The limitation of a fixed hierarchical structure is obvious; as the available grid resolution is measured by the product of the branching factors at each level, a higher resolution suggests either of the branching factors to be increased. A large branching factor at the top level inflates the number of tiles, and hence the storage. At the bottom level branching



(a) A 2-Level grid structure with 16^3 blocks, and each tile representing 16^3 voxels.



(b) Tiled grid structure in a sample 2D cross-section. In this example, the boundary blocks (yellow) are not shown divided into voxels.

Figure 2.2: Illustration of a tiled grid structure.

factor needs to be relatively small, as the ratio of inactive to active¹ voxels needs to be low for limiting the redundant storage. Hence, for extreme-scale volume modeling such a fixed two-level hierarchy reduces exploiting adaptivity in the geometry, and thus inflates topology storage for the active voxels (cf. Section 3.4.1).

To address the requirement of large top-level grid in a tiled grid structure, Konobrytskyi et al. [60] presented a GPU centric two-level grid implementation, called hybrid dynamic tree (HDT). In stead of using a large top-level grid, their implementation adopted *a list* of cells (*i.e.*, tiles), and each cell containing the boundary is partitioned into finer resolution sub-cells (*i.e.*, voxels). The presented two-level HDT structure scaled up to 256^3 cells, where each cell is a block of 16^3 voxels. While the two-level HDT implementation achieved up to a resolution of 4096^3 on a multi-GPU platform with a combined graphics memory capacity of 9 GB [60], being a variant of tiled grid structure their HDT inherits the same constraints as uniform 3D grids exhibit for extreme-scale voxel representation. Our presented HDT design [48] is a

¹In this research, we define the voxels lying on the solid boundary as *active* voxels, while the rest of the voxels in a tile, located either inside or outside of the object, are termed inactive voxels.

natural extension to the two-level HDT proposed by Konobrytskyi et al. [60] that integrates a complete adaptive refinement in between the top grid and the bottom grid. The details on the completely adaptive HDT design and implementation are presented in Chapter 3.

2.1.3 Octree and Generalized N^3 -tree

Given a solid object, to represent its volume in a structured but adaptive way an intuitive strategy will be to discretize space into voxels where the smallest voxels correspond to a desired resolution, but with a goal of being memory-efficient by only storing voxels necessary to represent the object. This is exactly what is conceived in a *sparse voxel octree* (SVO) [76, 53, 89, 39, 38, 33]. Octrees have an especially long history in the context of rendering, modeling, and mesh extraction. An octree is a hierarchical space-partitioning data structure that recursively subdivides space into units of 8 cells, or *octants*; to make it adaptive, a given cell subdivides only if it intersects with the target object. Thus, a sparse voxel octree retains the desired geometric resolution at the lowest level of the tree, while clusters similar regions (empty or solid) compactly at the nodes in intermediate levels. The memory footprint in an SVO scales with the number of voxels required to define the solid boundary, instead of the embedding volume.

With an extremely small branching factor at every level of the tree hierarchy, octree is clearly optimal for adaptive grid sampling. However, for optimal geometric processing it is not only the size of the data, but the layout of the computation and the data that also matters to harness the fullest capability of off-the-shelf hardware accelerators, such as GPUs. A fundamental challenge in any sparse adaptive data structure is to properly balance between computational efficiency and storage efficiency: storage effective algorithms tend to lower the computational efficiency, while computationally efficient algorithms tend to increase the storage requirements. Figure 2.3(a) demonstrates the concepts of an octree in 2D (i.e., a quadtree) for a sample

triangle that is spatially decomposed in successive levels of geometry approximations. The resulting quadtree is shown in Fig. 2.3(b), with the filled cells containing geometry, and the rest cells denoting empty space. To illustrate the complexity of algorithms operating on sparse voxel structure compared to a uniform grid, Figure 2.4 shows the same Boolean operations (as depicted for the uniform grid in Figure 2.1) that operate on two quadtree structures.

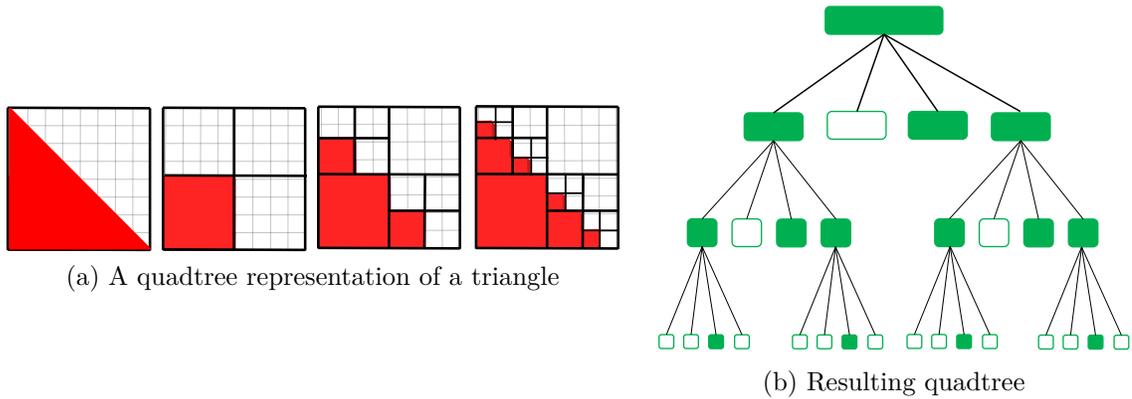


Figure 2.3: Illustration of a quadtree.

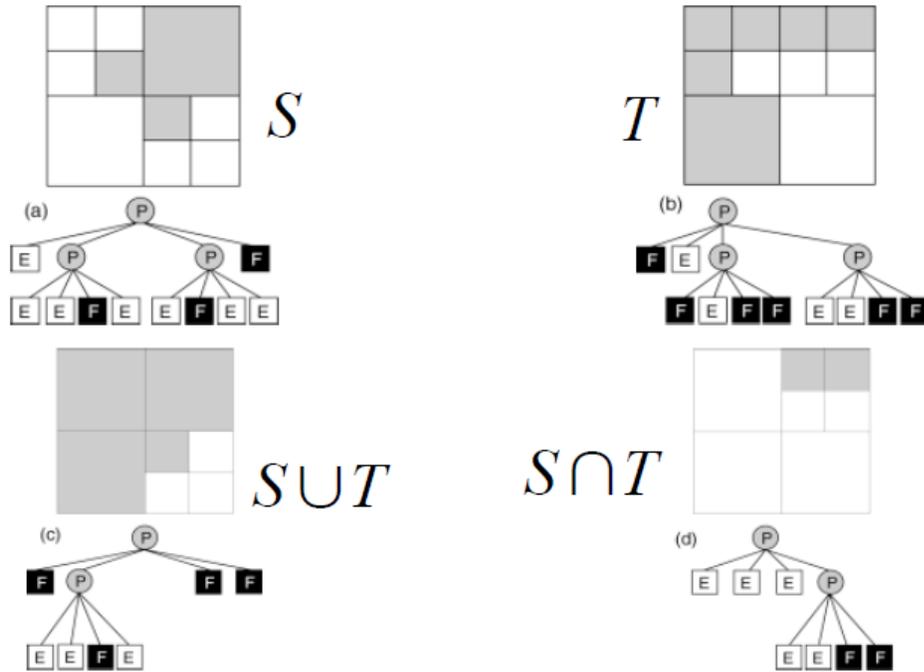
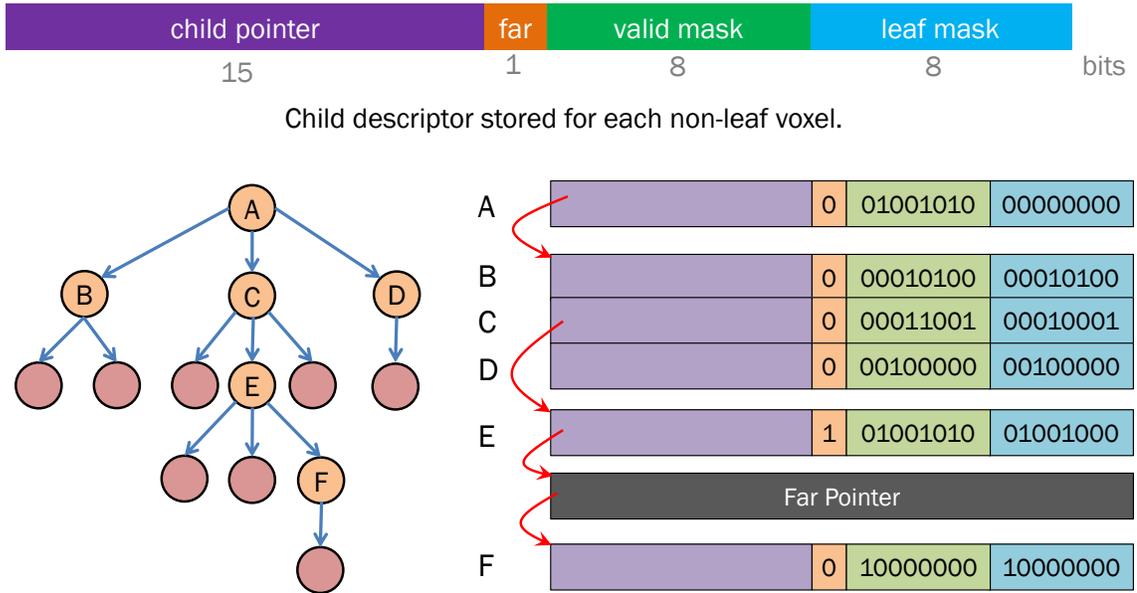


Figure 2.4: Boolean operations on two quadtree structures (illustration source: [36]).

While an octree is space-efficient, a relatively small branching factor (of just 2 per dimension) means that achieving a high resolution will require a relatively deep tree. Level-by-level tree construction and traversal can be a major bottleneck, especially on GPU platforms. The two main reasons are that a) the tree depth defines the critical path length, and b) the degree of parallelism is high near the lower levels (leaves) but low everywhere else. Hence, tall octrees make the construction and dynamic updates of the volume data problematic on GPUs. A natural generalization to octree is the N^3 -tree [67], where each non-leaf node has N^3 children (for instance, an octree has $N = 2$). The choice of N trades-off memory efficiency for traversal efficiency: a relatively small N yields a highly adaptable data structure with the potential for low storage at the cost of a deep tree with low traversal-efficiency, whereas a relatively larger N reduces adaptivity, thereby increasing storage but making a more shallow tree with high traversal-efficiency.

A sparse voxel octree encodes the cubical 3D grid by grouping empty regions, where each node stores an 8-bit mask denoting for every child if it exists – i.e., the corresponding voxel space is not empty. A pointer connects the parent to its children, which are ordered in memory. Thus, 8 bits are needed for the childmask, plus a pointer of typically 32 bits. Altogether, it consume $8 + 32 = 40$ bits (5 Bytes) per node in a typical SVO based voxel representation. To optimize the storage requirement, Laine et al. [63, 62] presented an efficient sparse voxel octree (ESVO) structure, where each leaf identifies an active voxel on the surface geometry. As depicted in Figure 2.5, the authors used a 32-bit child descriptor to store the topology of the octree, corresponding to every non-leaf node. Leaves (voxels) do not require a descriptor of their own, as they are described by their parents. The child descriptor contains two bitmasks, each storing one bit per child slot – *valid mask* tells whether each of the child slots contains any boundary geometry, while *leaf mask* further specifies whether each of these slots is a leaf (voxel). Based on the bitmasks, the status of a child slot

can be interpreted as either of — (a) *neither bit is set*: the slot is not intersected by a surface; (b) *the bit in valid mask is set*: the slot contains a non-leaf node; (c) *both bits are set*: the slot contains a leaf voxel. If the voxel contains any non-leaf children, a 15-bit child pointer is used to reference the descendant node.



Top: Layout of a child descriptor entry. Left: Example voxel hierarchy. Right: Child descriptor array containing one descriptor for each non-leaf voxel in the example hierarchy.

Figure 2.5: Efficient sparse voxel octree representation [63].

As Laine et al. [63, 62] analyzed the average storage per active voxel, it requires only 1.33 Byte (11 bits²) to maintain the topology of the voxel in a sparse 3D space, and 1 Byte for storing the state of individual voxel. Thus, in total $11 + 8 = 19$ bits is consumed per active voxel in the efficient sparse voxel octree, which is roughly $2\times$ compact than typical SVOs. While the ESVO is a compact representation and capable to represent extreme resolution, an SVO-like structure is not well suited for parallel volume editing application. For parallel voxel offsetting on GPUs, as in the case of our CAD/CAM application, a sparse data structure needs organizing the active voxels in

²The specific topology storage depends on the sparsity of the nodes in the octree. For an average four children per node the topology storage is shown to be 1.33 Byte.

a way to ensure that all the computing threads of a CUDA block must access spatially adjacent memory addresses to avoid any overhead caused by data divergence. In an SVO or variants thereon, threads in a CUDA warp access incoherent memory blocks corresponding to sparsely located voxels that pose computational challenges in time.

2.1.4 Hash Table Representation

In contrast to deep tree based hierarchical structure, voxels can be sparsely represented using hash tables [65, 13, 34, 24, 81, 59]. Different hashing schemes can be adopted in conjunction with above discussed voxel data structures to speedup the traversal for voxel lookup. Generally, a set of indexes referencing an active voxel (or a group of voxels) is used as the key that the hash function maps to a particular memory block. As an example, Figure 2.6 shows the voxel hashing data structure used for real-time 3D reconstruction by Nießner et al. [81].

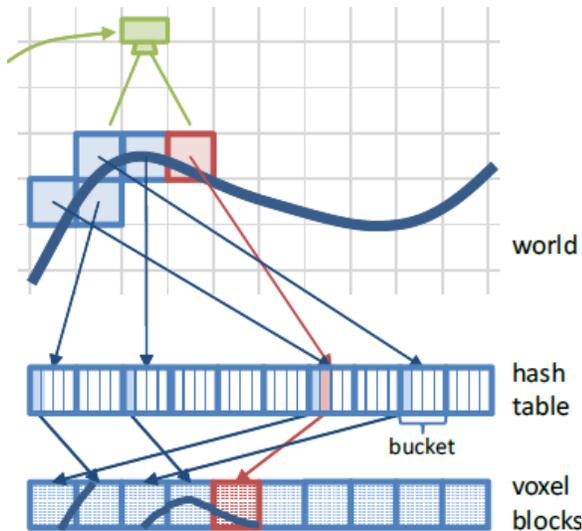


Figure 2.6: Voxel hashing data structure by Nießner et al. [81]. Here, the voxel world is conceptually partitioned into an infinite uniform grid. Using the hash function, integer world coordinates are mapped to hash buckets, which store a small array of pointers to regular grid voxel blocks. Each voxel block contains an 8^3 grid of signed distance field (SDF) values. When information for the red block gets added, a collision appears which is resolved by using the second element in the hash bucket.

Spatial hashing allows random access to sparse data in *expected* constant time, whereas in a dense voxel grid look-up can be done in *worst-case* constant time. In practice, the look-up time for some items in a hash table is $\Omega(\lg \lg n)$ with high probability [13]. The work by Lefebvre and Hoppe [65], among the first to develop a hash table on GPUs for efficient access to sparse voxel data, addressed the issue of variable lookup time by using a perfect hash table. The perfect hash table allows accessing an item in worst case $\mathcal{O}(1)$ time.

Though such sparse voxel implementation is fairly simple and flexible for GPU parallelization, it suffers several performance issues. Typically, a volume at high resolutions include tens to hundreds of millions of active voxels, and its fairly difficult, if not impossible, to avoid hash-key collisions for such large set of elements. Besides, the time to compute the hash-key for complex hash functions is often non-trivial. Moreover, as good hash functions in general prefer distributing the keys randomly, the cache performance is impaired even during sequential accesses. As demonstrated in the study by Eyiurekli et al. [24], the experimentations on a two-dimensional numerical simulation validated that a hash table based solution generally fails to match the performance of quadree data structure.

2.1.5 Hybrid Representation

As grids have large branching factor and octrees (N^3 -trees) are with small branching factor, hybrid schemes that combine the basic ideas underlying regular grids and octrees (N^3 -trees) can balance between storage and computation efficiencies. One such approach is *octree-grid* (also known as sparse block grids) [23, 74, 30, 26] where octree-like dyadic spatial division is adopted at top hierarchies, while at the lowest level of refinement each cell represents a block of non-overlapping voxels. Another alternative hybrid approach is called *grid-octree* [79, 99], which is conceptually a grid of small octrees — at the topmost hierarchy the volume is divided into uniform-shaped cells, and then each cell is adaptively divided in an octree fashion.

Though both of these hybrid schemes share a common goal: leveraging the implicit indexing in grid structure to bypass multiple accesses of slow tree traversals, from the viewpoint of parallel voxel editing scenario, the former approach is generally better suited to GPUs than the latter. In a hybrid grid-octree structure each CUDA thread processes a relatively shallow octree that although spans a quite limited depth, yet such recursive branchings likely to cause divergences and unaligned memory accesses. By contrast, for a hybrid octree-grid approach though divergence exists at the upper hierarchies of tree traversal, at the lowest level the set of grids containing the active voxels can be processed with tremendous efficiency. This is to be emphasized here that in our target use case of high-resolution voxel offsetting in CAD/CAM application, the geometric processing of the boundary voxels are generally the most computation-demanding operation. Hence, a sparse data organization resembling a hybrid octree-grid structure could be considered as a preliminary fit to our needs.

Among all the prior sparse data structures, the VDB approach proposed by Museth et al. [80] is most closely related to our work. Structurally, VDB is a hybrid scheme similar to octree-grids. VDB organizes a block of contiguous voxels at the leaf hierarchy. At intermediate levels VDB divides the spatial range in a large branching factor similar to B+ trees. Using a B+ tree as the underlying structure in place of an octree, VDB manages the hierarchical data organization to be shallow that improves the traversal efficiency as described earlier. Furthermore, the branching factors across different levels in VDB generally increase from the bottom up. Such a design is based on multi-level caching policy in modern CPU architecture. As VDB implements custom-tailored software-level caching, increasing branching factors from the bottom up improve the cache hit rates for sequential accesses. At the root level VDB suggests using a hash table based key (e.g., voxel indexes) distribution to support a virtually infinite resolution at the cost of scattered data allocation.

Figure 2.7 shows a rendering of a dragon model in VDB. In this example, the VDB

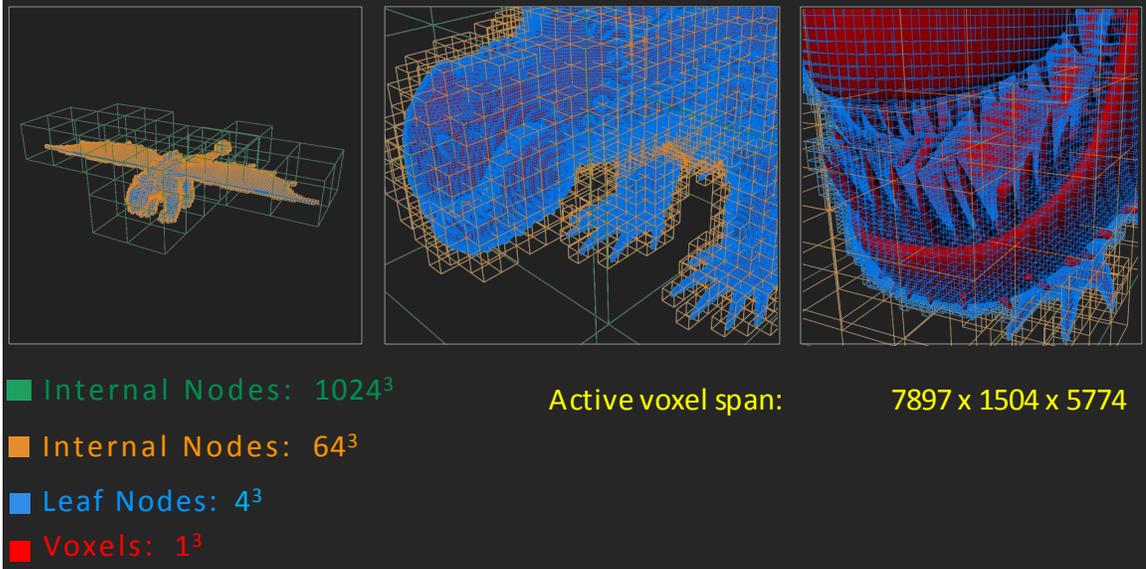


Figure 2.7: High-resolution VDB created by converting polygonal model from *How To Train Your Dragon*. Images are courtesy of *DreamWorks Animation*.

structure is configured in three levels. At the top level each internal node represents 1024^3 resolution, and each of the mid-level internal nodes represents 64^3 resolution. Thus, the top level node has a branching factor of $\frac{1024}{64} = 16$ along each dimension. The same branching factor is adopted for the middle level nodes, and thus each of the leaf nodes represents a volumetric space with a dimension of $\frac{64}{16} = 4$ voxels. Contrary to the VDB, our HDT design adopts a constant branching factor for the adaptive refinement of the nodes in between the top grid and bottom grid. A most subtle difference with our hybrid data structure is that VDB is designed for CPU platforms, which is typically equipped with an order of magnitude larger system memory than GPUs. While VDB design is quite sophisticated, the use of software-level caching and large branching factors introduces storage overhead of 8–9 Bytes per active voxel [80].

2.2 Compressed Voxel Representation through Geometric Redundancy

Although the scope of this dissertation is confined within efficient storage and processing of raw voxel data, and hence does not explore any compression mechanism

to reduce the memory footprint, a vast array of compression schemes are widely used in diverse application domains. While advanced compression algorithms can significantly cut down the storage requirement, for high-resolution voxel offsetting in an interactive CAD/CAM application the overhead of repeated compressions and decompressions may impose significant challenge, particularly for accelerated processing on graphics hardware. Generally compressed size of the data depends on the input, and hence runtime compressions-decompressions cause memory fragmentation due to the changing size of the compressed data. Another more subtle issue is that due to variable size of the compressed data, *computation homogeneity* across the threads on GPUs gets degraded. This part of the literature review focuses on voxel compression techniques that primarily targets optimizing storage of hierarchical voxel representation, and thus are suitable to sparse voxel octree (SVO) structures, and the variants thereon. As the hybrid voxel representations discussed above generally use octree or generalized N^3 -tree as the underlying data structure to leverage sparsity, these compression methods can be equally applied to hybrid structures, such as, VDB and HDT, among others.

Our discussion includes recently proposed state-of-the-art compression mechanism based on *directed acyclic graph* (DAG) that targets storing voxel data at extremely high resolution. While the SVO allows for efficient encoding of empty regions of space, the DAG additionally allows for efficient encoding of *identical regions* of space. Besides the structural sparsity in the voxel grid, recent works [57, 31, 101, 104] have exploited the resemblance among voxel blocks in disparate volume spaces to compress the volume data. The idea of compression through region merging has been studied earlier by Webber and Dillencourt [103], where quadtrees representing binary cartographic images are compressed by merging common subtrees. A similar approach was further extended by Parker et al. [84] to three dimensions to compress axis-aligned and regular voxel data. Alternative approaches to compactly representing trees, and

applications thereof, are presented and surveyed in an earlier work by Katajainen et al. [58].

Kämpe et al. [57] explored that geometric redundancy in the voxel structure, particularly in VFX applications, is common. The authors studied that a *binary* voxel grid can be represented more efficiently than using an SVO by generalizing the tree to a DAG, where nodes in a DAG are allowed to share pointers to identical subtrees. The proposed approach searches the tree for common subtrees in a bottom-up manner, references only the unique instances, and merge the identical regions. This transforms the tree structure into a directed acyclic graph (DAG), resulting in a reduction of nodes without loss of information. Figure 2.8 shows the high-level abstraction of this compression scheme. While the proposed technique of merging equivalent subtrees in an SVO demonstrates promising compression rates, it may incur significant overhead to transform an SVO to a DAG representation. The efficiency of this compression methodology depends on the geometric resemblance in the input voxel data. When there are abundant patterns matching, the compression takes typically 1 to 5 seconds at 8192^3 resolution, but for irregular models it takes over 40 seconds on a NVIDIA GTX 680 [57].

Besides the transformation overhead, one disadvantage of the DAG in comparison to an SVO is that pointers need to be stored for *each child*, because they can no longer be grouped consecutively in memory (in which case, a single pointer to the first child is sufficient). For instance, assuming a node has four children on average, it may require up to $8 + 4 \times 32 = 136$ bits per node in DAG (8-bit childmask and a 32-bit pointer), which would require maximum 40 bits per node in a typical SVO structure. Schnabel and Klein presented pointerless SVOs [91] that completely removes pointer overhead and are well suited for offline storage. The authors presented memory compaction technique of a point cloud of geometry discretized into an octree with 4096^3 resolution. By sorting the tree in an implicit order, e.g. breadth-first order,

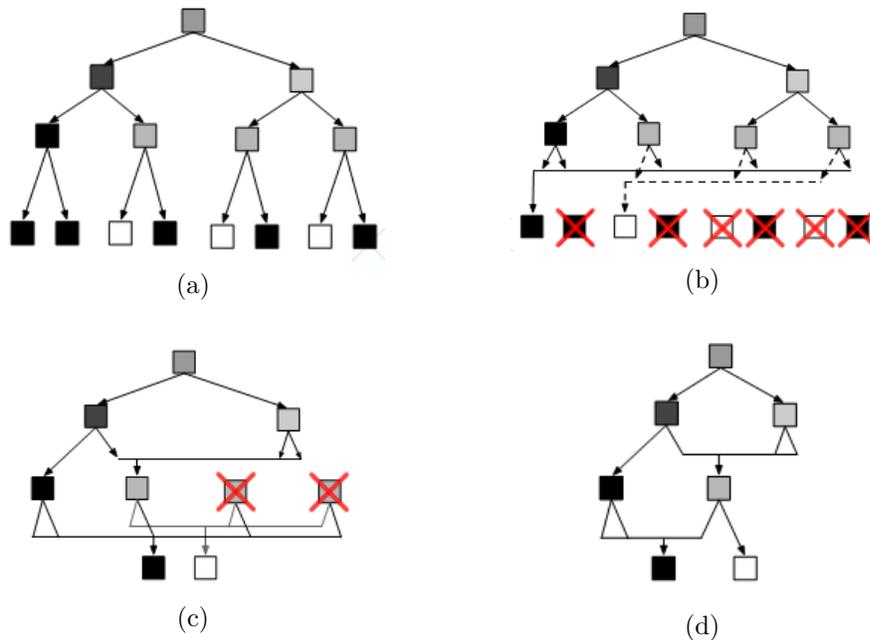


Figure 2.8: Reducing a sparse voxel tree, illustrated using a binary tree, instead of an octree, for clarity. a) The original tree. b) Non-unique leaves are reduced. c) Now, there are non-unique nodes in the level above the leaves. These are reduced, creating non-unique nodes in the level above this. d) The process proceeds until the final directed acyclic graph is deduced (illustration source: [57]).

they can store it without pointers between nodes. However, the implicit order do not support random access and cannot be extended to DAGs, as pointerless SVOs require a fixed, sequential memory layout of nodes. While several reduction approaches for pointers have been proposed [66, 63], they are typically not applicable to the DAG. These schemes assume that pointers can be replaced by small offsets, but in a DAG, a node's children are not in order but scattered over different subtrees. Another recent work presented a pointer entropy encoding and symmetry-based compression for DAGs, but does not support attributes [101].

A further limitation of the DAG based storage compaction proposed by Kämpe et al. [57] is that it is restricted to the compression of a single bit of (geometry) information per voxel. As discussed earlier, the method is particularly successful if the voxel data exhibit geometric repetition. Unfortunately, extending the information

beyond one bit (e.g., to store material properties) is challenging, as it reduces the amount of similar subtrees drastically [31]. To address this lack, recent works. [104, 31] have further extended the applicability of DAG based compression with graphics rendering attributes, such as, colors, normals etc. resulting into further memory reduction.

CHAPTER 3

THE DESIGN, IMPLEMENTATION AND ANALYSIS OF THE HYBRID DYNAMIC TREE

This chapter presents the hybrid dynamic tree (HDT) data structure, and discusses its condensed representation, a parallel algorithm for constructing the HDT on GPU, and thorough experimental evaluations to characterize the HDT.

3.1 HDT Fundamentals

3.1.1 Basic Scheme

The hybrid dynamic tree (HDT) is an adaptive tree based data structure for representing high-resolution sparse volumes. The HDT combines two contrasting data structures – *dense* grid and *sparse* octree – in such a way that makes it both more compact (i.e., *storage efficient*) and better-suited to GPUs (i.e., *computation effective*) than state-of-the-art alternatives. Figure 3.1 illustrates a sample layout of an HDT structure. Like a tiled grid, the topmost level of an HDT is a *root grid* (shown in green). The root grid is a 3D grid of uniformly-sized cells. If a given cell of the root grid (or *root cell*) intersects the target object, then it becomes the root node of an octree. Each cell in octree (*octree cell*) is then adaptively subdivided just as a regular octree would be. In the figure, blue and white colored cells represent space in a uniform state: either completely *full* (e.g., inside the solid object) or completely *empty* (e.g., outside the solid object). Cells in a uniform state are not subdivided, whereas the remaining cells (light blue) are. This adaptive refinement continues until the cell size reaches the target resolution, at which point each leaf-level cell represents a dense block of voxels, or *leaf grid*.

As the figure shows, an HDT effectively “sandwiches” octrees between the levels of a tiled grid. Though simple in comprehension, such a hybrid representation is yet capable of representing extreme-scale resolution with a reasonable memory footprint.

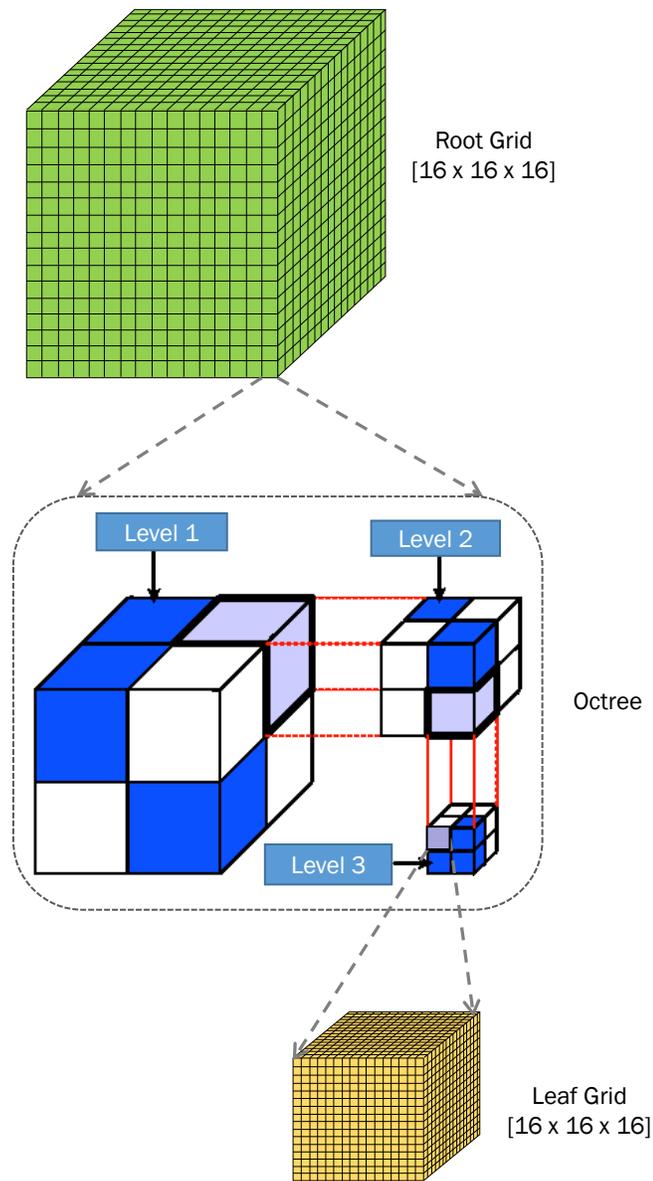


Figure 3.1: Illustration of HDT representation. Each intersecting root cell represents the root of an octree that is adaptively refined until the desired resolution is reached. Each octree (i.e., root cell) in this demonstration represents $(16 \times 2^3)^3$, i.e., 128^3 sub-volume, and hence the entire HDT represents a volumetric resolution of $16^3 \times 128^3$, i.e., 2048^3 . In this example, two small grids and a three-level octree jointly represent a volumetric resolution of 2048^3 that is much shallower than a hierarchy of eleven with a standard octree.

The HDT in Figure 3.1 has a root grid and leaf grids all of size 16^3 , but these need not be the same in general. These are tunable parameters that control the effective resolution and, given an object, the sparsity of the representation. For instance, suppose one wishes to represent a cubic volume at a resolution of $8192^3 \approx 550$ billion voxels. Choosing a root grid of size 64^3 and leaf grid of 16^3 would yield an HDT whose octrees have at most $\log_2 \frac{8192}{64 \times 16} = 3$ levels. Compare these numbers to pure tiled grid and pure octree representations. A tiled grid capable of the same overall resolution would need a top-level grid of size $\frac{8192^3}{16^3} = 512^3$, which makes it less adaptive than the HDT¹. A pure octree would need a $\log_2 8192 = 13$ levels, which makes it much “taller” (and therefore slower and with reduced parallelism per level) than the HDT. The impact of deep octree hierarchies will be further analyzed in Section 3.4 to highlight the choice we made to restrain taller tree structure for efficient construction and processing on GPUs.

3.1.2 Building Blocks

Conceptually, HDT is composed of two types of abstract data: (1) *tree cells* and (2) *leaf grids*. We store at each tree cell its (x, y, z) coordinates and its *depth* in the HDT – that together specify the non-overlapping volumetric space represented by the corresponding cell in HDT. In addition, each cell is in one of four possible states: *full*, *empty*, *branching*, and *boundary*. The full and empty states are terminal, but the branching and boundary states point to descendent tree cells and leaf grids, respectively. Our HDT implementation uses 24 Bytes of storage for each tree cell (Figure 3.2): 4 Bytes for each of the three coordinates, 2 bits to encode the 4-valued state, 30 bits to encode the depth, and 8 Bytes as a pointer.² Leaf grids are stored densely and so voxels within them may be referenced without additional storage or

¹Of course root grid size can be reduced by increasing the dimension of leaf grid from default 16. However, as described earlier arbitrarily large leaf grid size blows up the redundant voxels in HDT.

²These are just choices we made; a more compact representation may be possible using additional compression or delta encodings.

pointers.

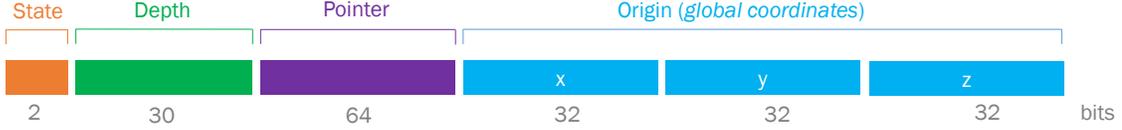


Figure 3.2: Compact 192 bits representation of an HDT cell; state (2), depth (30), descendant pointer (64), and cell origin (3×32).

A tree cell in a branching state intersects with the target solid object, and spans the HDT hierarchy by one level. Thus, each branching cell generates eight child cells. Our HDT implementation uses only a single pointer to these child cells, since it also allocates the descendant cells of a given cell together, as is done in prior studies [30, 63]. Allocation all the child cells in one such contiguous block not only improves the coherency of memory accesses, but also reduces the storage requirements tremendously. Finally, a tree cell in the boundary state is linked to a leaf grid, where each leaf grid subdivides the corresponding cubic space into 2^l cells along each dimension of the cube. Thus, a leaf grid is a small voxel grid of size of $2^l \times 2^l \times 2^l$ voxels, which approximates the part of the original volume that corresponds to the specific boundary cell. Thus, for $l = 4$ each leaf grid contains $16 \times 16 \times 16$ voxels (i.e., total 4096 voxels). Each voxel may be in one of three states: inside the target object, outside the target object, or at the surface. Thus, we use 2 bits to encode this state and a leaf grid needs $2^{3l} \times 2 = 2^{3l+1}$ bits of storage. For $l = 4$, $2^{3l+1} = 2^{13} = 8192$ bits = 1024 Bytes. Hence, each leaf grid in our HDT is allocated in a contiguous memory block of 1024 Bytes. The two-level HDT by Konobrytskyi et al. [60] adopted same design for the leaf grid, where 2 bits storage per voxel is embodied in the memory pool of 1024 Bytes.

3.1.3 Design Principles

3.1.3.1 Root Grid

We choose the root grid in the HDT structure with the same goal as adopted in the two-level HDT proposal by Konobrytskyi et al. [60]. The root grid serves two purposes. First and most obviously, the root grid eliminates multiple levels in the hierarchical tree structure that would otherwise impose significant overhead. For instance, even a root grid sizes of 32^3 and 64^3 can bypass tree traversal of 5 or 6 levels, respectively. Second, as the HDT is specifically tailored for parallel construction and processing on GPUs, the root grid exposes significant *thread level parallelism* (TLP) to be exploited. For instance, a root grid of $64 \times 64 \times 64$ has up to 262,144 cells in the grid that can be traversed for geometric processing in parallel on GPU.

3.1.3.2 Octree

To justify the choice of using Octree at the intermediate hierarchies, we show that a three-level grid is not optimal on storage requirement compared to the approach we adopt in the HDT. Our proof is based on the assumption that when a cell split occurs, storage for all the descendant cells are allocated at once. This helps tracking all the child cells through a single pointer indexing descendants at consecutive memory locations in the buffer.

For simplicity, let us assume the intermediate grid has a dimension of 4 (*i.e.*, 2^2). Hence, a root cell splitting accounts for 4^3 cell storage. If the intermediate grid is replaced with a 2-level Octree, a root cell splitting at the first level generates 2^3 cells. If the state of all these cells are uniform we stop splitting; otherwise each cell containing some part of the geometry is decomposed into the second level. Maximum number of such second level cells is (2^3-1) . Each of these (2^3-1) cells generates 2^3 cells when decomposed that results into a total of $2^3 + (2^3-1) \times 2^3 = 4^3$ cells. So, we come up that the number of cells generated in a 2 level octree cannot be larger than 4^3 —same as the other alternative. We can generalize the proof to an intermediate grid

of arbitrary size of 2^k that can be replaced by an Octree of depth k in HDT, and it can be guaranteed that the HDT approach consumes no more storage than the grid alternative.

3.1.3.3 Leaf Grid

The choice of clustering a group of consecutive voxels in a leaf grid is driven by our underlying GPU computing platform. GPUs are designed for massive parallelism, where threads in a computing block (called *warp*) should respect some constraints to achieve high performance. Using an underlying voxel representation, the computation across the threads in a warp naturally perform *homogeneous* computation, *i.e.*, exactly same set of operations on different voxels. However, the overall performance depends not only how the operations are executed on the processing core, but also how the data are fetched (or written back) from (to) the global memory on GPUs.

Let us consider the case of efficient sparse voxel octree (SVO) data structure [63, 62], where a leaf in the octree represents specific voxel in the 3D space. Although approaches like SVO is compact in memory footprint, they pose challenges on the way data is fetched on GPUs. For parallel volume editing on GPUs, as in the case of our CAD/CAM application, threads in a CUDA warp access incoherent memory blocks corresponding to sparsely located voxels in an SVO organization. For instance, we assume a simplistic volume representation that stores one bit data per voxel to identify two possible states — voxel located at boundary, or not on boundary. Without any coalesced memory addressing, only single bit data is used in this case per 32-byte device memory read or write — the smallest width of memory transaction³ on current generation GPUs [4]. This translates to a memory-bus usage efficiency of only $\frac{1bit}{32 \times 8bits} = 0.4\%$. Such a low bus utilization incurred by divergent memory accesses imposes severe performance penalty to achieving peak throughput on graphics devices.

³We consider the transaction between the processing core and the cache; the width of data transaction between the cache and the memory is even larger (128 bytes).

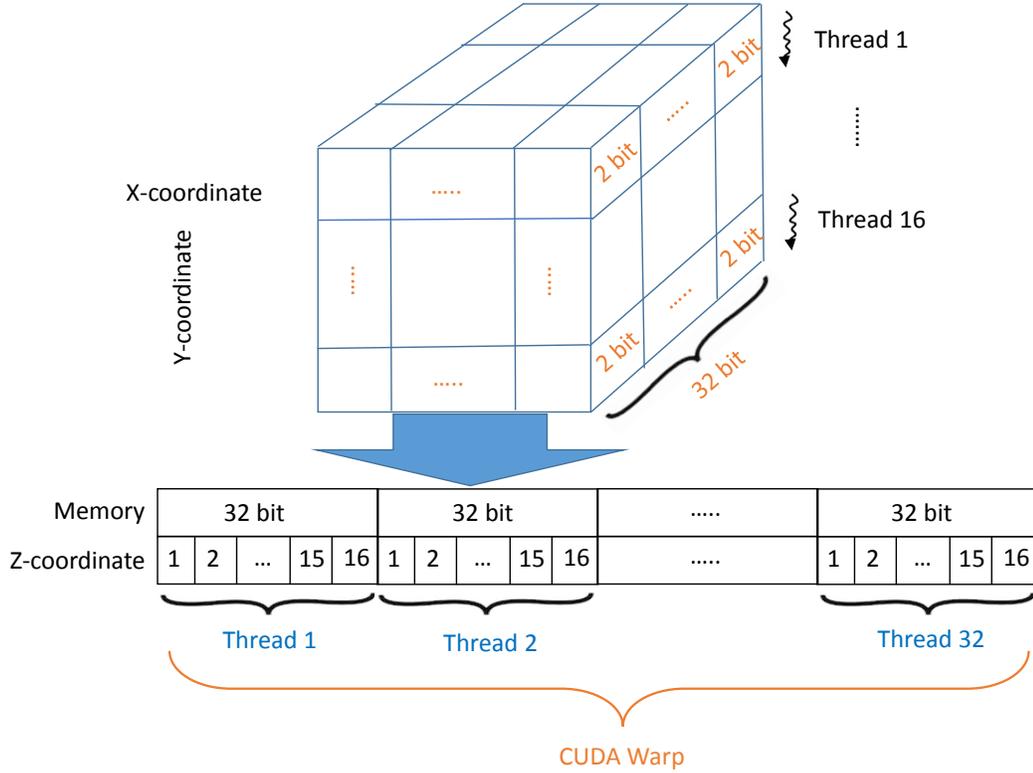


Figure 3.3: Distribution of the CUDA threads in the HDT leaf grid space during the HDT construction and volume processing on GPUs.

To address this limitation pertinent to SVO-like data representation, our HDT leaf grid adopts a similar principle proposed by Konobrytskyi et al. [60]. In our HDT design, a leaf grid consists a block of voxels in contiguous volumetric space. There are two major benefits from the grouping of voxels. First, as the state of multiple voxels are packed in a memory *word* (four Bytes), the data, fetched or written back, contain no *redundant* bits. Second and more importantly, as the adjacent threads access word-aligned and contiguous memory addresses, the memory read/write requests can be coalesced to achieve optimal bus utilization. Figure 3.3 illustrates how the GPU threads get distributed to the specific voxel coordinates in a leaf grid for parallel HDT construction and processing with a default leaf grid size of 16^3 . In our design, the leaf dimension in the HDT is a tunable parameter, unlike the two-level HDT by Konobrytskyi et al. [60] that works only with a fixed leaf size of $16 \times 16 \times 16$.

3.1.4 Sturcutre Storage: Memory Pools

Throughout the HDT construction process, multiple dynamic buffers are consistently used for storing constructed tree cells and leaf grids, and maintaining in-process splitting cells. A natural consequence of dynamic memory management is that it leads to memory fragmentation. Fragmentation is a critical performance limiter for GPU-algorithms [50, 95] than its impact on CPU counterpart, as the available memory on graphics device is generally an order of magnitude smaller than the system memory. Therefore, we need an efficient dynamic buffer management to reduce memory fragmentation on GPU.

For each type of the building blocks in HDT, i.e., the tree cell and the leaf grid, two memory pools are consistently managed on the graphics device. Our custom memory manager implementation maintains two buffers – the *element pool* for the tree cells, and the *leaf pool* for the leaf grids. During the HDT construction process, both the element pool and leaf pool buffers are continuously expanded. Element pool is initialized with the total number of cells in the root grid. The descendants of a branching tree cell are stored in a contiguous block of $2 \times 2 \times 2$ cells inside the element pool. On the other hand, leaf pool is initialized empty, and each boundary tree cell is mapped to an entry inside the leaf pool. Figure 3.4 demonstrates these two memory pools for sample HDT construction process.

For dynamic volumes in which both the topology and the values of the data can change over time, the layout of the tree cells and the leaf grids in HDT will alter during geometric processing. Due to such consistent allocations and deallocations, the memory pools gradually become fragmented. Hence, the specifically tailored memory manager periodically cleans the buffers, and consolidates empty slots to prevent fragmentation. Further, previous researches [50, 95] have observed that CUDA memory allocation (e.g. *cudaMalloc*) is order of magnitude slower than native CPU memory allotment. Therefore, allocating a small chunk of memory by each of the thousands

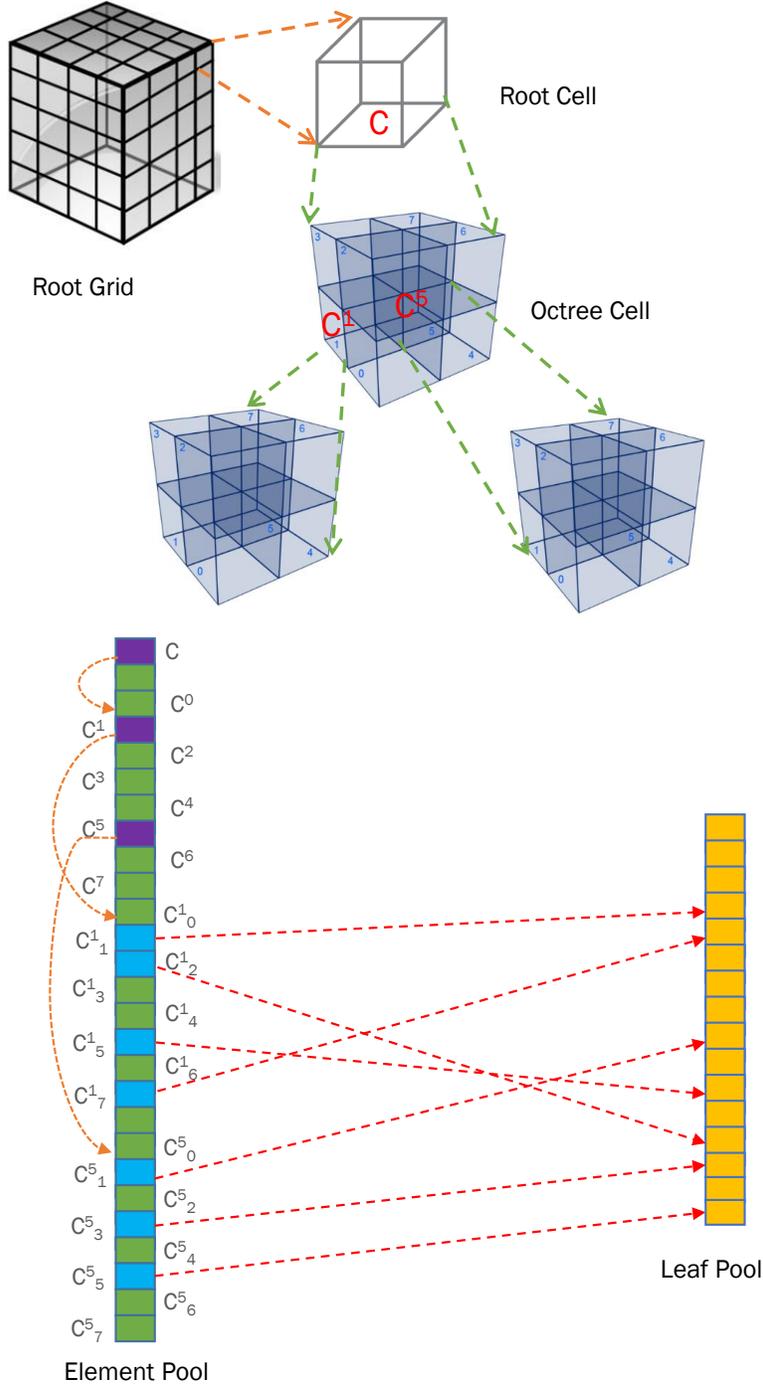


Figure 3.4: Storage implementations for HDT structure on GPU memory. Demonstration shows a sample root cell (C) branching into eight octree cells – all allocated in a contiguous block in the element pool. Two of these first-level octree cells (i.e., C^1 and C^5) get branched into eight octree cells in deeper hierarchy. Out of the eight octants of cell C^1 , four cells (C^1_1 , C^1_2 , C^1_5 and C^1_7) are in *boundary* state, and each is mapped to a leaf grid in the leaf pool. Similarly, for the cell C^5 three descendants (C^5_1 , C^5_3 and C^5_5) are mapped to respective leaf grids in the leaf pool. Except the purple (branch) and blue (boundary) blocks in the element pool, the rest blocks (green) are either full or empty, and not partitioned.

of CUDA threads impose severe performance penalty. To alleviate this bottleneck, HDT allocates a large chunk of GPU memory at a time, and then uses CUDA atomic operations to grant access to the shared memory pool. In the current implementations, GPU memory is allocated for 8,192 entries at a time both for the element pool and the leaf pool.

3.1.5 Storage per Active Voxel

This section presents a theoretical analysis on the storage for each active voxel in HDT. In a sparse voxel representation, such as, HDT, storage can be classified into two types: (1) data and (2) topology. *Data* storage is pertinent to the memory required to store the states of *all* the cells in the leaf grids in HDT. This includes the storage overhead of *non-active voxels* in the leaf grids.

In order to approximate the memory consumption for the topology of each active voxel in HDT, we must first estimate how many cells there are in total in a hierarchy, compared to the number of leaf grids. If we assume an *active* branching factor of K for every HDT cell⁴, and a perfectly balanced hierarchy, we see that for each leaf grid, its parent is shared among K leaves; the parent’s parent among K^2 leaves; and so on. Figure 3.5 demonstrates such a perfectly balanced hierarchy for $K = 4$. During HDT branching, at each hierarchy all the eight child cells are allocated together in the element pool that collectively accounts for 24×8 , i.e., 192 Bytes. Then, for an HDT with a depth d we can formulate the per leaf grid storage (in Bytes), S_L as below:

$$S_L = \frac{192}{K} + \frac{192}{K^2} + \frac{192}{K^3} + \dots + \frac{192}{K^d} \quad (3.1)$$

Simplifying Eq. 3.1 results:

$$S_L = \frac{192}{K-1} \left(1 - \frac{1}{K^d} \right) \quad (3.2)$$

⁴This assumption implies that among the eight descendants of an HDT cell, K cells are of branching state, while the rest $(8 - K)$ cells are of non-branching states.

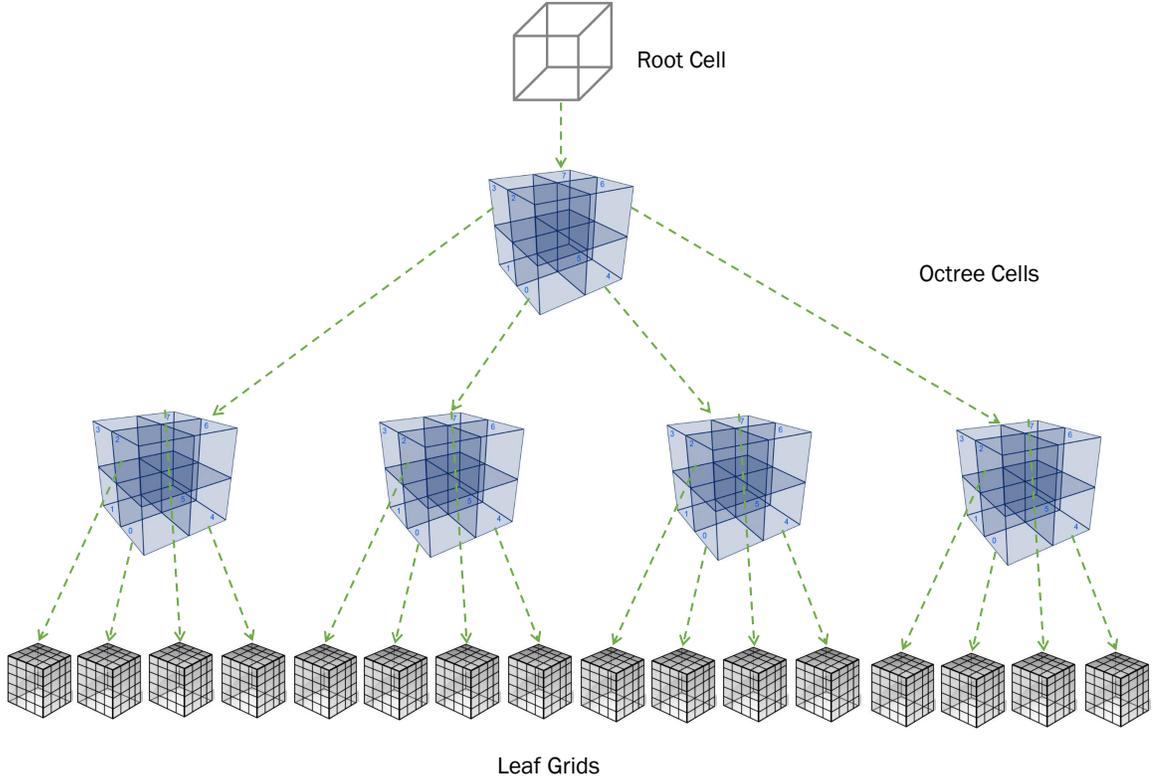


Figure 3.5: Storage analysis of active voxels in HDT. In this example, a balanced octree partitioning (with $K = 4$) for a given root cell is illustrated. Here, a leaf grid is shown as a block of $4 \times 4 \times 4$ voxels instead of general convention (i.e $16 \times 16 \times 16$) for better visualization.

At high-resolutions the value of $\frac{1}{K^d}$ can be neglected to derive an approximate storage per leaf grid in HDT to be $\frac{192}{K-1}$ Bytes, assuming $K > 1$. For possible values of K in range of 2 to 8, Table 3.1 reports the topology storage per leaf grid required for corresponding K . If we assume $K = 4$, similar to the assumption presented in [63, 62], Table 3.1 suggests an average storage of 64 Bytes per leaf grid for maintaining the topology in the sparse hybrid dynamic trees.

Table 3.1: Topology storage per leaf grid for different values of active branching factor.

Active Branching Factor (K)	2	3	4	5	6	7	8
Storage per leaf grid in Bytes (S_L)	192	96	64	48	38	32	27

Now, for the default leaf grid configuration of $16 \times 16 \times 16$, our evaluations (Section 3.3) observe that out of 16^3 voxels in a leaf grid on average roughly 16^2 voxels are active voxels⁵. Considering this observation, we find that each active voxel needs a topology storage of $\frac{64 \times 8}{256} = 2$ bits. Finally, we need to account for the data storage for each active voxel. As each leaf grid with size of $16 \times 16 \times 16$ takes 1024 Byte in the leaf pool, per active voxel data storage is $\frac{1024 \times 8}{256} = 32$ bits. Hence, with the default leaf grid configuration, it requires in total $2 + 32 = 34$ bits of storage per active voxel. We will further show in Section 3.4.3 how the storage can be condensed with alternative leaf grid configuration.

⁵Due to the approach of surface representation in the HDT structure, it can be generalized that a leaf grid of size $2^l \times 2^l \times 2^l$ contains on average $2^l \times 2^l$ active voxels.

3.2 HDT Construction

To make the use of HDTs practical, one needs a scalable way to build it. We present such an algorithm to construct an HDT at high-resolution from a triangle mesh represented in STL (*STereoLithography*⁶ [5]) format. In this voxelization process the input mesh geometry is defined with a set of HDT cells and a set of leaf grids that combinedly capture the surface of the solid object. The STL input is represented as an indexed mesh in CUDA memory, termed as *triangle pool*. And, the resultant HDT is constructed on the two buffers, namely, the element pool and the leaf pool – both buffers are *dynamically* built during the HDT construction process. Figure 3.6 shows a sample STL input (left), and an HDT representation of the mesh at specific resolution (middle); the part of which is zoomed in (right) for better visualization of the voxelized surface.

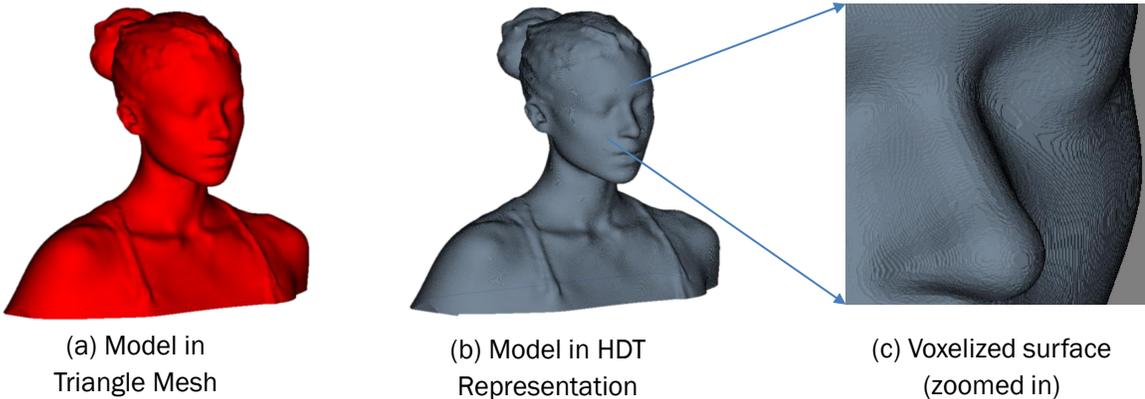


Figure 3.6: Demonstration of a sample 3D model in HDT (source: [2]).

First, we present a high-level illustration on the HDT construction processes in Figure 3.7 on the same 2D cross-section earlier presented in Figure 2.2(b). At the top-most level of HDT construction, each triangle of the mesh is checked for intersection with the bounding box of each root cell [Figure 3.7(a)]. Root cells with no surface

⁶STereoLithography format is one of the most common standards mesh layout widely used in rapid prototyping, where meshes are represented as triangle soups, i.e., as sets of triangles without any additional connectivity information.

intersection (i.e., no overlapping triangle in the input mesh) are not subdivided: the red outlined cells are outside, whereas the green regions lie within the object. In this 2D case, the overlapped root cells are partitioned into four child cells. The cells after two levels of subdivision are shown in the middle [Figure 3.7(b)]. The subdivision continues until the desired resolution is achieved; for this example, the completely refined HDT after another two levels of partition is shown in the right [Figure 3.7(c)]. In the resultant HDT the cells appeared in yellow are the leaf grids each representing a block of $16 \times 16 \times 16$ voxels. For each leaf grid, the triangle intersection check [12] sets the state of individual voxel. The voxels with at least one overlapped triangle are determined as ‘boundary’ (i.e., active). Collectively the set of active voxels defines the solid surface at the finest approximation. Any algorithm operating on HDT represented models requires processing these yellow leaf grids, along with the tree cells that lay out the topology of these leaf grids in the sparse HDT hierarchy. We implemented our parallel HDT construction procedure for NVIDIA GPUs using CUDA. Our implementation consists of three kernels, corresponding to the three major steps of the procedure described below: triangles mapping, HDT branching, and leaf processing.

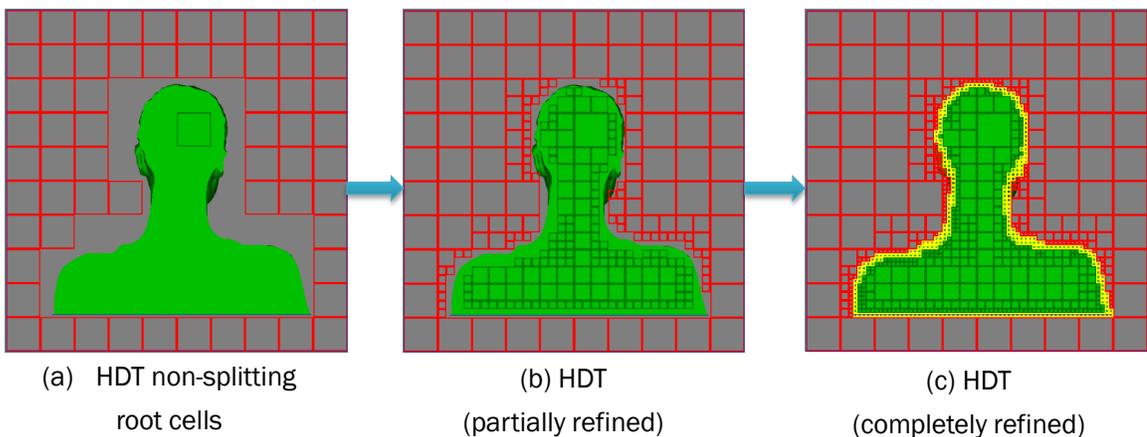


Figure 3.7: HDT construction on a 2D cross-section of a sample 3D model (source: [2]).

3.2.1 Triangle Mapping

HDT construction starts with mapping the triangles of the input mesh to the root cells in the output HDT. As exploiting fine-grained parallelism is a key ingredient to unlock the concurrent processing capability on GPUs, it is critical to distribute the workload across as many CUDA threads as possible. For the triangle mapping, we have two choices: each thread can either process a mesh primitive (i.e., triangle), or an HDT element (i.e., root cell). For a typical configuration, the number of triangles generally outnumbers the the number of root cells by two orders of magnitude. Hence, the most suitable approach to mapping the mesh primitives is to parallelize on the triangles.

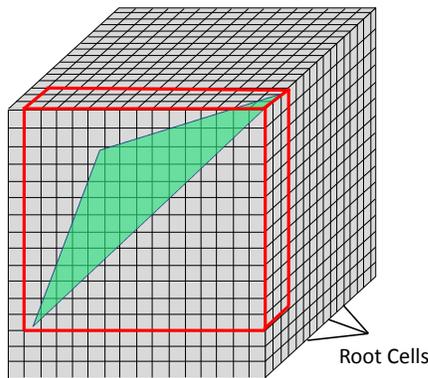


Figure 3.8: HDT triangles mapping.

To map each triangle efficiently onto root cells, the idea is to compute an axis-aligned bounding box for each triangle, and then only check root cells that lie within the bounding region. This step can be performed efficiently using the triangle-box overlap test [12, 11]. Figure 3.8 illustrates this check for a sample triangle. The bounding box is outlined in red; only root cells within this bounding box need to be checked for possible intersection with the triangle boundary. In reality, each triangle overlaps only a small fraction of the total number of root elements (in this example a gigantic triangle is shown for better visual illustration). Then, for each root cell that does intersect, this step records the index of the root cell and a list of indexes of

the overlapping triangles in the triangle pool. To sum up, in first part of the triangle mapping step, we compute the *list of root cells* each triangle intersects; and then subsequently we have to generate the *list of triangles* that each root cell overlaps. Our approach is similar to the scheme presented by Kalojanov et al. [56] in context of a GPU-centric uniform grid construction.

3.2.2 HDT Branching

We refer to the second step as the HDT branching step. It constructs the HDT level-by-level, as other hierarchical tree construction methods for GPUs have done [106, 64, 49]. Unlike the triangle mapping phase, this step parallelizes the computation over the HDT cells. At every iteration, the HDT branching kernel processes the tree cells of the current level in parallel, and generates the child cells for the next iteration. This level-order construction approach removes inter-level data dependencies. At successive tree levels, the size of the cells are halved along each of the three dimensions. Once the size of the cell reaches the target resolution, the hierarchal HDT spanning is terminated; the cell then refers to a leaf grid in the leaf pool.

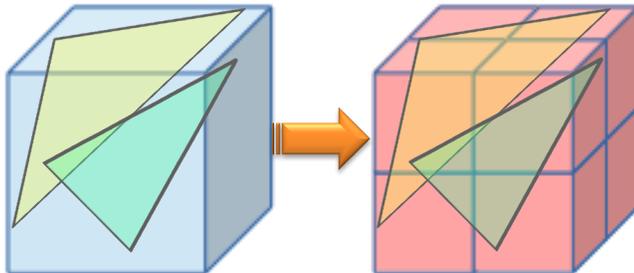


Figure 3.9: Hierarchical branching in HDT construction.

Figure 3.9 demonstrates an example with an HDT cell having two overlapped triangles (left). When this ‘parent’ cell splits into eight child cells (right), all the descendant cells need to check for intersection with the two triangles that intersect their parent. Thus, for the successive level of HDT branching the element pool size is expanded by a factor of eight. Recall that all the device (GPU) memory allocation

is carried on the host (CPU) side at the synchronization point, which is in between the CUDA kernel invocations for successive hierarchies.

3.2.3 Leaf Processing

The third step is to process each voxel in the leaf grids to determine its state (inside, outside, or boundary). As noted above, the voxels in a leaf grid are stored contiguously, and the state of each voxel uses 2 bits. So, a leaf grid of 16^3 voxels would require $16^3 \times 2$ bits or 1024 Bytes. Each leaf grid is assigned to a CUDA thread block. In order to achieve best performance using the CUDA programming model, it is necessary to arrange sequential threads to perform sequential write operations, thus enabling the hardware to amortize memory latency by coalescing several small write requests into one large memory operation. This optimization cannot be achieved when CUDA threads in a warp write to incoherent memory addresses.

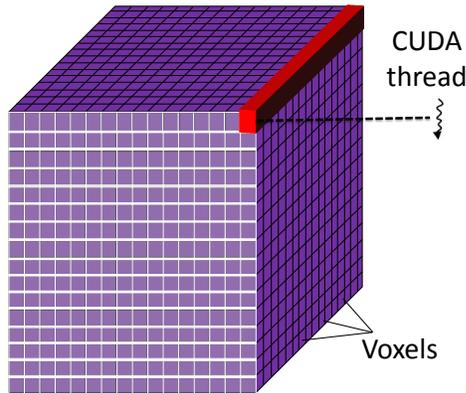


Figure 3.10: Leaf grid processing in a CUDA thread block.

For the leaf processing, each CUDA thread block is configured with 16×16 threads. As, we have 16^3 voxels in a leaf grid, each CUDA thread processes 16 consecutive voxels as illustrated in Figure 3.10. As each voxel state is encoded in 2 bits, the states of 16 contiguous cells are stored in a block of 16×2 bits or 4 Bytes in GPU memory. Hence, each thread in a CUDA warp reads a 4 Byte word, works on the 16 corresponding voxels, and writes back the modified word to memory. Thus, a CUDA thread block processes altogether 256×4 bytes, i.e., 1024 bytes data of 16^3 cells of a

grid. To optimally utilize the GPU memory bandwidth, any CUDA kernel working on the leaf grids adopts such GPU execution-model centric design and implementation.

3.3 Experiments and Analysis

We analyzed the GPU implementation of our HDT construction procedure on a benchmark suite of freeform CAD models [60], suitable for rapid prototyping by 3D printing and computer numerical control (CNC) milling. The experiments were carried out on a workstation with a quad-core Intel Core i7-4770K 3.5 GHz CPU and an NVIDIA GTX 780Ti GPU with 3 GB of GPU memory.

3.3.1 HDT Benchmarking

A summary of the input meshes and their detailed characteristics after conversion to HDT appears in Tables 3.2 and 3.3. Each model name is appended with the number of triangles in respective STL inputs. The first row in the tables shows the mesh surface area, followed by two rows presenting the model’s physical dimension and the bounding volume, respectively ⁷. For each model, we evaluated 4 different HDT configurations, specified by target resolution (1024^3 to 8192^3), assuming root grid and leaf grid dimensions of 16^3 . Thus, the maximum height of the octree in the respective HDT configurations falls between 2 to 5 levels, shown as “HDT Octree-Height” in the tables. The “Voxel Size” row lists the corresponding effective physical voxel sizes, set between $120\ \mu\text{m}$ and $15\ \mu\text{m}$ across the HDT configurations.

Tables 3.2 and 3.3 also report the total number of tree cells and the number of leaf grids in the HDT at each resolution. As expected, the number of leaf grids correlates with the surface area of the triangle mesh, showing that sparsity is being exploited. For instance, the Head model has $6\times$ the number of triangles as Candle Holder, but the mesh surface area in the latter is 22% larger than the former. The leaf grid counts in Candle Holder proportionally (20%) higher than Head at 8192^3 resolution. Moreover, as the surface voxels of a solid geometry increase quadratically, at $2\times$ grid resolution the number of leaf grids grows by a factor of four.

The total number of leaf voxels is the product of the leaf grid count and the voxels

⁷The dimensions are arbitrary to within a constant scaling factor.

Table 3.2: Geometric statistics of CAD models: Head and Dragon.

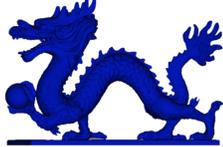
	 Head (230k)				 Dragon (173k)			
Mesh Surface Area (mm^2)	8,956				10,747			
Dimensions XYZ (mm)	48.6 x 46.0 x 64.4				46.7 x 74.6 x 50.4			
Bounding Volume (mm^3)	143,923				175,577			
Effective Resolution	1024	2048	4096	8192	1024	2048	4096	8192
HDT Octree-Height	2	3	4	5	2	3	4	5
Voxel Size (μm)	120	60	30	15	120	60	30	15
HDT Cells ($\times 10^3$)	10	37	146	581	10	36	150	621
HDT Leaf Grids ($\times 10^3$)	3.32	13.7	54	218	3.33	14.3	60	240
HDT Leaf Voxels ($\times 10^6$)	14	56	223	894	14	59	245	985
Active Voxels in HDT ($\times 10^6$)	0.87	3.51	14.0	56.0	0.98	3.96	15.8	63.4
Topology Storage (bits)	2.21	2.01	1.99	1.99	1.92	1.74	1.82	1.88

Table 3.3: Geometric statistics of CAD models: Turbine and Candle Holder.

	 Turbine (58k)				 Candle Holder (38k)			
Mesh Surface Area (mm^2)	12,346				10,987			
Dimensions XYZ (mm)	48.9 x 48.9 x 31.1				48.4 x 48.9 x 57.7			
Bounding Volume (mm^3)	74,367				136,515			
Effective Resolution	1024	2048	4096	8192	1024	2048	4096	8192
HDT Octree-Height	2	3	4	5	2	3	4	5
Voxel Size (μm)	120	60	30	15	120	60	30	15
HDT Cells ($\times 10^3$)	8	36	153	684	11	44	172	698
HDT Leaf Grids ($\times 10^3$)	3.03	14.9	66	291	3.99	16.6	65	262
HDT Leaf Voxels ($\times 10^6$)	12	61	271	1193	16	68	267	1074
Active Voxels in HDT ($\times 10^6$)	1.17	4.67	18.7	74.7	1.05	4.20	16.8	67.2
Topology Storage (bits)	1.30	1.46	1.57	1.76	1.94	1.99	1.96	1.99

per grid ($16^3 = 4096$). Thus, at an effective 8192^3 resolution with the Turbine model, the HDT stores 1.19 billion voxels instead of $8192 \times 8192 \times 8192 \approx 550$ billion in a fully dense representation. To study the storage effectiveness of HDT, the final two rows of Tables 3.2 and 3.3 show the number of active voxels and the topology storage per active voxel, respectively. At 8192^3 resolution, each boundary voxel in HDT accounts for an average of 1.76 – 1.99 bits storage, which closely matches with the theoretical analysis on the topology storage presented in Section 3.1.5.

3.3.2 Storage Comparisons

We compare the storage requirement in the HDT with prior state-of-the-art voxel data structures suited for high-resolutions volume modeling. The efficient sparse voxel octree (ESVO) representation by Laine et al. [63, 62] derived an storage of 1.33 Byte ≈ 11 bits for the topology of each active voxel. For the data storage, each voxel requires a minimum of 1 Byte memory in the ESVO representation, thus consuming a total of $11 + 8 = 19$ bits per active voxel. It is no surprise that per active voxel storage in the ESVO is significantly lower than that is required in the HDT with the default configurations of root grid and leaf grid. The HDT is designed to harness the power of massive parallel computing fabric on GPUs to push the capability of the computation efficiency beyond the realm of multicore CPUs. On the other hand, while spatial structure like VDB [80] is amenable for parallelization on graphics hardware, it consumes significant storage overhead than that of HDT. The per active voxel storage in VDB is in range of 60-72 bits due to the use of large branching factors, custom-tailored software-level caching, among others. Thus, HDT demonstrates much efficient storage-compact representation, which is critical for extreme-scale volume modeling on GPUs equipped with an order of magnitude smaller memory than the system memory available to CPUs.

3.3.3 Discussion on HDT Construction

The execution time of the three HDT construction steps versus resolution appears in Fig. 3.11. We denote the triangles mapping time by T_1 , the HDT branching time by T_2 , and the leaf processing time by T_3 . Per Section 3.2.1, triangles mapping scales with the product of the face count by the number of elements in the root grid. Hence, for a specific model the triangles mapping time is independent of the target resolution, which appears as T_1 remaining unchanged across different HDT configurations. However, T_1 does vary with the number of triangles. At 8192^3 resolution, T_1 consumes about 2% of total HDT construction time for Head and Dragon, whereas for Turbine and Candle Holder T_1 is less than 1% of total GPU time. Nevertheless, T_1 is overall a small fraction of the total HDT construction time.

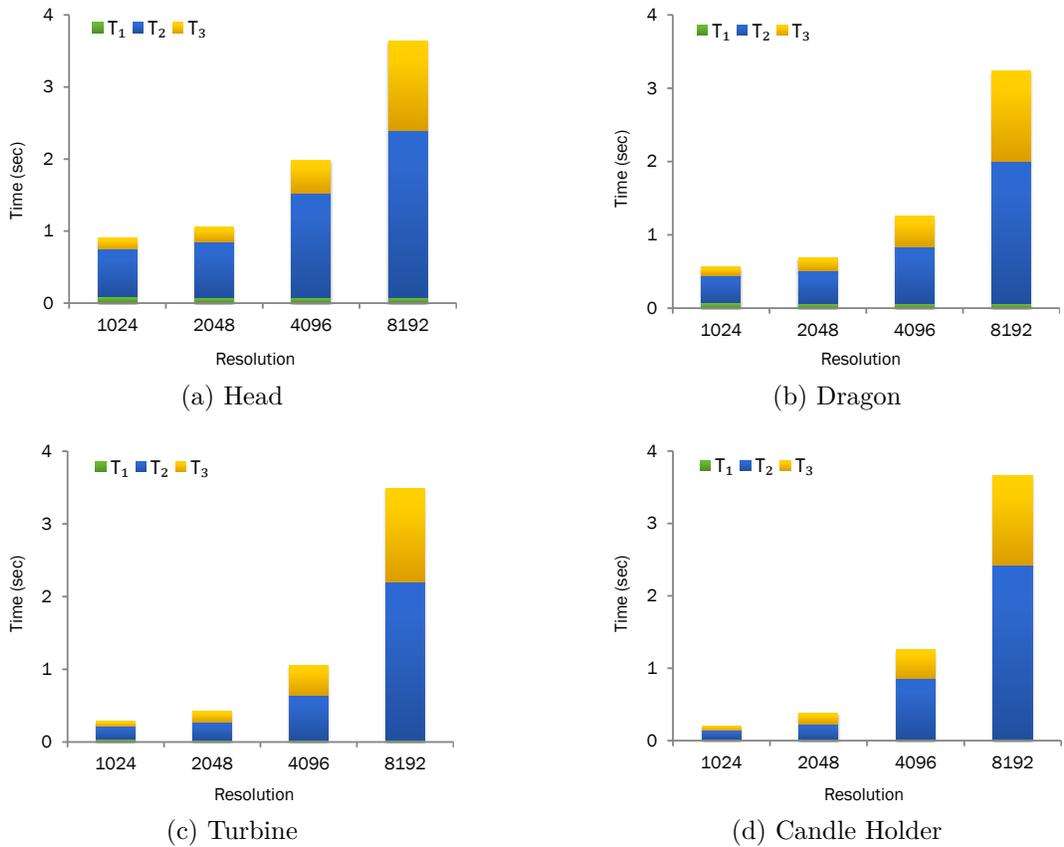


Figure 3.11: Measurements on HDT construction time.

The HDT branching time (T_2) tends to dominate the total time. As shown in Fig. 3.11, the relative magnitudes of T_2 values for different models correlates with the number of tree elements reported in Tables 3.2 and 3.3. For instance, at 8192^3 resolution, the Dragon, Turbine and Candle Holder models have 621, 684 and 698 thousands tree elements respectively; and this trend in HDT cell counts scales with the corresponding T_2 values of 1.93 seconds, 2.17 seconds and 2.39 seconds.

Similarly, the HDT leaf processing times (T_3) correspond to the number of leaf grids. However, the differences in T_3 across models is hardly distinguishable in Fig. 3.11. Since the leaf processing step parallelizes relatively easily, the GPU can process several hundreds of millions of voxels per second. Hence, the T_3 values lie in a tight interval; for example, at 8192^3 resolution, the T_3 measurements for the four models all lie between 1.23 to 1.29 seconds.

One notable observation in Fig. 3.11 is that the complete HDT construction times across different CAD models are not directly related to the triangle counts. For instance, consider the comparative measurements of T_1 , T_2 and T_3 for Head and Candle Holder at 8192^3 resolution: the total construction times for Head and Candle Holder are similar, even though the former makes up $6\times$ the number of triangles than the latter. The triangles mapping time only accounts for 1-2% of total time, which is the only component of HDT construction affected by triangle count. Also, the 20% higher leaf grid counts in Candle Holder relative to Head make no noticeable difference in leaf processing time (T_3). Lastly, the branching time (T_2) in Candle Holder is marginally higher due to more tree cells that offset the timing difference of triangles mapping time (T_1). These findings are consistent with those of the efficient sparse voxel octree study, where two particular models take similar amounts of time while they differing in triangle count by $3\times$ (“Fairy” and “Conference” models [63]).

3.3.4 Discussion on GPU Speedups

We also measured the speedups of the GPU-accelerated HDT construction against a single-thread CPU implementation, and report these results in Figure 3.12. GPU implementation of triangles mapping is $2.9\text{--}4.7\times$ faster; and curiously, higher speedups are demonstrated for meshes with more triangles. These experimental results corroborate a general hypothesis in GPGPU parallelization that suggests a higher acceleration with larger parallel workloads [43]. Due to amortization of the CUDA related overhead across massive workloads, the speedups in triangle mapping step are higher when the number of faces are higher; e.g., compare Head and Dragon to Turbine and Candle Holder.

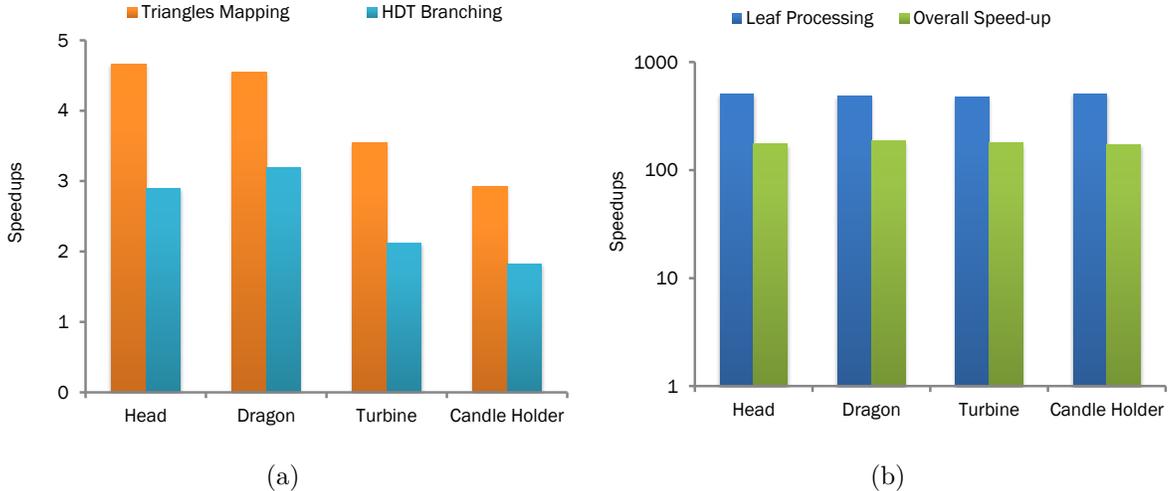


Figure 3.12: GPU-speedups of HDT construction at $8192 \times 8192 \times 8192$ resolution.

For HDT branching, GPU computation demonstrates a modest speedup between 1.8 and $3.2\times$. Unlike the speedups of triangles-mapping step, the HDT branching phase achieves higher acceleration for models with lower number of tree cells. This is due to the level-order HDT branching that incurs significant overhead for repeated transfers of memory pools across the boundary of the host and the device, particularly for the test configuration at 8192^3 resolution where the octree height in HDT reaches a depth of 5 levels. Lastly, as shown in Fig. 3.12(b) (on a logarithmic scale),

leaf processing benefits significantly from GPU parallelization, exceeding over $500\times$ speedup for all the sample models. Since the overall speedups are closer to the leaf processing speedup than the triangles-mapping and HDT branching speedups, leaf processing evidently dominates the CPU implementation's execution time by a large margin. Interestingly, leaf processing becomes so much faster on the GPU that it no longer dominates the execution time (cf. Figure 3.11). Comprising all the three operations, it should be noted that the overall speedups are very high, in the range of $173\text{--}187\times$.

3.4 Tunable Hybrid Dynamic Tree

3.4.1 Impact of the Root Grid Dimension

No single configuration of any spatial data structure can claim to handle all applications equally well, and HDT is no exception. Different combinations of the cells and their parameters can alter the tree depth and branching factors, which in turn impact characteristics like adaptivity, performance of tree traversal, memory footprint, available grid resolution, and even hardware efficiency. Conceptually, the characteristics of the HDT structure can be tuned by varying three parameters: (1) the root grid dimension, (2) the branching factor of intermediate-level tree cells, and (3) the leaf grid dimension. For the convenience of discussion, let us first formulate a notation to specify an HDT configuration with these three parameters. Let us assume an HDT with configuration of root grid dimension $R = 2^r$, branching factor $B = 2^b$ and leaf grid dimension $L = 2^l$, then for a given resolution X we get an HDT with height h that satisfies the following condition:

$$2^{(r+b \times h+l)} \geq X \tag{3.3}$$

For a modeling resolution X and particular assignments to two of the three parameters, the remaining parameter can be tuned to realize its impact. A particular interest is to study the impact on the storage required for HDT representation, and the behavior of HDT branching time, which is the dominating part among the HDT construction steps. Our first study focuses on the root grid dimension (R), which affects the depth of the HDT hierarchy that in turn impacts the HDT branching time. For a default leaf grid dimension (L) of 16 and branching factor (B) of 2, we analyze the impact of the root grid dimension in Fig. 3.13 with R set in between 128 and 4. Hence, the HDT hierarchy spans between 2 (for dimension of 128) and 7 (for dimension of 4) at 8192^3 resolution.

For all the four test models, we observe a similar trend in HDT branching time and

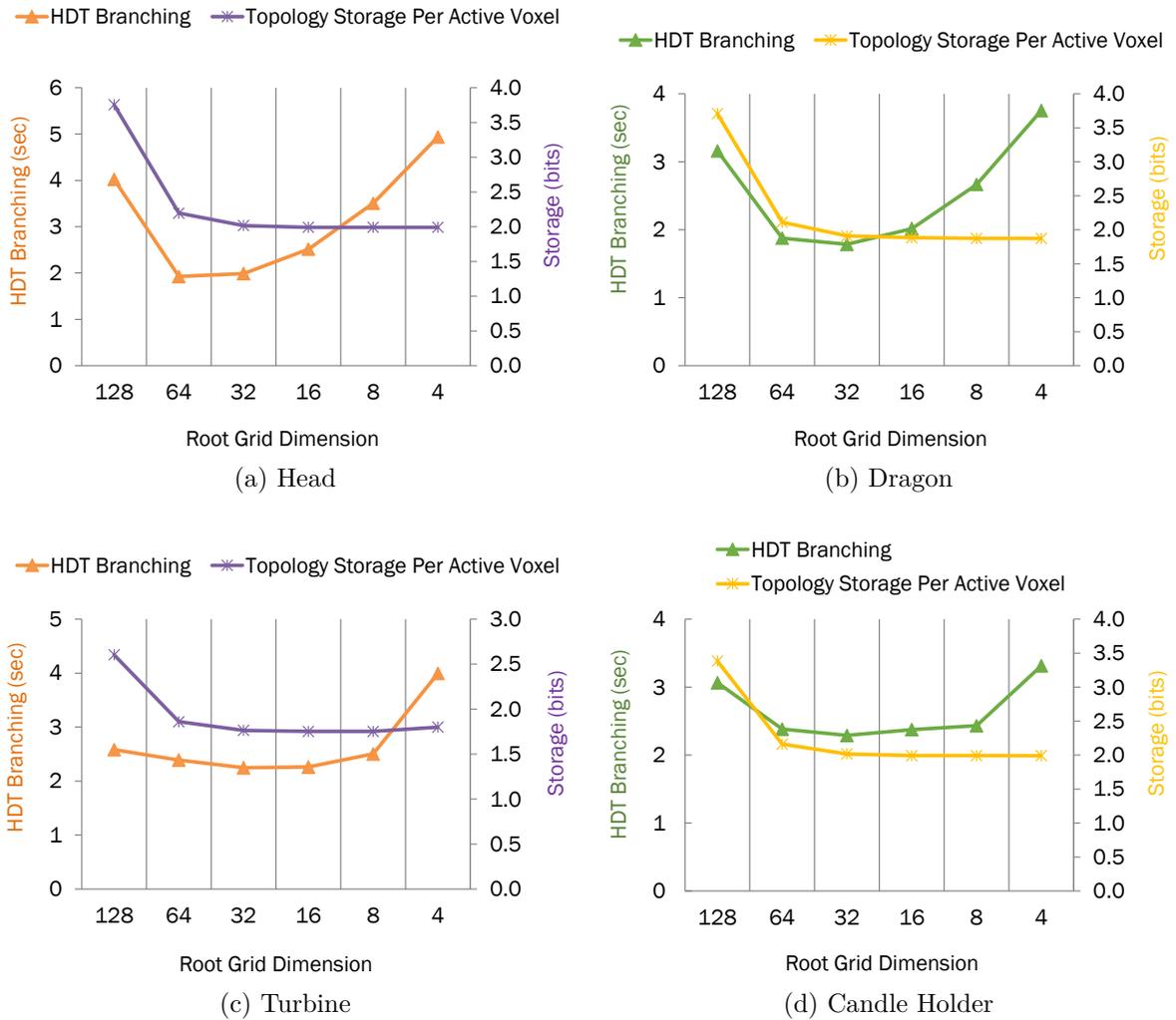


Figure 3.13: Impact of different root dimensions at $8192 \times 8192 \times 8192$ resolution.

the topology storage requirement per active voxel. HDT branching time is optimal for R in between 16 and 64. With smaller R , HDT gets taller; that slows down HDT branching process. On the other hand, with a very large root grid dimension ($R = 128$) the adaptivity in spatial refinement decreases, which leads to the rise in HDT branching time as shown in Fig. 3.13. Curiously, the reduced adaptivity at larger R translates to relatively higher number of tree cells in the HDT. This is reflected in Fig. 3.13 with the jump in bits per active voxel for the topology storage.

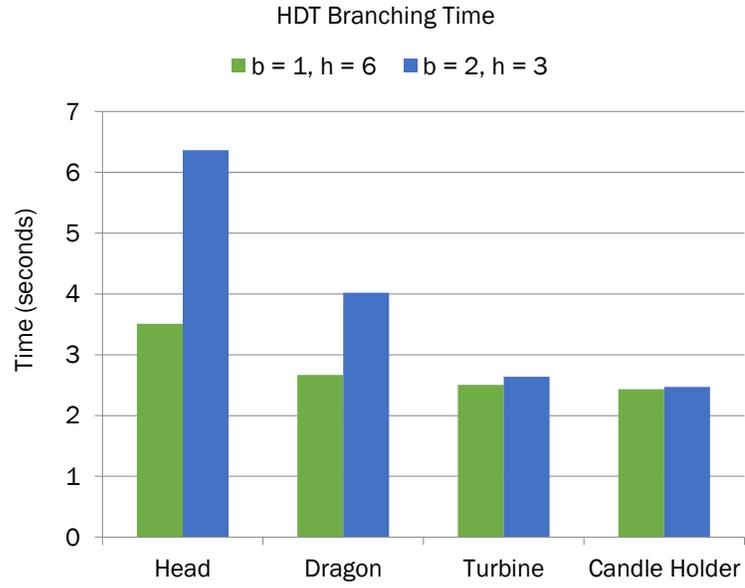
Here, an interesting observation is that with very small root grid dimension, such as, $R = 4$, HDT branching takes significantly longer than the optimal configuration

for the models with non-regular pattern, such as, Head and Dragon, compared to the trend observed in models having a repeated pattern, like, Turbine and Candle Holder. Our experimentation on the HDT height with tunable root dimension parameter thus validates the choice for HDT’s shallow hierarchy in extreme-scale volume representation. These results suggest that a shallower tree is preferred. However, one cannot necessarily choose arbitrarily large root grid, since storage overhead grows with the root grid dimension. In this case, a root grid dimension of 128 is exactly the point at which storage per active voxel begins to increase for this input model, as Fig. 3.13 demonstrates.

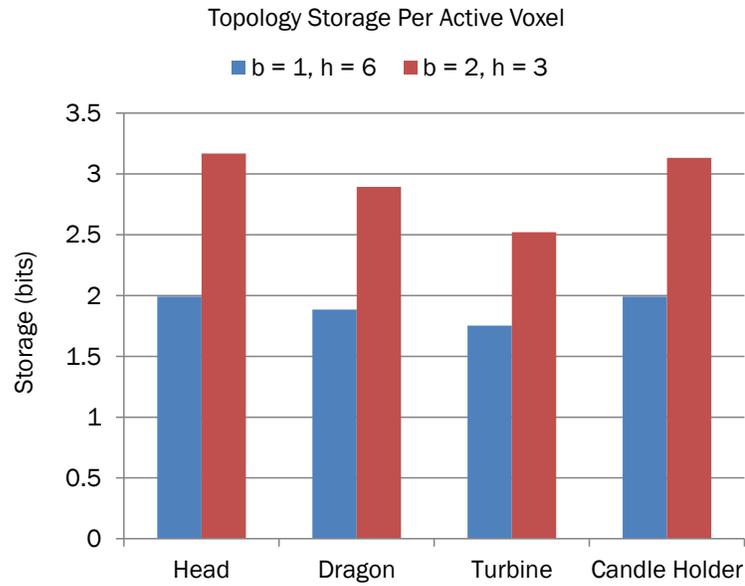
3.4.2 Impact of the Branching Factor

In this study, with the root grid dimension (R) set to 8 and the leaf grid dimension (L) set to default 16, HDT branching time and topology storage are analyzed at two branching factors: for $B = 2$ (i.e., $b = 1$) and $B = 4$ (i.e., $b = 2$). For these settings, the tree hierarchy in HDT is respectively 6 and 3 (cf. Eq. 3.3). From the HDT branching times shown in Fig. 3.14(a), we see that with larger B branching takes longer due to decrease in spatial adaptivity. Interestingly, for the models with regular patterns (i.e., Turbine and Candle Holder) larger B does not impact the adaptivity noticeably, unlike in the case of Head or Dragon.

However, for the topology storage the bits per active voxel increase in similar scale for all the models, as depicted in Fig. 3.14(b). With a larger branching factor, for instance $B = 4$, a cell is decomposed into 4^3 child cells. As each cell occupies 24 Bytes in the element pool, each cell partitioning consumes a total storage of $4^3 \times 24$ Bytes = 1536 Bytes. As with larger B , the adaptivity in spatial partitioning scales down, thus the ratio of the number of HDT cells to the number of leaf grids increase. For instance, with Candle Holder the above ratio rises to 4.23 (for $B = 4$) from 2.66 (for default $B = 2$) that translates to the proportionate increase in the topology storage with larger B .



(a) HDT Branching Time



(b) Topology Storage Per Active Voxel

Figure 3.14: Impact of different branching factors at $8192 \times 8192 \times 8192$ resolution.

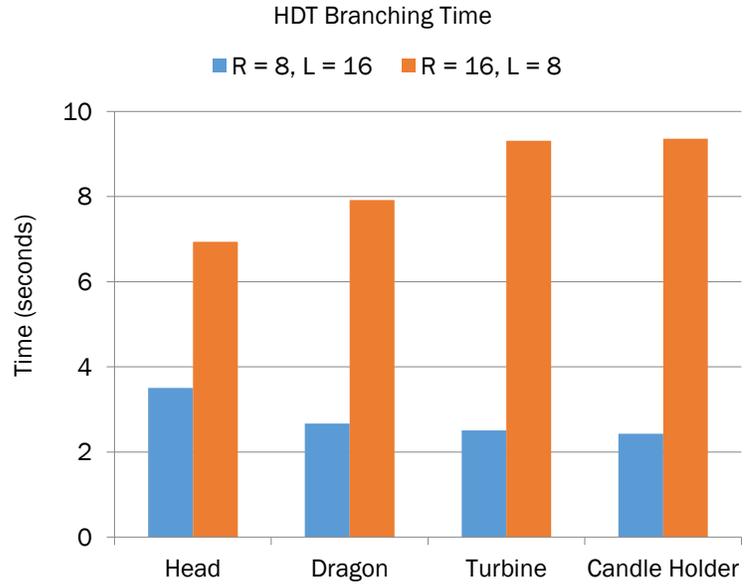
3.4.3 Impact of the Leaf Grid Dimension

In this study, with the tree branching factor fixed to $B = 2$ (i.e., $b = 1$), the root grid dimension (R) and the leaf grid dimension (L) are set to two different settings. In the first configuration, R and L are set to 8 and 16 respectively, while in the second configuration R and L are set to 16 and 8 respectively. Such assignments to R and L are chosen so that for these two configurations, the tree hierarchy in HDT remains fixed to 6 (cf. Eq. 3.3) to avoid any discrepancy incurred on HDT branching time due to different depths in HDT hierarchy.

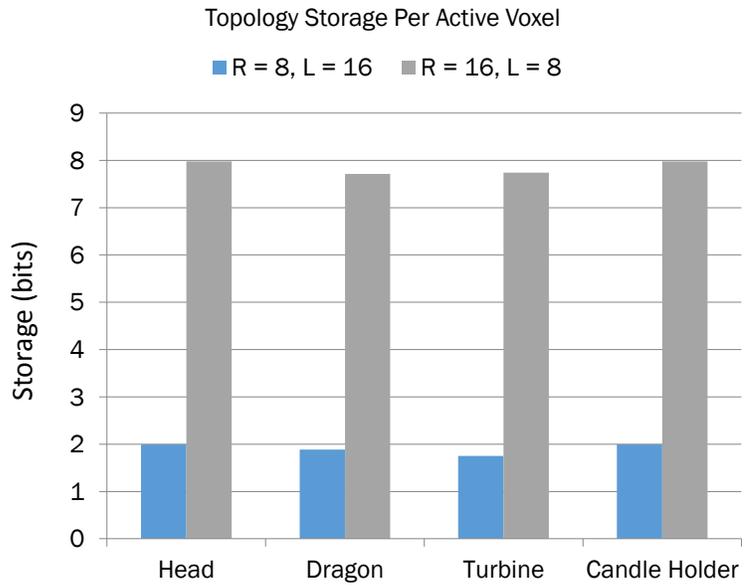
With the dimension of leaf grid gets halved, the number of leaf grids in HDT roughly increases by a factor of four⁸. With the leaf grids count increased to $4\times$, the number of the cells in HDT roughly increases by a similar factor with the $L = 8$ compared to $L = 16$. This translates to $4\times$ bits required for the topology storage of each active voxel in the HDT configured with $L = 8$, as shown in Fig. 3.15(b). From the branching times presented in Fig. 3.15(a), as we expect branching takes significantly longer with $L = 8$ due to processing approximately $4\times$ more tree cells during the HDT construction. Curiously, for the CAD models with non-regular patterns, such as, Head and Dragon HDT branching times increase modestly ($2.0 - 2.8\times$) than the models with repeated patterns, like, Turbine and Candle Holder where the branching times jump to roughly $4\times$.

To complete the analysis on different leaf dimensions, we studied the total storage requirement per active voxel for $L = 8$ relative to that with $L = 16$. As described earlier, with $L = 8$ the topology storage goes up by $4\times$. Thus, with reference to the theoretical storage analysis in HDT as presented in Section 3.1.5, we get an average topology storage of 8 bits per active voxel with $L = 8$. Then, to quantify the data storage per active voxel, we need to count two measurements: (a) the storage

⁸As HDT models the surface of a solid, this trend in number of leaf grids with smaller leaf grid dimension mimics the trend in statistics of the leaf grids at successively higher resolutions (cf. Tables 3.2 and 3.3).



(a) HDT Branching Time



(b) Topology Storage Per Active Voxel

Figure 3.15: Impact of different leaf dimensions at $8192 \times 8192 \times 8192$ resolution.

requirement for each leaf grid with $L = 8$ in the leaf pool, and (b) the average number of voxels in each leaf grid with ‘boundary’ state. With dimension set to 8, each leaf grid accounts total $8^3 \times 2$ bits, i.e., 1024 bits = 128 Bytes. Now, for the leaf grid configuration of $8 \times 8 \times 8$, on average roughly 8^2 voxels are active among 8^3 voxels in a leaf grid. Considering this observation, we find that per active voxel data storage is $\frac{128 \times 8}{64} = 16$ bits.

Hence, with the leaf grid dimension of $L = 8$, it requires in total $8 + 16 = 24$ bits of storage per active voxel, which is 29% smaller than the total storage needed with the default leaf grid configuration of $L = 16$. Thus, HDT is designed as a configurable sparse data structure to trade-off between computation efficiency and memory efficiency. As we have just analyzed here, while the use of a smaller leaf grid dimension takes longer to construct the HDT, voxels can be presented more compactly to improve the memory efficiency of the underlying data representation. Further, a smaller grid dimension of $L = 8$ bridges the gap between the per active voxel storages between HDT and that of the efficient sparse voxel octree [63, 62], as presented in Section 3.3.2.

By contrast to other approaches, like SVO [63] or VDB [80], different components in HDT cell are assumed to be extensible beyond the practical resource limits. For instance, in SVO a non-leaf node uses 16-bit pointer to locate the descendants that can be extended to maximum 32 bits using a level of indirection through “far” flag bit. The size of this pointer field indicates how far two related parent-child nodes can be spaced in the buffer. Practically, in HDT we can adopt 32-bits pointer in place of 64-bits that seems perfectly sufficient, as the number of tree cells for the highest resolution is less than even 2^{20} . Hence, the maximum possible distance between index of two cells in the memory pools can be well accommodated in far less bits than currently used 64 bits pointer.

Similarly, for “depth” field adoption of even 5 bits can hold the depth information

of an octree with $2^5 - 1 = 31$ levels. In practice, for instance at 8192^3 resolution with a root dimension of 64, and leaf dimension of 16, the tree depth is only $\log_{64 \cdot 16} \frac{8192}{64 \cdot 16} = 3$. Finally, 19 bits for each of X, Y, and Z coordinates can represent up to $2^{19} = 512K$ resolution along each dimension; which is in line with VDB's 20 bits for each coordinate.

With such more realistic assumption, per cell storage in the HDT can be reduced to 2 (state) + 5 (depth) + 32 (pointer) + 3×19 (x, y and z) = 96 bits = 12 bytes. Using 12 bytes storage per cell, in stead of previously adopted 24 bytes per cell, the average topology storage with default leaf dimension of 16 reduces to 1. Table 3.4 reports the average data, topology and total storage per boundary voxel with leaf dimensions of 16 and 8. Using a compact cell storage and adopting a smaller leaf dimension of 8, thus the theoretical voxel storage per active voxel in HDT comes to 20 bits — closely matched with that of SVO's 19 bits.

Table 3.4: Total storage per active voxel with different leaf dimensions using the compact storage for cell components.

	Average Data Storage	Average Topology Storage	Average Total Storage
Leaf dimension of 16	$16 \times 2 \text{ bits} = 32 \text{ bits}$	1 bit	33 bits
Leaf dimension of 8	$8 \times 2 \text{ bits} = 16 \text{ bits}$	$4 \times 1 \text{ bit} = 4 \text{ bits}$	20 bits

CHAPTER 4

VOXEL OFFSETTING USING HYBRID DYNAMIC TREE

4.1 Overview on Offsetting Techniques

The goal in this dissertation is to enable the execution of computation-intensive voxel offsetting within a practical time period in interactive CAM applications. Offsetting is a fundamental geometry processing, and has been studied in CAD/CAM, robotics and related areas for more than three decades [?]. The mathematical basis for offsetting of solids was comprehensively studied in an earlier work by Rossignac and Requicha [88]. There the offsetting operation is introduced as a new solid-to-solid transformation and associated with methods like filleting and rounding of solids. While the offsetting operation for curves and surfaces is well known [45, 75], the complexity of offsetting increases significantly for a 3D model, because the offsetting must handle both the individual surfaces in the model as well as topological reconstruction by trimming and reconnecting the offset surfaces into a closed model.

Although an offsetting operation is mathematically well defined, existing state-of-the-art CAD/CAM packages lack the capability of automatic computation of offset surface for a free-form solid. Such a lack of automation in modern CAD/CAM software increases manufacturing costs, because it takes more human time and expertise to generate collision free tool trajectories manually. For a given triangle mesh input, the offset can be represented as a composition of the primitives associated with each component of the input. For two-dimensional case, as shown in Figure 4.1, every point is associated with a circle and every line is associated with a rectangle. Similarly, generating offsets of a triangular mesh for a given 3D model can be decomposed into a set of spheres, cylinders, and prisms corresponding to respectively vertices, edges, and faces of the mesh. A constructive solid geometry approach to offset surface

computation can be defined with the union of all these elements. Though this looks simple mathematically, a robust and efficient implementation of the union operation of a large number of higher order surfaces has been proven difficult because of its computational complexity and numerical instability.

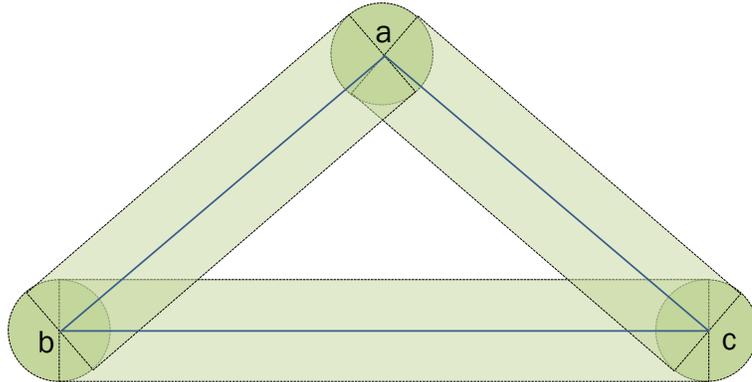


Figure 4.1: Offsetting a triangle in 2D.

4.1.1 Exact Offsetting

Early approaches rely on convolutions to compute offset surfaces and Minkowski sums [6, 94]. These methods obtain a superset of primitives of the offset surface that are trimmed and filtered to form the final boundary [37]. The Minkowski sum of two sets $A, B \in \mathbb{R}^3$, denoted $A \oplus B$, is defined as the set $\{a + b \mid a \in A, b \in B\}$. Planar Minkowski sums are used in many applications, such as, motion planning and computer-aided design and manufacturing. If A and B represent polygons in \mathbb{R}^2 or polyhedra in \mathbb{R}^3 , $A \oplus B$ can be generated by *sweeping* A along the boundary of B and then taking the union of the sweep with B (or vice versa). From the definition of Minkowski sum, offsetting can be considered as a special case, with one operand being an origin-centered disc (in two-dimensional case) or sphere (in three-dimensional case). In the general case, the actual offset surface of a polygonal surface is not polygonal: it contains cylindrical and spherical parts. However, polygonally approximated spheres can be used to obtain (arbitrarily precise) polygonal approximations of an offset surface.

Although the Minkowski sum of two convex polyhedra has complexity of $\mathcal{O}(mn)$ (here m and n denote the numbers of triangles of each input polyhedron) and can be computed very efficiently [40], the Minkowski sum of two non-convex polyhedra can have complexity as high as $\mathcal{O}(m^3n^3)$ and becomes much more difficult to compute. Most algorithms for non-convex objects first decompose the input non-convex polyhedra into convex pieces, computing all the pairwise Minkowski sums of these convex pieces, and then taking their union. As there are $\mathcal{O}(mn)$ pairwise Minkowski sums, and for given p polyhedral objects their union can have combinatorial complexity of $\mathcal{O}(p^3)$, the union step dominates the whole algorithmic complexity [17].

4.1.2 Approximate Offsetting

To overcome the difficulties with exact offset computation, many offset computation methods based on the discrete representation of the 3D model have been proposed. These representation schemes utilize points, voxels [85], dexels [100] and rays [77], among others. As discrete 3D models do not have surface elements, the topological reconstruction step, which is the most critical process in conventional offsetting, is not necessary. After offsetting, a polyhedral model of the offset shape is derived by applying some surface extraction technology, such as marching cubes [73] to the discrete model.

4.1.2.1 Point-based Offsetting

Chen et al. proposed a point-based offsetting method [27, 28] in which points are densely sampled on the surface of the input polyhedral model. Candidate points on the offset surface are generated by simply shifting the points on the polygon or replacing points on the vertices and edges with points on spheres and cylinders. After removing points located inside the offset model, a polygonal offset mesh is generated by connecting the remaining points. The authors used a regular voxel grid, i.e., the complexity grows cubically with the voxel resolution.

As offsetting a 3D object can be recognized as a Minkowski sum between the object and a sphere of the offset radius, Lien [71] proposed a point-based Minkowski sum operation whereby the surfaces of two objects are converted to two groups of points and then summed. Points located inside the Minkowski sum object are discarded by applying a series of filters to determine the offset surface. The most expensive process in point-based offsetting is this filtering step. An inherent limitation with point set based representation is that analyzing a point cloud is challenging as no explicit topological space is available. Recently, Calderon and Boubekur [25] introduced a morphological analysis framework for point clouds, which is able to perform morphological dilations and erosions. These operations remain expensive on point sets as the interior of the solid is not explicitly available.

4.1.2.2 Voxel-based Offsetting

Another group of methods generates a voxelization of the offset surface. Li and McMains [68, 69, 70] presented GPU approaches to compute the Minkowski sum of polyhedra by computing pairwise Minkowski sums, and obtaining a voxelization of its union. Unfortunately, the use of such spatial grid-based offsetting methods can only support mid-scale resolution, as the memory requirements of these methods rise rapidly as the voxelization resolution increases. The authors demonstrated GPU-accelerated Minkowski sum computation in tens of seconds for two triangle meshes having small number of faces at resolutions up to 512^3 [70].

4.1.2.3 Ray-based Offsetting

There exist few offsetting methods that consider ray representations. VanHook introduced the dixel structure [100] as a medium of ray representation of solids. For a single direction and a uniform grid of rays parallel to that direction, the dixel structure stores the intervals of the rays lying inside the solid. These intervals, called dexels (depth elements), collectively represent the solid. Menon and Voelcker [78] suggested approximating the Minkowski sum between A and B by computing the

union of some ray-rep instances of A over the boundary of B .

4.1.2.4 *Offsetting with Distance Field*

The offset surface can also be extracted from the distance field of the embedding space of the solid. A distance field can be represented as a spatial grid structure that implicitly represents the shape, in which the distance from the point to the closest surface of the object is recorded at each grid point. The distance may be signed to distinguish between the inside and outside of the shape. Among other operations, distance field is able to perform surface offsetting [22, 21, 96, 54, 83, 72]. For a given offset radius r , the offset surface of the model goes across an edge connecting a grid point with a distance greater than r with another point whose distance is less than r .

While the distance field is an effective representation of shape, regularly sampled distance fields have drawbacks because of their size and limited resolution. Because fine detail requires dense sampling, immense volumes are needed to accurately represent classical distance fields with regular sampling when any fine detail is present, even when the fine detail occupies only a small fraction of the volume. To overcome this limitation, Frisken et al. [39] presented the adaptively sampled distance field (ADF) that uses adaptive, detail-directed sampling, with high sampling rates in regions where the distance field contains fine detail and low sampling rates where the field varies smoothly. The demonstrated approach of ADF implementation stores distance values at cell vertices of an octree data structure.

Although ADF based implicit model representation is storage-efficient, yet the offsetting construction on an adaptively sampled distance-field still can be extremely computation intensive [96, 18]. To address the computational challenge, Bastos and Celes [18] leveraged GPUs to speedup the adaptive distance field computation that employed a 3D hashing scheme to store the underlying data structure. For the Stanford Armadillo model [1], the authors reported an execution time of 562 seconds to compute the ADF at a resolution of 237^3 . Pavić and Kobbelt [85] presented an

hierarchical algorithm where the offset surface is intermediately represented by an adaptively refined octree. Their approach traverses an octree and splits each cell depending on min/max operations applied to distance functions. For a given offset distance r , a cell is potentially intersected by the offset surface if the minimum distance is less than r and the maximum distance is greater than r . The authors reported an execution time of 3100 seconds to compute the dilated surface of the Stanford Buddha model [1] at a resolution of 478^3 for an offset distance of 2% of the diagonal length of the bounding box of the model. Inspired by the work of Bastos and Celes [18], another GPU-accelerated implementation of ADF was presented by Yin et al. [105] that used octrees as the underlying data representation. Use of an octree in place of 3D hashing based data representation proved to be effective to sparsely represent and compute the adaptive distance fields that scale the problem size to a resolution up to 512^3 .

4.2 Convolution and Mathematical Morphology

4.2.1 Convolution

Convolution is an important mathematical tool heavily used in both fields of signal and image processing. Since our use case lies in discrete domain, convolution in 3D space can be defined as Eq. 4.1. In Eq. 4.1, $w(x, y, z)$ is a filter of size $(2a + 1) \times (2b + 1) \times (2c + 1)$ that will be convoluted with a 3D grid $f(x, y, z)$, denoted as $w(x, y, z) \star f(x, y, z)$.

$$w(x, y, z) \star f(x, y, z) = \sum_{r=-a}^a \sum_{s=-b}^b \sum_{t=-c}^c w(r, s, t) f(x - r, y - s, z - t) \quad (4.1)$$

The direct approach to solving the 3D convolution will be implementing the above mathematical definition. For a filter size of $M \times M \times M$ and grid size of $N \times N \times N$, this gives an asymptotic running time of $\theta(N^3 M^3)$. In our use case for an offset distance of 100 voxels at $4096 \times 4096 \times 4096$ resolution, the grid has 4096 discrete points and the kernel has a $2 \times 100 + 1 = 201$ discretized values along each of the three dimensions. This leads the complexity of a naive 3D convolution to an order of $4096^3 \times (2 \times 10^2)^3 \approx 5.5 \times 10^{17}$ computations. Even an accelerator-integrated advanced computing platform with a sustained throughput of 1 TFLOPS (10^{12} floating point operations per second) will take over six days to compute single convolution.

While convolution is a general technique applicable to diverse types of inputs, our application deals with a particular type of 3D data. In our use case of voxel offsetting, the voxels represented in the HDTs can be considered as *binary* data, where the voxels with *boundary* state define the solid geometry and the rest of the voxels are *non-boundary* (*i.e.*, either interior or exterior to the object). Thus, to cope with the performance requirement of 3D convolution, we could use mathematical morphology to compute the offsets of the 3D voxel model [46].

4.2.2 Mathematical Morphology

Mathematical Morphology (MM) appears as one of the most powerful tool among the various shape analysis frameworks that has been extensively studied in [92, 93, 41]. MM mostly deals with the mathematical theory of describing shapes using sets and is used to simplify, enhance, extract or describe structured 2D and 3D geometrical data. Although MM is most commonly used in digital images, it can be equally applied to surface meshes, solids, and many other spatial structures [7]. It has been used in a number of applications, including image filtering, segmentation, skeletonization among others, in medical imaging, metrology, video surveillance, industrial control, video compression, to name a few.

As the morphological operators are only defined for sets, we can define a relationship between sets and 3D voxels to comprehend the interpretation of these operation on our HDT-represented voxel models. In general, MM is often used for *binary* images, where the set of all black (or white) pixels defines a complete morphological description of the image. In binary images, the sets are members of the 2D integer space \mathbb{Z}^2 , where each element of the set is a tuple whose elements are the (x, y) coordinates of a black (or white) pixel in the image. Similarly, our application of mathematical morphology lies in the binary domain, where the set of the boundary voxels in the HDTs completely define the solid geometry. Here, the sets are members of the 3D integer space \mathbb{Z}^3 , where each element of the set is a tuple whose elements are the (x, y, z) coordinates of the boundary voxel in the solid.

In morphology, non-linear transformation operators are used to intuitively alter the object at every point with another set of a known shape, generally known as *structuring element* (SE) or *template*. In the case of digital data, typically simple binary structuring elements are used. The shape of the SE defines the result of any morphological operation. Each SE has one particular element marked as the origin. In Figure 4.2, a 3×3 SE is shown, where each shaded block denotes a member of the SE,

and the center block containing the black dot is the origin of the SE. Conceptually, a SE in morphological filtering serves the same purpose as a kernel does in convolution. In this thesis, the scope has been limited to binary morphology with ring structuring element of different radii, where each radius corresponds to an offset distance. The ring SE captures the connectedness relationship between the origin of the SE, and any point on the surface of the 3D sphere having specific distance from the origin in the voxel unit.

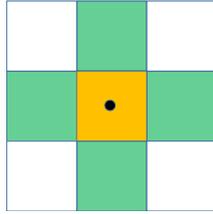


Figure 4.2: 3×3 morphology kernel.

4.2.3 Dilation and Erosion

The dilation and erosion are fundamental operations of mathematical morphology, and are frequently used in convolution based image filtering. In our context of HDT-represented voxel offsetting, these operations add or delete extra layers of voxels around the existing boundary cells of the solid, analogous to adding or removing rings of an onion. Since offsetting can be understood as a morphological operation, it is intuitive to extend the 2-D pixel based erosion and dilation operations to 3D resulting in a very simple volumetric offsetting approach, where the 26-neighborhood in a voxel-grid is used to propagate distance information [42].

Given the shape A of an object, the two basic operators are the *dilation* $D_{A,S} = A \oplus S$ and the *erosion* $E_{A,S} = A \ominus S$ where \oplus and \ominus are Minkowski sum and subtraction respectively. The binary dilation answers the question “*Does the structuring element hit the set?*” (quote from Soille et al. [93]). The result set contains the points where the answer is affirmative. The dilation $\delta_S(X)$ of a set X by the structuring element

S is defined by Eq. 4.2.

$$\delta_S(X) = \{x + s | x \in X \wedge s \in S\} \quad (4.2)$$

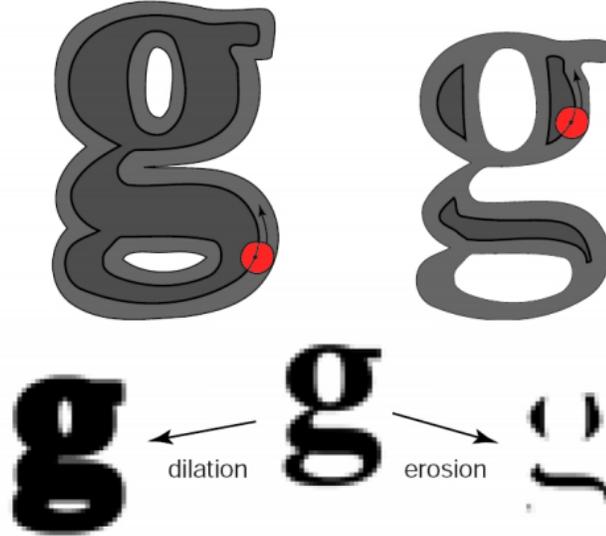


Figure 4.3: Illustration of morphological dilation and erosion [3].

The dual operation of dilation is called erosion. The output of the binary erosion is a set of points where the answer to “*Does the structuring element fit the set?*” is positive. The binary erosion $\epsilon_S(X)$ of a set X by a structuring element S is defined by Eq. 4.3. In our context of HDT based voxel representation, morphological dilation and erosion add or delete extra layers of voxels around the existing boundary cells, analogous to adding or removing rings of an onion. An example of morphological dilation and erosion is illustrated in Figure 4.3, where a *disc* structuring element (colored in red) is applied to the image.

$$\epsilon_S(X) = \{x | \forall s \in S, x + s \in X\} \quad (4.3)$$

Let us revisit the case of a triangle offsetting in discrete representation as depicted in Figure 4.4. In Figure 4.4 (a), a triangle is shown discretized on a 2D grid, where the set of red pixels define the boundary of the geometry. Like the state of voxel in the

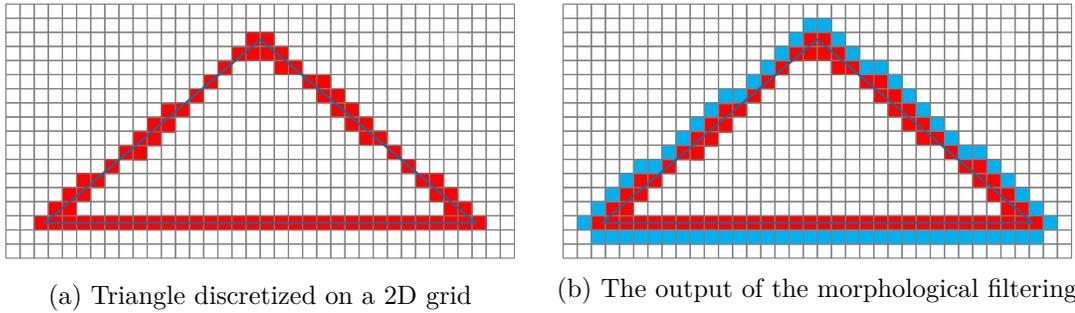


Figure 4.4: The case of convolution filtering on a discretized triangle to dilate by one voxel.

HDT, we assume the pixels enclosed within the geometry are set to **FULL** state and the pixels exterior to the geometry are set to **EMPTY** state. Our goal is to dilate the triangle by one voxel, for which we can apply the above 3×3 SE (shown in Figure 4.2) such that the origin of the SE coincides with each of the red pixels. Now, for the given 3×3 SE, the two horizontal and the two vertical blocks define the *connectivity* or *neighborhood* relationship with respect to the origin. All the neighboring cells with originally **EMPTY** state are colored in blue that define the dilated geometry as shown in Figure 4.4 (b).

4.3 HDT Offsetting using Mathematical Morphology

Given a solid represented in high-resolution HDT, our aim is to develop an efficient and robust implementation of morphological dilation and erosion on GPUs that generates the expanded or contracted representation of the input HDT. Figure 4.5 visually demonstrates this objective, where the cross-section of a 3D solid is shown getting dilated for a given offset distance. Thus, our offset algorithm takes an input HDT and an offset distance r , and produces another hybrid dynamic tree called *offset HDT*. The offset HDT maintains the property that every boundary voxel in the offset HDT has a minimum distance of $|r|$ from any boundary voxel in input HDT. Positive values of r correspond to dilation, while negative values correspond to erosion. Without loss of generality, only the case of dilation is considered in this discussion.

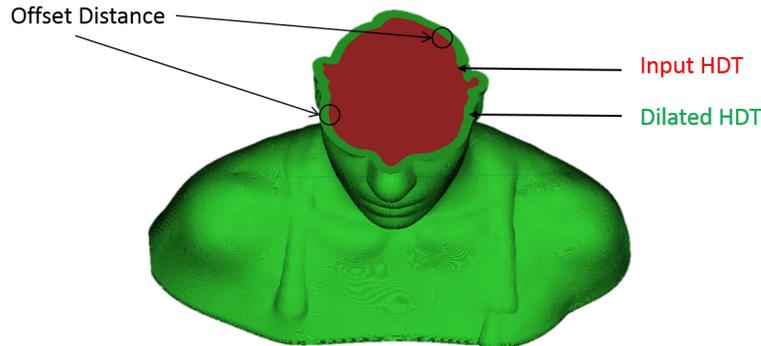


Figure 4.5: 3D Illustration of offsetting. The dilated portion is colored in green, and the original component is colored in red. The surfaces of these two HDTs maintain the given offset distance.

In Figure 4.6, we illustrate the offsetting operation on a 2D cross-section of an HDT representing a 3D cube. A 2D ring structuring element (magnified for better visualization) is shown in Figure 4.6(a). In this 2D illustration, the radius of the ring corresponds to the offsetting distance in pixels. Figure 4.6(b) shows a cross-section of the input HDT, where the yellow blocks represent the leaf grids containing the boundary voxels. The cross-section of the dilated HDT is shown overlaid with

the original HDT in Figure 4.6(c), where the outer yellow blocks are generated from sweeping the ring template, i.e. *convolving the the boundary voxels* of the input HDT with the morphological filter. Conceptually, our HDT offsetting algorithm consists of two steps. Algorithmic details of these two computations are presented below.

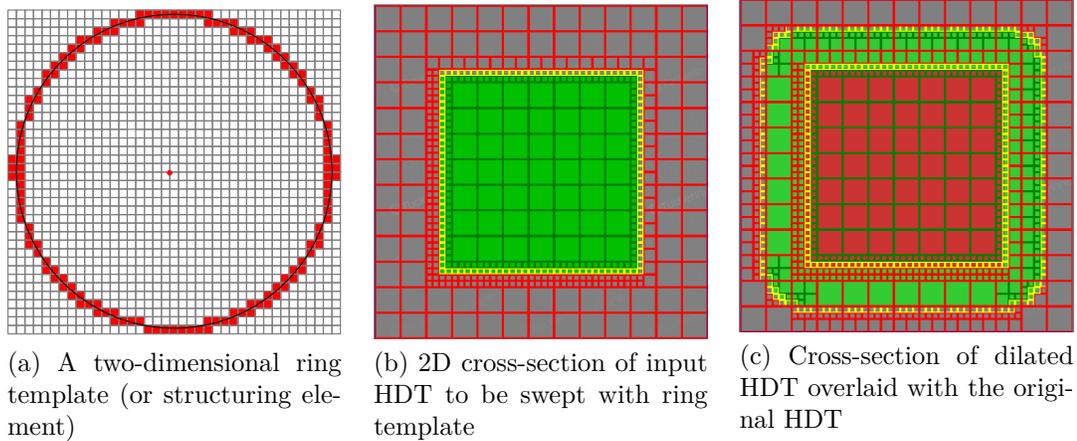


Figure 4.6: Offsetting illustration on HDT.

4.3.1 Constructing the Skeleton of the Offset HDT

In the first step of offset computation, a skeleton of the dilated HDT¹ is built that contains a conservative estimate of all the leaf grids necessary to represent the boundary of the offset HDT. As listed in Algorithm 1, the skeleton of the offset HDT (`hdtSkeleton`) is first initialized with an empty HDT (line 1) that has the same number of root cells as in the original HDT and no leaf grids. For all the root cells in the skeletal HDT, the cell boundaries are expanded by the given offset distance (line 4), and the dilated cell boundaries are checked for overlap with *any* leaf grid in the original HDT. A resulting intersection with the expanded root cell indicates the possibility of having an active voxel within the offset value (`distance`) from the surface of the original HDT. Such an intersecting root cell is processed in `SPLIT` procedure. Otherwise, the state of the root cell is set to either `full` (inside) or `empty` (outside)

¹The terms dilated HDT and offset HDT are used synonymously in this description.

depending on its location relative to the surface (line 8 in Algorithm 1).

Algorithm 1: Compute the Skeleton of the Dilated HDT

Input: A hybrid dynamic tree `hdtOriginal` and an offset value `distance`

Output: A skeleton of the offset hybrid dynamic tree `hdtSkeleton` and a list of leaf grids `leafList`

```

1 hdtSkeleton ← empty HDT
2 leafList ← empty list
3 foreach root cell elem in hdtSkeleton do
4   | elemBound ← GROWCELLBOUNDS( elem, distance )
5   | if elemBound overlaps any leaf in hdtOrig then
6   |   | SPLIT( hdtOriginal, elem, distance, hdtSkeleton, leafList )
7   | else
8   |   | elem.state ← either FULL or EMPTY
9   | end
10 end
11 return hdtSkeleton and leafList

```

The SPLIT procedure takes in a root cell that overlaps the original HDT when dilated by `distance`, and it recursively subdivides the cell and its descendants so long as a cell overlaps with any leaf grid in the original HDT. It should be emphasized here that during the construction of the skeletal HDT the intersection of a cell in `hdtSkeleton` is checked with all leaf grids in original HDT, *not* with all the boundary voxels in the original HDT. The latter is prohibitively expensive computation, as the set of boundary voxels typically includes tens to hundreds of millions of voxels.

The SPLIT routine executes MAKEBRANCH procedure (line 7) to subdivide each cell into eight child cells in the skeletal HDT. Now, like the parent cell, if a descendant child overlaps with any leaf grids in the original HDT when dilated by `distance`, the child is recursively partitioned in SPLIT (line 10). This recursive hierarchical cell partitioning continues until the cell size reaches the target resolution. Finally, at the

deepest hierarchy in the skeletal HDT the state of a cell is set to **boundary** (line 2), as it is estimated to have boundary voxels in the resultant offset HDT. For each boundary cell, a leaf grid is allocated in the Leaf Pool, as discussed in Section 3.1.4, and the boundary cell is mapped to the allocated *empty* leaf grid, which is a block of voxels where the state of each voxel is set to the default. The index to the allocated leaf grid is added to a list (line 5), which is passed back to Algorithm 1 along with the skeletal HDT that will be further processed in the following step to construct the offset HDT.

Procedure *Split*(*hdtOriginal*, *elem*, *distance*, *hdtSkeleton*, *leafList*)

Input: An HDT *hdtOriginal* along with a given cell *elem* to be dilated, an offset value *distance*, an HDT under construction *hdtSkeleton* along with its list of leaf grids *leafList*

Output: A hierarchical dilated representation of the given cell *elem* in the *hdtSkeleton*

```

1 if depth of elem  $\geq$  MAX_DEPTH then
2   | elem.state  $\leftarrow$  BOUNDARY
3   | allocate an empty leaf grid
4   | elem.pointer  $\leftarrow$  pointer to the allocated buffer
5   | leafList  $\leftarrow$  leafList  $\cup$  index of the grid in the Leaf Pool
6 else
7   | MAKEBRANCH( hdtOriginal, elem, distance, hdtSkeleton )
8   | foreach child of elem do
9     |   if child.state is BOUNDARY then
10    |     | SPLIT ( hdtOriginal, child, distance, hdtSkeleton, leafList )
11    |   end
12  | end
13 end

```

Next, in MAKEBRANCH procedure the state of each branching cell is set to **branch**

(line 1), and memory is allocated for the eight² descendants in the Element Pool, as detailed in Section 3.1.4. The branching cell stores the address of the allocated memory (line 3) to access the descendants through linear addressing. Finally, the state of each child cell is set to the corresponding state based on the intersection between the dilated bounds of the child cell and leaf grids in the original HDT, and the cells with BOUNDARY state are processed in SPLIT routine as described above.

Procedure MakeBranch(*hdtOriginal*, *elem*, *distance*, *hdtSkeleton*)

Input: An HDT *hdtOriginal* along with a given cell *elem* to be dilated, an offset value *distance*, an HDT under construction *hdtSkeleton*

Output: One level branching of the given cell *elem* in the *hdtSkeleton*

```

1 elem.state ← BRANCH
2 allocate BranchingFactor * BranchingFactor * BranchingFactor child cells
   in contiguous memory block
3 elem.pointer ← pointer to the allocated buffer
4 for i ← 1 to BranchingFactor do
5     for j ← 1 to BranchingFactor do
6         for k ← 1 to BranchingFactor do
7             set the state of child cell indexed at [i, j, k]
8         end
9     end
10 end
```

4.3.2 Constructing the Offset HDT through Morphological Filtering

The second step of our offset algorithm performs the morphological filtering on each voxel in the leaf grids of the input skeletal HDT, and produces the offset HDT where the state of each voxel is set to appropriate value (inside, outside, or at the boundary). Algorithm 2 presents the high-level abstraction of the CUDA implementation of the

²In general, for a branching factor of b the number of descendant cells are b^3 .

Algorithm 2: Compute the State of Voxels in the Dilated HDT

Input: An HDT `hdtOriginal`, an offset value `distance`, list of leaf grids in skeletal HDT `leafList`, and the skeleton of the offset HDT `hdtDilated`

Output: The offset HDT `hdtDilated` with the voxels state set to either FULL, EMPTY or BOUNDARY

```
1  $kernelHalfSize \leftarrow \frac{distance}{voxelSize}$ 
2  $kernelSize \leftarrow 2 \times kernelHalfSize + 1$ 
3  $kernelPoints \leftarrow \text{FILLKERNEL}(kernelSize)$ 
4 Allocate and Initialize CUDA buffers
5  $numLeafs \leftarrow leafList.size()$ 
6  $numPoints \leftarrow kernelPoints.size()$ 
7 OFFSET(hdtOriginal, hdtDilated, numLeafs, leafList, numPoints,  
kernelPoints )
```

morphological filtering. The execution starts at the host side (*i.e.*, on CPU) that takes in the original HDT, the offset distance, the skeletal HDT (`hdtDilated`), and the list of leaf grid indexes in the skeletal HDT. It should be noted that we denote the skeletal HDT with `hdtDilated` here, as opposed to `hdtSkeleton` in Algorithm 1, because the output of the Algorithm 2 represents the dilation of the input HDT. For a given offset value (`distance`), first the offset radius in voxel unit is computed (line 1); and then the kernel size is measured (line 2) as twice the radius (in voxel unit) plus one (for the center voxel). In `FILLKERNEL`, the discretized kernel boundary points are computed (line 3), which defines the boundary of the 3D ring structuring element. Once the buffers are allocated and initialized on GPU, these relevant parameters are passed to the device side `OFFSET` routine.

`OFFSET` procedure configures the CUDA execution parameters: (a) the number of blocks in the CUDA grid is set to the number of leaf grids in skeletal HDT, and (b) the number of threads per block is set to `LEAF_BRANCHING` \times `LEAF_BRANCHING`.

Thus, for the default LEAF_BRANCHING value of 16, each CUDA block is configured with $16 \times 16 = 256$ threads, which is mapped onto eight CUDA warps each of 32 threads on the GPU.

Procedure Offset(*hdtOriginal*, *hdtDilated*, *numLeafs*, *leafList*, *numPoints*, *kernelPoints*)

```

1 blocksInGrid  $\leftarrow$  numLeafs
2 threadsPerBlock  $\leftarrow$  LEAF_BRANCHING  $\times$  LEAF_BRANCHING
3 MORPHCUDA  $\lll$  blocksInGrid, threadsPerBlock  $\ggg$  (hdtOriginal,
    hdtDilated, numLeafs, leafList, numPoints, kernelPoints)

```

Procedure MorphCUDA(*hdtOriginal*, *hdtDilated*, *numLeafs*, *leafList*, *numPoints*, *kernelPoints*)

```

1 blockId  $\leftarrow$  CUDA block id
2 x  $\leftarrow$  threadIdx.x
3 y  $\leftarrow$  threadIdx.y
4 if blockId  $\geq$  numLeafs OR x  $\geq$  LEAF_BRANCHING OR y  $\geq$  LEAF_BRANCHING
   then
5   | return
6 end
7 leafElem  $\leftarrow$  GETELEMENT( hdtDilated, leafList[blockId] )
8 for z  $\leftarrow$  0 to (LEAF_BRANCHING - 1) do
9   | cellIndex  $\leftarrow$  make_int3 (x, y, z)
10  | cellCenter  $\leftarrow$  COMPUTECELLCENTER (hdtDilated, leafElem, cellIndex)
11  | cellState  $\leftarrow$  MORPHFILTER (numPoints, kernelPoints, cellCenter,
    | hdtOriginal)
12  | SETLEAFSTATE (hdtDilated, leafElem, cellIndex, cellState)
13 end

```

The entry point for each thread on GPU is MORPHCUDA procedure that is executed in parallel, where each thread processes specific block of voxels in the assigned

leaf grid. In MORPHCUDA procedure, first each thread retrieves the information (*i.e.*, memory address) of the assigned leaf grid (line 1), and the X-Y indexes of the thread in the CUDA block (lines 2-3), and then the boundary conditions are checked (line 4). Similar to the leaf processing in HDT construction (cf. Section 3.2), each CUDA thread works on 16 neighboring voxels (all Z values) for fixed $\langle X, Y \rangle$ location. To determine the state of each voxel, MORPHFILTER procedure is invoked for all the voxels located at $\langle X, Y, Z \rangle$ positions (line 11). MORPHFILTER iterates over all the kernel points, and checks if specific voxel, convolved with any point in the kernel, belongs into the set of boundary voxels in the original HDT. Thus, the state of a voxel in the offset HDT is set to BOUNDARY based on the existence of any active voxel within `distance` in the original HDT. Collectively, all these boundary voxels define the surface of the dilated HDT.

Procedure MorphFilter(*numPoints*, *kernelPoints*, *cellCenter*, *hdtOriginal*)

```

1 for  $i \leftarrow 0$  to ( $numPoints - 1$ ) do
2   |  $lookUpPoint \leftarrow \text{add3}(kernelPoints[i], cellCenter)$ 
3   | if STATEOFPOINT( $hdtOriginal, lookUpPoint$ )  $\neq$  EMPTY then
4   |   | return BOUNDARY
5   | end
6 end
7 return EMPTY
```

4.4 Experimental Results and Analysis

This section presents performance evaluations of the presented morphological offsetting algorithm on HDT based voxel representations. We perform the experiments both for dilation and erosion at 1024^3 , 2048^3 and 4096^3 resolutions with offsetting distance up to 100 voxels. These magnitudes of resolution are much higher than used in prior works [85, 72], which demonstrated evaluations only in low to mid-range voxel resolutions. For the experimental evaluations reported in the following Section 4.4.1 and Section 4.4.2, a NVIDIA GTX 780Ti GPU is used as the underlying CUDA platform. Finally, as the input models, we considered the same CAD data sets used in Chapter 3. As a visual demonstration of offsetting, dilations and erosions for Candle Holder model with an offset distance of 10 voxels, 20 voxels and 40 voxels are shown in Figures 4.7 and 4.8 respectively.

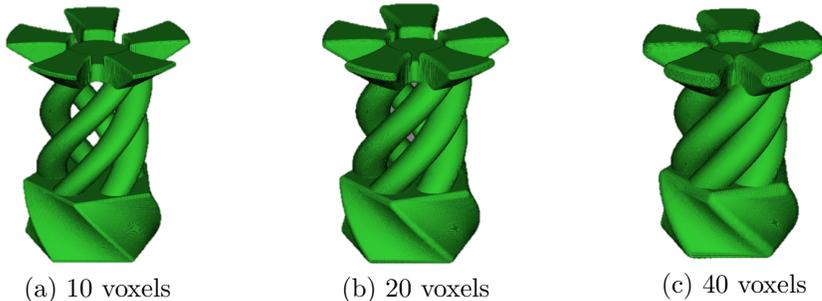


Figure 4.7: Dilations of Candle Holder at 2048^3 resolution.

4.4.1 Dilation Performance Evaluations

At different modeling resolutions, Table 4.1 (for model Head and Dragon) and Table 4.2 (for model Turbine and Candle Holder) report the offset computing time (middle five rows) and the number of leaf grids processed (last five rows). To keep the offset distance independent of the physical dimension of specific model, the algorithms are tested at five different offset distances: 20 voxels, 40 voxels, 60 voxels, 80 voxels and 100 voxels.

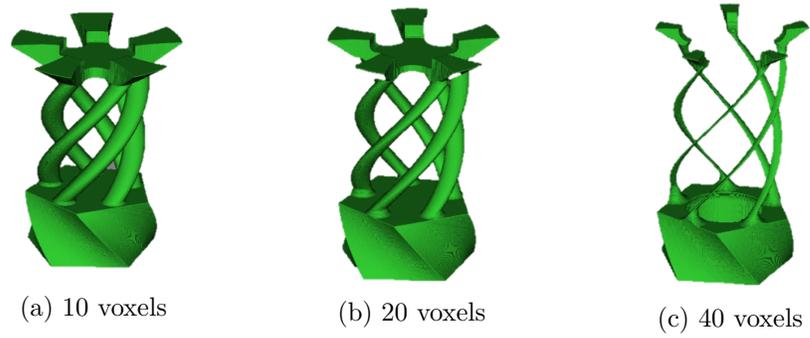
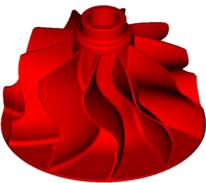


Figure 4.8: Erosions of Candle Holder at 2048^3 resolution.

Table 4.1: Dilation results: Head and Dragon.

	 Head			 Dragon		
Resolution	1024	2048	4096	1024	2048	4096
Offsetting Time (sec)						
20 voxels	4	15	59	4	17	75
40 voxels	15	61	254	17	71	325
60 voxels	38	151	612	42	172	762
80 voxels	81	289	1147	85	325	1416
100 voxels	186	629	2295	194	674	2731
Leaf Grids Processed (thousand)						
20 voxels	12.1	44.2	168.4	13.6	52.4	214.2
40 voxels	17.7	61.8	229.3	19.7	72.9	287.6
60 voxels	24.2	81.0	293.0	26.7	94.6	364.5
80 voxels	31.6	101.8	359.9	34.5	117.9	444.1
100 voxels	49.0	148.6	503.0	53.1	168.9	611.1

Table 4.2: Dilation results: Turbine and Candle Holder.

						
	Turbine			Candle Holder		
Resolution	1024	2048	4096	1024	2048	4096
Offsetting Time (sec)						
20 voxels	3	15	76	4	19	77
40 voxels	13	57	314	17	78	347
60 voxels	28	119	703	40	184	843
80 voxels	57	207	1273	80	325	1531
100 voxels	132	408	2323	178	645	2885
Leaf Grids Processed (thousand)						
20 voxels	11.2	51.1	214.8	14.3	57.5	220.4
40 voxels	15.2	66.5	289.2	19.8	77.9	307.6
60 voxels	20.0	81.3	360.0	26.1	99.0	390.1
80 voxels	25.4	96.5	426.0	33.2	120.4	470.9
100 voxels	38.7	130.4	546.0	50.3	166.5	635.1

By our approach of voxel offsetting using morphology filtering, the computation complexity to generate the dilated HDTs scales in proportion to two parameters: a) the number of leaf grids created in COMPUTESKELETON procedure that are processed in following CONVOLUTIONCUDA procedure (Section 4.3), and b) the number of boundary points on the discretized structuring element. At our target resolutions, the number of boundary points on the discretized structuring element—shorthand as *kernel boundary points*—can be considered invariant across the modeling resolution. Table 4.3 reports the kernel boundary points for different offset distances. By the depiction of the ring morphological filter in Figure 4.6 (a), a two-fold increase of the radius of the structuring element indicates the number of discretized points rises

approximately to $2\times$ (in 2D case) and $4\times$ (in 3D space)³. In our target application of offsetting in toolpath generation, conceptually the kernel represents the spherical surface of a ball-end milling tool. Thus, the number of boundary points in the structuring element roughly increases by a factor of four at $2\times$ offset distance, as shown in Table 4.3. For instance, the ratio of the kernel boundary points between offset distances of 40 voxels and 20 voxels is $\frac{2375}{581} = 4.09$. Similarly, the ratio of the kernel boundary points between offset distances of 80 voxels and 40 voxels is $\frac{9674}{2375} = 4.07$.

For a given offset distance, understanding how the time to compute offset scales with the target resolution is useful on two grounds. First, it helps to validate the parameters that define the theoretical complexity of the morphological offsetting. Second, it helps to predict the growth in offsetting time at finer target resolutions. To address that goal, we analyze the dilation computation times at different resolutions for the individual models. As the number of kernel boundary points remain almost same across the target resolutions, the dilation times for a given offset distance is expected to scale in proportion to the number of leaf grids processed. As an example, consider the case of 100 voxel dilations at 2048^3 and 4096^3 resolutions. The ratio of the number of processed leaf grids for the Head, Dragon, Turbine and Candle Holders are respectively 3.73, 3.62, 4.19 and 3.81, while the ratio of the dilation times for the models are 4.11, 4.05, 5.70 and 4.47 respectively. Thus, the offsetting time at 4096^3 grow faster than the scale of processed leaf grids. As at the $2\times$ resolution the HDT depth increase by 1, the average processing time for each leaf grid increases at the higher resolution. We will revisit the impact of the HDT depth in details in Chapter 5.

Now, for a given target resolution, understanding how the dilation computation scales with the offset distance is crucial to comprehend the aspects of parallel-execution efficiency of the morphological filter based offsetting algorithm on GPU

³These are analogues, respectively, to the ratio of the circumferences of a circle (in 2D case) and the surface areas of a sphere (in 3D case) with the radius gets doubled.

Table 4.3: Boundary points of the structuring element at different offset distances.

Offset Distance (in voxel unit)	20	40	60	80	100
Number of Kernel Boundary Points	581	2375	5370	9674	14948

hardware. For instance, 100 voxel offsetting at 4096^3 resolution takes about 48 minutes for the Candle Holder. To develop a deep insight into the parallelization efficiency of our implemented morphological offsetting on the HDT, we study the dilation computation times at the same resolution for different offset distances. For instance, consider the dilations of the Candle Holder at offset distances of 20 voxels and 40 voxels at 4096^3 resolution. For these tests, the number of leaf grids processed are 220.4 and 307.6 thousands (Table 4.2), while the numbers of kernel boundary point are 581 and 2375 for corresponding offset distances (Table 4.3). Taking these values into analysis, the theoretical computational complexity uplifts by a factor of $\frac{307.6}{220.4} \times \frac{2375}{581} = 5.7$ for offsetting 40 voxels relative to 20 voxels. Now, the ratio of the execution times for the Candle Holder at these configurations is $\frac{346.5}{76.8} = 4.5$, which lies within the bound of the theoretical complexity in order of 5.7.

By contrast, the same comparative analysis between the offset distances of 20 voxels and 100 voxels, depicts a different scenario. As it appears, the dilation time with 100 voxels at 4096^3 resolution grows by $\frac{2885}{76.8} = 38$ times, whereas the complexity scales by a factor of $\frac{635.1}{220.4} \times \frac{14949}{581} = 74$. This demonstrates that with larger problem size, the algorithmic acceleration increases significantly compared to their theoretical complexity. With larger workload the overhead of pre-processing operation (*i.e.*, the skeletal HDT construction), the overheads of CUDA related issues, such as, data allocation on GPUs, data transfer between CPU and GPU, etc. are amortized over larger number of computing threads. Thus, the computation efficiency of the offsetting kernel on GPUs generally scales with larger offset distances.

4.4.2 Erosion Performance Evaluations

Similar to the dilation experiments, erosion computation times at different modeling resolutions are presented in Table 4.4. Unlike the dilation experiments, erosions are evaluated at three different offset distances in voxel units: 10 voxels, 20 voxels and 40 voxels, and at resolutions of 2048^3 and 4096^3 only. This is because for our CAD models large-scale erosion shrinks the solid entirely, and so does not reflect a practical offset operation. Figure 4.8 depict the outcome of erosions on Candle Holder with an offset distance of 10 voxels, 20 voxels and 40 voxels.

Table 4.4: Erosion results: Head, Dragon, Turbine and Candle Holder.

								
Resolution	2048	4096	2048	4096	2048	4096	2048	4096
Offsetting Time (sec)								
10 voxels	3.0	13.0	2.1	9.1	1.5	8.5	3.2	15.6
20 voxels	11.6	55.6	5.7	32.6	2.9	22.1	10.5	57.2
40 voxels	40.5	226.3	12.4	104.2	5.2	60.3	28.1	211.5
Leaf Grids Processed (thousand)								
10 voxels	25.4	105.8	18.6	86.5	17.7	88.4	26.8	118.9
20 voxels	35.1	152.6	20.3	103.7	18.2	100.6	31.5	161.0
40 voxels	43.0	195.3	20.5	114.8	18.2	107.0	32.7	189.3

As it is expected, erosions are faster to compute than dilations for corresponding offset distances (*i.e.*, offset distances of 20 and 40 voxels). This is because the sample inputs require more leaf grids to represent the dilated surface compared to the shrunk surface for a given offset distance, that in turn implies less leaf grids are processed in the offset algorithms for erosions. These can be validated by comparing the leaf grids count processed for the dilation relative to that for the erosion with the offset

distance of 20 voxels and 40 voxels (cf. Tables 4.4 with Tables 4.1 and 4.2).

4.4.3 Computational Complexity of HDT Offsetting

The asymptotic complexity of the presented morphological offsetting scales in $\mathcal{O}(L^3NM)$, where L is the size of the leaf grid, N is the number of leaf grids processed in the computation, and M is the number of kernel boundary points. While M depends on the offset distance in voxel unit, N depends on the target resolution and the geometric sparsity in the input voxel representation. Hence, we pursue a model-driven computational comparison between the morphology algorithm with the underlying HDT representation compared to the complexity of 3D convolution.

For comparison, we consider the test case of 100 voxel offsetting at 4096^3 resolution. Out of the four inputs, the Candle Holder model requires processing the highest number of leaf grids—approximately 637 thousands leaf grids, where each grid consists of $16 \times 16 \times 16$ voxels. Now, the kernel with a radius of 100 voxel comprises about 15×10^3 boundary points as shown in Table 4.3. Thus, the theoretical complexity for the Candle Holder model lies in order of $(637 \times 10^3 \times 16^3) \times (15 \times 10^3) \approx 3.9 \times 10^{13}$. Compared to a computational complexity in order of 5.5×10^{17} with a dense 3D grid, as presented Section 1.2, for our benchmarked CAD models the theoretical complexity is about *four orders of magnitude* faster. However, as discussed in Chapter 1, while the use of the HDT as the underlying data structure solves voxel offsetting problem at higher resolutions than the prior state-of-the-arts, it comprises the peak parallelism available with a regular 3D grid based voxel representation.

4.4.4 Cross-Platform Scalability

To investigate the performance scalability of our offset algorithm on different GPUs, we conduct the dilation experiments on another graphics hardware of comparable performance with that of a GTX 780Ti (the default setup). As the second GPU we chose a GTX Titan card. Figure 4.9 highlights the comparative performance of the

dilation algorithm for the four models at 2048^3 resolutions. For all the four models, the performance gap between the execution times on 780Ti and Titan increases with larger offset distances. For instance, for 20 voxel offsetting Titan takes respectively 16%, 16%, 20% and 16% more time than 780Ti for the Head, Dragon, Turbine and Candle Holder, whereas for 100 voxel offsetting the former takes respectively 21%, 41%, 40% and 22% longer than the latter.

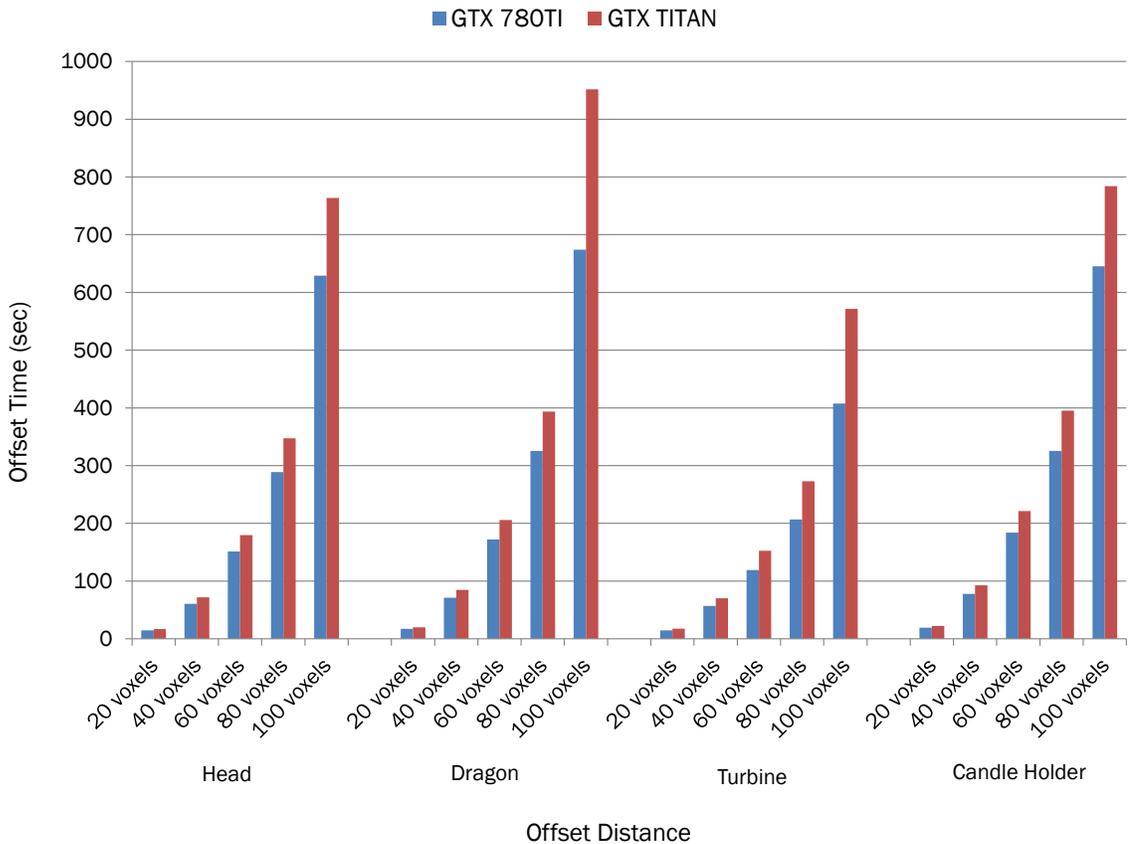


Figure 4.9: Dilation times comparison between GTX 780Ti and GTX Titan.

Now, to rationalize these relative performance gap between these two tested GPUs, we analyze the key architectural specifications [8, 9] of these selected cards as reported in Table 4.5. As Table 4.5 provides, a GTX 780Ti unleashes $\frac{5040}{4494} \approx 1.12\times$ peak throughput of a GTX Titan, and on memory bandwidth the former demonstrates

$\frac{336.0}{288.4} \approx 1.16\times$ higher bandwidth than the latter. Thus the performance gap, as observed in Figure 4.9 seems to be well justified—particularly for small-scale offsetting—that fall in line with the differentiable key metrics appeared in Table 4.5. With larger offset distance, the size of the morphological filter increases roughly quadratically, as we reported in Table 4.3. This results in an order of magnitude higher data to be loaded from the global memory on GPU. Our experimentation and analysis on the CUDA profiler outputs demonstrate that the memory throughput on GTX Titan significantly under perform at these test scenarios. Thus, with larger offset distances the performance penalty incurred due to lower memory bandwidth seems obvious in some cases, like we observed with the Dragon and Turbine models.

Table 4.5: Key architectural specifications of NVIDIA GTX 780Ti and Titan.

	GTX 780Ti	GTX Titan
Number of streaming multi-processors (SM)	15	14
Number of CUDA cores per SM	192	192
Total CUDA cores	2880	2688
Floating-Point ⁴ performance (GFLOPS)	5040	4494
Memory bandwidth (GB/s)	336.0	288.4

4.5 Comparison with Prior Studies

4.5.1 Multi-Core CPU Offsetting on Distance Field based Representation

To demonstrate the relative performance of the presented GPU accelerated surface offsetting algorithm, a recent CPU based study [72] is considered as the basis for comparison. Since the work of Liu et al. [72] uses a publicly available CAD model from the Stanford Repository [1] for their experimental analysis, it makes possible to conduct *apples-to-apples* comparison with a complex mesh input to validate the relative speedup achieved through GPU acceleration. Liu et al. benchmarked the performance of their offsetting algorithm on a distance field based volume representation. The experiment used the Buddha model [1], and the offsetting operation dilated the model at 512^3 resolution with an offset distance of 2% of the diagonal length of the bounding box. The same test configuration is used for our offsetting experiments. The resultant dilated model appears in Fig. 4.10.

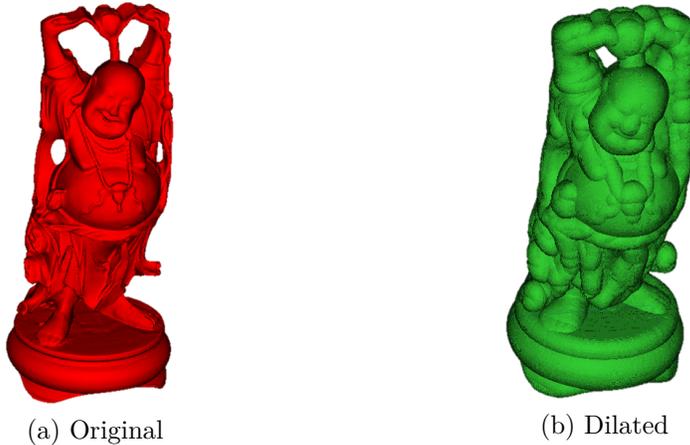


Figure 4.10: Surface dilation of a Buddha model (polygonal mesh comprising 1.1 million triangles).

The work by Liu et al. achieved significant speedup compared to another prior study [85] that took over 3000 seconds. To realize the comparison fair, only the time of distance field computation is considered here, and the times of auxiliary filtering steps and mesh reconstruction are excluded from the reported results. Table 4.6 shows

the comparative measurements of our GPU accelerated offset implemented relative to the single-core and eight-core CPU implementations. The leftmost two columns in Tab. 4.6 present offsetting times for CPU implementations at 512^3 resolution as reported in [72], while the rightmost three columns present offsetting times for the proposed GPU algorithm at 512^3 , 1024^3 and 2048^3 resolutions respectively. Here, the reported GPU offsetting times sum up the execution times of Algorithm 1 and Algorithm 2.

Table 4.6: Time comparison for dilation of Stanford Buddha [1]

CPU Offsetting Time at 512^3 [72]		GPU Offsetting Time at Resolution		
Single-core	Eight-core	512^3	1024^3	2048^3
114.5	22.8	0.46	7.6	109

Liu et al. reported 114.5 seconds with a single-core CPU, and 22.8 seconds on a dual-socket quad-core CPU for computing the distance field. First, the relative computation times on single-core and octa-core (2 x quad core) CPU emphasize that the computational performance not necessarily scales proportionately to the number of cores used. For instance, as the single-core and 8-cores timing results of Liu et al. reflect with employing eight cores, the performance could be accelerated only by a factor of $(114.5 \text{ sec} / 22.8 \text{ sec}) \approx 5$ only. By contrast, our GPU implementation of morphology based offsetting takes only 0.46 second at 512^3 resolution leading to respectively $50\times$ and $249\times$ speed-up than eight-core and single-core CPU results [72]. Further, at two-fold resolution presented algorithm achieves a speedup of $3\times$ and $15\times$ relative to eight-core and single-core CPU implementations respectively. Even at four-fold resolution it takes almost similar time relative to the single-core implementation. It should be emphasized here that in 3D space a four-fold resolution along each dimension raises the complexity of the problem by a factor of $4 \times 4 \times 4 = 64$.

Although both approaches work with volumetric representations, the presented

GPU implementation of the offsetting algorithm achieves significant speedup due to following reasons. First, use of a voxel based solid representation in the hybrid dynamic trees allows storing high-resolution volumetric data very compactly compared to the distance field based representation used in [72]. As a distance field represents 3D volume implicitly, volume processing algorithm, such as, surface offsetting is more computation intensive compared to voxel based alternatives. Secondly, presented scheme benefits from GPU acceleration. The peak throughput on a GTX 780Ti card is over an order of magnitude higher than the dual-socket quad-core CPU used in the study by Liu et al. Finally, as Liu et al. adopted a uniformly-indexed 3D grid to represent the distance field, it incurs high redundancy in computation due to processing of each point in the volume space. To the contrary, our presented technique represents the 3D space adaptively in HDT structures, and thus can eliminate avoiding redundant computations. Due to the use of a uniform grid, the works by Liu et al. scaled only up to a resolution of 512^3 on a CPU platform with much larger system memory than the graphics cards used in our benchmark results.

Table 4.7: Test configurations for 2% dilation of the diagonal length at different resolutions.

Target Resolutions on GPU	512^3	1024^3	2048^3
Offset Distance (number of voxels)	12	23	46
Number of Kernel Boundary Points	251	842	3210
Processed Leaf Grids	4933	18192	72304

To understand how the morphological offsetting scales for the *same offset distance* across multiple resolutions, we investigate the GPU offsetting times at the tested configurations for the specific offset distance of 2% of the diagonal length. As, the dimension of voxel gets halved at $2\times$ resolution, two-fold more number of voxels are required to make up the same offset distance as observed in Table 4.7. With $2\times$ larger offset distance (in *voxel unit*), as detailed earlier, the number of kernel boundary points

roughly increases by four times, specially for high target resolutions. Further, the leaf grid count processed during this dilation process grows approximately four-fold with increasing resolution. These contributing components raise the computational complexity of voxel offsetting approximately by $4 \times 4 = 16$ times for the *same offset distance* at twice higher resolution, which is reflected in the results shown in Table 4.7. For instance, the comparisons between the test data at resolutions 1024^3 and 2048^3 reveal that the complexity to compute the same dilation distance scales by a factor of $\frac{3210}{842} \times \frac{72304}{18192} \approx 15.2$, whereas Table 4.6 reports $\frac{109}{7.6} \approx 14.3 \times$ more computation time at the $2 \times$ resolution. However, it should be pointed that the real offset computation time may vary from the theoretical complexity depending on the parallel execution efficiency on the target graphics hardware.

4.5.2 Comparative Performance Analysis with GPU-Accelerated 3D Convolution

While a tight bound between the computational complexity of morphology based HDT offsetting and 3D convolution is not quite viable, we yet contrast our results with prior performance benchmarks conducted by Aqrabi et al [16]. The authors evaluated the performance of highly-tuned 3D convolution (in spatial domain) both on multi-core CPU and many-core GPU, and experimented resolution up to 2000^3 with a filter size up to 13^3 . It took 4,000 seconds to compute the 3D convolution on a quad-core Intel i7 CPU, and approximately 820 seconds on a NVIDIA Tesla C2050 GPU [10].

As we discussed earlier, the complexity of convolution depends on the target resolution and the size of the kernel. With higher resolution there are more voxels in the grid. Besides, the larger the filter, the more computation there is to do per voxel. To investigate the execution efficiency between our methodology and that by Aqrabi et al. [16], the target problem configurations and peak throughput of the respective GPU hardwares are enlisted in Table 4.8. As Table 4.8 reports, at our target

Table 4.8: Comparisons of the target problem configurations and respective GPU platforms.

	Work by Aqrawi [16]	Our Study	Corresponding Ratio
Resolution (along each dimension)	2000	4096	2.0
Filter Size (along each dimension)	13	100	7.7
Floating-Point Performance (GFLOPS)	1030	5040	4.9

resolution of 4096^3 with a filter size of 100^3 the complexity of the problem increases by $\frac{4096}{2000} \times \frac{100}{13} \approx 2.0 \times 7.7 = 15.4$ times along each dimension. Considering the problem space of three dimensions, we evaluate the problem at a scale of $15.4 \times 15.4 \times 15.4 \approx 3,652$ times more computationally intensive.

From Table 4.8 it should be noted that the Tesla C2050 is roughly 4.9 times less powerful than a GTX 780Ti. Hence, the normalized time for the GPU convolution translates to $\frac{820}{4.9} = 167$ seconds. Now, multiplying the scale of the problem size, we can estimate an execution time⁵ of $3652 \times 167 = 609,884$ seconds (over *7 days*) to compute 3D convolutions at 4096^3 resolution with a filter size of 100^3 . This is roughly $211\times$ larger than our peak reported time of 2,885 seconds (Candle Holder) for offset computation at 4096^3 resolution with a distance of 100 voxels (cf. Table 4.2). Thus, even compared to a hand-tuned GPU implementation of 3D convolution, our technique demonstrates *two orders of magnitude* faster offsetting on the hybrid dynamic trees. Further, compared to the quad-core CPU measurement, our approach gains a speedup of $\frac{4000}{820} \times 211 \approx 1031$ (over *three orders of magnitude* faster).

These observations are not surprising. First, our offsetting approach uses morphological filtering and only processes a fraction of data compared to the total discrete

⁵Arguably, this is a simplified estimation where we consider the disparity in peak throughput of the two GPUs as the only differentiating factor behind the overall computation efficiency. However, it should be noted that many of the optimization adopted in [16] is not applicable for large problem size. For instance, the authors used *fast constant memory* on GPUs to store the convolution kernel. When the filter size increases, even the latest generation GPUs cannot hold the entire filter on the constant or shared memory due to limited capacity.

points in a 3D grid. For instance, to compute 100 voxel offsetting at 4096^3 resolution, our algorithm processes roughly $(635 \times 10^3 \text{ leaf grids}) \times (4096^3 \text{ voxels/leaf grid}) = 2.6 \times 10^{12}$ voxels for the Candle Holder model, while a 3D convolution deals with $4096 \times 4096 \times 4096 = 69 \times 10^{12}$ voxels. Additionally, for 100 voxel offsetting a 3D convolution iterates over all the $(2 \times 100 + 1)^3 = 8,120,601$ discretized kernel points, whereas our morphological filtering only iterates over roughly fifteen thousand points (cf. Table 4.3). Thus, while our comparative analysis in Section 4.5.1 demonstrates a $50\times$ speedup over the octa-core CPU algorithm that works on the widely-studied Buddha model at a relatively lower 512^3 resolution, current comparison shows much greater speed-ups over multi-core CPU and hand-tuned GPU implementations of 3D convolutions at a higher resolution of 2000^3 .

CHAPTER 5

TUNABLE VOXEL OFFSETTING WITH HDT PARAMETERS

The process of performance tuning of a computation kernel with a sparse data representation is surprisingly complex [102]. The efficiency and performance of an application are determined not only by the algorithm and the hardware architecture that runs the algorithm, but critically also by the organization of computations and data on that architecture [87]. And, often that performance comes through a rigorous tuning of optimum trade-offs between the pertinent parameters, like in the *Halide* work [86] the authors have analyzed the tensions between parallelism, locality, and redundant work to maximize the overall efficiency in computational photography applications. Optimization becomes challenging because the best trade-offs are rarely obvious, and finding them often requires extensive experimentation. The ideal balance depends on the interaction between the underlying data structure, individual algorithms, the hardware architecture onto which they are mapped to.

Superficially, voxel-processing applications appear embarrassingly parallel, as the operations are often independent and thus perfectly suit to GPU’s parallel architecture. While GPUs can exploit abundant data-level parallelism to accelerate the applications, it has limitation as well. For high-resolutions voxel processing the low memory capacity on GPUs is a severe constraint. Although GPUs aim at hiding the long memory latency by switching the computing resources effectively across tens of thousands resident threads, the limited locality, as in the case of our morphological filter based offsetting algorithm, can throttle the memory throughput and eventually the execution pipelines. Driven by these challenges, designing high-performance voxel processing algorithm with a underlying sparse data structure is challenging, because

it requires a holistic insight into the data representation, and optimum mapping between the computations and the data to leverage the underlying parallel hardware most effectively.

Our research leverages the configurability of the HDT structure as a tunable knob to demonstrate high-performance voxel offsetting at extreme resolution. Selection of optimal choices for the HDT parameters not only minimizes the memory footprint of the voxelized representation of the solid, but also maximizes the parallel code execution efficiency on the parallel GPU hardware. To that end, we conduct offset experimentations with tunable HDT parameters, namely the size of the root grid, the branching factor of the nodes, and the size of the leaf grid. While the similar experimentations as demonstrated in Section 3.4 revealed the relationship between the memory footprint in the HDT and the tunable parameters of the underlying HDT data structure, here our goal is to explore similar relationship between the offset algorithm and the tunable HDT parameters.

First, for the convenience of discussion we revisit the same notations to specify an HDT configuration, as adopted in Section 3.4. Let, we formulate an HDT with configuration of root grid size $R = 2^r$, branching factor $B = 2^b$ and leaf grid size $L = 2^l$, then for a given resolution X we get an HDT with height h that satisfies the following condition:

$$2^{(r+b \times h+l)} \geq X \tag{5.1}$$

Our goal in this chapter is to analyze the impacts of the tunable HDT parameters to achieve higher computing efficiency in the HDT offset computation. The morphological filter processing is the most dominating step in the overall HDT offset computation; hence we emphasize our experiment and analysis with the tunable HDT parameters on the filter computation step. The following sections respectively investigate the impact of the size of the root grid, the branching factor of the nodes and the size of the leaf grid in HDT.

5.1 HDT Offsetting with Tunable Root Grid

5.1.1 The Impact of the Root Grid on HDT Offsetting

Our first study focuses on the impact of the root grid dimension (R) in the HDT offsetting computation. As expressed in Eq. 5.1, the depth of the HDT reduces by one for a two-fold larger root grid. The impact of the shallower HDT hierarchy in the morphological filter computation can be realized from the pseudocode of MORPHFILTER routine presented in Section 4.3.2. As we describe in Section 4.3, the state of individual boundary voxel in the offset HDT is determined in the MORPHFILTER procedure. To determine the state of each voxel, this CUDA routine iterates over all the kernel points, and checks if specific voxel, convolved with any point in the kernel, belongs into the set of boundary voxels in the given HDT. As we see in the code snippet of MORPHFILTER, the STATEOFPOINT is invoked to look up for every kernel point for all the voxels in the skeletal HDT. Thus, the depth of the HDT controls the code execution efficiency of the morphological filter on the GPUs.

Besides the impact on the depth of HDT, the size of the root grid also dictates the *magnitude of parallelism* in the skeletal HDT construction as appeared in Algorithm 1. As we described in Section 4.3.1, Algorithm 1 processes each cell in the root grid in parallel. Once the dilated bound of the cell intersects with the original HDT, a hierarchical octree structure is constructed that is estimated to contain some boundary voxels in the resultant offset HDT. Thus, with a two-fold larger root grid, the parallel construction of the skeletal HDT can be accelerated up to $8\times$.

5.1.2 HDT Dilations with Variable Root Grid Size

To study this impact of the HDT depth through tunable size of the root grid, we configure the HDT with a default leaf grid size (L) of 16 and branching factor (B) of 2, and set the root size (R) in between 8 and 64. As can be derived from the Eq. 5.1, at a target resolution of 4096^3 with $R = 8$, $L = 16$ and $B = 2$, the HDT spans a depth of five, whereas with $R = 64$ the HDT has a depth of two for the same value of

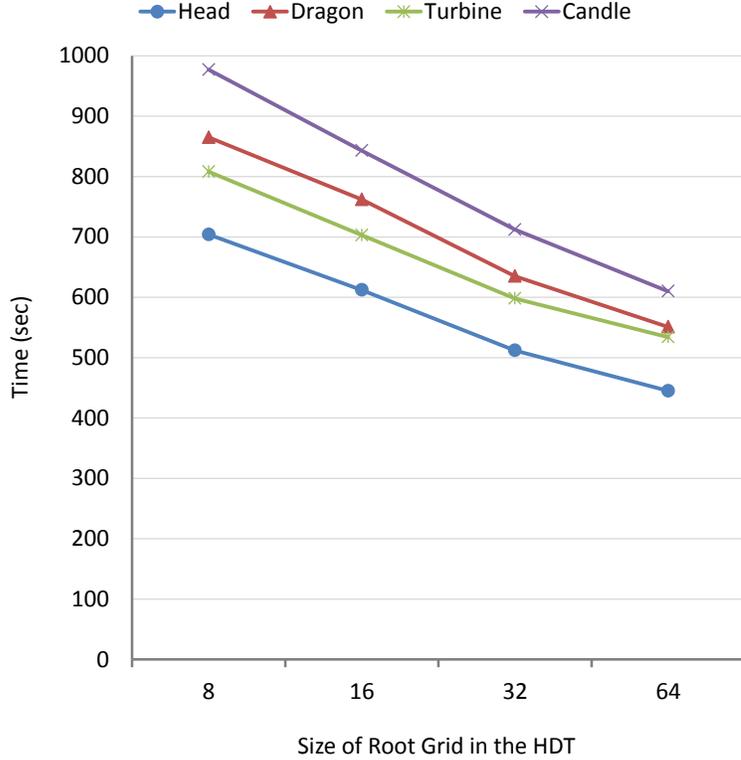


Figure 5.1: Impact of different root grid size on 60 voxels dilations at $4096 \times 4096 \times 4096$ resolution.

L and B . The dilation times with these different configurations of HDTs are reported in Figures 5.1 and 5.2. The former shows the offsetting times at 4096^3 resolution, while the latter presents the times at 2048^3 resolution.

As Figure 5.1 shows, all the models demonstrate a similar trend of linear reductions in the dilation times across the experimented sizes of the HDT root grid at a resolution of 4096^3 . Compared to the default HDT configuration with a root size of 16^3 , dilatations take 14 – 15% more with a root size of 8^3 . With the scaling of the root grid size to 32^3 , dilation computations take 15 – 17% lower, and for the root grid size of 64^3 offsetting is 34 – 38% faster than the default configuration.

Now, compared to the dilations times at 4096^3 resolution, the reported times in Figure 5.2 at 2048^3 resolution show that offsetting times in general decrease till some point, and then it observes not significant change in the computation efficiency. Between the root grid sizes of 8^3 and 32^3 , the dilation times decreases linearly –

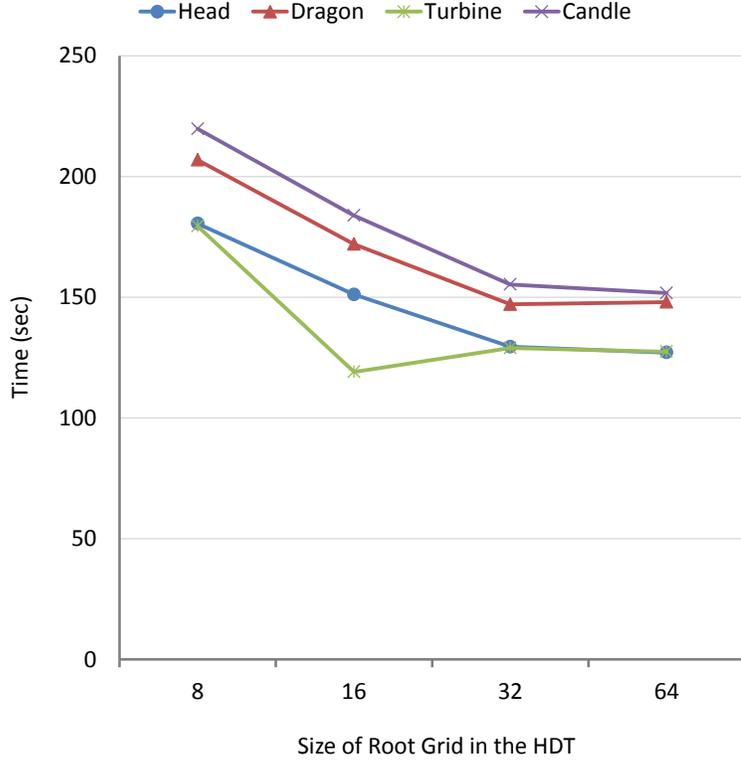


Figure 5.2: Impact of different root grid size on 60 voxels dilations at $2048 \times 2048 \times 2048$ resolution.

similar to the trend as observed with the case of 4096^3 resolution. The Turbine model is an outlier here that takes noticeably lower time with the default root grid of size 16^3 than the HDTs configured with a larger root grid. At 2048^3 resolution, with $R = 32$, $L = 16$ and $B = 2$, the HDT spans a depth of only two, and hence no noticeable performance gain is observed with an even larger root grid of size 64^3 . Thus, our offsetting experimentation with tunable size of the root grid in HDTs reveals the opportunity of high-performance offset computation by adopting a suitable grid size such that the overall HDT depth is relatively low (typically two for the peak performance).

5.1.3 Analysis on the CUDA Profiler Statistics

As the filter computation is the dominating component of the overall offset algorithm, we investigate the impact of the tunable root grid on the parallel execution efficiency

of the filter algorithm on GPUs with different HDT configurations. For the purpose of the analysis, we examine two CUDA performance statistics that represent the execution efficiency of the filter computation on GPUs at different HDT settings — a) *warp execution efficiency*, and b) *global memory transactions*. The warp execution efficiency indicates the distribution of the availability of eligible warps per cycle across the GPU. For our study, we investigate the number of cycles that a warp scheduler had at least one eligible warps to select from. At a given cycle, no warp may be available for scheduling, where warp can be stalled due to pipeline busy, execution and memory dependency, memory throttling, synchronization, among others. Thus, the *higher* the percentage of cycles with some eligible warps in the GPU the *more* efficient the code runs on the target device. Next, the global memory transactions indicates the pressure of the data read and write request on the GPU memory system. A lower number of memory transactions are preferred to avoid saturating the memory pipeline that may eventually throttle the instruction executions.

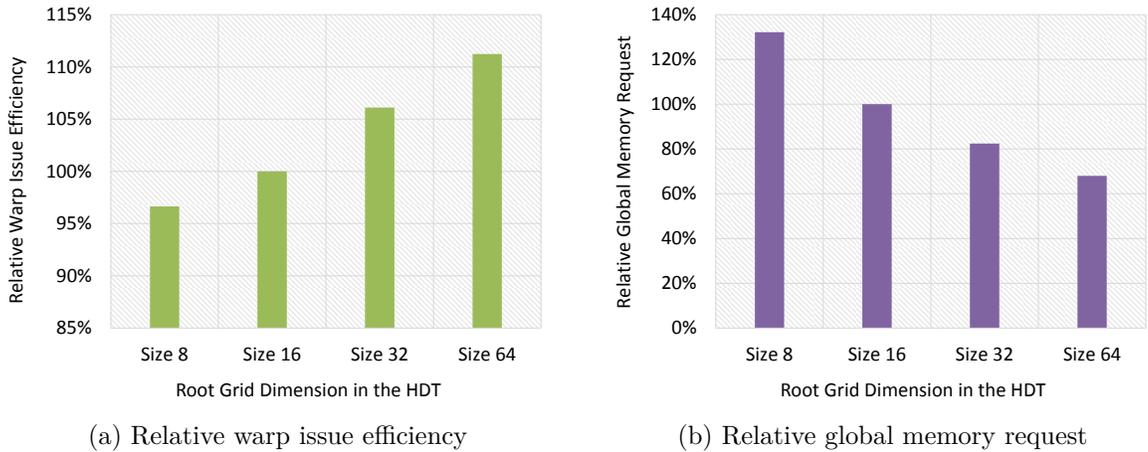


Figure 5.3: Impact on CUDA warp issue efficiency and global memory transactions with different sizes of root grid at 60 voxels dilations for the Candle Holder model.

Figure 5.3 shows the relative warp execution efficiency and relative global memory transactions performance metrics for 60 voxels dilation of the Candle Holder model. All these results are averaged across 10 kernel launches to avoid spurious hardware

statistics. As demonstrated in Figure 5.3, relative to the default size of root grid of 16^3 , the warp execution efficiency increases by 6% and 11% for HDTs with root grid of 32^3 and 64^3 respectively, while warp execution efficiency decreases by 3% in HDTs with root grid of 8^3 . On the other hand, the global memory requests decrease by respectively 18% and 32% in HDTs with root grid of 32^3 and 64^3 compared to the default grid size of 16^3 . With increasing depth of the HDT hierarchy, the rate of the number of memory transactions grow higher, as can be validated that the memory requests raised by 32% for the next deeper HDTs with root grid of size 8^3 . Thus, our analysis on the tunable root grid exposes the importance of shallow hierarchy in the voxel modeling through larger possible root grid to to achieve high-performance voxel offsetting in HDTs.

5.2 HDT Offsetting with Tunable Node Branching

As expressed in Eq. 5.1, with a two-fold size of the branching factor, the HDT gets wider and hence the height of the HDT decreased by half. For instance, if the node branching is changed to four from the default setting of octree structure (*i.e.*, $B = 2$), then the depth of the leaf nodes in the new structure will be $\frac{h}{2}$, where h is the original HDT height with octree configuration. In this study, with the root grid size (R) set to 4 and the leaf grid size (L) set to default 16, offsetting is performed at the HDTs with different branching configurations. We experiment and analyze the dilation experiment at 4096^3 resolution for three different offset distances (40 voxels, 60 voxels and 80 voxels) with HDTs configured at three branching factors: for $B = 2$ (*i.e.*, $b = 1$), $B = 4$ (*i.e.*, $b = 2$) and $B = 8$ (*i.e.*, $b = 3$). For these settings, the tree hierarchy in HDT is respectively 6, 3 and 2, as expressed from Eq. 5.1.

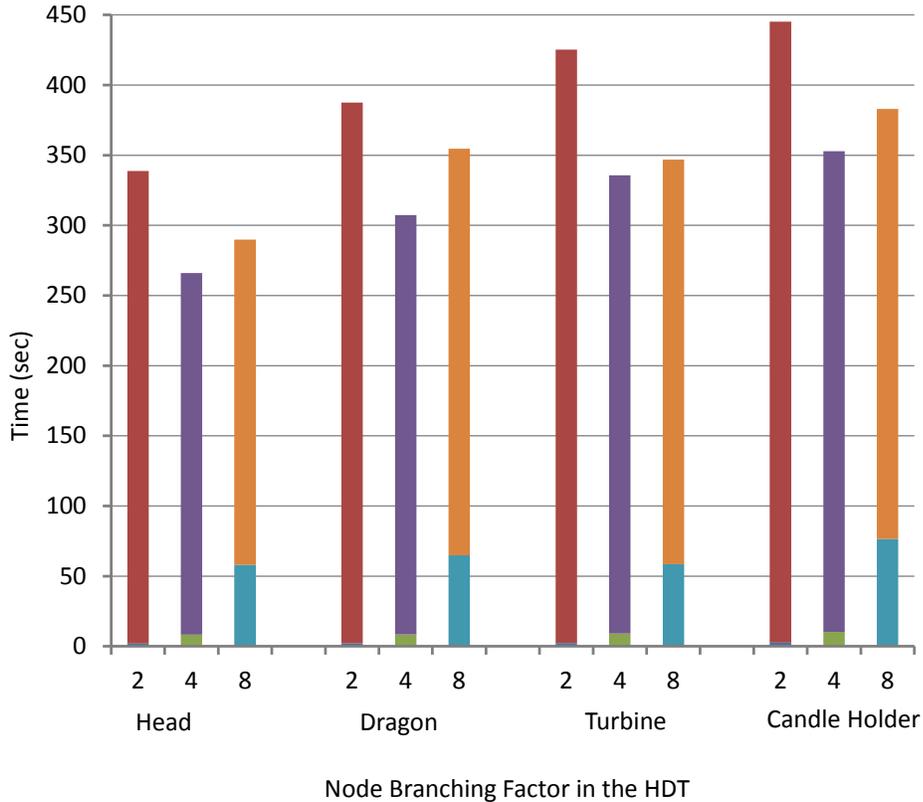


Figure 5.4: Impact of different branching factors on 40 voxels dilations at $4096 \times 4096 \times 4096$ resolution.

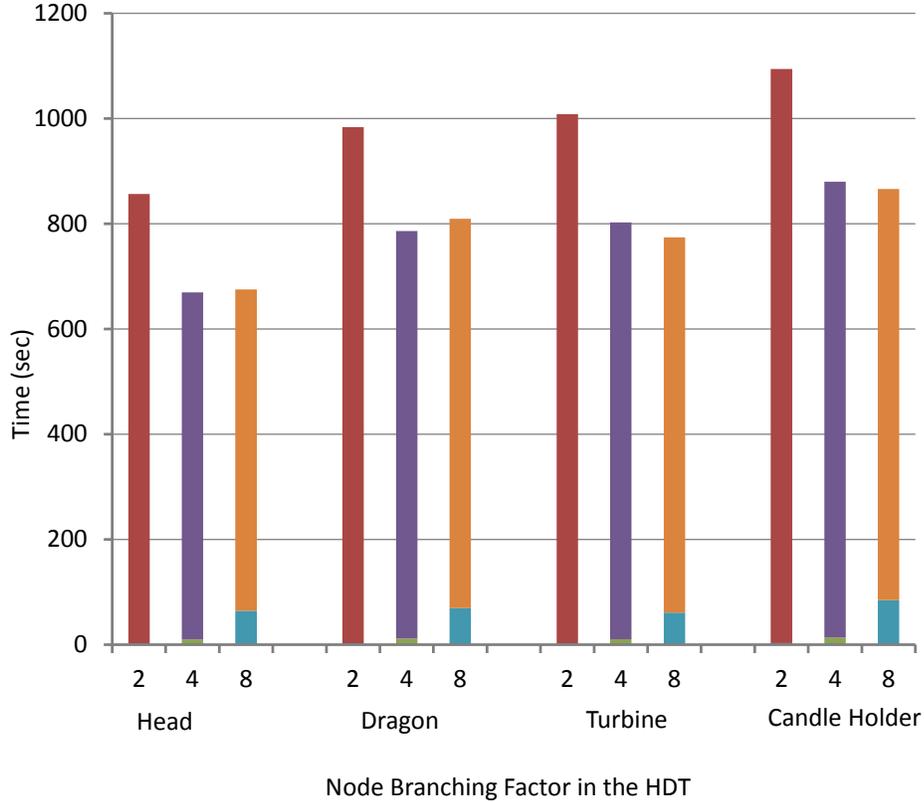


Figure 5.5: Impact of different branching factors on 60 voxels dilations at $4096 \times 4096 \times 4096$ resolution.

5.2.1 HDT Dilations with Node Branching of 4

The dilation times reported in Figures 5.4, 5.5, and 5.6 show the breakdown of the two steps of the presented HDT offsetting algorithm in Section 4.3. For each stacked bar in these figures, the taller component represents the offsetting time with morphological filters, while the bottom component indicates the preprocessing time to generate the skeletal dilated HDT. As it obvious that with the default HDT configuration with a node branching of two, the time to compute the skeleton of the offset HDT is quite negligible compared to the the computation time of the the morphological filtering. With larger branching values the skeletal computation time increases, this observation is similar to what we discussed in Section 3.4.2 as an analysis on the increases of branching time in HDT construction process.

As detailed in Section 4.3, the construction of the skeletal HDT is a hierarchical

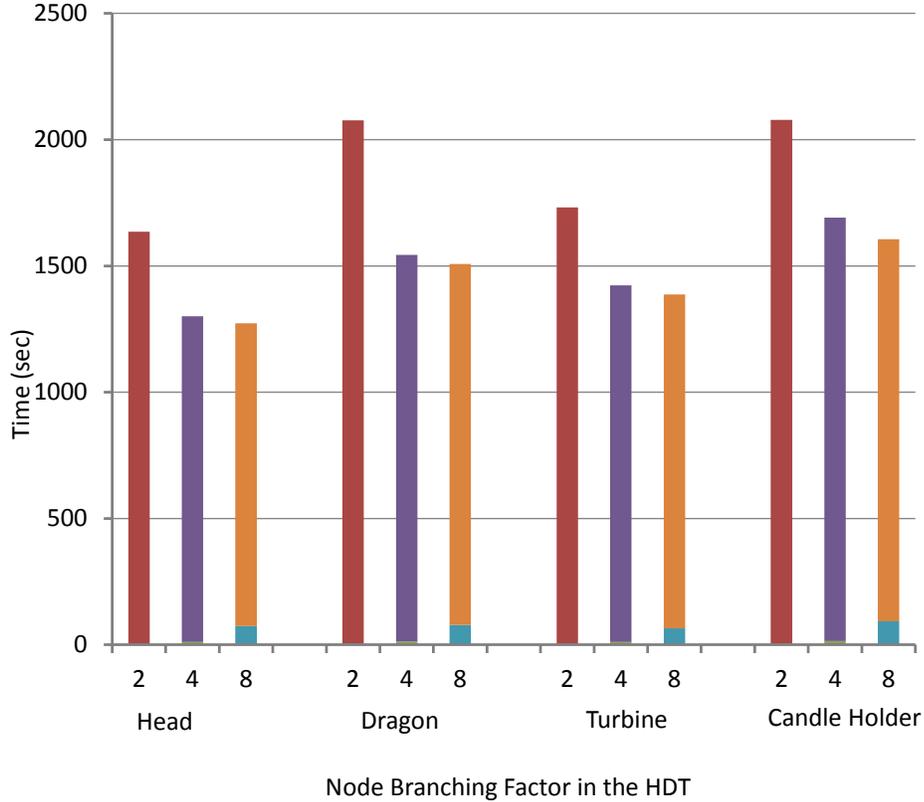


Figure 5.6: Impact of different branching factors on 80 voxels dilations at $4096 \times 4096 \times 4096$ resolution.

process, where the parallelism at the top levels are severely limited by the low number of cells in the root grid. With a larger branching factor, both the original HDT and the dilated HDT contain more tree cells for the same number of leaf grids. This is due to that each cell partitioning creates $8\times$ more children for a two-fold larger value of B . For instance, with $B = 8$ each cell in HDT will have $8 \times 8 \times 8=512$ children. Such proliferation of cells in the underlying HDT result into proportionate growth in the skeletal computation time. Now, for a larger branching factor, the HDT gets shallower that positively impacts the filter computation time. As shown in Figures 5.4, 5.5, and 5.6, the dilation times with $B = 4$ reduce by 18 – 26% compared to default branching of $B = 2$ across different offset distances. Thus, even with the overhead of higher skeletal computation time the overall dilation times with $B = 4$ improves by a factor of 1.22 – 1.35 \times .

5.2.2 HDT Dilations with Node Branching of 8

However, once the branching becomes too wide (*i.e.*, $B = 8$), the sharp increase in skeletal computation time offsets the relative modest gain in the filter computation time. For instance, the filter computation time for 40 voxels dilation (Figure 5.4) reduces by 3 – 12% with $B = 8$, however the skeletal computation time increases by a factor of 7 – 8 \times compared to that with $B = 4$. The overall dilation times thus observe a slowdown between 3% to 15%.

Now, the skeletal computation time increase just modestly for larger offset distances, whereas the kernel computation time scales at a higher rate with larger filters. Thus, with larger offset distances, the overall HDT dilation times with a branching factor of 8 get faster than the results with a branching factor of 4. For instance, as we see in Figure 5.6, the total offsetting times achieve bit higher performance (2 – 5%) compared to the results with a branching factor of 4. An interesting observation is that, for the models with regular patterns (*i.e.*, Turbine and Candle Holder) larger B in general demonstrate higher performance compared to the non-regular shapes (Head or Dragon). For instance, as shown with 60 voxels dilation results (Figure 5.5), the overall offsetting times with $B = 8$ for the regular shapes are faster than with $B = 4$, whereas for the non-regular shapes the former HDT configuration is slower than the latter.

5.2.3 Analysis on the CUDA Profiler Statistics

As the filter computation is the dominating component of the overall offset algorithm, we investigate the impact of the tunable node branching on the parallel execution efficiency of the filter algorithm on GPUs with different HDT configurations. Toward that goal, similar to the case with tunable root grid size, we analyze two CUDA performance statistics, namely the *warp execution efficiency* and the *global memory transactions*. Figure 5.7 shows the relative warp execution efficiency and relative global memory transactions performance metrics for 60 voxels dilation of the Head

model. To avoid spurious hardware statistics, all these results are averaged across 10 kernel launches. As demonstrated in Figure 5.7, relative to the default branching factor of 2, the warp execution efficiency increases by 5% and 8% for branching factor of 4 and 8 respectively. On the other hand, the global memory requests decrease by respectively 23% and 28% for branching factor of 4 and 8 compared to the default branching factor of 2. Thus, the compounding impacts with the larger branching factors demonstrate better code execution efficiency on parallel GPU hardware, which earlier reflected in the reported results in Figures 5.4, 5.5, and 5.6.

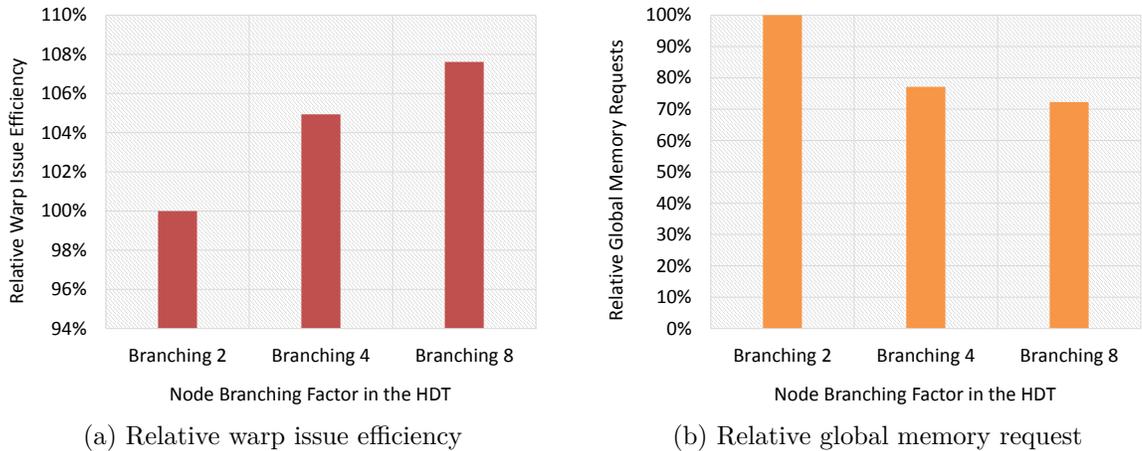


Figure 5.7: Impact on CUDA warp issue efficiency and global memory transactions with different branching factors at 60 voxels dilations for the Head Model.

The performance statistic charts closely reflect the observed trend in the dilation data reported in Figure 5.5. As the depth of the HDT reduces to three with branching value of 4 that originally spans a deep hierarchy of level six with default octree setting, both the warp execution efficiency and the relative global memory request demonstrate noticeable performance improvements. On the other hand, when the HDT is configured with a branching of 8 the height of the HDT reduces to two from 3 with branching of 4, thus the improvements in performance statistic observe only modest gain. In Figure 5.5, this gain is reflected in the relative magnitude of the filter computation times (purple bar with $B = 4$ and orange bar with $B = 8$).

5.3 HDT Offsetting with Tunable Leaf Grid

5.3.1 Parallelism, Redundant Computation and Storage Trade-Offs

As discussed in Chapter 1, on graphics hardware the organization of computations and data for a given algorithm are constrained by fundamental tensions between parallelism, storage and redundant computation. We argued that with the traditional approach of grid based voxel representation while the parallelism can be best leveraged, it imposes tremendous challenge on computational and storage requirement to deal with the *non-active* voxels. In Section 3.4.3, we presented detailed analysis on the trade-offs between parallelism and storage to highlight the need of a hybrid approach that laid the foundation of the HDT structure. In this section, our study focuses on trading off the parallelism to optimize the redundant computations in the voxel offsetting through tunable size of the leaf grid in the HDT.

Table 5.1: Comparisons of different voxel data representations.

	Parallelism	Storage	Redundant Computation
Uniform Grid	High	High	High
List of Active Voxels	Medium	Medium	Low
Octree (SVO)	Low	Low	Low

For better comprehending these contrasting trade-offs, we enlist some fundamental choices of voxel data representation in Table 5.1. As obvious, a regularly sampled 3D grid provides perfect parallelizability, while both the storage and redundancy in computations are high. A simple alternative to a regular grid is a flat list of active voxels, where each element in the list basically encodes the coordinates of individual voxels. List based sparse representation avoids redundant computation, as it only stores the boundary voxels; however the storage requirement could be relatively high as it requires $\log_2 R$ bits to encode single dimension with R discrete points. Thus, unlike the octree, the topology storage in a list based representation scales with the

resolution, and requires a total of $3 \times \log_2 R$ bits for identifying the coordinates in volumetric space. With sparse voxel octree (SVO), similar to the list based sparse representation, computations are done only with active voxels and the storage is optimum as the topology information from the parent to the descendants can be compactly stored. However, the deep hierarchy of SVO, typically with a tree height of $\log_2 R$, challenges the data parallel processing on GPUs.

By design, the HDT structure is a hybrid representation of dense grid and sparse octree, and hence tuning the size of the leaf grid we can optimize the voxel offsetting algorithm through fine trade-off between parallelism and redundancy in computation. While larger size of the leaf grid makes the HDT more towards a uniform grid and offers generally higher parallelizability, a reduced size of the leaf grid makes the HDT more akin to a sparse octree and incurs lower computation redundancy.

5.3.2 HDT Dilations with Leaf Grid of 8^3

In this study, with the tree branching factor fixed to $B = 2$, the root grid size (R) and the leaf grid size (L) are set to two different settings. In the first configuration, both R and L are set to the default values of 16, while in the second configuration R and L are set to 32 and 8 respectively. Such assignments to R and L are chosen so that for these two configurations, the depth of the HDT hierarchy remains same. The dilations times with these HDT setups at 2048^3 and 4096^3 resolutions for our CAD models appear respectively in Tables 5.2, 5.3, 5.4 and 5.5.

As the speedup column in Tables 5.2, 5.3, 5.4 and 5.5 demonstrate, offsetting can be significantly accelerated using a leaf grid of $8 \times 8 \times 8$. For our experimented benchmarks, at 2048^3 resolution an average speedup of $1.7 - 1.9\times$ is observed, while an average speedup of $2.0 - 2.2\times$ is achieved at 4096^3 resolution. Relatively higher speedups at 4096^3 resolution for larger offset distances, *i.e.*, 60 – 100 voxels dilations happen due to relatively greater acceleration in the morphological filtering step of the HDT offsetting. For instance, the filtering computation for 60 voxels offsetting

Table 5.2: Dilation times for the *Head* model with different leaf sizes.

Resolution	2048 ³			4096 ³		
Leaf Grid Size	$L = 16$	$L = 8$	Speedup	$L = 16$	$L = 8$	Speedup
20 voxels	14.7	6.4	2.3×	59.0	25.8	2.3×
40 voxels	60.7	25.3	2.4×	254.2	99.7	2.6×
60 voxels	151.1	89.8	1.7×	611.9	317.8	1.9×
80 voxels	288.6	187.4	1.5×	1147.1	653.4	1.8×
100 voxels	629.0	417.7	1.5×	2294.5	1347.5	1.7×

Table 5.3: Dilation times for the *Dragon* model with different leaf sizes.

Resolution	2048 ³			4096 ³		
Leaf Grid Size	$L = 16$	$L = 8$	Speedup	$L = 16$	$L = 8$	Speedup
20 voxels	17.2	7.5	2.3×	75.2	35.4	2.1×
40 voxels	71.1	29.2	2.4×	325.3	125.2	2.6×
60 voxels	172.0	97.8	1.8×	761.9	376.4	2.0×
80 voxels	325.4	197.2	1.7×	1415.7	759.4	1.9×
100 voxels	674.1	428.5	1.6×	2730.5	1484.0	1.8×

at 2048³ resolution with $L = 8$ accelerates by 1.75×, 1.83×, 1.69× and 1.87× respectively for the Head, Dragon, Turbine and the Candle Holder, whereas we observed respective speed-ups of 1.98×, 2.08×, 2.02× and 2.05× at 4096³ resolution. This relative acceleration at higher resolution is likely due to the impact of larger root grid with $L = 8$ compared to that with $L = 16$ at different modeling resolutions, as we experimentally analyzed in Section 5.1.2 (cf. Figures 5.1 and 5.2).

The performance gain in HDT offsetting with a smaller leaf size of $8 \times 8 \times 8$ comes from the optimization in computation redundancy, while maintaining the same level of parallelism offers with the HDTs configured with default leaf grid size of $16 \times 16 \times 16$. First, the computational redundancy in the HDT stems from the grouping of voxels

Table 5.4: Dilation times for the *Turbine* model with different leaf sizes.

Resolution	2048 ³			4096 ³		
Leaf Grid Size	$L = 16$	$L = 8$	Speedup	$L = 16$	$L = 8$	Speedup
20 voxels	14.7	7.6	1.9×	75.6	32.9	2.3×
40 voxels	56.8	26.2	2.2×	314.2	119.6	2.6×
60 voxels	119.1	73.9	1.6×	702.6	357.3	2.0×
80 voxels	206.9	135.1	1.5×	1273.4	647.1	2.0×
100 voxels	407.5	283.8	1.4×	2322.7	1183.9	2.0×

Table 5.5: Dilation times for the *Candle Holder* model with different leaf sizes.

Resolution	2048 ³			4096 ³		
Leaf Grid Size	$L = 16$	$L = 8$	Speedup	$L = 16$	$L = 8$	Speedup
20 voxels	19.1	8.6	2.2×	76.8	33.2	2.3×
40 voxels	77.6	32.4	2.4×	346.5	132.8	2.6×
60 voxels	183.9	102.6	1.8×	843.2	424.5	2.0×
80 voxels	325.4	203.2	1.6×	1531.3	824.4	1.9×
100 voxels	645.3	425.1	1.5×	2885.0	1599.2	1.8×

in the leaf grid. As we earlier discussed in the computational complexity analysis on the presented morphological offsetting algorithm in Section 4.4.3, the asymptotic complexity of filter computation scales in $\mathcal{O}(L^3NM)$, where L is the size of the leaf grid, N is the number of leaf grids processed, and M is the number of kernel boundary points. Here, M depends on the offset distance (in voxel unit), and can be considered unchanged across our experimented modeling resolutions. Now, for a reduced leaf grid size of $\frac{L}{2}$, the number of processed leaf grids N roughly increases by a factor of four. Thus, the overall computation reduced by a factor of two, which is reflected in Tables 5.2, 5.3, 5.4 and 5.5.

As we described in morphological filtering algorithm (Section 4.3.2), in our offsetting implementation a thread block — unit of workload grouping in CUDA — is configured with $L \times L$ threads for a leaf grid size of L^3 . Thus, each CUDA block has total $16 \times 16 = 256$ threads with the default setting of $L = 16$. On the GPU hardware, as 32 threads are grouped in a *warp*, a thread block comprising 256 threads are mapped to $\frac{256}{32} = 8$ warps that can be concurrently executed on the parallel hardware. Contrast to the default setting, once the size of the leaf grid is reduced to 8^3 , there are 8×8 threads in each thread block that are mapped to $\frac{64}{32} = 2$ warps. While it is obvious that for peak hardware utilization each warp must be fully occupied with 32 threads, on the macro-level it also requires to have sufficient number of concurrent warps to be eligible for scheduling to hide long latency operations, particularly, memory reads and writes, among others. As we process voxel data at high resolutions, typically our offsetting algorithm deals with tens of thousands of leaf grid with the default $L = 16$ setup, and the number of leaf grids with $L = 8$ is roughly four times larger. Such a high data-parallel workload ensures that the parallelizability is not affected while we tune down the leaf grid size to $8 \times 8 \times 8$ for optimization of the redundant computations.

A natural approach to further optimize the redundant computations in offsetting is to adopt even smaller leaf grid in the HDTs, for instance, a size of $4 \times 4 \times 4$. As we discussed above, the constraint here is that in our filtering algorithm a thread block consists of $L \times L$ threads, and so with $L = 4$ we get a total of 16 threads per block. Thus, with $L = 4$ each thread block becomes too small to fully occupy even a single warp, as it requires a multiple of 32 threads to optimally map the CUDA threads to the warps in the hardware. As it is not permitted to bundle threads from multiple blocks into one warp in the current generation graphics hardware, configuring thread block to 4×4 results into 50% theoretical efficiency on the GPU hardware. Our experimental evaluations thus suggest the leaf grid size of $8 \times 8 \times 8$ to be an optimal

choice that allows perfect mapping of the CUDA threads to the GPU warps, and simultaneously optimizes the redundancy in the filtering computation.

CHAPTER 6

VOXEL OFFSETTING WITH TUNABLE SPEED AND PRECISION

6.1 Offsetting Error Measurement

6.1.1 Errors in Voxel Offsetting

Error or discrepancy in the geometric computations can be generated from diverse sources. The most obvious source is the approximation error induced from the discretized representation of the solids. The application of our research is in the field of rapid prototyping, where approximated geometry with tens of microns discrepancy is typically permitted. Our objective in this dissertation is to demonstrate the voxel offsetting at sufficiently high resolutions such that the voxel size inherently meets the accuracy constraint in the target HDT representation. Hence, in this study we do not consider analyzing the error caused by the discretization of the triangle mesh model into the HDT voxel representation.

We confine the scope of the error analysis in the voxel offsetting to be the errors caused by the limited resolution in the discretization of the different morphological filters used in our study. As can be realized from Figure 4.6(a), with a larger offset distance the structuring element introduces less approximation error than a structuring element representing a shorter offset value. Thus, the use of the different sizes of structuring elements in the morphological filtering introduces different magnitudes of approximation error in the overall computation.

To study the accuracy of the offsetting results, the distances between all the active voxels in the dilated HDT and the boundary surface of the given solid are measured. In our analysis, the offsetting discrepancy is quantified as *the average error*. Let V denotes the set of active voxels in the offset HDT, and for an active voxel $v \in V$ the distance to the nearest boundary voxel in the input HDT is denoted by D_v . Then, for a given offset distance r , the average error, E_{avg} is defined as below.

$$E_{avg} = \frac{1}{|V|} \sum_{\forall v \in V} |D_v - r| \quad (6.1)$$

As formulated in Equation 6.1, to measure the offsetting error we need an efficient method to compute the distance to the nearest boundary voxel in the given HDT for all the active voxels in the dilated HDT. This effectively translates the error analysis to the nearest neighbor search problem.

The nearest neighbor search is a fundamental computational primitive widely used in diverse applications dealing with massive datasets [14]. The nearest neighbor problem can be defined as follows: given a collection of points and a target query point, find the data point that is closest to the query. A particularly interesting and well-studied instance is where the data points live in a d -dimensional Euclidean space. This problem has a broad set of applications in data processing and analysis. For instance, it forms the basis of a widely used classification method in machine learning: to give a label for a new object, find the most similar labeled object and copy its label. Other applications include information retrieval, searching image databases, finding duplicate files and web pages, and many others.

Many efficient approaches have been proposed in the literature that pre-processes the dataset so the nearest neighbor can be identified efficiently. The first such data structure, called *kd-trees* was introduced by Jon Bentley [20], and remains one of the most popular data structures used for searching in multidimensional spaces. Since then many other multidimensional data structures have been proposed [90]. However, despite the decades of intensive effort, the standard solutions suffer from either space or query time that is exponential to the size of dimension (d). In recent years, several researchers have proposed methods for overcoming the running time bottleneck by using approximation, such as, *locality-sensitive hashing* [51, 15]. The appeal of this approach is that, in many cases, an approximate nearest neighbor is almost as good as the exact one.

6.1.2 Nearest Neighbor Search in the HDT

Naively implemented, finding a closest boundary voxel requires iterating over the entire candidate dataset, i.e., it is in $\mathcal{O}(n)$, where n is the number of voxels in the original HDT. At our target resolutions, typically a solid is represented with tens of millions of boundary voxels, and thus it turns out quite inefficient to iterate over all the voxels to determine the closest one. As in the HDT the voxels are represented in a hierarchical form, the search for the closest neighboring voxels in the HDT can leverage the property of the underlying data structure to avoid the expensive approach of enumeration of the entire dataset. We present a simple and intuitive approach of computing the nearest boundary voxel search in the HDT that leverages GPU’s massive parallelism. Algorithm 3 presents a nearest neighbor search technique that computes the distance to the closest boundary voxel in the original HDT for all the boundary voxels on the dilated HDT. Since the HDT is a hybrid representation combining grid and octree, conceptually our algorithm is similar to prior proposal that works with underlying octree structure [19].

Algorithm 3: Compute the Offsetting Error

Input: A given HDT *hdtOriginal*, an offset value *distance*, and the offset HDT *hdtDilated*

Output: The offsetting error for the boundary voxels in the dilated HDT

- 1 *leafs* \leftarrow List of leaf grids in *hdtDilated*
 - 2 *errorTable* \leftarrow Buffer that stores the computed error values
 - 3 *boundaryVoxels* \leftarrow Buffer that counts the number of boundary voxel in *hdtDilated*
 - 4 Allocate and Initialize CUDA buffers
 - 5 `FINDCLOSEST(hdtOriginal, hdtDilated, leafs, distance, errorTable, boundaryVoxels)`
-

As listed in Algorithm 3, for the *average* offset error computation it requires to

accumulate the individual cell error, and the total number of boundary voxels in the dilated HDT. Once the buffers are allocated for storing these values on GPU, the host side `FINDCLOSEST` routine is invoked that configures the CUDA grid and thread block setup. Like the case of HDT skeleton and HDT offset computation, we configure each thread block to be 16×16 (for the default leaf configuration 16^3). The entry point for the GPU execution is `FINDCLOSESTBOUNDARYCELL` procedure that computes the offsetting error for the adjacent 16 cells sharing the same Z-coordinate (line 8).

Procedure `FindClosest`(*hdtOriginal, hdtDilated, leafs, radius, errorTable, boundaryVoxels*)

```

1 blocksInGrid  $\leftarrow$  leafs.size()
2 threadsPerBlock  $\leftarrow$  LEAF_BRANCHING  $\times$  LEAF_BRANCHING
3 FINDCLOSESTBOUNDARYCELL  $\lll$  blocksInGrid, threadsPerBlock  $\ggg$ 
   (hdtOriginal, hdtDilated, leafs, radius, errorTable, boundaryVoxels)

```

The `FINDCLOSESTBOUNDARYCELL` procedure checks the state of individual cell in the dilated HDT (line 12), and for every boundary voxel it increments the local counter `voxelCount` (line 13), and invokes the `FINDCLOSESTINNODE` routine that finds the nearest boundary voxel in the original HDT. Depending on the computed distance to the nearest boundary cell in the original HDT, the cell error is accumulated in the local counter `errorSum` (line 17). Finally, the accumulated values of `voxelCount` and `errorSum` are written to the corresponding index on the GPU buffers (lines 22-23).

For a given cell center in the dilated HDT (*i.e.*, the center of a boundary voxel), the `FINDCLOSESTINNODE` procedure computes the dilated bounds of the cell (line 2) depending on the given offset `radius`. It checks for possible neighbor search within the range of overlapped root cells in the original HDT (lines 6-8). For every intersecting root cell in the original HDT, `SEARCHNEARESTNEIGHBOR` is invoked (line

Procedure FindClosestBoundaryCell(*hdtOriginal*, *hdtDilated*, *leafs*, *radius*,
errorTable, *boundaryVoxels*)

```

1 blockId ← CUDA block id
2 x ← threadIdx.x
3 y ← threadIdx.y
4 voxelCount ← 0
5 errorSum ← 0
6 leafElem ← GETELEMENT( hdtDilated, leafs[blockId] )
7 leafGrid ← GETLEAFGRID( hdtDilated, leafElem )
8 for z ← 0 to (LEAF_BRANCHING - 1) do
9     cellIndex ← <x, y, z>
10    cellCenter ← COMPUTECELLCENTER ( hdtDilated, leafElem, cellIndex )
11    cellState ← GETLEAFCELLSTATE ( leafGrid, cellCenter )
12    if cellState == BOUNDARY then
13        Increment voxelCount
14        foundNeighbor ← FINDCLOSESTINNODE ( hdtOriginal, cellCenter,
15                                             shortestDistance, radius )
16        if foundNeighbor then
17            cellError ← |shortestDistance - radius|
18            errorSum ← errorSum + cellError
19        end
20    end
21 index ←
    ( blockId × LEAF_BRANCHING × LEAF_BRANCHING ) + ( x × LEAF_BRANCHING ) + y
22 boundaryVoxles[index] ← voxelCount
23 errorTable[index] ← errorSum

```

10) that recursively partitions the root cell, and checks for overlapping in the spatial hierarchy. If any nearest neighbor is found, the shortest distance to the closest voxel is updated (line 11), which gets accumulated in the caller FINDCLOSESTBOUNDARYCELL routine.

```

Procedure FindClosestInNode(hdtOriginal, point, shortestDistance, radius)
1  sphereBounds  $\leftarrow$  Box with bounds [point, point]
2  sphereBounds  $\leftarrow$  GROWELEMENTBOUNDS( sphereBounds, radius )
3  foundNeighbor  $\leftarrow$  FALSE
4  shortestDistance  $\leftarrow$   $\infty$ 
5  rootRange  $\leftarrow$  FINDROOTRANGE( sphereBounds, hdtOriginal );
6  for i  $\leftarrow$  rootRange.min.x to rootRange.max.x do
7      for j  $\leftarrow$  rootRange.min.y to rootRange.max.y do
8          for k  $\leftarrow$  rootRange.min.z to rootRange.max.z do
9              root  $\leftarrow$  GETROOT(hdtOriginal, <x, y, z>)
10             if SEARCHNEARESTNEIGHBOR( hdtOriginal, root, point,
11                 shortestDistance, radius) then
12                 shortestDistance  $\leftarrow$  sqrtf( neighborDistance )
13                 foundNeighbor  $\leftarrow$  TRUE
14             end
15         end
16 end
17 return foundNeighbor

```

The SEARCHNEARESTNEIGHBOR procedure is a major computation block that hierarchically subdivides a HDT cell until the leaf grid is reached, and computes the distance to the boundary voxels in the leaf grid of the original HDT. Out of the computed distances to the active voxels, it updates the minimum distance that reflects the nearest boundary voxel in the original HDT for a given active voxel in the

dilated HDT. As outlined in `SEARCHNEARESTNEIGHBOR` code snippet, depending on the state of an element in the HDT different actions are taken. As obvious, if an element with `EMPTY` or `FULL` state is reached (lines 2-3), the volumetric space does not contain any boundary voxels, and hence no closest neighbor is found. If an element with `BRANCH` state is reached (lines 4-8), for each of the 2^3 descendant child cells, recursively `SEARCHNEARESTNEIGHBOR` is invoked.

Conversely, if an element with `BOUNDARY` state is reached (lines 9-25), which indicates a leaf grid in the HDT, the search for active voxels are conducted in the range of overlapped voxels. For each of the voxels of the overlapped region (lines 15-25), if it is is a `BOUNDARY` voxel, distance to the voxel center is computed, and the nearest distance to the voxel in the original HDT is updated. This value is passed to the caller `FINDCLOSESTINNODE`, which is eventually accumulated in the caller `FINDCLOSESTBOUNDARYCELL` routine.

Thus, as expected, the complexity of the closest neighbor search is confined within the leaf grids in the original HDT that may overlap the bounded region of a cell in the dilated HDT. In our approach of finding the nearest neighbor in the HDT, at a particular modeling resolution the computation time is dependent on two factors: (1) distance of offset distance, as it affects the number of leaf grids in the original HDT that may contain some overlapping active voxels, and (2) the modeling parameters of the HDT that govern the depth of hierarchical tree traversal. In the following section, we study some of these impacts on the nearest neighbor search in the HDT.

```

Procedure SearchNearestNeighbor(hdtOriginal, element, point,
shortestDistance, radius)


---


1 elementBounds  $\leftarrow$  GETELEMENTBOUNDS (hdtOriginal, element)
2 if elementState == EMPTY OR elementState == FULL then
3   return FALSE
4 else if elementState == BRANCH then
5   for i  $\leftarrow$  1 to  $2^{\text{BranchingFactor}}$  do
6     child  $\leftarrow$  GETCHILDELEMENT(hdtOriginal, element, i)
7     if SEARCHNEARESTNEIGHBOR( hdtOriginal, child, point,
8       shortestDistance, radius) then
9       return TRUE
9 else if elementState == BOUNDARY then
10   neighborFound  $\leftarrow$  FALSE
11   leafVolume  $\leftarrow$  GETLEAFVOLUME( hdtOriginal, element )
12   sphereBounds  $\leftarrow$  Box with bounds [point, point]
13   sphereBounds  $\leftarrow$  GROWELEMENTBOUNDS( sphereBounds, radius )
14   [start, end]  $\leftarrow$  voxel range leafVolume that overlaps with sphereBounds
15   for i  $\leftarrow$  start.x to end.x do
16     for j  $\leftarrow$  start.y to end.y do
17       for k  $\leftarrow$  start.z to end.z do
18         cellBox  $\leftarrow$  cubical space for  $\langle i, j, k \rangle$ 
19         if BOXOVERLAPSPHERE( cellBox, point, radius) then
20           cellState  $\leftarrow$  GETLEAFCELLSTATE( leafVolume, cellIndex)
21           if cellState == BOUNDARY then
22             Compute the distance from point to the center of cellBox
23             Update the distance to the shortest voxel
24             neighborFound  $\leftarrow$  TRUE
25   return neighborFound
26 return FALSE

```

6.1.3 Evaluations on Voxel Offsetting Error

To measure the accuracy of the morphological offset operation using the hybrid dynamic trees, we consider the average error metric, as defined in Equation 6.1, to benchmark the experimental evaluations. Figure 6.1 presents the offsetting accuracy results for dilation at 2048^3 resolution. Here the *normalized* average error is reported with respect to the offset distance. The values of offset distance r are chosen to be 40 voxels, 60 voxels and 80 voxels. As illustrated in the figures, the normalized offsetting errors linearly decline with larger offset distances, which in turn implies that for a particular input model the absolute average error remain same for different values of dilation. For instance, as shown in Figure 6.1 with 80 voxel offsetting the normalized average errors are in between 0.005 and 0.007, whereas the errors for 40 voxel dilation are between 0.010 and 0.014.

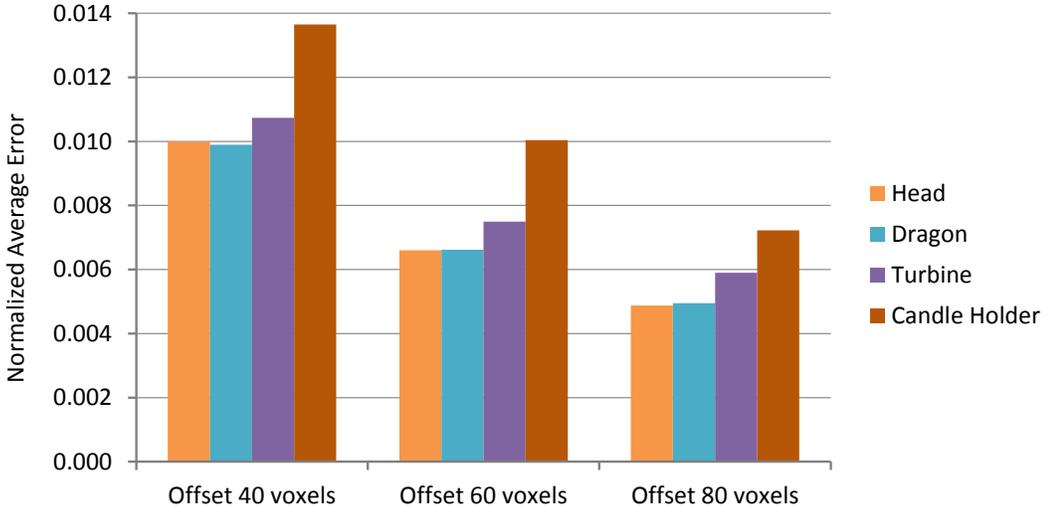


Figure 6.1: Offsetting error for different dilation distances.

To understand the complexity of the nearest neighbor computation in the HDTs, we further study the time required to compute the closest boundary voxel for different offset distances. Figure 6.2 reports the time of error computation averaged over the all the boundary voxels in the dilated HDT. Naturally, the dilations with larger

values take longer to compute the offsetting error, as the number of candidates in the proximity bound scales with the magnitude of offset distance. An interesting observation here is that with irregular models, such as, Head and Dragon the relative computation time scales much faster than the regular models, such as, Turbine and Candle Holder. For instance, the offset error computation with 60 voxels grow by a factor of 1.6 – 2.3 compared to the error timings with 40 voxels for the Head and Dragon, whereas for the Turbine and Candle Holder they increase by a factor of 1.3 – 1.4. Similarly, relative to the offset error computation with 40 voxels, the nearest voxel search with 80 voxels dilation takes 2.7 – 3.6 \times longer for the Head and Dragon, and for the Turbine and Candle Holder it takes 2 \times higher. This seems to be due to spatial variation in the geometric patterns of the respective models. For the Turbine and Candle Holder the boundary voxels on the original HDTs likely to be in close proximity of the dilated surface than in the case of the Head and Dragon model.

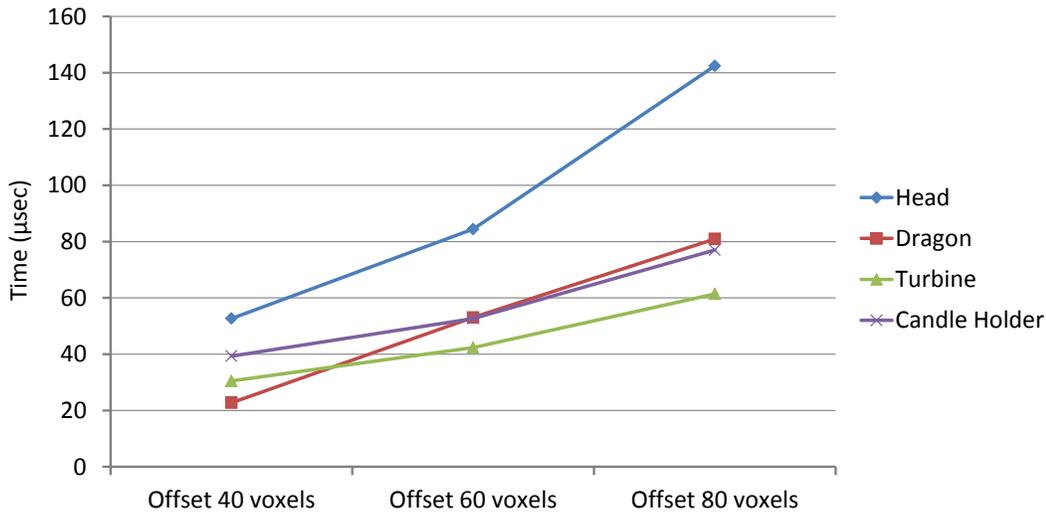


Figure 6.2: Offsetting error for different dilation distances.

6.2 Speed and Precision Trade-Offs in Voxel Offsetting

6.2.1 High-Performance Offsetting with Kernel Decomposition

Voxel offsetting with three-dimensional morphological filtering for a very large offset distance is computationally challenging, as both the number of processed leaf grids and the size of the kernel boundary points increase with larger offset values. We presented in Chapter 5 that voxel offsetting using the hybrid dynamic trees can be significantly accelerated through careful selection of the HDT configuration parameters. For the scenario of 100 voxel dilations at 4096^3 resolution, we showed that using a smaller leaf, for instance, can be particularly useful to reduce the dilation times roughly by a factor of two. Yet a computation time in range of 20 – 27 minutes for a single offsetting operation at 4096^3 resolution deems not to be practical for an interactive CAM application. To further optimize the computations, particularly for offsetting at high resolution, a suitable knob is to tune the size of the morphological filters.

As examined in the Chapter 4, for the ring morphological template the computational complexity grows quadratically with the size of the structuring element¹. The size of kernel is a critical parameter in morphological filtering, which we can exploit as a performance tuning knob to trade-off between speed and quality in offsetting operations. The general approach to deal with the efficiency problem is to decompose a large structuring element into several smaller ones, as it was explained in Zhuang and Haralick [107]. The basic relations for morphological decompositions that are applied here are described in [107] as:

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C \quad (6.2)$$

$$A \ominus (B \oplus C) = (A \ominus B) \ominus C \quad (6.3)$$

This means, if we have a decomposition of a large structuring element S into

¹This is analogous to the ratio of the areas of a sphere (in 3D case) with $2\times$ larger radius.

several smaller ones:

$$S = H_1 \oplus H_2 \oplus H_3 \oplus \dots \oplus H_N \quad (6.4)$$

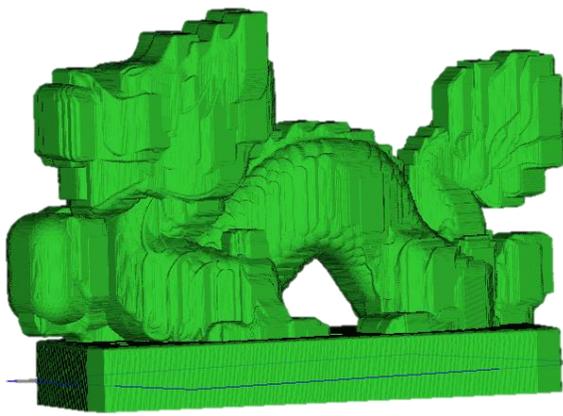
Morphological operation with a large structuring element can be computed efficiently by the application of several smaller ones:

$$A \oplus S = (((A \oplus H_1) \oplus H_2) \dots) \oplus H_N \quad (6.5)$$

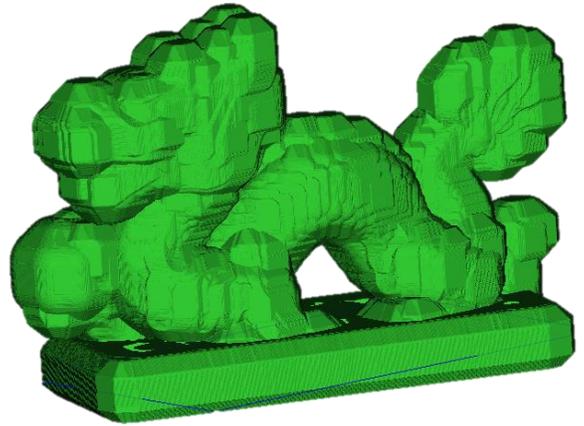
$$A \ominus S = (((A \ominus H_1) \ominus H_2) \dots) \ominus H_N \quad (6.6)$$

Thus, an offset computation with a distance r can be decomposed into n successive offsetting operations with distances r_1, r_2, \dots, r_n , if the offsetting distances satisfy: 1) $\sum_{i=1}^n r_i = r$, and 2) all r_i s and r have the same sign. The impact of splitting a large offset distance over a set of successively smaller offset distances can be realized from Figure 4.6(a) that entails that as the radius of the structure element is reduced by half, the number of discretized structuring element points also decreases by approximately by a factor of two for the 2D case, while in the 3D space the number of discretized structuring element points decreases by factor of four. However, with a halved size of the kernel, the morphological filtering needs to be applied twice, which introduces the geometric approximation errors twice in the offset computation.

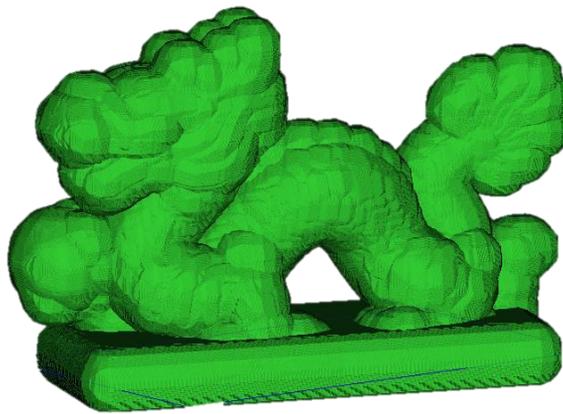
Figure 6.3 illustrates the impact of successive dilations on the quality of offsetting outcomes. In each of the six test scenarios in Figures 6.3 (a)-(f), the Dragon model is dilated by 64 voxels applying different sizes of the filter. Fig. 6.3(a) demonstrates the dilated Dragon when a small filter size of $2 \times 2 \times 2$ is applied successively for 32 iterations. Application of a $2 \times$ larger filter results into more accurate outcome, as can be clearly distinguished in Fig. 6.3(b). As expected, the quality of the offsetting scales with successively larger filters as shown in Figures 6.3(c)-(d). However, after certain point the outcomes of the voxel dilations using even larger filters can be hardly distinguishable visually, as the outputs of Figures 6.3(e) and 6.3(f) depict.



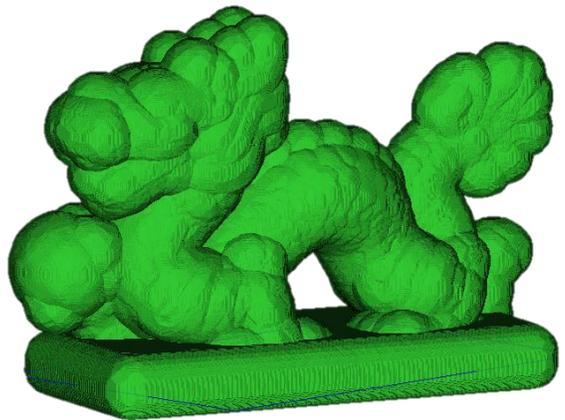
(a) 32×2 voxels offsetting



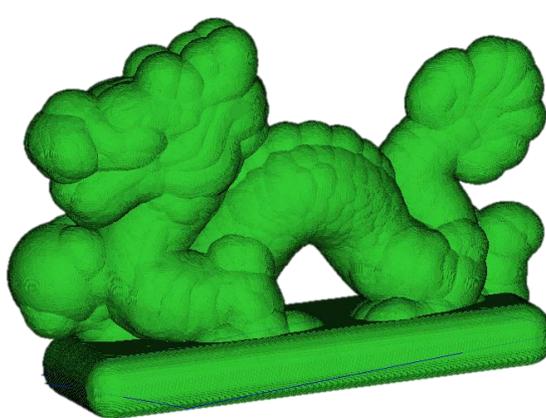
(b) 16×4 voxels offsetting



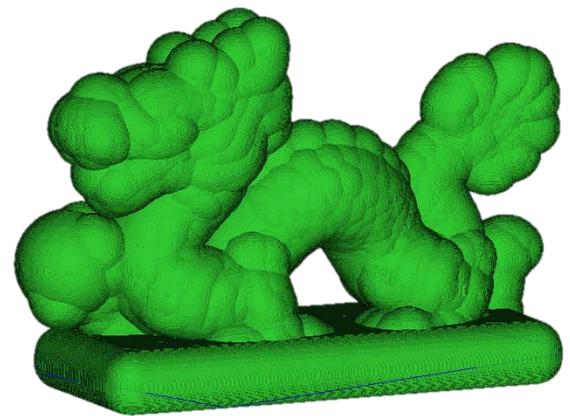
(c) 8×8 voxels offsetting



(d) 4×16 voxels offsetting



(e) 2×32 voxels offsetting



(f) 1×64 voxels offsetting

Figure 6.3: Visualization of the impact of filter size in morphological voxel-offsetting.

6.2.2 Impact of Kernel Decomposition on Offsetting Complexity

As we pointed in Section 4.4.3 that the asymptotic complexity of our presented morphological offsetting scales in $\mathcal{O}(L^3NM)$, where L is the size of the leaf grid, N is the number of leaf grids processed in the computation, and M is the number of kernel boundary points. While in successive offsetting, the size of leaf grid (L) remains unchanged across different experiments, the number of kernel boundary points (M) and the number of processed leaf grids (N) are dependent on the size of the morphological filters.

Table 6.1: Boundary points of the structuring element at different offset distances.

Offset Distance (in voxel unit)	13	25	50	100
Number of Kernel Boundary Points	251	946	3720	14948

The impact of the size of the morphological filters on the dilation time can be realized from the reported number of kernel boundary points used in our successive offsetting operations. As Table 6.1 reveals that with a two-fold larger offset distance, the kernel boundary points roughly increase by a factor of four that proportionately accelerate the computation. For instance, the number of kernel boundary points increase by $3.93\times$ between filters of size 50 and 25, and by $4.02\times$ between filters of size 100 and 50 at 4096^3 resolution. As there are twice number of offsetting operations with a halved size of filter, the overall dilation time with a $2\times$ smaller filter is expected to accelerate by a factor of two. However, as shown in Fig. 6.4(a), a speedup over three is observed with a filter size of 50 voxels compared to that of a filter size of 100 voxels.

Besides the acceleration through reduced kernel boundary points, a secondary impact of using a smaller filter comes from the reduced number of processed leaf grids in the offsetting operation. As reported in Table 4.1 and Table 4.2 (Section 4.4.1), the

number of processed leafs for different offset distances not necessarily increase linearly. For instance, at 4096^3 resolution with 40 voxel and 80 voxel offsetting respectively 229.3 thousand and 359.9 thousand leaf grids are processed for the Head model, *i.e.*, for a $2\times$ larger filter the processed leaf grids increased by a factor of $\frac{359.9}{229.3} = 1.57$. This observation is consistent for all the models across all the target modeling resolutions. Thus, the compounding impacts of the reduce filters on the size of the processed leaf grids and the number of the kernel boundary points unleash noticeable acceleration in voxel offsetting.

6.2.3 Experimentations on Accuracy and Performance Trade-Offs

We study the impact of successive offsetting at 2048^3 and 4096^3 resolutions on the HDTs with optimal leaf grid size of $8 \times 8 \times 8$. In our experiment, successive offsetting is evaluated on three cases. For a dilation of 100 voxels, the offset distance is split into 2, 4 and 8 iterations respectively. The dilation times for the different offsetting choices are presented in Fig. 6.4(a). For the four models, replacing one dilation of 100 voxels with two consecutive dilations each of 50 voxels yields $2.7 - 3.1\times$ speedup as shown in Fig. 6.4(b). With a smaller distance of 25 voxels in four successive offsetting, speedups in range of $5.7 - 7.4\times$ are observed. Continuing the successive offsetting with more iterations, for instance, in eight offsetting operations each of 13 voxels yield $10.0 - 13.6\times$ acceleration.

While speedups are possible through successive offsetting, using too many iterations will increase the approximation error as shown in Fig. 6.5. Not surprisingly, the normalized average errors ($\frac{E_{avg}}{r}$) in Fig. 6.5 monotonically increase with more convolutions of smaller radius. For instance, the offsetting errors increase three-fold for the Turbine model between dilations using a filter of size 13 voxels in stead of a filter of size 50 voxels. Thus, by trading off the geometric quality through successive offsetting with smaller filters, the performance of offset computing can be enhanced significantly. As depicted in Fig. 6.5, splitting a dilation of 100 voxels with two

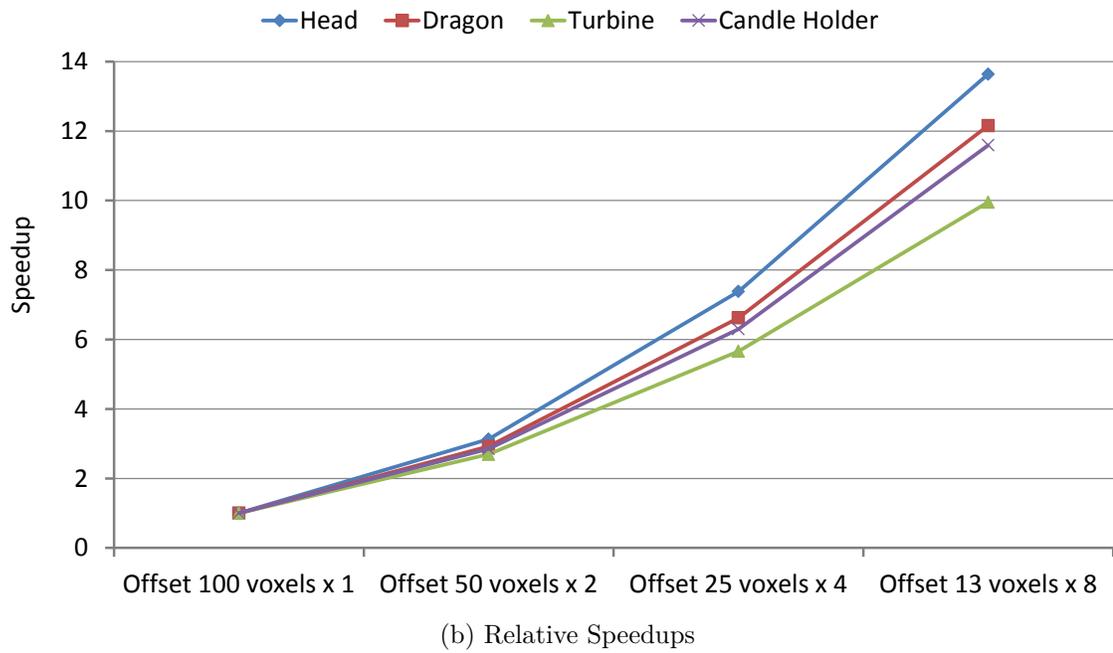
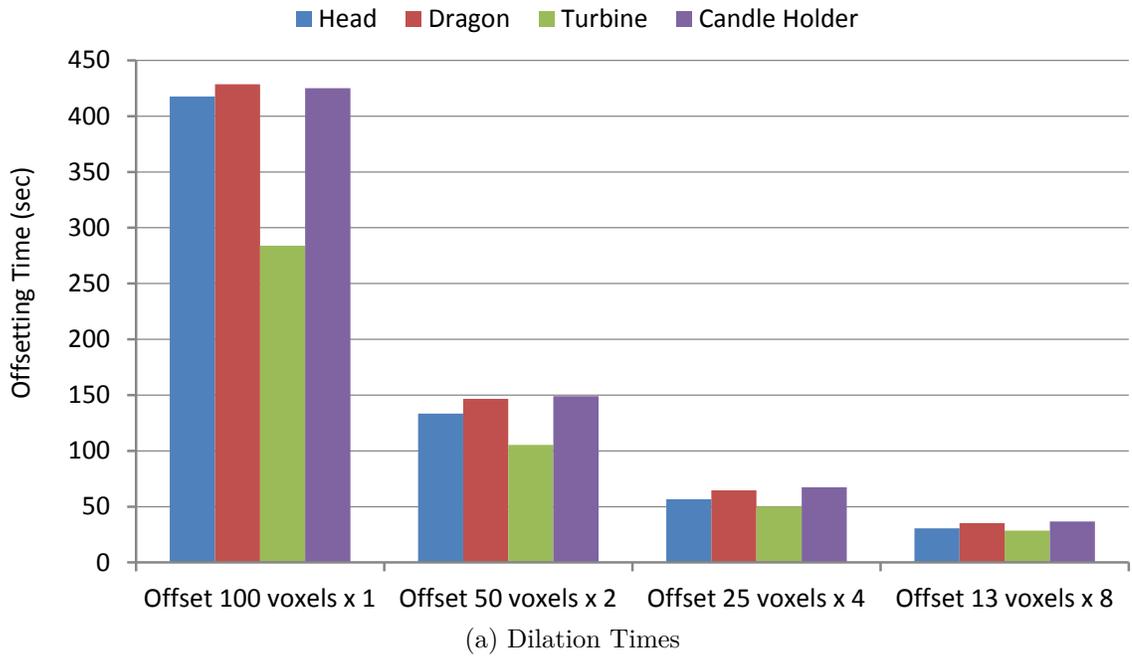


Figure 6.4: Successive offsetting performance at 2048^3 resolution.

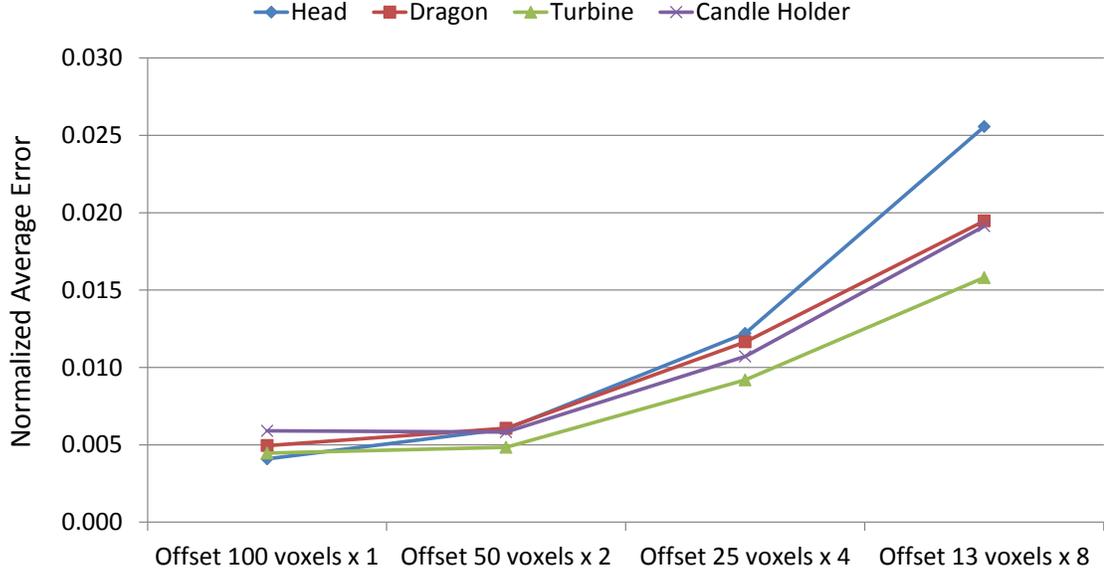


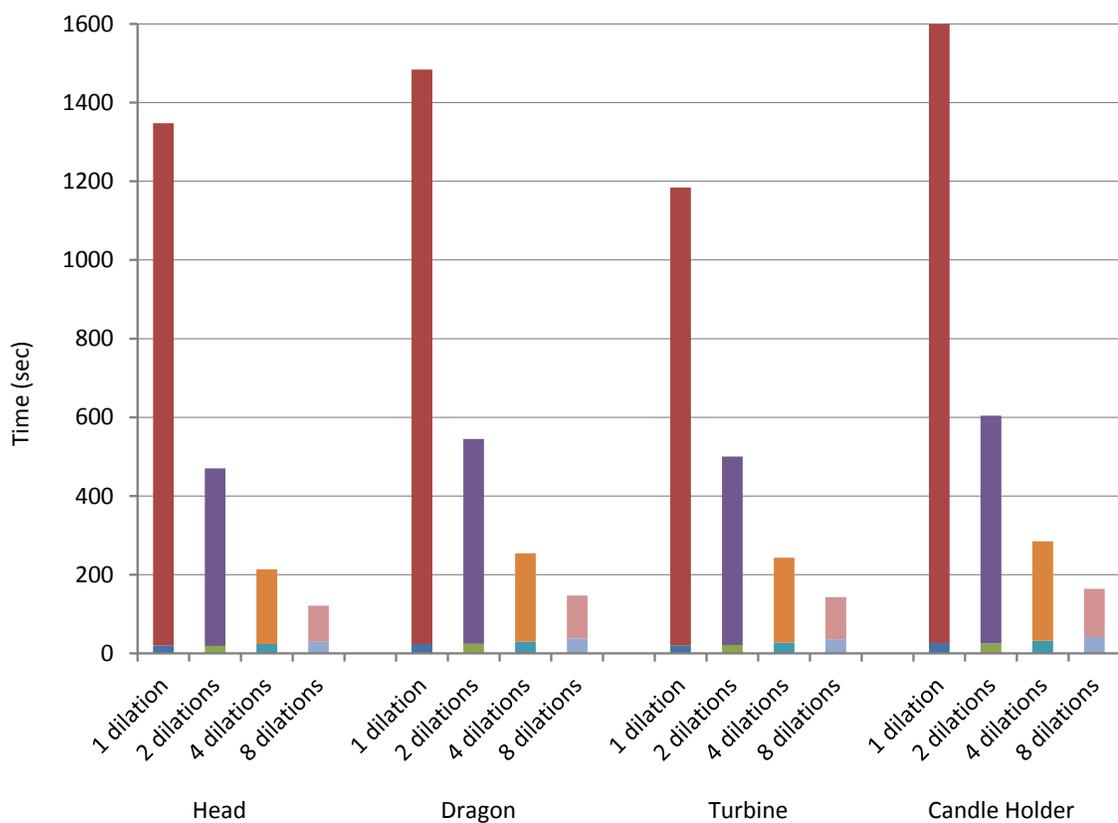
Figure 6.5: Successive offsetting error.

successive dilations each of 50 voxels results in over $3\times$ speedup without incurring noticeable accuracy in the computation.

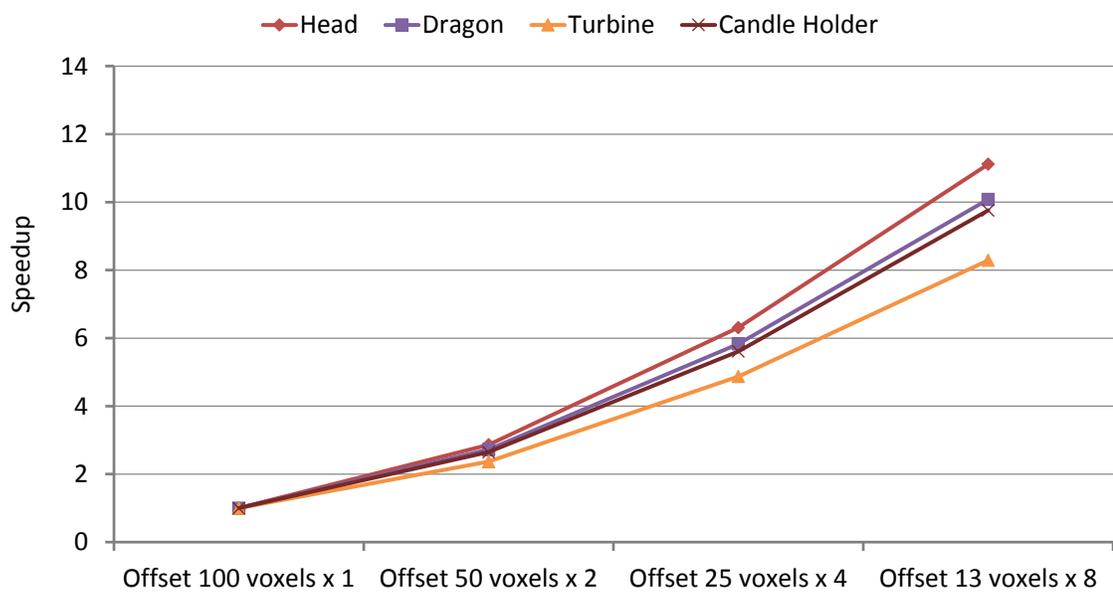
To investigate the opportunity of high-performance voxel offsetting through filter decomposition, we further experiment the successive dilations at 4096^3 resolutions. The results are appeared in Figure 6.6 that shows the breakdown of the two steps of the offsetting algorithm. For each stacked bar in these figures, the taller component represents the offsetting time with morphological filtering, while the bottom component indicates the preprocessing time to generate the skeletal dilated HDT. While at 4096^3 resolution with the HDTs configured with leaf of size $8 \times 8 \times 8$, as reported in Section 5.3, 100 voxel dilation takes between 20 and 27 minutes. Two successive dilations each of 50 voxel instead take a total of 7.8 – 10.1 minutes. Further, four successive dilations each of 25 voxel take 3.6 – 4.8 minutes in total, and thus make it suitable for interactive application setup. Applying even larger number of successive offsetting operations, for instance, eight dilations to replace the single dilation of 100 voxel collectively take between 2.0 to 2.7 minutes, as shown in Figure 6.6.

Overall, at the higher resolution a similar trend in dilation times of that at 2048^3

resolution is observed with successive offsetting. However, the rate of acceleration in offsetting at the 4096^3 resolution shows relatively lower gain using smaller filters. For instance, while at 2048^3 resolution we observed a $2.7 - 3.1\times$ speedup from replacing one dilation of 100 voxels with two consecutive dilations each of 50 voxels, it demonstrates $2.4 - 2.9\times$ speedup at 4096^3 resolution, as shown in Figure 6.6(b). The computation breakdowns in Figure 6.6(a) depicts that with increasing number of dilations, the preprocessing time (lower component in the bar chart) gradually becomes a significant part of the total offsetting computation.



(a) Dilation Times



(b) Relative Speedups

Figure 6.6: Successive offsetting performance at 4096^3 resolution.

CHAPTER 7

MULTI-GPU VOXEL OFFSETTING

7.1 Scale-Out Voxel Offsetting on Multiple GPUs

In the previous chapters, we have presented that voxel offsetting using hybrid dynamic trees can be greatly accelerated exploiting the massive parallelism on modern graphics hardware that can be further optimized through tuning the parameters of underlying HDT structure, and also by controlling the pertinent algorithmic parameters (such as, the size of the structuring element in the morphological operation). Naturally, the next step to unleashing even more computing capacity comes from the effective use of multiple GPUs, where the execution of the morphological filtering can be scaled out seamlessly. While multiple GPUs can lead to a significant speedup over a single GPU, it requires efficient memory management, synchronization-free workload scheduling, and perfect load balancing to ensure that a program takes full advantage of the massive computing fabric.

Just as scientific computing can be done on clusters composed of a large number of CPU nodes, in some cases problems can be decomposed and run in parallel on multiple GPUs within a single host machine, achieving correspondingly higher levels of performance. One of the drawbacks to the use of multi-core CPUs for scientific computing has been the limited amount of memory bandwidth available to each CPU socket, often severely limiting the performance of bandwidth-intensive scientific codes. In the recent years, this problem has been further exacerbated since the memory bandwidth available to each CPU socket has not scaled in proportion to the increasing number of cores. Since GPUs are packaged with on-board high performance memory, the usable memory bandwidth available for computational tasks scales with the number of GPUs deployed. This architectural flexibility allows single-system multi-GPU codes to scale much better than their multi-core CPU based counterparts.

Contrary to multi-core CPUs, one of the big benefits of GPU architectures is its scalability — it automatically scales the number of CUDA thread blocks to be processed concurrently onto the number of streaming multiprocessors the GPU contains. Hence, higher parallelism seems to be readily available with larger number of CUDA cores across multiple GPUs co-hosted on a single system. However, to achieve linear scalability leveraging more and more computing resources, not only the hardware but both the data structure and the algorithms need to be scalable as well.

For the case of voxel offsetting using the hybrid dynamic trees, superficially each leaf grid in the HDT seems to be independently processable, and hence the scale-out execution of the morphological filtering appears to be straightforward. However, in practice the execution scalability of an algorithm that deals with *sparse* data representation is nontrivial. Contrary to a regular 3D voxel grid structure where the topology of each voxel is implicitly encoded, in our sparse representation the topology of the leaf grids in the HDT—the path that tracks the nodes from the root to the leaf—needs to be accessible on all the GPUs. Hence, even though the leaf grids can be conceptually processed distributedly, the data that encode the HDT representation need to be replicated across all the GPUs, then get independently updated, and finally get merged to reflect the combined result. While there may have different ways to implement the morphological filtering algorithm for computing the offsets of voxel models, in this chapter we present a simple intuitive method to implement offsetting on multiple GPUs.

7.2 Multi-GPU Implementation of Morphological Filtering

To get maximum performance on the many-core graphics processors it is important to have an even balance of the workload so that all processing units contribute equally to the task at hand. This can be hard to achieve when the cost of a task is not known beforehand. With a voxel representation, the computation cost of the morphological filtering for individual voxel is about to be same. This research explores an intuitive approach to distribute the computation of voxel offsetting across multiple GPUs.

The conceptual design of voxel offsetting on multiple GPUs is presented in Figure 7.1. Three key components in the system are shown: *Load Distributor*, *Kernel Launcher*, and *Result Merger*. As we discussed in Section 4.3.2, the morphological filtering step takes in the list of leaf grids representing the skeleton (*i.e.*, outline without setting proper state values assigned to the voxels) of the dilated HDT, and produces the offset HDT where the state of each voxel is set to appropriate value (Algorithm 2).

The task of the load distributor is to divide the given list of leaf grids into a non-overlapping set of leaf grids, where each set is mapped onto unique GPU device. While different policies can be adopted that can be configured either *statically* or *dynamically*, in our study we contrast between two simple choices: 1) *equal* distribution, and 2) *weighted* distribution. While the equal distribution offloads same number of leaf grids across all the devices, weighted distribution targets optimizing the imbalance into the execution times caused by the disparity among architectural specifications of the GPUs. Many sophisticated schemes can be adopted to identify and assign weights to the respective hardware specifications to devise an intelligent weighted distribution policy. However, in the following experimental section we demonstrate that even a simple weighted distribution based multi-GPU voxel offsetting can achieve near-linear acceleration in practice. We consider solely the difference in the computing throughputs of the devices as an effective metric to derive a simple weighted load distribution policy.

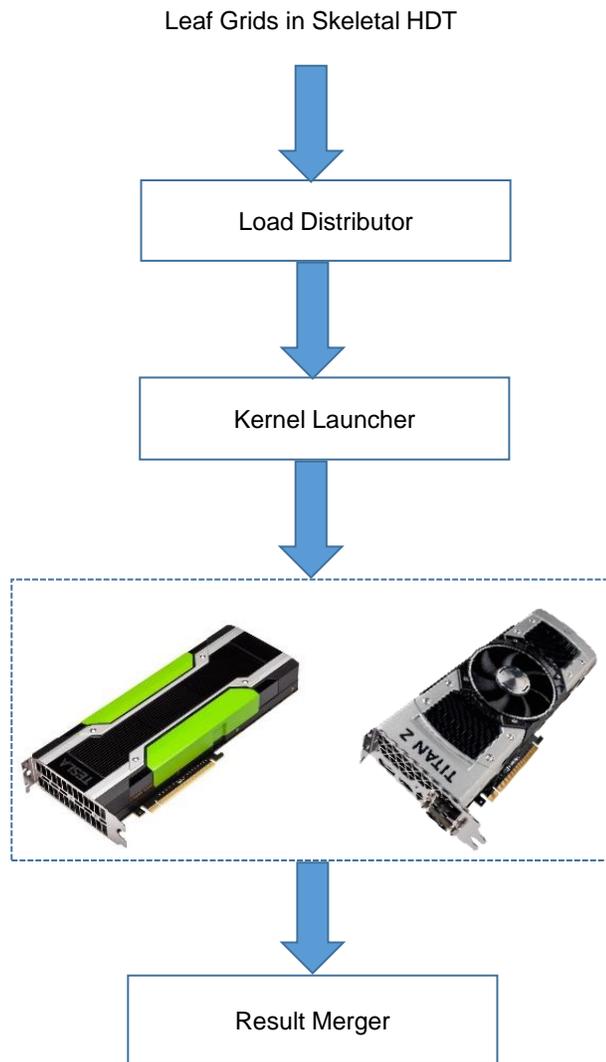


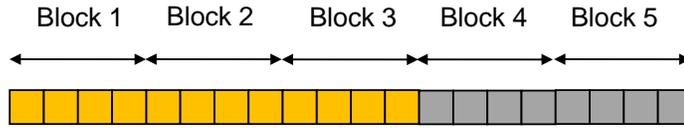
Figure 7.1: The schematic design of the components in multi-GPU offset implementation.

Once the load is properly divided across all the devices, the next component is the *kernel launcher* that controls how the CUDA kernels on individual GPU are initiated for execution. First, we need to elaborate how the morphological filtering kernel is launched on a single device. Although we listed in OFFSET procedure (Section 4.3.2) that all the leaf grids are offloaded to the GPU *at once*, in actual implementation we have to break down the list of leaf grids in small chunks¹. Thus, in practice even with a single GPU the leaf grids are offloaded into multiple steps, and the kernel is launched the same number of iterations.

Now, for the controlling of all the kernel launches our research explores two alternative choices. In the first scheme, all the CUDA kernel launches are controlled by a *single* CPU thread, which requires synchronization after each step of kernel execution. As our experimentations in the following section reveal that this synchronization overhead can impose a bottleneck on the achievable peak performance. To avoid this overhead, in the alternative scheme CUDA kernel launches on individual GPU are controlled by the corresponding CPU threads.

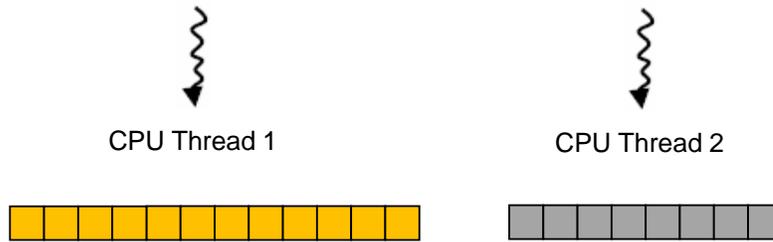
Finally, once the filtering is done on the devices, the results from all the GPUs are combined on a single device (called *master* GPU). In our example of Figure 7.1, thus the results from the second GPU (called *slave* GPU) are copied onto the GPU memory of the master. Figure 7.2 shows an example to demonstrate how the load distributor, the kernel launcher, and the result merger process the leaf grids of the input skeletal HDT on a dual GPU setup. The morphological filtering starts with a list of input leaf grids, as shown in Figure 7.2(a). For efficient dynamic memory allocation on GPU, in our HDT implementation the buffer on the leaf pool is allocated at a bigger chunk than the size of a leaf grid, as we discussed in Section 3.1.4. The example in Figure 7.2(a) depicts each *block* in the HDT leaf pool comprising four contiguous leaf grids.

¹Offloading all the leaf grids at once makes the device occupied for a very long time that often triggers GPU driver time out in our test setup

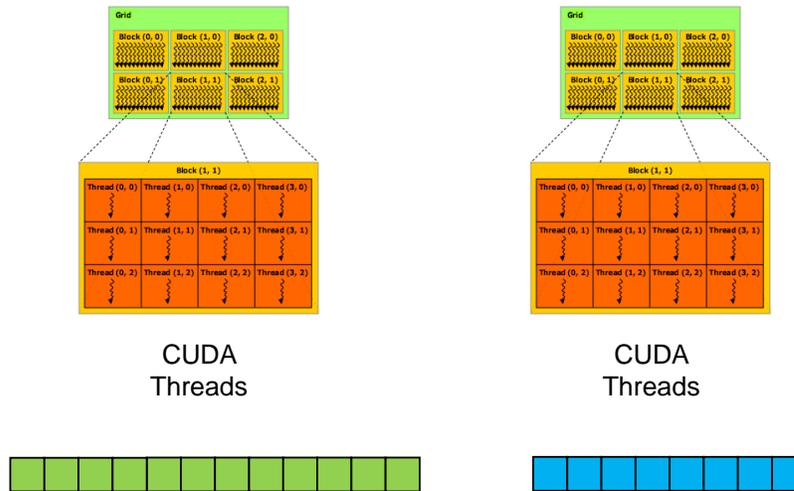


Leaf Grids in Skeletal HDT

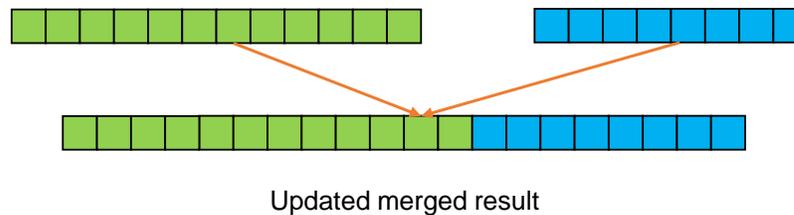
(a) Weighted Distribution of Leaf Grids (Blocks)



(b) Kernel Launched by Two CPU Threads



(c) Kernels Executed by the CUDA Threads



(d) Modified Leaf Grids are Merged

Figure 7.2: Illustration of how the leaf grids in skeletal HDT are processed on a dual GPU setup.

For the purpose of illustration, in this scenario we adopted a weighted load distribution policy that divides the input five leaf blocks into two sets: one of size 3 and the other of size 2. Each set of leaf blocks are launched by different CPU threads, as shown in Figure 7.2(b). Like the single GPU case, all the leaf blocks mapped to specific device are processed independently (Figure 7.2(c)). Finally, the updated results are merged as shown in Figure 7.2(d).

7.3 Evaluations on Multi-GPU Voxel Offsetting

While the presented methodology is applicable to any number of GPUs on a single node, our experimentations were evaluated on a platform with two GPUs: one GTX 780Ti and one GTX Titan. The comparative specifications of the peak throughput and memory bandwidth of the two graphics hardware are appeared in Table 7.1. By aggregating the computing throughputs of the two GPUs, we get the normalized throughput of the dual GPU platform as: $\frac{5040+4494}{5040} = 1.89$. Similarly, by aggregating the memory bandwidths of the two GPUs we get the normalized bandwidth of the dual GPU platform as: $\frac{336.0+288.4}{336.0} = 1.86$.

Table 7.1: Comparative throughput and memory bandwidth of NVIDIA GTX 780Ti and Titan.

	GTX 780Ti	GTX Titan	Normalized Dual GPU
Floating-point throughput (GFLOPS)	5040	4494	1.89
Memory bandwidth (GB/s)	336.0	288.4	1.86

7.3.1 Impact of Load Distribution Policy

Our first study on dual GPU offsetting investigates the impact of different load distributions on the overall computation time. Figure 7.3 shows the results for the two load distributions policies. With the equal distribution policy, equal number of the leaf grids of the skeletal HDT are offloaded onto each GPUs in the platform. This is the simplest policy that work assumed homogeneous set of graphics card in the cluster of GPUs, and hence distributing equal load to each of them appears to be optimal. However, as in our experimental setup the GPUs are of different computing capacity and memory efficiency, an equal distribution policy is sub-optimal. Thus, a workload distribution policy that takes the disparity of deployed GPU accelerators into consideration performs superior, as reflected in the comparative speedup results

in Figure 7.3.

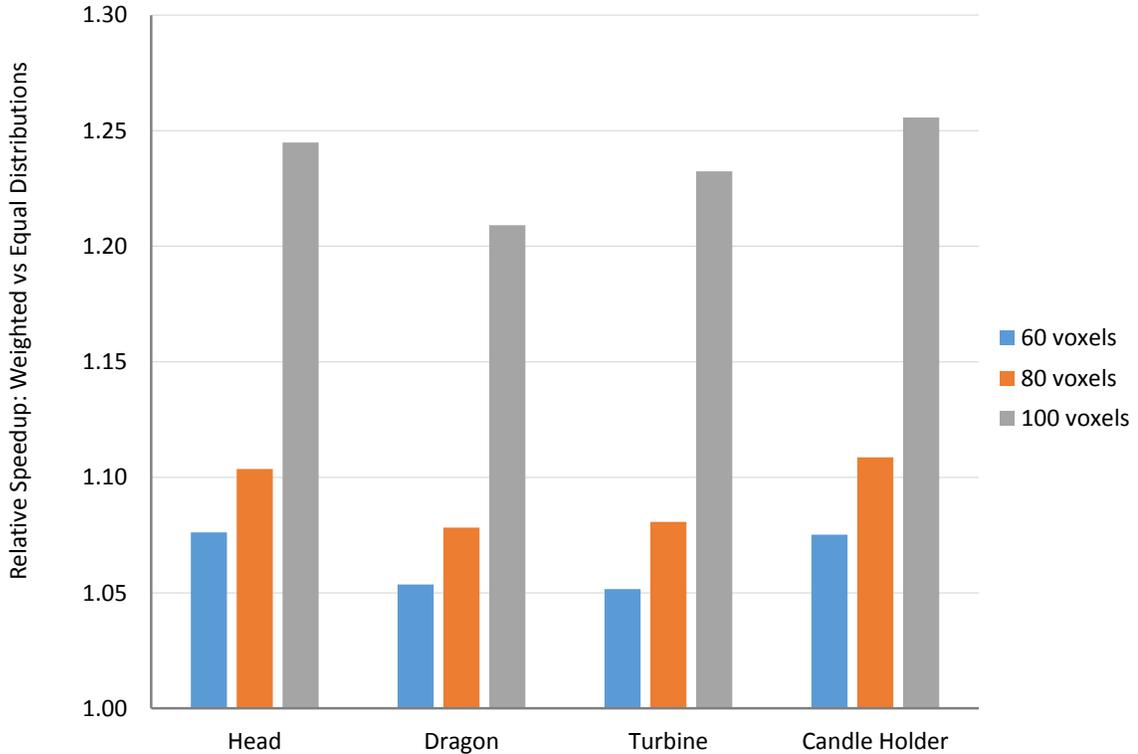


Figure 7.3: Impact of load distributions on dual GPU setup.

For the dilations of 60-100 voxels at 2048^3 resolution, our evaluations depict a relative speedup between $1.05\times$ and $1.26\times$. A non-obvious observation here is that with larger offset distances, the gap between the weighted and equal distribution increases. For instance, as Figure 7.3 shows with 60 voxel dilation weighted distribution achieves $1.05 - 1.08\times$ better than equal distribution, whereas with 100 voxel dilation the former executes $1.21 - 1.26\times$ faster than the latter. Interestingly this observation supports the results that we discussed in Section 4.4.4 to study the scalability of the morphological offsetting across different graphics cards, where we examined that the performance gap between the execution times on GTX 780Ti and GTX Titan scaled with the larger offset distances (cf. Figure 4.9). As we rationalized in Section 4.4.4

that with larger offset distance, both the size of the morphological structuring element and the number of the leaf grids in the skeletal HDT increase significantly (cf. Table 4.3). These result in an order of magnitude higher data to be loaded from the global memory on GPU, which was validated with our experimentation and analysis on the CUDA profiler outputs that demonstrated the memory throughput on GTX Titan significantly under perform for test scenaiors with larger offset distance.

7.3.2 Impact of Kernel Execution Alternatives

In the next study, we examine the impact of different approaches of CUDA kernel execution on the voxel offsetting computation. As we have two GPUs in the system, we have two choices here: 1) launching the CUDA kernels from a single CPU thread, and 2) launching the kernels for respective GPUs from two CPU threads. The comparative results for these two alternatives are appeared in Figure 7.4. In both of the setups, a weighted workload distribution policy is adopted to offload the leaf grids.

With the first option of kernel execution, after each step of the kernel launches the two devices get synchronized before the the next step can proceed. While in the weighted distribution, the overall load is distributed in proportion to the computing capacity of the devices to minimize the imbalance in the overall offsetting time, the kernel execution times for each individual step may yet differ by large margin. Hence, such step-by-step synchronization loses some performance as depicted in Figure 7.4.

With a multi-threaded implementation, where each CPU thread controls the specific kernel launches on the corresponding device, the synchronization overhead after each iteration can be eliminated. As Figure 7.4 shows this improved approach of voxel offsetting on dual GPU setup can improve the performance significantly. Among the evaluated test scenarios, for the case of 60 voxels dilation at 2048^3 resolution kernel execution by two independent CPU threads can achieve $1.05 - 1.11\times$ speedup. With larger offset distances, higher relative speedups are observed. For instance, with the case of 100 voxels dilation dual threaded implementation accelerates the offsetting

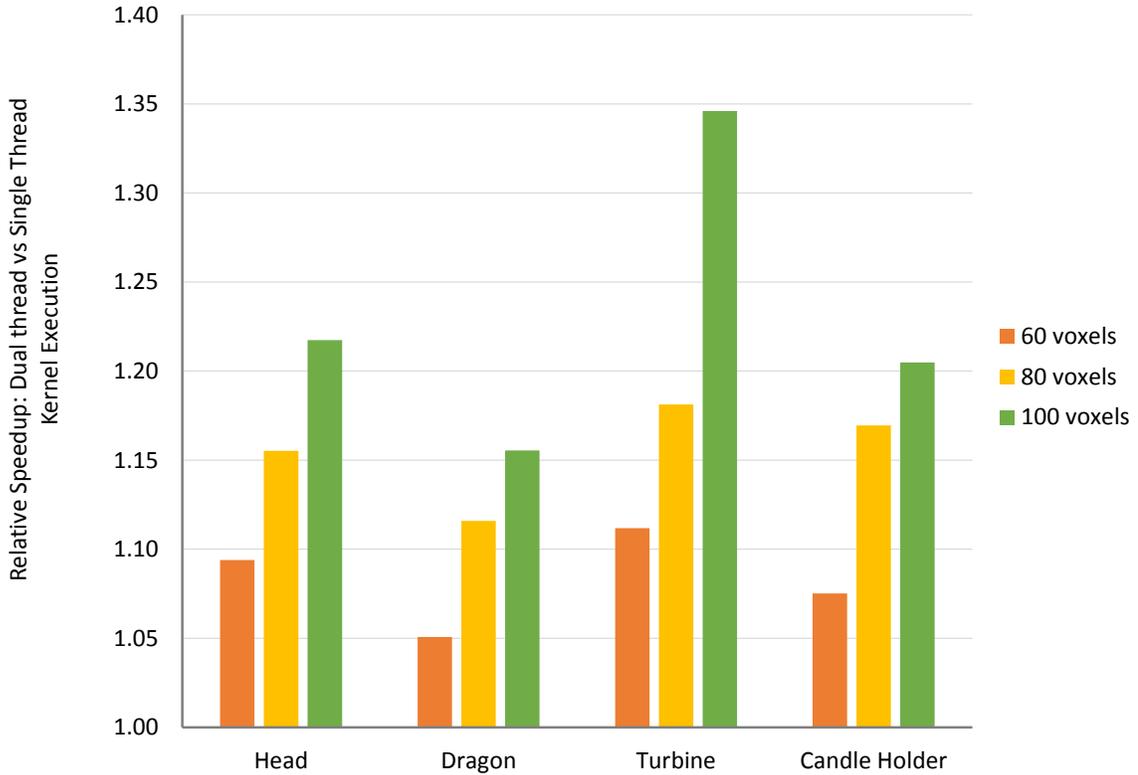


Figure 7.4: Comparisons between single threaded and dual threaded kernel executions on dual GPU setup.

computation by 16–35%. With larger offset distances, as we just discussed above, the impact of the lower memory bandwidth on the GTX Titan becomes more prominent, which supposedly incurs more overhead from the step-wise synchronization between the launched kernels on the dual GPU setup.

7.3.3 Performance Comparisons between Single and Dual GPU Setups

In this final study, we analyze the performance of voxel offsetting from the scale-out implementation of the morphological filtering algorithm on the dual GPU platform. The results appear in Tables 7.2 and 7.3 for offset distances of 60 voxels, 80 voxels and 100 voxels. The first column represents the dilation times on the default single GPU setup (*i.e.*, GTX 780Ti), and the second column represents the offsetting times on dual GPU platform with a weighted load distribution, where the CUDA kernel

execution is controlled by two independent CPU threads. And, the third “Speedup” column reflects the ratio of the respective execution times on single and dual GPU setups.

Table 7.2: Dilation times comparisons for the *Head* and *Dragon* models at 2048³ resolution.

	Head			Dragon		
	1 GPU	2 GPUs	Speedup	1 GPU	2 GPUs	Speedup
60 voxels	151.1	82.4	1.83×	172.0	93.1	1.85×
80 voxels	288.6	158.6	1.82×	325.4	181.1	1.80×
100 voxels	629.0	346.9	1.81×	674.1	383.5	1.76×

Table 7.3: Dilation times comparisons for the *Turbine* and *Candle Holder* models at 2048³ resolution.

	Turbine			Candle		
	1 GPU	2 GPUs	Speedup	1 GPU	2 GPUs	Speedup
60 voxels	119.1	65.1	1.83×	183.9	102.0	1.80×
80 voxels	206.9	114.8	1.80×	325.4	180.3	1.80×
100 voxels	407.5	228.2	1.79×	645.3	358.0	1.80×

As depicted in the speedup column in Tables 7.2 and 7.3, dual GPU setup achieves an acceleration in range between 1.76 and 1.85. These speedups should be contrasted with the normalized computing throughput of 1.89 and normalized memory bandwidths of 1.86 on the dual GPU platform. Thus, compared to the theoretical normalized computing throughput of 1.89, our multi-threaded dual GPU morphological offsetting achieves a scalability in between $\frac{1.76}{1.89} = 93.1\%$ and $\frac{1.85}{1.89} = 97.8\%$. Similar to the observation discussed in Section 7.3.1, with larger offset distances the relative speedup sometimes deteriorates noticeably, as in the case of Dragon model. As we pointed earlier, this is due to the same reasoning that with larger offset distance as

the size of the global memory transaction increases significantly, the lower memory bandwidth on the GTX Titan incurs noticeable performance overhead. To overcome this performance drop on dual GPU setup with larger offset distances, one solution could be to do in-depth analysis on the impact of the disparate memory systems with the size of the data to be processed on the GPUs for the voxel offsetting computation. This insight then can be exploited to devise even better distribution policy that not only considers the disparity of computation throughputs across multiple graphics cards, but also considers the overall system design towards the development of more intelligent workload distribution.

CHAPTER 8

CONCLUSIONS AND FUTURE WORKS

In the recent years, digital manufacturing has experienced the wave of rapid prototyping through the innovation and ubiquity in 3D printing technology. While such advancement liberates the constraints of shape selection in physical objects, 3D printing is yet to mature to match the precision, robustness and vast applicability offered by the classical subtractive manufacturing process. To simplify the toolpath planning in conventional multi-axis CNC machining, recent research has proposed adopting voxel-based geometric modeling. Inherently, such voxel representation is amenable for parallel acceleration on modern ubiquitous GPU hardware that has grown tremendously in the last few years to the level to offer supercomputing-scale computation capability of 100 TFLOPS (100×10^{12} floating-point operations per second) in a single computing node.

This dissertation has contributed to this nascent field by developing practical approaches of efficient voxel offsetting computation, which is an integral component of collision-free toolpath generation for advanced CAM systems. Below, we summarize our main results on high-resolution voxel offsetting using the hybrid dynamic trees in Section 8.1, and sketch future research works in Section 8.2.

8.1 Conclusions

While there can be many different approaches to represent voxel models, our research is based on a novel voxel data structure called hybrid dynamic tree (HDT). In the first part of this dissertation, in Chapter 3 we presented a parallel method to construct the HDT representation on GPU for a CAD input modeled in triangle mesh. At the highest modeling resolution of 8192^3 , we demonstrated the complete GPU-acceleration of the mesh to voxelization process achieving over *two orders of*

magnitude speedup for a practical set of CNC-manufacturable parts. As the memory footprint of HDT representation can be challenging to offer extreme resolutions, our research explored theoretical limit on the storage analysis for different active node branchings in the octree structure. Such tunability into the HDT organization helps devising the optimal parameter selections for compact HDT representation.

The next part of the thesis presented a mathematical morphology based offsetting algorithm using the hybrid voxel representation. For the CAD benchmarks, in Chapter 4 we showed that large-scale offsetting consumes 7 – 11 minutes for single dilation of 100 voxels at a resolution of 2048^3 . Our theoretical complexity analysis, which was further substantiated with experimental results demonstrated that at the higher resolution of 4096^3 a dilation of 100 voxels may take impractical time as high as 48 minutes for one of the four tested models. Thus, we emphasize the need for practical approaches to develop a *robust, efficient* and *tunable* voxel offsetting method to make large-scale volume dilation and erosion practical at our target resolution of $4096 \times 4096 \times 4096$.

Capability of fine-tuning of a data structure is crucial for understanding and thereby optimizing the developed computation-intensive algorithm that uses the HDT as the underlying voxel representation. Towards that end, the next part of the thesis has focused on exploring different techniques to achieve high-performance voxel offsetting. First, we studied the impact of the different HDT configurations on the voxel offsetting in Chapter 5. Our experimentations with tunable root size revealed up to 34 – 38% acceleration by enlarging the size of the root grid four-fold in the HDTs at 4096^3 resolution, while the evaluations with tunable node branching demonstrated an attainable speedup of 1.22 – 1.35 \times with twice larger branching in the HDT. Further, we showed that using a smaller leaf grid can be particularly useful to reduce the dilation times roughly by a factor of two. For the test scenario of 100 voxel dilations at 4096^3 resolution—the most computation intensive benchmark— tuning of the size

of the leaf grid to 8^3 in the HDTs reduces the offsetting time to 20 – 27 minutes from the original runtime of 38 – 48 minutes.

With the goal of enabling intensive voxel offsetting in an interactive scenario of CAM applications at a modeling resolution of 4096^3 , in Chapter 6 the thesis focused on leveraging the controllable size of the morphological structuring element. Further, to deeply analyze the impact of this tunability of algorithmic parameter on the geometric precision of the computed result, we implemented a GPU-accelerated error measurement technique. To devise a fast voxel-based offsetting algorithm, we analyzed the trade-offs between speed and accuracy through tunable size of the filter. Our evaluations revealed that by trading away a bit of precision over 6x speedup can be achieved, just as we studied with successive offsetting for the case of decomposing a dilation of 100 voxels into 4 successive dilations each of 25 voxels. This constrains the offsetting time within 5 minutes on a single GPU for a practical set of CAD models. While higher speedups are attainable by splitting a large structuring element into even larger number of successive offsetting operations, geometric precisions might be compromised quite high beyond acceptable level of manufacturing tolerance, as depicted with the case of 8 successive dilations.

Finally, to enable even faster voxel offsetting, in Chapter 7 we presented the principles of offloading the offset computation in the HDTs across a cluster of GPUs co-hosted on the same computing node. We analyzed the impact of different approaches for CUDA kernel execution controlled through either single or multiple independent CPU threads. We experimented with different load distribution policies—equal workload partitioning and weighted workload partitioning. While different advanced mechanisms could be adopted to dynamically tune the workload distribution across multiple GPUs, in our evaluations we observed even a simple multi-threaded weighted load distribution policy could achieve near-linear (above 90%) performance acceleration. With more and more GPUs integrated on a single computing node, such

exploration of algorithmic speedup through load-balanced implementation of offsetting across multiple GPUs emphasizes the high scalability of the HDT’s hybrid voxel representation.

8.2 Future Work

We outline possibilities of future research directions that can be built on the work and ideas presented in this dissertation.

Although the underlying hybrid dynamic tree structure is a compact volume representation, the limited memory capacity on the modern GPUs yet challenges voxel modeling at extreme resolutions (cf. Section 2.2). Although we confine the scope of this dissertation within efficient storage and processing of raw voxel data, many existing voxel compression approaches could be applied to reduce the memory footprint in the HDT. While the presented literature reviews on geometric redundancy compression seem the most promising route, the standard compression techniques, such as, Arithmetic Coding, Run Length Encoding (RLE), Huffman Coding could be leveraged to compactly represent the octree cells and the leaf grids in the HDT.

While advanced compression algorithms can significantly reduce the memory footprint, for high-resolution voxel offsetting in an interactive CAD/CAM application the overhead of repeated compressions and decompressions may impose significant challenge, particularly for accelerated processing on graphics hardware. Hence, an alternative approach to deal with voxel modeling beyond the capacity of the deployed accelerators, it is possible to construct and process the HDT represented volumes in a streaming fashion. Out-of-core streaming can reduce the in-memory footprint by storing the leaf grids *out-of-core* and only keeping the cell topology in memory. Then, the leaf grid data, *i.e.*, the states of the voxels are loaded on demand. For out-of-core voxel construction and processing it is important to efficiently manage the data transfer between the host and the accelerators to overcome the burden of slow PCIe

communication channel.

The focus of this dissertation is developing high-performance voxel offsetting on modern graphics hardware attached to single computing node. As vertical scaling of computation acceleration is limited, the natural route to extend our work is the distributed offsetting that exploits GPUs on a cluster of nodes. This can possibly support the construction of high-resolution HDTs over multiple nodes that effectively overcomes the limitation of GPU memory capacity. With multiple accelerator devices deployed across a cluster of nodes, development of load-balanced HDT construction and HDT offsetting should consider the overhead of inter-node communication. The research can further explore the scalability of the voxel processing algorithm, such as, the morphological offsetting studied in this dissertation.

REFERENCES

- [1] The Stanford 3D Scanning Repository: <http://graphics.stanford.edu/data/3Dscanrep/>.
- [2] “Stoya”, <http://www.thingiverse.com/thing:9255>.
- [3] Visual illustrations of morphological dilation and erosion: <http://maverick.inria.fr/Membres/Adrien.Bousseau/morphology/morphomath.pdf>.
- [4] CUDA C Programming Guide (Version 7.5).
- [5] The STL Format, http://www.fabbers.com/tech/STL_Format.
- [6] Minkowski Sum: <http://www3.cs.stonybrook.edu/algorithm/files/minkowski-sum.shtml>.
- [7] www.wikipedia.org/mathematical_morphology.
- [8] NVIDIA GeForce GTX 780 Ti: <https://www.techpowerup.com/gpubdb/2512/geforce-gtx-780-ti>,.
- [9] NVIDIA GeForce GTX Titan: <https://www.techpowerup.com/gpubdb/1996/geforce-gtx-titan>,.
- [10] NVIDIA Tesla C2050: <https://www.techpowerup.com/gpubdb/923/tesla-c2050>,.
- [11] AKENINE-MOLLER, T., “Source Code of Fast 3D Triangle-box Overlap Testing,” http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/tribox2.txt.
- [12] AKENINE-MOLLER, T., “Fast 3D Triangle-box Overlap Testing,” *Journal of Graphics Tools*, 2002.
- [13] ALCANTARA, D. A., SHARF, A., ABBASINEJAD, F., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., and AMENTA, N., “Real-time parallel hashing on the gpu,” *ACM Trans. Graph.*, vol. 28, pp. 154:1–154:9, Dec. 2009.
- [14] ANDONI, A., *Nearest Neighbor Search: the Old, the New, and the Impossible*. Ph.D. Thesis, Massachusetts Institute of Technology, 2009.
- [15] ANDONI, A. and INDYK, P., “Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions,” *Commun. ACM*, vol. 51, pp. 117–122, Jan. 2008.

- [16] AQRAWI, A. A., “Effects of Compression on Data Intensive Algorithms,” 2010.
- [17] ARONOV, B. and SHARIR, M., “The Union of Convex Polyhedra in Three Dimensions,” in *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*, pp. 518–527, 1993.
- [18] BASTOS, T. and FILHO, W. C., “GPU-accelerated Adaptively Sampled Distance Fields,” in *Shape Modeling International*, pp. 171–178, IEEE, 2008.
- [19] BEHLEY, J., STEINHAGE, V., and CREMERS, A. B., “Efficient Radius Neighbor Search in Three-dimensional Point Clouds,” in *ICRA*, pp. 3625–3630, IEEE, 2015.
- [20] BENTLEY, J. L., “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, pp. 509–517, Sept. 1975.
- [21] BREEN, D. E. and MAUCH, S., “Generating Shaded Offset Surfaces with Distance, Closest-Point and Color Volumes,” in *In Proceedings of the International Workshop on Volume Graphics*, pp. 307–320, 1999.
- [22] BREEN, D. E., MAUCH, S., and WHITAKER, R. T., “3D Scan Conversion of CSG Models into Distance Volumes,” in *Proceedings of the 1998 IEEE Symposium on Volume Visualization, VVS '98, (New York, NY, USA)*, pp. 7–14, ACM, 1998.
- [23] BRIDSON, R., *Computational aspects of dynamic surfaces*. Ph.D. Thesis, Stanford University, 2003.
- [24] BRUN, E., GUITTET, A., and GIBOU, F., “A Local Level-set Method Using a Hash Table Data Structure,” *J. Comput. Phys.*, vol. 231, pp. 2528–2536, Mar. 2012.
- [25] CALDERON, S. and BOUBEKEUR, T., “Point Morphology,” *ACM Trans. Graph.*, vol. 33, pp. 45:1–45:13, July 2014.
- [26] CHEN, J., BAUTEMBACH, D., and IZADI, S., “Scalable Real-time Volumetric Surface Reconstruction,” *ACM TOG*, 2013.
- [27] CHEN, Y., WANG, H., ROSEN, D. W., and ROSSIGNAC, J., “Filletting and rounding using a point-based method,” in *ASME 2005 International Design Engineering Technical Conference*.
- [28] CHEN, Y., WANG, H., ROSEN, D. W., and ROSSIGNAC, J., “A Point-Based Offsetting Method of Polygonal Meshes,” tech. rep., Georgia Institute of Technology, 2005.
- [29] COOLEY, J. and TUKEY, J., “An Algorithm for the Machine Calculation of Complex Fourier Series,” *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

- [30] CRASSIN, C., NEYRET, F., LEFEBVRE, S., and EISEMANN, E., “GigaVoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering,” in *Interactive 3D Graphics and Games*, 2009.
- [31] DADO, B., KOL, T. R., BAUSZAT, P., THIERY, J.-M., and EISEMANN, E., “Geometry and Attribute Compression for Voxel Scenes,” *Computer Graphics Forum (Proc. Eurographics)*, vol. 35, pp. 397–407, may 2016.
- [32] DOUBROVSKI, E., TSAI, E., DIKOVSKY, D., GERAEDTS, J., HERR, H., and OXMAN, N., “Voxel-based Fabrication through Material Property Mapping: a Design Method for Bitmap Printing,” *Computer-Aided Design*, vol. 60, pp. 3 – 13, 2015.
- [33] ELSEBERG, J., BORRMANN, D., and NÜCHTER, A., “One billion points in the cloud \dot{U} an octree for efficient processing of 3D laser scans,” *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 76, no. Complete, pp. 76–88, 2013.
- [34] EYIYUREKLI, M. and BREEN, D. E., “Data Structures for Interactive High Resolution Level-set Surface Editing,” in *Proceedings of Graphics Interface 2011, GI ’11*, pp. 95–102, 2011.
- [35] FIELD3D 2009. Version 1.2.0. <https://sites.google.com/site/field3d>.
- [36] FOLEY, J. D., VAN DAM, A., FEINER, S. K., and HUGHES, J. F., *Computer Graphics: Principles and Practice (2Nd Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.
- [37] FORSYTH, M., “Shelling and Offsetting Bodies,” in *ACM Symposium on Solid Modeling and Applications*, 1995.
- [38] FRISKEN, S. F. and PERRY, R. N., “Simple and efficient traversal methods for quadtrees and octrees,” *Journal of Graphics Tools*, vol. 7, 2002.
- [39] FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., and JONES, T. R., “Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics,” in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 249–254, 2000.
- [40] GHOSH, P. K., “A unified computational framework for Minkowski operations,” *Computers Graphics*, vol. 17, no. 4, pp. 357–378, 1993.
- [41] GONZALEZ, R. C. and WOODS, R. E., *Digital Image Processing*. Addison-Wesley Longman Publishing Co, 2001.
- [42] GURBUZ, A. Z. and ZEID, I., “Offsetting operations via closed ball approximation,” *Computer Aided Design*, 1995.
- [43] HARRIS, M., SENGUPTA, S., and OWENS, J. D., “Parallel Prefix Sum (Scan) with CUDA,” *GPU Gems 3*, pp. 851–876, August 2007.

- [44] HERHOLZ, P., MATUSIK, W., and ALEXA, M., “Approximating Free-form Geometry with Height Fields for Manufacturing,” *Comput. Graph. Forum*, vol. 34, no. 2, pp. 239–251, 2015.
- [45] HOFFMANN, C. M., *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Pub., 1989.
- [46] HOSSAIN, M. M., NATH, C., TUCKER, T. M., KURFESS, T. R., and VUDUC, R. W., “A Graphical Approach for Freeform Surface Offsetting with GPU Acceleration for Subtractive 3D Printing,” in *In Proceedings of the ASME Manufacturing Science and Engineering Conference, MSEC ’16*, 2016.
- [47] HOSSAIN, M. M., TUCKER, T. M., KURFESS, T. R., and VUDUC, R. W., “A GPU-parallel Construction of Volumetric Tree,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA3 ’15*, pp. 10:1–10:4, 2015.
- [48] HOSSAIN, M. M., TUCKER, T. M., KURFESS, T. R., and VUDUC, R. W., “Hybrid Dynamic Trees for Extreme-Resolution 3D Sparse Data Modeling,” in *In Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS ’16*, 2016.
- [49] HOU, Q., SUN, X., ZHOU, K., LAUTERBACH, C., and MANOCHA, D., “Memory-Scalable GPU Spatial Hierarchy Construction,” *IEEE T. on Visualization and Computer Graphics*, 2011.
- [50] HUANG, X., RODRIGUES, C., JONES, S., BUCK, I., and HWU, W.-M., “XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines,” in *IEEE 10th International Conference on Computer and Information Technology (CIT)*, pp. 1134–1139, June 2010.
- [51] INDYK, P. and MOTWANI, R., “Approximate nearest neighbors: Towards removing the curse of dimensionality,” in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC ’98*, (New York, NY, USA), pp. 604–613, ACM, 1998.
- [52] INUI, M. and OHTA, A., “Using a gpu to accelerate die and mold fabrication,” *IEEE Comput. Graph. Appl.*, vol. 27, pp. 82–88, Jan. 2007.
- [53] JACKINS, C. and TANIMOTO, S., “Oct-trees and their use in representing three-dimensional objects,” *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.
- [54] JONES, M. W., BAERENTZEN, J. A., and SRAMEK, M., “3D Distance Fields: A Survey of Techniques and Applications,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 581–599, July 2006.
- [55] KALOJANOV, J., BILLETER, M., and SLUSALLEK, P., “Two-Level Grids for Ray Tracing on GPUs,” *Computer Graphics Forum*, 2011.

- [56] KALOJANOV, J. and SLUSALLEK, P., “A Parallel Algorithm for Construction of Uniform Grids,” in *High Performance Graphics, 2009*.
- [57] KÄMPE, V., SINTORN, E., and ASSARSSON, U., “High Resolution Sparse Voxel DAGs,” *ACM Trans. Graph.*, vol. 32, pp. 101:1–101:13, July 2013.
- [58] KATAJAINEN, J. and MÄKINEN, E., “Tree compression and optimization with applications,” *Int. J. Found. Comput. Sci.*, vol. 1, no. 4, pp. 425–448, 1990.
- [59] KLINGENSMITH, M., DRYANOVSKI, I., SRINIVASA, S., and XIAO, J., “Chisel: Real Time Large Scale 3D Reconstruction Onboard a Mobile Device using Spatially Hashed Signed Distance Fields,” in *Robotics: Science and Systems*, 2015.
- [60] KNOBRYTSKYI, D., *Automated CNC Toolpath Planning and Machining Simulation on Highly Parallel Computing Architectures*. Ph.D. Thesis, Clemson University, 2013.
- [61] KOSHELEVA, O., SERGIO, C., GIBSON, G., and KOSHELEV, M., “Fast implementations of morphological operations using fast fourier transform,”
- [62] LAINE, S., “Efficient Sparse Voxel Octrees - Analysis, Extensions, and Implementation,” Tech. Rep. NVR-2010-001, NVIDIA Technical Report, 2010.
- [63] LAINE, S. and KARRAS, T., “Efficient Sparse Voxel Octrees,” in *Interactive 3D Graphics and Games, 2010*.
- [64] LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., and MANOCHA, D., “Fast BVH Construction on GPUs,” in *EUROGRAPHICS '09*, 2009.
- [65] LEFEBVRE, S. and HOPPE, H., “Perfect spatial hashing,” *ACM Trans. Graph.*, vol. 25, pp. 579–588, July 2006.
- [66] LEFEBVRE, S. and HOPPE, H., “Compressed Random-access Trees for Spatially Coherent Data,” in *Proceedings of the 18th Eurographics Conference on Rendering Techniques, EGSR'07*, (Aire-la-Ville, Switzerland, Switzerland), pp. 339–349, Eurographics Association, 2007.
- [67] LEFEBVRE, S., HORNUS, S., and NEYRET, F., “Texture Sprites: Texture Elements Splatted on Surfaces,” in *I3D 2005*.
- [68] LI, W. and MCMAINS, S., “A GPU-based Voxelization Approach to 3D Minkowski Sum Computation,” in *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling, SPM '10*, (New York, NY, USA), pp. 31–40, ACM, 2010.
- [69] LI, W. and MCMAINS, S., “Voxelized Minkowski Sum Computation on the GPU with Robust Culling,” *Computer Aided Design*, 2011.

- [70] LI, W. and MCMAINS, S., “A sweep and translate algorithm for computing voxelized 3d minkowski sums on the gpu,” *Computer-Aided Design*, vol. 46, pp. 90–100, 2014.
- [71] LIEN, J.-M., “Covering Minkowski sum boundary using points with applications,” *Computer Aided Geometric Design*, vol. 25, no. 8, pp. 652–666, 2008.
- [72] LIU, S. and WANG, C. C. L., “Fast Intersection-Free Offset Surface Generation From Freeform Models With Triangular Meshes,” *IEEE T. Automation Sci. and Engr.*, 2011.
- [73] LORENSEN, W. E. and CLINE, H. E., “Marching Cubes: A High Resolution 3D Surface Construction Algorithm,” in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’87, (New York, NY, USA), pp. 163–169, ACM, 1987.
- [74] LOSASSO, F., GIBOU, F., and FEDKIW, R., “Simulating Water and Smoke with an Octree Data Structure,” *ACM Trans. Graph.*, vol. 23, pp. 457–462, Aug. 2004.
- [75] MAEKAWA, T., “An overview of offset curves and surfaces,” *Computer-Aided Design*, 1999.
- [76] MEAGHER, D., “Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer,” Tech. Rep. IPL-TR-80-111, Rensselaer Polytechnic Institute, October 1980.
- [77] MENON, J., MARISA, R. J., and ZAGAJAC, J., “More Powerful Solid Modeling Through Ray Representations,” *IEEE Comput. Graph. Appl.*, vol. 14, pp. 22–35, May 1994.
- [78] MENON, J. and VOELCKER, H., “Mathematical foundations I: set theoretic properties of ray representations and Minkowski operations on solids,” tech. rep., Cornell University, 1991.
- [79] MILLER, A., JAIN, V., and MUNDY, J. L., “Real-time Rendering and Dynamic Updating of 3-d Volumetric Data,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pp. 8:1–8:8, 2011.
- [80] MUSETH, K., “VDB: High-resolution Sparse Volumes with Dynamic Topology,” *ACM Trans. Graph.*, vol. 32, pp. 27:1–27:22, July 2013.
- [81] NIESSNER, M., ZOLLHÖFER, M., IZADI, S., and STAMMINGER, M., “Real-time 3D Reconstruction at Scale Using Voxel Hashing,” *ACM Trans. Graph.*, vol. 32, pp. 169:1–169:11, Nov. 2013.
- [82] OPPENHEIM, A. V., SCHAFER, R. W., and BUCK, J. R., *Discrete-time Signal Processing (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

- [83] ORTIZ, C. G., “Fast and accurate computation of the Euclidean distance transform in medical imaging analysis software,,” 2007.
- [84] PARKER, E. and UDESHI, T., “Exploiting self-similarity in geometry for voxel based solid modeling,,” in *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, (New York, NY, USA), pp. 157–166, ACM, 2003.
- [85] PAVIC, D. and KOBELT, L., “High-Resolution Volumetric Computation of Offset Surfaces with Feature Preservation,,” *Computer Graphics Forum*, vol. 27, pp. 165–174, 2008.
- [86] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., and AMARASINGHE, S., “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines,,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 519–530, ACM, 2013.
- [87] RAGAN-KELLEY, J. M., *Decoupling algorithms from the organization of computation for high performance image processing*. Ph.D. Thesis, Massachusetts Institute of Technology, 2014.
- [88] ROSSIGNAC, J. R. and REQUICHA, A. A. G., “Offsetting Operations in Solid Modelling,,” *Computer Aided Geometric Design*, 1986.
- [89] SAMET, H., *The Design and Analysis of Spatial Data Structures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.
- [90] SAMET, H., *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.
- [91] SCHNABEL, R. and KLEIN, R., “Octree-based Point-cloud Compression,,” in *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'06, (Aire-la-Ville, Switzerland, Switzerland), pp. 111–121, Eurographics Association, 2006.
- [92] SERRA, J., *Image Analysis and Mathematical Morphology*. Academic Press, Inc., 1983.
- [93] SOILLE, P., *Morphological Image Analysis: Principles and Applications*. Springer, 2004.
- [94] SÍR, Z., GRAVESEN, J., and JÜTTLER, B., “Computing Convolutions and Minkowski Sums via Support Functions,,” 2005.

- [95] STEINBERGER, M., KENZEL, M., KAINZ, B., and SCHMALSTIEG, D., “ScatterAlloc: Massively parallel dynamic memory allocation for the GPU,” in *Innovative Parallel Computing (InPar)*, pp. 1–10, May 2012.
- [96] SUD, A., OTADUY, M. A., and MANOCHA, D., “DiFi: Fast 3D Distance Field Computation Using Graphics Hardware,” *Computer Graphics Forum*, vol. 23, no. 3, pp. 557–566, 2004.
- [97] TARBUTTON, J., KURFESS, T. R., TUCKER, T., and KONOBRYTSKYI, D., “Gouge-free voxel-based machining for parallel processors,” *The International Journal of Advanced Manufacturing Technology*, vol. 69, no. 9, pp. 1941–1953, 2013.
- [98] TARBUTTON, J. A., KURFESS, T. R., and TUCKER, T. M., “Graphics Based Path Planning for Multi-Axis Machine Tools,” *Computer-Aided Design and Applications*, vol. 7, no. 6, pp. 835–845, 2010.
- [99] ULUSOY, A. O., BIRIS, O., and MUNDY, J. L., “Dynamic Probabilistic Volumetric Models,” in *Proceedings of the 2013 IEEE International Conference on Computer Vision, ICCV ’13*, (Washington, DC, USA), pp. 505–512, IEEE Computer Society, 2013.
- [100] VAN HOOK, T., “Real-time Shaded NC Milling Display,” in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH ’86*, (New York, NY, USA), pp. 15–20, ACM, 1986.
- [101] VILLANUEVA, A. J., MARTON, F., and GOBBETTI, E., “Ssvdags: Symmetry-aware sparse voxel dags,” in *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D ’16*, (New York, NY, USA), pp. 7–14, ACM, 2016.
- [102] VUDUC, R. W., *Automatic performance tuning of sparse matrix kernels*. Ph.D. Thesis, UNIVERSITY OF CALIFORNIA, BERKELEY, 2003.
- [103] WEBBER, R. E. and DILLEN COURT, M. B., “Compressing quadtrees via common subtree merging,” *Pattern Recognition Letters*, vol. 9, no. 3, pp. 193–200, 1989.
- [104] WILLIAMS, B., *Moxel DAGs: Connecting material information to high resolution sparse voxel DAGs*. M.Sc. Thesis, California Polytechnic State University, 2015.
- [105] YIN, K., LIU, Y., and WU, E., “Fast Computing Adaptively Sampled Distance Field on GPU,” in *Pacific Graphics Short Papers*, 2011.
- [106] ZHOU, K., HOU, Q., WANG, R., and GUO, B., “Real-time KD-tree Construction on Graphics Hardware,” *ACM TOG*, 2008.

- [107] ZHUANG, X. and HARALICK, R. M., “Morphological Structuring Element Decomposition,” *Computer Vision, Graphics and Image Processing*, 1986.