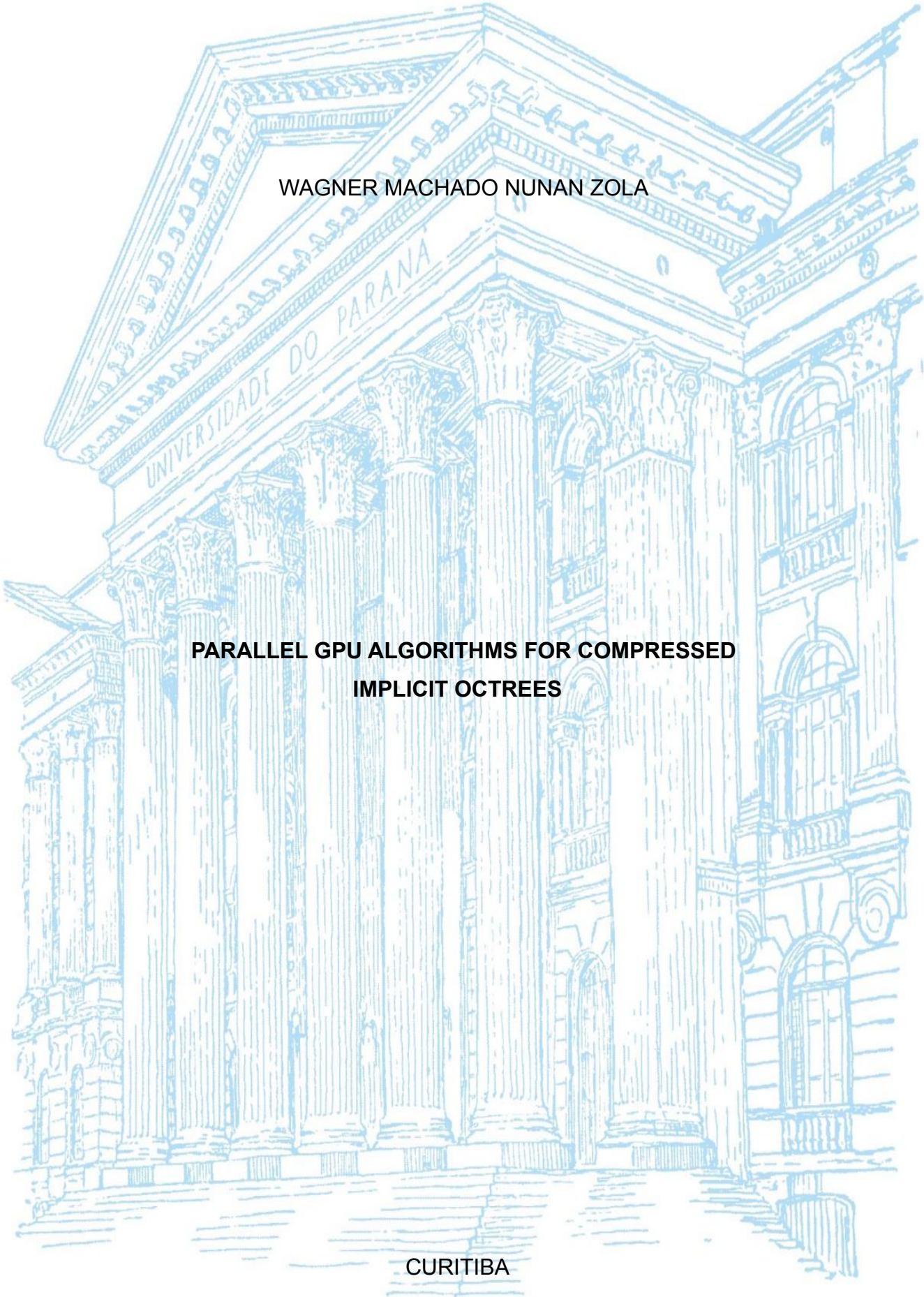
UNIVERSIDADE FEDERAL DO PARANÁ



WAGNER MACHADO NUNAN ZOLA

PARALLEL GPU ALGORITHMS FOR COMPRESSED IMPLICIT OCTREES

CURITIBA

2015

WAGNER MACHADO NUNAN ZOLA

PARALLEL GPU ALGORITHMS FOR COMPRESSED IMPLICIT OCTREES

Tese apresentada ao Programa de Pós-graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Orientador:
Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

Setembro/2015

Z86p Zola, Wagner Machado Nunan
Parallel GPU algorithms for compressed implicit octrees / Wagner Machado Nunan Zola. – Curitiba, 2015.
103 f. : il. color. ; 30 cm.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática , 2015.

Orientador: Luis Carlos Erpen de Bona .
Bibliografia: p. 97-101.

1. Algorítmos. 2. Algorítmos computacionais. 3. Algoritmos paralelos. 4. Problemas de n-corpos. 5. Barnes-Hut. I. Universidade Federal do Paraná. II.Bona, Luis Carlos Erpen de. III. Título.

CDD: 518.1

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Comissão Examinadora da defesa do(a) aluno(a) de Doutorado em Ciência da Computação, Wagner Machado Nunan Zola, avaliamos a de tese de doutorado intitulado "PARALLEL GPU ALGORITHMS FOR COMPRESSED IMPLICIT OCTREES", cuja defesa pública, foi realizada no dia 10 de setembro de 2015. Após avaliação, decidimos pela:

(X) **Aprovação** do(a) candidato(a). ( ) **Reprovação** do(a) candidato(a).

Curitiba, 10 de setembro de 2015.

Prof. Dr. Luis Carlos Erpen de Bona
PPGINF/UFPR – Orientador

Prof. Dr.Antonio Roberto Mury
LNCC – Membro Externo

Prof. Dr. David Menotti
UFOP – Membro Externo

Prof. Dr. Bruno Schulze
PPGINF – Membro Interno

Prof. Dr. Fabiano Silva
PPGINF – Membro Interno

Prof. Dr. Daniel Weingaertner
PPGINF – Membro Interno

À Carmem, Pedro, Eurica e Levy

Agradecimentos

Gostaria de manifestar meus agradecimentos ao Prof. Luis Carlos Erpen de Bona, primeiramente por ter me convencido a avançar nessa etapa acadêmica, bem como pela dedicada e eficiente orientação, pelas valiosas contribuições, sempre estando pronto a discutir os assuntos do meu trabalho.

Agradeço também aos membros da minha banca examinadora, professores Antonio Roberto Mury, Bruno Richard Schulze, Daniel Weingaerttner, David Menotti Gomes e Fabiano Silva, pela revisão da minha tese e pelas inúmeras sugestões feitas ao trabalho.

Muitos colegas e amigos do Departamento de Informática da UFPR me incentivaram a prosseguir com esses estudos. Sem correr o risco de esquecer alguém, gostaria de agradecer a todos.

Gostaria de lembrar e agradecer meus amigos e professores da UFMG, por minha formação acadêmica, em especial ao Prof. José Monteiro da Mata pela experiência transmitida nos saudosos anos de mestrado e pela amizade.

Agradeço a minha família, aos meus pais Levy e Eurica, e a todos os meus irmãos pelo amor e incentivo em todas as etapas. Agradeço especialmente a minha esposa Carmem e ao meu filho Pedro pelo amor e compreensão que foram indispensáveis nessa jornada.

Resumo

O algoritmo Barnes-Hut é um método aproximado amplamente usado para na simulação gravitacional de N-Corpos, que envolve a construção e caminhamento de árvores esparsas a cada passo de simulação e assim reduzindo a complexidade computacional e possibilitando a solução de problemas práticos de grande escala. A natureza irregular desse código de caminhamento em árvore apresenta desafios interessantes na sua computação em sistemas paralelos. Desafios adicionais ocorrem nesse tipo de padrão de computação paralela quando se deseja utilizar de maneira eficaz a capacidade computacional de arquiteturas de GPUs (processadores gráficos multicore de propósito geral).

Octrees são estruturas de dados que representam de maneira eficiente as informações de dados espaciais em várias áreas tais como computação científica, computação gráfica, processamento de imagens, dentre outras.

Nosso enfoque nesse trabalho é de tratar explicitamente os padrões dinâmicos irregulares de acesso a dados em memória com o remapeamento de dados e transformações de layouts, dependendo das estruturas acessadas. Também é feito o controle explicito, por programa, de fluxos divergentes de execuções em *threads*.

Apresentamos uma nova estrutura de dados compacta para layouts de octrees esparsas, bem como algoritmos paralelos para GPUs, tanto para transformações de layouts como para caminhamento paralelo usando a técnica de simulação de "warps"-largos (SWW, Simulated Wide-Warps).

Os benefícios de nossas técnicas ocorrem devido à transposição do algoritmo de caminnhamento na árvore para execução em padrões mais regulares, possibilitando uma melhor adaptação ao modelo GPU paralelo. A estrutura de dados permite explorar localidades de acessos à memoria durante os percursos, ao mesmo tempo conservando espaço em memória cache ou em memoria compartilhada (*scratchpad*). Desta forma a memória rápida intra-core pode ser dedicada a acelerar caminhamentos. Controle divergência de fluxos também é delimitado de maneira algorítmica, impondo uma execução uniforme na maior parte dos segmentos de execução. Nossos experimentos mostram melhoria de desempenho significativa em relação às soluções em GPU mais conhecidas para este algoritmo.

Desenvolvemos um novo algoritmo paralelo eficiente que gera diretamente de uma só vez as *octrees implícitas comprimidas*, como um método massivamente paralelo. Este método traz uma nova visão para tratar de forma eficiente com a natureza irregular também presente na construção de octrees esparsas.

O algoritmo proposto de geração massivamente paralela de octrees esparsas tem aplicação imediata em nossa implementação GPU paralela da simulação Barnes-Hut e em

outros métodos de N-corpos. As técnicas e algoritmos propostos nesta tese também poderão ser aplicadas em outros contextos.

Palavras-chave: Algoritmo Massivamente Paralelo para Geração de Octrees; Octrees esparsas; Octree implícita; Probleamas de N-Corpos; Barnes-Hut; GPGPU; Warps-Largos Simulados em Software; CUDA; Algoritmo Paralelo irregular; Algoritmos paralelos; Manycore Computing; Acelerador de Computação;

Abstract

The Barnes-Hut algorithm is a widely used approximation method for the N-Body simulation problem, which involves the construction and traversal of sparse trees at each simulation step and thus reducing the complexity to solve large/practical problems. The irregular nature of this tree walking code presents interesting challenges for its computation on parallel systems. Additional problems arise in effectively exploiting the processing capacity of GPU architectures.

Octrees are data structures that efficiently represent spatial data in many fields such as scientific computing, computer graphics and image processing, among others.

In this work we explicitly deal with dynamic irregular patterns in data accesses with data remapping and data transformation, depending on the data structures being accessed, and by controlling the execution flow divergence of threads.

We present a new compact data-structure for sparse octree layouts, and also GPU parallel algorithms for tree transformation and parallel walking using software *Simulated Wide-Warps* (SWW). Benefits of our techniques are in transposing the tree algorithm to execute regular patterns to match the GPU parallel model. The data structure allows exploring localities during traversals, at the same time conserving space in caches or scratchpad memory. This way fast intra-core memory can be dedicated to speed up traversals. Control flow divergence is also algorithmically constrained, enforcing a mostly uniform execution of threads. Our experiments show significant performance improvement over the best known GPU solutions to this algorithm.

We have developed a novel efficient parallel algorithm that directly generates entire *compressed implicit octrees* at once, as a massively parallel method. This method brings new insight on how to efficiently deal with the irregular nature of algorithms for constructing sparse octrees. The proposed algorithm has immediate application to our GPU parallel Barnes-Hut implementation and other N-Body methods.

We envision that the techniques and algorithms proposed in this dissertation can also be applied in other contexts.

Keywords: Massively Parallel Octree Generation Algorithm; Sparse Octrees; Implicit Octree; N-Body; Barnes-Hut; GPGPU; Software Simulated Wide-Warp; CUDA; Irregular Parallel Algorithm; Parallel algorithms; Manycore Computing; Accelerator Computing;

Contents

List of Figures

List of Tables

List of Algorithms

Chapter

1

Introduction

Parallel processing has attained major advancements and endorsement with the arise of GPU computing. Although there is an inherent difficulty in shifting from sequential to parallel programming, obtaining reasonable speedups is the common case when using GPUs in the parallelization of algorithms. The CUDA (NVIDIA Corporation, 2013) environment allows diverse styles of parallelism to coexist in GPU programming: multithreading, MIMD (Multiple Instruction Multiple Data), SIMD (Single Instruction Multiple Data), and instruction-level parallelism (Hennessy and Patterson, 2011). We believe that the immediate speedup witnessed in GPU parallel programs, even in initial versions of parallel code is, in great measure, due to the relatively low latency of operations in these architectures when compared to multicore multithreaded parallel computing. GPU multiprocessors contains thousands of cores, in which thousands of active threads are put to work in cooperation to solve large problems. On the low side, when aiming to obtain top performance, traditional programming patterns in sequential programming do not immediately follow in parallel GPU computing environments, mainly due to the difficulty in selecting the appropriate set of techniques to harness the huge available computational throughput on these architectures.

Due to the SIMD (Single Instruction Multiple Data) nature of GPUs, massively parallel programs (with little data dependency and no use of synchronization primitives)

are naturally better candidates to achieve near peak performance on these architectures. Groups of "contiguous" threads denominated warps execute their instructions in lock step using SIMD instructions. GPU processors allow for the computation flow to diverge in a warp, i.e. threads in a warp follow different paths depending on values on thread local variables. In this case, some threads in the warp are allowed do proceed execution in some path, while others are automatically disabled. All divergent paths are exercised with only a fraction of cores in the warp in effective use, until all threads reconverge execution at some point, retaking lock step execution. This mechanism, denominated SIMT (Single Instruction Multiple Thread), while allowing the simulation of MIMD (Multiple Instruction Multiple Data) execution, produces sub-utilization of the GPU cores, precluding peek performance. Different warps need not follow convergent execution paths, in such case, divergence of execution of different warps incur no performance penalty.

Irregular algorithms present dynamic irregular patterns in data accesses (Jenkins et al., 2011), which depend on the data structures being accessed. Data dependence can cause control flow divergence of threads (Coutinho et al., 2011) which particularly affects execution performance on GPU architectures in the case of intra-warp divergence.

We have investigated algorithms that generate and traverse sparse tree data structures in GPUs. In such problems, the dynamic irregular data access patterns are common, and limit the effective use of the available computational power. We have particularly studied algorithms of sparse octrees in GPUs. Octrees form a class of tree data structure used extensively to represent spatial data (e.g. 3-dimentional data).

We have selected the Barnes-Hut simulation problem as an important test case for our octree algorithms, as it generates octrees at each simulation step. Efficient traversal of the octrees are also of crucial importance on these simulations. Barnes and Hut (Barnes and Hut, 1986) proposed an $\mathcal{O}(NlogN)$ N-Body approximate force-calculation algorithm based on the traversal of octrees. Groups of particles are organised in subtrees, with a parent node summarizing particle cluster properties. Octrees space subdivision data structures reduce the computational complexity of the force-calculation algorithm.

Besides its importance as a computational science tool (Groen et al., 2008), the Barnes-Hut method (Barnes and Hut, 1986) is traditionally investigated in parallel benchmarks. A modern view of parallel computing research (Asanovic et al., 2006) includes N-Body/Barnes-Hut methods among thirteen classes of algorithmic methods (so called, *dwarfs*) exhibiting common patterns of computation and communication of significance in the study of parallel programming models and architectures. Researchers have extensively studied the applicability of exact, approximated and combined N-body methods in different scales (Ishiyama et al., 2012), and using different architectures (Tanikawa et al., 2013)(Jetley et al., 2010).

In our work, efficient generation of entire trees are possible in applications if done at once. We expect that incremental methods or dynamic data structures incur in considerable overhead. To this extent, operations are sometimes aggregated to be done in larger batches. In such case, amortized analysis or experimental evaluation seems more appropriate to assess performance.

The main contributions of this work are:

- A new implicit data layout for octrees and techniques to minimize uncoalesced accesses in Barnes-Hut simulations.

- A synchronization free parallel GPU algorithm to convert any pointer based sparse tree to its equivalent implicit layout. We also show that the extra time taken in the layout transformation is amortized in the forces-calculation phase (equivalent to do N traversals of the tree, in parallel), as our implicit layout allows tree traversals to be executed without using a stack data structure.

- Application of our *software simulated wide warp* (SWW) technique (Nunan Zola et al., 2014) for accelerating the tree traversal/forces calculations phase. Execution of simulated wide warps of threads can be applied to some blocks of code and mixed with regular physical warps in GPU kernels. To the best of our knowledge this is the first work to demonstrate a simple and effective method to simulate very large warps in software.

- Use of AoS (*Array-of-Structs*) layout for the octrees implicit data layout with SWW to constrain SIMT threads divergence. This approach differs from common GPU practice. Recent work (Lange and Fortin, 2014) on N-Body simulation proposes a parallel implementation of a dual tree traversal algorithm with an efficient implementation on CPU/SIMD and multicore accelerators but shows that GPUs still outperform these machines for this task. We expect that our compact AoS layout and SWW can also be effective on other multicore architectures.

- A novel efficient parallel algorithm that directly generates entire *compressed implicit octrees* at once, as a massively parallel method. This algorithm differs from previous ones as it utilizes massively parallel steps and primitives, in opposition to traditional methods that incrementally traverse and insert nodes on partially constructed trees using many threads. We believe that, while efficient, this methodology of processing data is in conformance with enormous parallel processing power present in modern throughput computing systems.

We believe that the *parallel implicit octree direct generation methods and traversal algorithms* presented in this dissertation are generally applicable in many GPU computing contexts.

1.1 Structure of this dissertation

This dissertation is organized as follows. In chapter 2 we define octrees and discuss algorithms and techniques, from the literature, for their construction. In chapter 3 we propose a compact data representation for compressed implicit octrees and present a parallel algorithm to transform any pointer based tree to this format. In chapter 4 we propose efficient methods for traversing the implicit octree on GPUs with application in Barnes-Hut simulations. In chapter 5 we present experimental results on the application of implicit octree layouts and our SWW techniques in tree traversals in the context of Barnes-Hut simulations. In chapter 6 we present a new parallel algorithm that directly

generates entire octrees at once, as a massively parallel method. Chapter 7 presents our conclusions and plans for future work.

Chapter 2

On generating octrees

In this chapter we define and discuss algorithms, from the literature, for building octrees. In section 2.1 we present possible variations for defining octrees, and present the definition that we use in our work. We also define the class of *compressed octrees* and the benefits of using them in section 2.4. Section 2.2 describes an existing sequential algorithm for generating octrees. As the main focus of our work is on parallel processing of octrees, we present in sections 2.3 to 2.6, different parallel methods, from the literature, for octree generation. *Morton Space Filling Curves (SFC)* is an important mathematical tool for parallel construction, and in load balancing phases of parallel octree processing. We define and discuss these methods in sections 2.3 to 2.6.

2.1 Octrees

Octrees form a class of tree data structure used extensively to represent spatial data, e.g. in computer graphics, pattern recognition and scientific computing. In 3-dimensional (3D) data applications, octrees represent recursive subdivisions of the space in octants. A cubic region of space is subdivided in 8 octants of equal geometry, also known as cells. Each octant can be represented by an internal node in the tree. The root node represents the bounding box of the entire set of 3D data.

There are different representations for octrees:

- linear octrees: They discard internal nodes and leaves are represented *only* by Morton codes (section 2.3) a.k.a. z-order codes. This class of representation is commonly adopted in "voxel" (Volumetric Elements) computer graphics applications.

- sparse octrees: They are structures with "pointers", possibly not complete at each level, i.e. an internal node has at most 8 children. We distinguish between the structure of *internal* nodes and *leaf* nodes. Leaf nodes have no children.

We can have different possibilities for defining sparse octrees:

Definition 2.1.1 (Complete tree)**.** *An octree is a structure where internal nodes have* **exactly** *8 children.*

Definition 2.1.2 (Sparse tree)**.** *An octree is a structure where internal nodes have* **at most** *8 children.*

Definition 2.1.3 (Sparse tree with blocking factor at the bottom level)**.** *An octree is a structure where internal nodes that are* **not** *in the last level have* **at most** *8 children. Internal nodes immediately above the* **bottom level** *have* **exactly M** *children.*

Definition 2.1.4 (Sparse tree not restricted at the bottom level)**.** *An octree is a structure where internal nodes that are* **not** *in the last level have* **at most** *8 children. Internal nodes immediately above the* **bottom level** *have two or more children (leaves).*

In this work we consider sparse octree representations as in definition 2.1.4. More specifically we present parallel algorithms for generation and traversal of this class of *compressed* (section 2.4) representation for sparse octrees.

2.2 Tree Construction

We adapt from Warren and Salmon (1993) a simple description on how to sequentially build an octree from a set of leaf nodes using its geometrical data: "The higher level nodes in the tree can be constructed in a variety of ways. The simplest is analogous to the one described in Barnes and Hut (1986). Each *leaf node* is loaded into the tree by starting at the root, and traversing the partially constructed tree. When two particles fall within the same leaf node, the leaf is converted to a cell (i.e. an *internal node*), and new leaves are constructed one level deeper in the tree to hold each of the original *leaves*. This takes O(log N) steps per *leaf* insertion. After the topology of the tree has been constructed, the *application specific* contents (mass, charge, moments, etc.) of each *internal node* may be initialized by a post-order tree traversal."

2.3 Parallel octree construction using Morton codes, Morton ordering or Z-space filling curve (SFC) ordering

A space filling curve (SFC) is a mapping of points (or coordinates) on d dimensions to a one-dimensional sequence of values. Locality properties of space filling curves make them useful in linearization of data from d-dimensional spaces (e.g. 3D points).

Morton ordering or Z-space filling curve (SFC) ordering use Morton codes ($a.k.a$ Morton keys or Morton indexes) to map d-dimensional points (from d-dimensional hypercubes) to a 1-dimensional ordering. Morton codes are easily obtained by interleaving bits from coordinate values of each dimension. As in figure 2.1 for example, suppose a 3D point (x, y, z) with 8bit representation of each coordinate number. The Morton code, or Morton index, of (x, y, z) is a 24bit value formed by alternating bits from each value x, y and z.

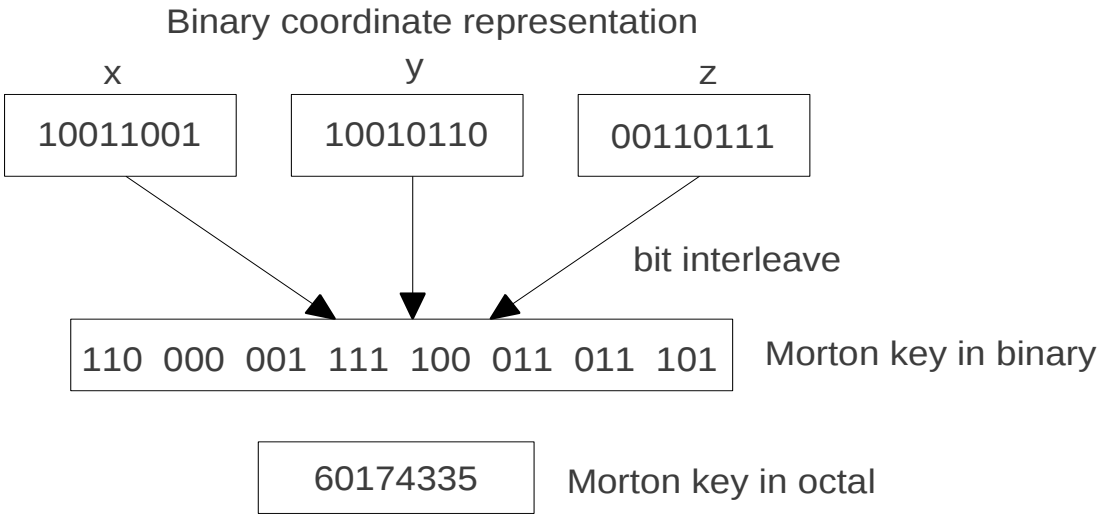


Figure 2.1: Construction of a Morton key from integer $3D$ coordinates

If we take a sequence of Morton keys $[m_1, m_2, m_3, ...m_k]$ in ascending numerical order and draw a corresponding polygonal line with endpoints in coordinates $p_1, p_2, p_3, ...p_k$ where m_i is the Morton key of points p_i we obtain a Morton space filling curve (SFC). As an example, in figure 2.2 we have Morton keys 0 to 63 and the respective SFC in a 2-dimensional space. We can conclude that a sequence of Morton keys can represent an ordering of points in space.

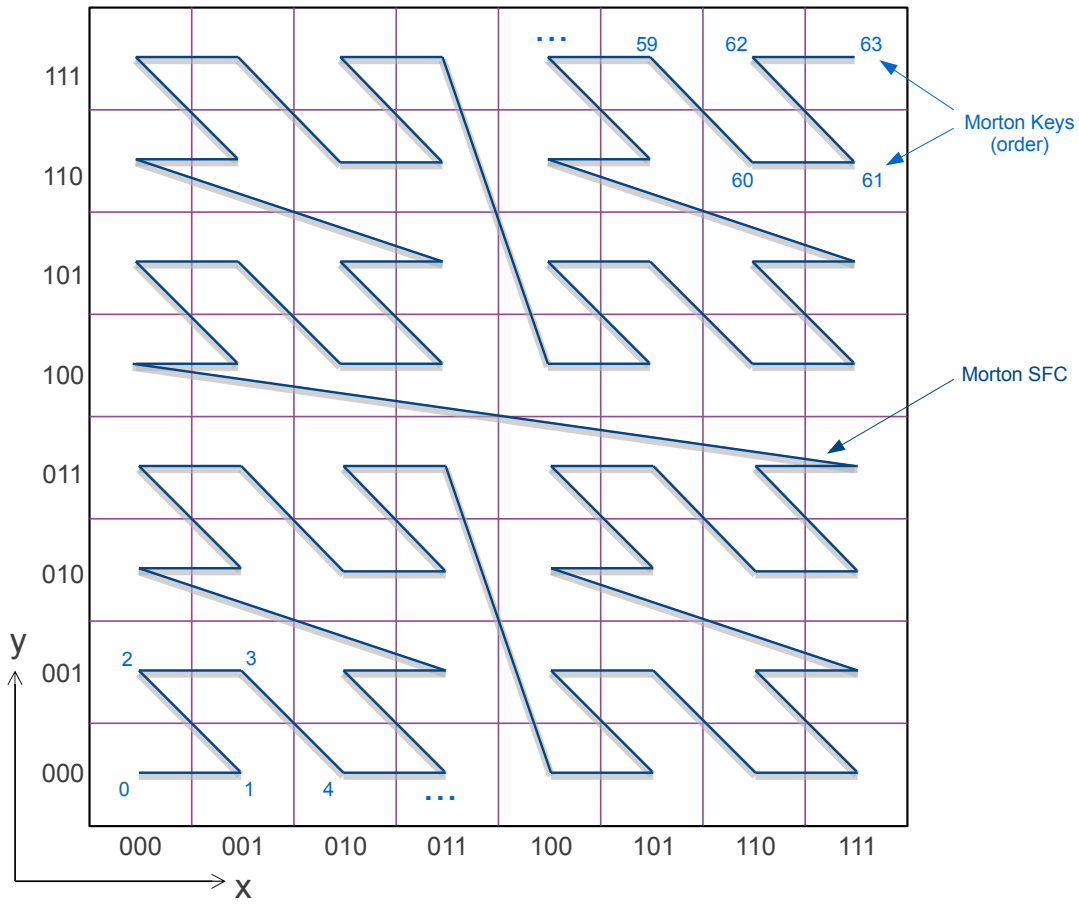


Figure 2.2: A 2-dimensional Morton space filling curve, obtained by interleaving y followed by x bits (from high to low order bits)

Conversely, if we take a set of $n-$dimensional points and sort them using their respective Morton keys, we produce a sequence of points in the order of their occurrence in the Morton SFC. The advantage of this procedure is that, by taking advantage of Morton's SFC spacial locality property, we can take points that are closer to each other in space and make them close to each other (to some extent) in a uni-dimensional sequence (as in an array of objects). This reflects in performance for applications accessing such data, as we can take advantage of the locality of reference in computer memory systems. In figure 2.3 we have some points in a 2-dimensional space. If sorted according to their Morton keys, the sequence of points will be: $i \rightarrow f \rightarrow j \rightarrow e \rightarrow c \rightarrow g \rightarrow h \rightarrow d \rightarrow a \rightarrow b$, as this is their order in the SFC. The Morton keys were obtained as described for figure 2.2.

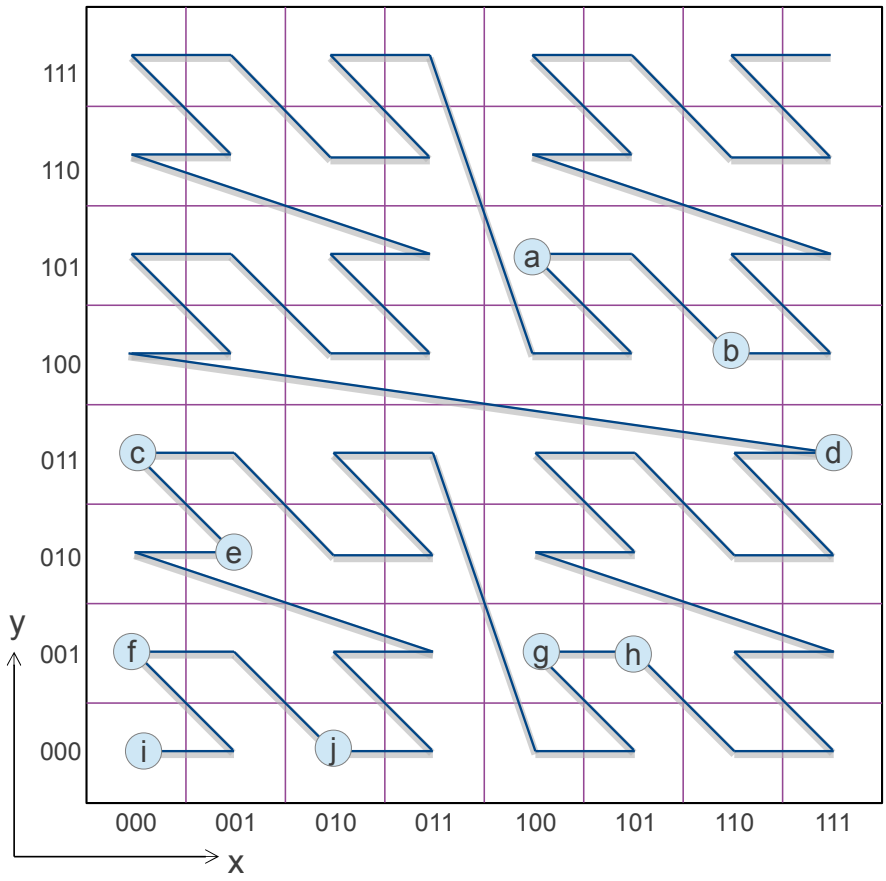


Figure 2.3: Some points in the 2-dimensional space and their order in the Z-curve.

An ordered sequence of Morton keys induces a *hyper*−octree. As an example in figure 2.4 we have a 2-dimensional space and a set of points in the bottom of the figure. These Morton keys induce a quad-tree. Parents of nodes or *least common ancestor* (LCA) nodes can be obtained from adjacent Morton keys. For 3D space, the Morton keys induce octrees.

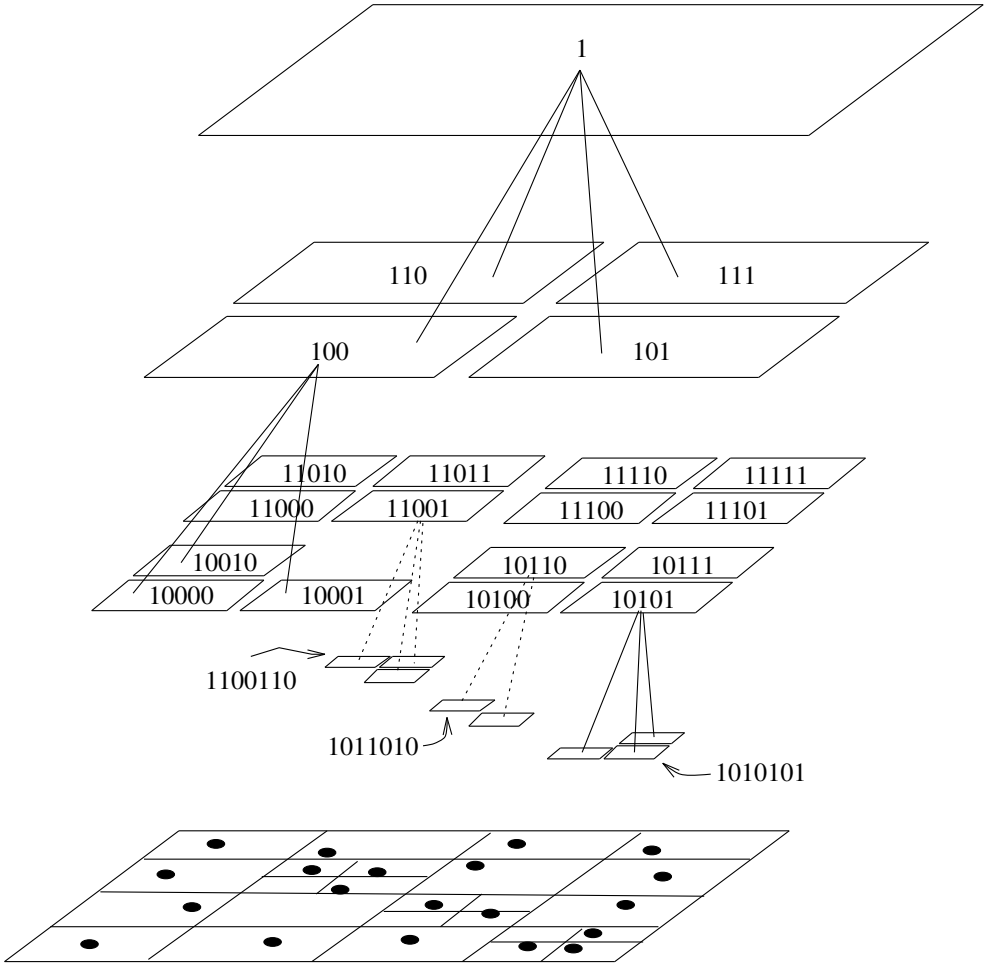


Figure 2.4: A set of points (bottom of the figure) in 2−dimesnional space and their respective Morton Keys. These Morton keys induce a quad-tree. Figure taken from reference (Warren and Salmon, 1993)

The next algorithms in this subsection work with multi-processors. In both algorithms a global tree is constructed in parallel, but each subtree is built sequentially on each processor from a subset of points distributed to each machine.

Warren and Salmon (1993) proposed a different method in multiple processors parallel construction of pointer based octrees using Morton codes as keys to hash leaf nodes. To distribute the nodes across processors, all leaves are sorted according to their Morton codes to allow a simple way to partition the tree among the processors. The idea is to partition the one-dimensional list of sorted keys/leaves into N_p equal chunks, where N_p is the number of processors.

Hariharan and Aluru (2005) followed the same approach of Warren and Salmon (1993) with respect to using Morton codes and Morton ordering, also known as Z-space (Morton,

1966) filling curve ordering of leaf nodes, to construct octrees in parallel on N-Body simulation code. Note that this algorithm also constructs a global tree is in parallel by sequentially building a subtree on each processor. This algorithm is described in figure 2.5. We further discuss the method of (Hariharan and Aluru, 2005) and of a derived work in section 3.4.

Parallel Algorithm for Compressed Octrees with Applications to Hierarchical Methods
(Hariharan and Aluru, 2005)

Input: The algorithm takes as input the number of allowed levels (l) in the tree to determine the size of the smallest cell (internal node cube).

Output: The compressed octree is constructed as follows:

1. Each processor is initially given $\frac{n}{p}$ points.
2. For each point, the corresponding leaf cell is generated.
3. These leaf cells are sorted in parallel, eliminating duplicates.
4. Each processor:
   (a) borrows the first leaf cell from the next processor and runs a *sequential algorithm* described in (Aluru and Sevilgen, 1999) to construct the compressed octree for its leaf cells together with the borrowed cell in O($\frac{n}{p}$ log $\frac{n}{p}$) run-time.
   (b) This local tree is stored in postorder traversal order in an array. Each node stores the indices of its parent and children in the array.

Figure 2.5: Multi-processor Parallel Algorithm for Compressed Octrees proposed by (Hariharan and Aluru, 2005)

2.4 Compressed Octrees

The term *distribution-independent* was introduced in Aluru and Sevilgen (1999) to characterize data structures in which the size is a function of the number of points only and is not dependent upon the distribution of the input points. Aluru and Sevilgen (1999)

also present a sequential deterministic algorithms to construct a compressed hyperoctree in $\mathcal{O}(NlogN)$.

Linear chains are formed in octrees when points form separate clusters in a small region of space. Since octrees are built as a result of successive subdivisions of space in octants, these linear chains tend to be formed when there exist non uniform distributions of points. In figure 2.6 we have a quadtree formed by subdivision of a $2D$ space, with up to 3 subdivisions of the space (outer rectangle). Each subdivision produces 4 identical subregions (rectangles). In a $d-$dimentional space each subdivision produces 2^d identical subregions. In the case shown in the figure, we limit the quadtree to 5 levels, and linear chains are formed. The presence of linear chains makes the size of octrees dependent upon the distribution of the points. If we limit the number of space subdivisions we also impose a limit on the maximum height of the trees. The fact that leaves can be clustered in a small subregion in distributions of points, can result in more than 2^d leaves in the last level of the three. For the case of octrees, the number of points inside a smallest region can be larger than 8 and, if we don't want to eliminate points in a small cell, we have to allow more than 8 leaves in the last tree level. For this reason we use our octree as in definition 2.1.4.

The term compressed octrees implies the absence of linear chains (Aluru and Sevilgen, 1999). Internal nodes in a chain represent redundant information in terms of the subdivisions, and we can eliminate chains by using only the smallest subdivision that encloses the points. In figure 2.6 we show the compressed quadtree for our initial set of points. This not only avoids information duplication but also makes the size distribution independent. The size of a compressed octree for n points is O(n). A compressed octree for n points (leaves) can be constructed sequentially in O(n log n) time, which is optimal (Aluru and Sevilgen, 1999).

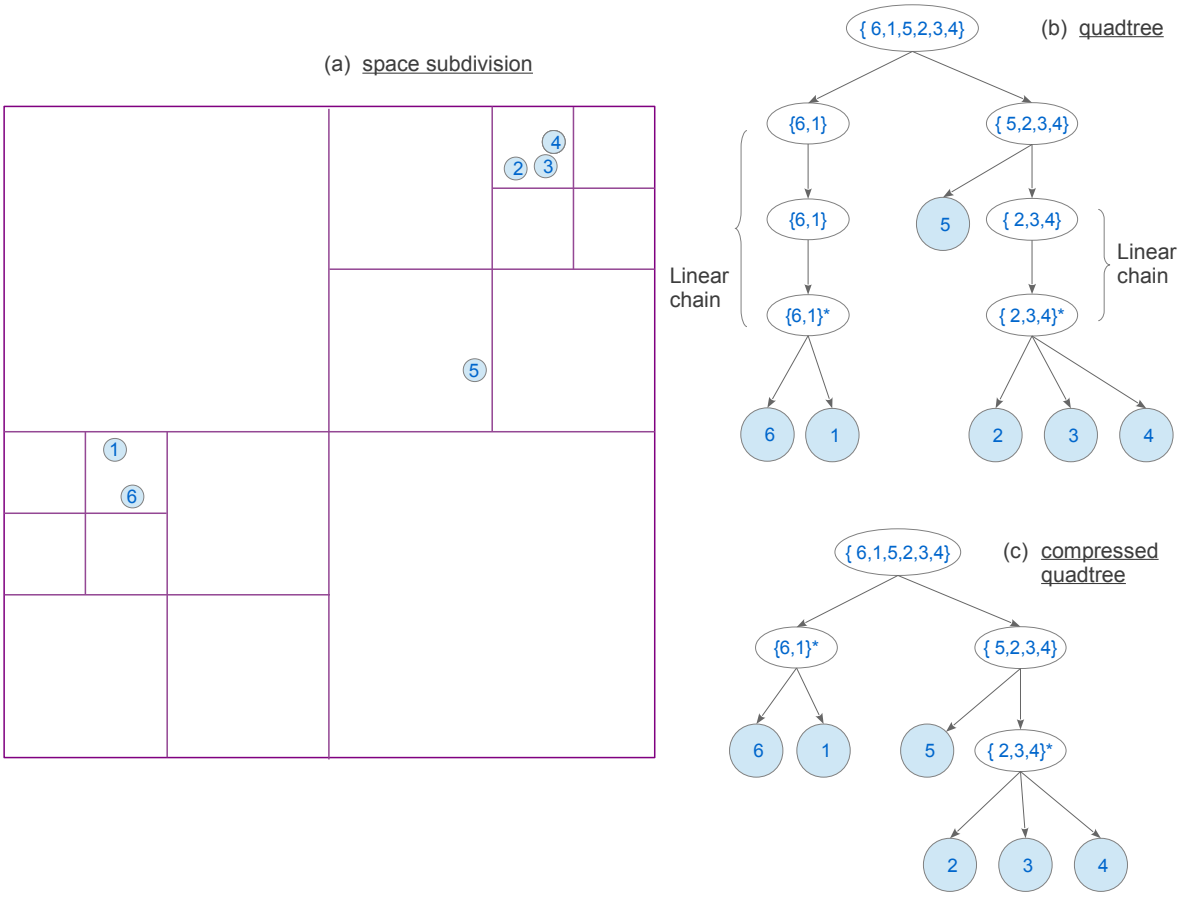


Figure 2.6: (a) set of points and 2D space subdivision, (b) a quadtree with the points and linear chains and (c) a correspondent compressed quadtree

2.5 Stackless tree traversals

Tree traversal algorithms are usually defined recursively, and the algorithm implementation can be coded as such. Any recursive algorithm can be equivalently be written using iterations (loops). Iterative versions of tree traversal algorithms make use of stack data structures.

Recursion is not an issue in current GPUs as it is supported in CUDA since version 3.0, but instead, equivalent iterative algorithms are generally used for performance reasons. Recursive traversals of trees use the programming language stack, whereas their iterative equivalents need to use a stack data structure which is explicitly controlled in the program.

Moreover, the stack structure is maintained in fast intracore memory (e.g. scratchpad memory) taking precious space and yet, introducing extra costs in data movements.

On parallel tree traversals each thread must maintain a stack, which is generally kept in main memory in multi-processors. In manycore parallel processors (such as GPUs), stacks are maintained in scratchpad memory (a.k.a. shared memory in GPUs). Scratchpad space is a scarce resource, more so when a massive amount of threads are used as in the case of GPUs. For a relatively small size of shared memory and large number of threads per multiprocessor, as is the case of GPUs, we have to limit the maximum size of each stack, which limits the maximum tree height.

The work of Burtscher and Pingali (2011) uses regular pointer based sparse octrees, and iterative octree traversals on GPUs, using a small stack per thread (kept in shared memory). On this work, the use of the stack limits maximum tree height in N-Body simulations, since the stack needs to be maintained per thread in shared memory. The space limitation on maximum stack size per thread imposes the same limited on tree height, during parallel traversals. Our proposed compressed octree layout presented in chapter 3 allows octree traversals without a stack.

Foley and Sugerman (2005) presented two methods for stackless kd-tree traversal in GPUs, (a.k.a kd-restart and kd-backtrack). By chasing additional pointers (e.g. to parent nodes, in the case of kd-backtrack) and not using a stack, their implementation reduced the working set size needed to trace rays while traversing a KD-tree and resulted in higher throughput in their parallel raytracer, although still not outperforming CPUs in this application. Popov *et al.* achieved higher speedups in GPU raytracing with stackless KD-tree (Popov et al., 2007) traversals by using extra pointers (*ropes*) in the tree structure. Figure 2.7 shows nodes of a KD-tree, augmented with *ropes* pointers to "link each leaf node of the kd-tree via its six faces (of a cube) directly to the corresponding adjacent node of that face or to the smallest node enclosing all adjacent nodes if there are multiple". The work of dos Santos et al. (2009) further studied this method, with comparison to others, and suggested an optimized algorithm to insert ropes in the KD tree achieving better performance on traversals.

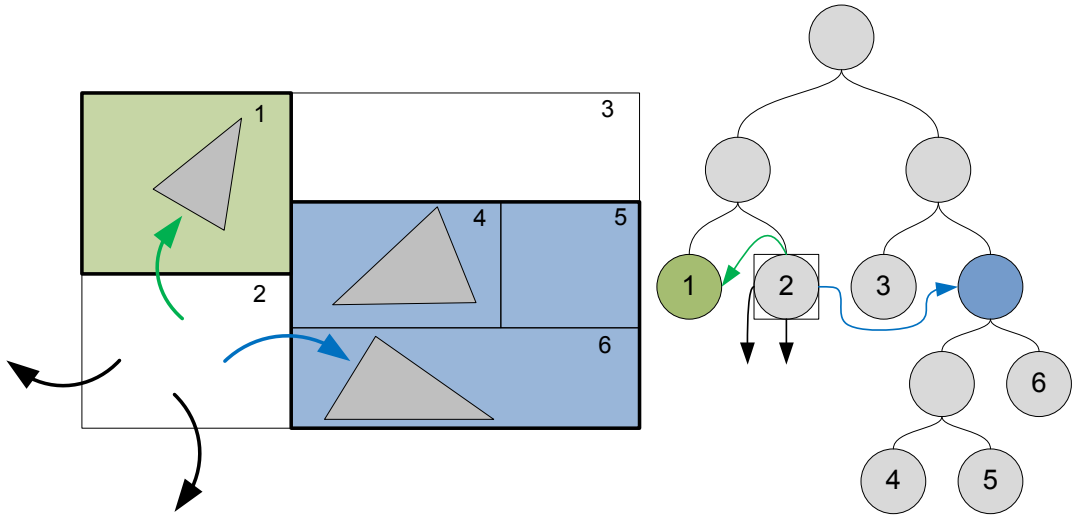


Figure 2.7: A kd-tree augmented with "ropes" from Popov et al. (2007). Ropes drawn as arrows. A *2*-dimensional tree is shown, with ropes to 4 faces only from one leaf node. Black ropes show the case when pointing to the enclosing region (node).

2.6 "Massivelly parallel" algorithms for octree construction

A GPU parallel method for generating octrees was studied by (Ajmera et al., 2008). The algorithm (presented in figure 2.8) generates octree nodes in post-order using Morton Space Filling Curves (SFC). Step 2a of the algorithm allocates a large number of threads, relative to the cells resolution (or octree levels). The maximum number of threads in a GPU kernel is a limiting factor. Considering the recent GTX680 GPU model shown in table 5.1, the maximum number of threads per multiprocessor is 2048, with 8 available multiprocessors, resulting in a maximum of 2^{14} threads per GPU. Assuming this limitation can be overcome in programming (e.g. by using persistent threads) we reach another limitation on GPU memory allocation, as we can see in step 1, which allocates arrays with a total space proportional to 8^{k+1} cells, where k is the octree resolution (i.e. octree height). The complexity of this algorithm is exponential in space and time, although it is feasible to build small octrees (Ajmera et al., 2008).

GPU Parallel Compressed Octree Construction Algorithm
(Ajmera et al., 2008)

Input: SFC based sorted ordering of cells at resolution k
Output: An Adaptive Octree (Leaves present at different levels)

1. Allocate $L_0, L_1, ..., L_k$ arrays of sizes $8^0, 8^1, ..., 8^k$ respectively
2. Loop for i=k to i=1
   - 2a. Allocate 8^{i-1} threads
   - 2b. Each thread checks 8 elements in L_i from SFC ids $(8 * ThreadId)$ to $(8 * ThreadId + 8)$
   - 2c. If all 8 elements are empty then make all the elements NULL and their PARENT at level L_{i-1} as leaf

Figure 2.8: A GPU-based Parallel Octree Construction Algorithm proposed by Ajmera et al. (2008)

A recent, still unpublished work (Goradia, 2012) that is based on (Aluru and Sevilgen, 1999), (Hariharan and Aluru, 2005) and (Ajmera et al., 2008) proposes a bottom up GPU-based adaptive octree construction algorithm. We present this algorithm in two parts in figures 2.9 (part A) and 2.10 (part B). This work also uses Morton space filling curve (SFC) to sort tree leaves, and constructs the tree in *postorder* as in its seminal work. We reproduce bellow this algorithm's steps from the reference. The paper also describes how to generate an octree from the compressed octree. A comparison to our massively parallel algorithm for *direct* generation of the compressed implicit octrees is presented in section 3.4.

GPU-based Adaptive Compressed Octree Construction Algorithm
PART A - steps 1-2 (Goradia, 2012)

1. **Constructing leaves:**

   1a. Read n points in the first n locations of an array A of size $2n - 1$.

   1b. Assuming a point per leaf, for every point, in parallel, do:

      i. Generate the 3D coordinate of leaf cell to which it belongs

      ii. Generate SFC index for the leaf cell

   1c. Sort the first n elements of array A, in parallel, based on SFC indices of leaves.

2. **Generating internal nodes & post order:** In Parallel, for every adjacent leaves, find their LCA using the common bits in their SFC indices:

   2a. Allocate $n - 1$ GPU threads.

   2b. For every two adjacent leaves (say at locations i and $i + 1$) in array A, in parallel, generate the internal node and store it at location $n + i$ in array A.

   2c. Sort, in parallel, the internal nodes generated, across levels based on their SFC indices. To do the same, we need to establish a total order on the cells across levels. If one is contained in the other, the subcell is taken to precede the supercell; if they are disjoint, they are ordered according to the order of the immediate subcells of the smallest supercell enclosing them, sorted internal nodes with duplicates might be generated.

   2d. Allocate $n-2$ threads for a maximum of $n-2$ consecutive internal node pairs in the later half of array A to remove the duplicates.

   2e. For every two adjacent internal nodes not having same SFC indices, in parallel, traverse back in the later half of array A starting from the current node to look for its duplicates and eliminate them.

   2f. Sort array A, in parallel, based on SFC indices across levels to get the postorder traversal of a compressed octree.

3. **... continues on PART B**

Figure 2.9: A GPU Parallel Compressed Octree Construction Algorithm (PART A) proposed by Goradia (2012)

GPU-based Adaptive Compressed Octree Construction Algorithm
PART B - steps 3-4 (Goradia, 2012)

... (continued from PART A)

3. **Generate the parent-child relationship in the compressed octree:**

   3a. Allocate an array B twice the size of the num- ber of leaves and internal nodes (atmost $4n - 2$). Copy the first half of array B with the current post-ordered array A of leaves and nodes.

   3b. Allocate threads one less than the $(NumberOfLeaves + InternalNodes)$.

   3c. For every two adjacent nodes in the first half of array B, in parallel, do:

      i. Generate the LCA from the SFC indices.

      ii. Copy the new node (copy of the parent of the first node in the pair considered) into the corresponding location in second half of the array B.

      iii. Write in this new node, the SFC index of the first node of the node-pair which generated it, along with the location of that node in array A. This location information will eventually give the index of the child this parent node-copy was generated from.

   3d. Sort array B across levels, in parallel based on newly generated SFC indices. All the parents and their copies will come together.

   3e. For every two adjacent nodes both having same SFC indices and atleast one of them not being a generated copy, in parallel, do:

      i. Establish the parent-child relationship. Here we see that one of the nodes is the original node and another is its copy (generated in step 3c.ii). The copy will give the location of the child in array A while we get the location of the parent from the original.

      ii. Scan ahead in array B and repeat step 3e.i for all the copies of the original to establish the relationship between the parent and all its children. Step 3e.i will be repeated atmost 7 times since in an octree, a parent can have atmost 8 children.

Figure 2.10: A GPU Parallel Compressed Octree Construction Algorithm (PART B) proposed by Goradia (2012)

Burtscher and Pingali (2011) use a similar approach as described before in section 2.2 for the *sequential algorithm* to build octrees on a GPU *parallel* code for N-Body Barnes-Hut simulation, i.e. this work does *not* use SFC on the tree construction, although it separates leaf nodes and internal nodes in different global memory regions, and sorts them

in *postorder*. To adapt for the GPU massively parallel processor, this method introduces locking on tree nodes and busy waiting, to make it work for the parallel threads in cases when different threads try to simultaneously insert in the same tree branch. Due to the automatic scheduling of threads and the SIMT (*Single Instruction Multiple Thread*) mechanism present on GPUs (which executes a set of threads in lockstep), there is a possibility of deadlock in algorithms that use locking and busy waiting. Burtscher and Pingali (2011) take extra precautions against deadlocks by using different ordering of tree nodes, providing different regions of memory for internal and leaf nodes in memory, and providing guarantees on visiting tree nodes in deadlock free order, which make parallel programming even harder to code and maintain. The authors (Burtscher and Pingali, 2011) argue that these guarantees make the code free of deadlocks independent of thread scheduling. Our parallel algorithm for *direct* generation of the compressed implicit octrees presented in chapter 6 is built with massively parallel primitives only, using no busy waiting nor locking mechanisms. We have studied the performance of this algorithm in comparison to our work in chapter 5 and in section 6.4

Another recent N-body gravitational simulator (Bédorf et al., 2012) that runs entirely on GPUs was presented by Bedorf *et al.* This code also builds and traverse sparse pointer based octrees at each simulation step but with a different structure. Their trees are generated and traversed top down, per tree level. The tree layout process needs to sort all leaf (body) nodes in Morton order. We have also studied the performance of these algorithms with respect to tree-traversals in N-Body simulations. We present our conclusions and further discuss this work in section 5.1.

Chapter

3

A Compressed Implicit Octree Layout for Stackless Traversal

In this chapter we propose a compact data representation for compressed implicit octrees in section 3.1. This tree layout allows stackless traversals. In section 3.6 we present a parallel algorithm that transforms any pointer based tree to this format. In Section 3.3 we prove useful properties about this layout and compare to possible alternatives from the literature, in sections 3.4 and 3.5.

3.1 Compressed Implicit Tree Layout

Compact representations of data structures are commonly used to enhance the efficiency of algorithms, mainly when sparse data is represented. Examples range from compact representations for sparse matrix algorithms to compressed representations for graph processing, just to name a few.

We propose a compact data representation for compressed octrees, named *Compressed Implicit Tree* layout. Our representation allows efficient stackless parallel preorder traversals on sparse octrees.

A sparse pointer based octree and its equivalent implicit layout, corresponding to a *preorder* disposal of nodes is presented in figure 3.1.

Our implicit tree structure avoids dynamic allocation by juxtaposition of internal and leaf nodes in a single linear array. Although heterogeneous, this linear mixture of different types places the nodes in the proper traversal order, making most accesses to the tree hit the data cache present in modern GPUs. Most pointers are implicit, e.g. leaf nodes implicitly point to the next node, thus occupying no memory space, with the exception of one explicit pointer, present in each internal node, which we call *skip link*. Traversals are performed from left to right, corresponding to a preorder walk. The *skip link* points the node to "skip to", in cases where the algorithm decides to detour from (i.e. not visit) the subtree ahead. The implicit octree data structure requires only one pointer per internal node besides application data, making it very compact and perhaps, most importantly, placing less demand for memory bandwidth in traversals. Information stored at the root of each subtree are accessed during traversals to take decisions of weather to *visit* the subtree or to skip it.

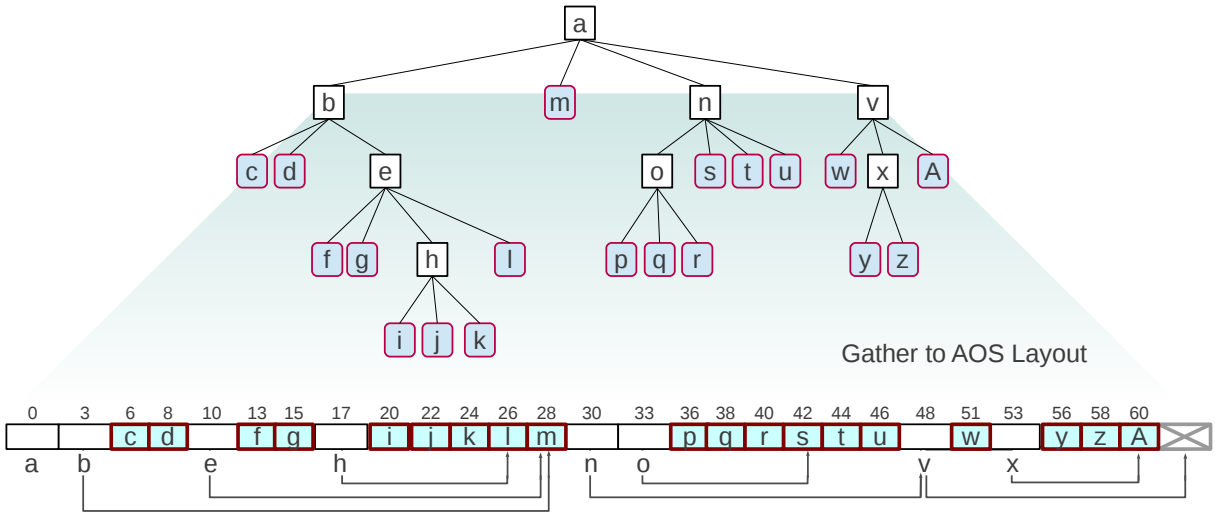


Figure 3.1: Implicit tree layout for stackless traversal: The numbers above the nodes in the implicit layout represent word addresses of each node. Internal and leaf nodes can have different sizes. In the example we assume internal nodes and leaves have size 3 and 2 words respectivelly.

In figure 3.2 we see another pointer based tree T_{p2}. A different correspondent view of the compressed tree representation for T_{p2} is presented in figure 3.3, with implicit pointer relations shown in dashed lines and skip links in green. A next cell in the linear sequence

implicitly represents either a father-son relation as in $a \rightarrow b$, a next-sibling relation as in $m \rightarrow n$, or a next-in-preorder relation as in $k \rightarrow m$.

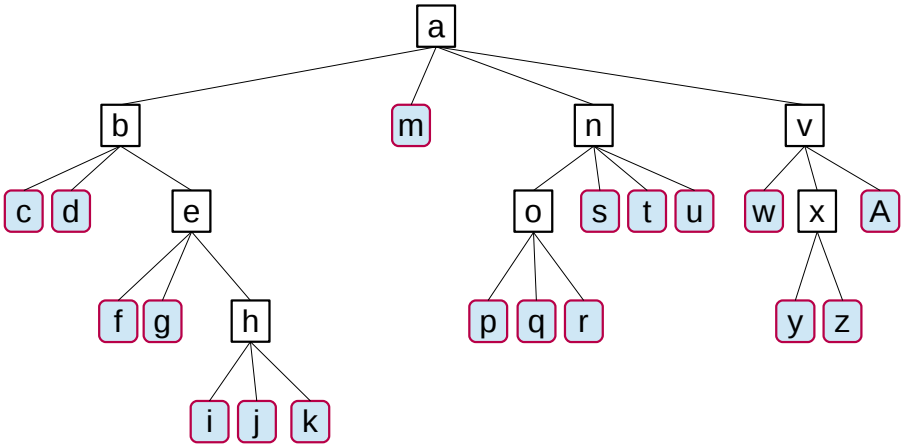


Figure 3.2: An example of a Pointer Based Tree: T_{p2}

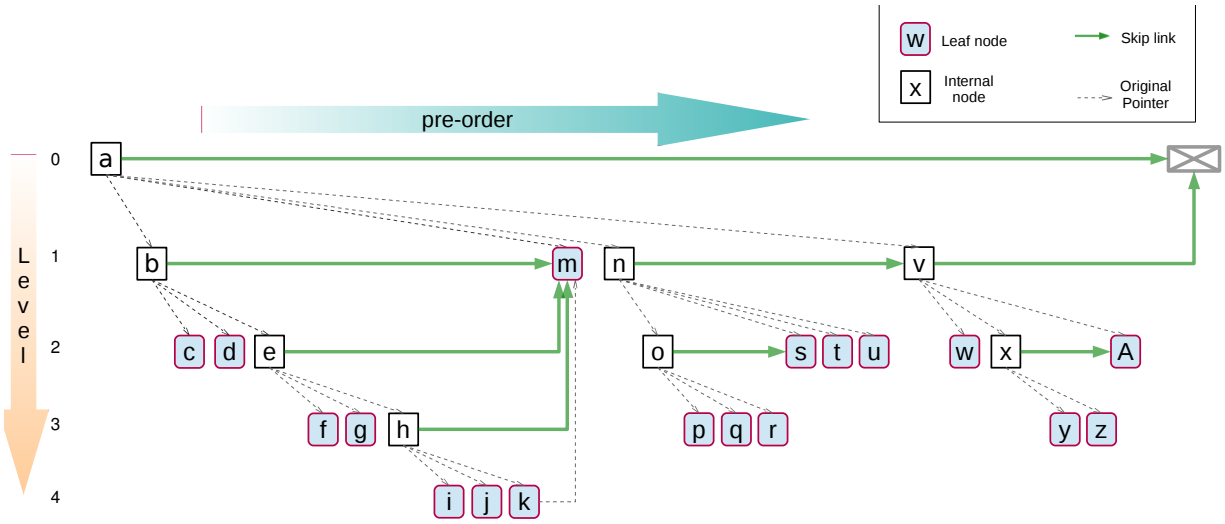


Figure 3.3: A first view of the implicit relations in tree T_{p2}.

In the view of Figure 3.3 we have represented the linear layout in 2 dimensions. It is easy to visualize that by walking in sequence (horizontally) we can either traverse a subtree without accessing any explicit pointers or we can skip the entire subtree. When entering a subtree we can directly walk to lower levels (vertically) without accessing explicit pointers, by just accessing the next cell in the sequence.

During the traversal, it is possible to decide, by accessing the information in each internal node, whether to keep visiting subtrees or to skip to a next-in-preorder node (i.e. skip a subtree) in this case by accessing an explicit pointer.

A more accurate view of the representation is depicted in Figure 3.4, where the dashed arrows correspond to the real implicit relations, which is from a node n_i to a next node n_{i+1} in the linear sequence. The final representation of the compreesed implicit three T_{p2} is shown in figure 3.5. In this representation we assume that leaf nodes are of size 2 (e.g. in words) and internal nodes are of size 3. The numbers above nodes in the linear view correspond to the word address of each node, relative to the beginning of the tree.

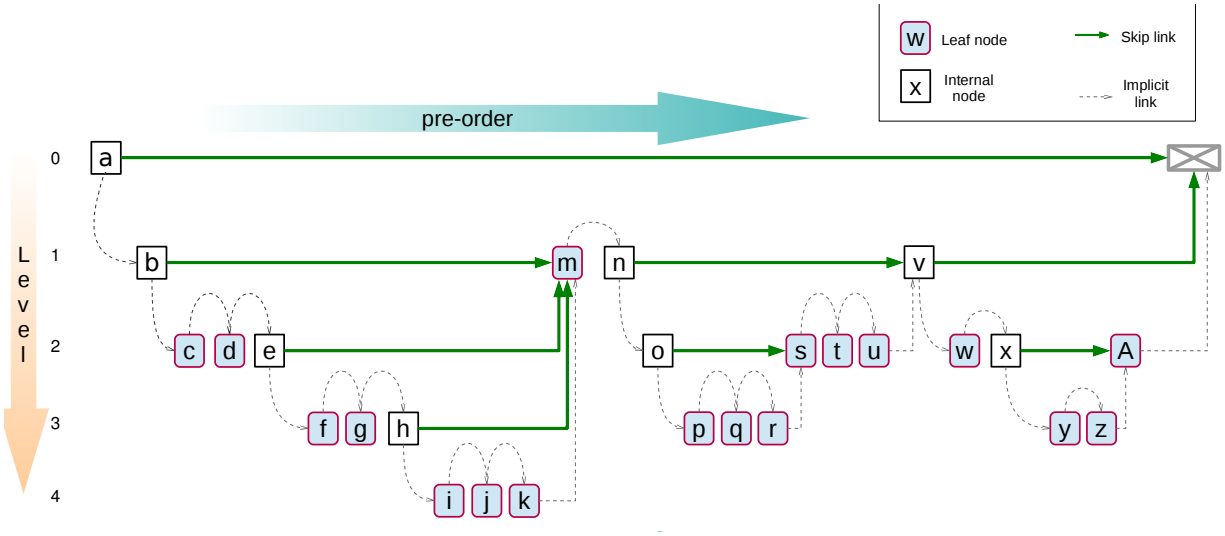


Figure 3.4: A more accurate view of the implicit relations in tree T_{p2}.

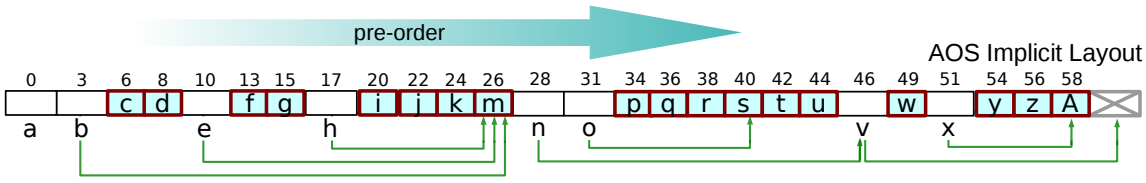


Figure 3.5: The final representation for example tree T_{p2} in the implicit layout. In this representation we assume that leaf nodes are of size 2 (e.g. in words) and internal nodes are of size 3.

From figure 3.4 and with this final representation in figure 3.5 we can observe that by visiting nodes in sequence we can efficiently take advantage of locality of references in caches, as the information that will be necessary in a traversal will be present in the cache line, in this case, if we keep walking down the subtree. Note also that the automatic loading of a cache line in this case is equivalent to an automatic prefetch of up to w bytes ahead, where w is the width of a cache line (or load granularity in case of GPUs). We

could also experiment with insertion of extra prefetching instructions at this point in the algorithm, to hide the latency of memory accesses during traversals.

We describe the parallel *Tree Transformation Kernel* in section 3.6. The algorithm takes as input any sparse pointer based tree and produces the implicit layout equivalent. At each step of the Barnes-Hut simulation method a pointer based octree representation of the particles configuration in 3D space is generated and traversed by the *Traversal/Force Calculation Kernel*. Algorithm 2 is employed at each step to transform the pointer based octree to the equivalent implicit octree layout, with the order of nodes corresponding to a preorder traversal of the tree. Traversal of the implicit octree layout can be performed iteratively without the use of any stack data structure, lowering "working set" requirements per particle by saving on register and shared memory space used per thread and producing an efficient execution in the GPU.

Our stackless *Traversal Kernel* employs other techniques to produce higher granularity per thread by grouping virtual threads in simulated wide warps (section 4.3) and to walk in more coherent tree walking patterns, lowering thread divergence effects and producing mostly coalesced accesses in recent cache based GPU architectures.

3.2 On the efficiency of the implicit layout for coalesced memory accesses in preorder (top-down) traversals

GPUs work with SIMT (*Single Instruction Multiple Thread*) hardware and execute the same instruction on a group of w cores. On current GPUs the so called with of a warp w equals 32 cores. This is used to run 32 threads in lockstep, the group of 32 consecutive numbered threads is called a *warp*.

When issuing a memory transaction all w threads in a warp can read its own destination address. In this case, if we have n different addresses then n memory transactions are started incurring in low utilization of memory bandwidth. If the set of w threads

in the warp access addresses in a "window" of w contiguous address space, the memory operation is denominated *coalesced*. In coalesced accesses only one memory transaction is issued, and a set of $w = 32$ words are read to a common buffer (or cache line), this way making efficient use of memory bandwidth.

The authors in (Zhang et al., 2011) prove that finding the optimal data layouts or thread-data mappings that minimize the number of memory transactions for any algorithm is NP-complete. They also argue that this result does not preclude optimality of accesses in specific cases.

Consider N tree traversals in parallel as in the algorithm below.

Algorithm 1 Generic parallel preorder traversal with the implicit tree layout.

```
 1: for each thread in parallel do
 2:     node ← root;
 3:     while stoppingCriterion( node ) == false do
 4:         nodeData ← loadNode( node );
 5:         if visitingCriterion( node ) == true then
 6:             visit( node );
 7:             node ← left_sibling( node );          ▷ follow implicit pointer
 8:         else
 9:             node ← next_in_preorder( node );      ▷ follow skip link
10:         end if
11:     end while
12: end for
```

SIMT many-core (GPU) threads work in lock step in warps, and each warp fetches a set of contiguous bytes at a time, a block per thread (equivalent to fetching a cache line in uniprocessors).

When all threads fetch contiguous addresses or if their accesses are in the same order the access pattern is denominated *coalesced*, i.e. only one block is fetched from memory (one memory transaction generated) and the content is "shared" by all warp threads, maximizing the use of global memory bandwidth.

All threads in the warp access next nodes ahead in the proper order (e.g. preorder Barnes-Hut octree traversals) which is fetched at once by the warp.

From figure 3.4 and with the final representation in figure 3.5 we can observe that by visiting nodes in sequence, as in algorithm 0, we can efficiently take advantage of locality

of references in caches, since the information that will be necessary in a traversal will be present in the cache line, in this case, if we keep walking down the subtree. Note also that the automatic loading of a cache line in this case is equivalent to an automatic prefetch of up to w bytes ahead, where w is the width of a cache line (or load granularity in case of GPUs). We could also experiment with insertion of extra prefetching instructions at this point in the algorithm, to hide the latency of memory accesses during traversals.

3.3 On the number of "skip links" going between consecutive partitions (processors) in the Implicit octree

A tree is commonly partitioned among processors, each processor being responsible for a section of the tree. The number of *skip links* going from one partition to the next in sequence in our linear format implies the amount of information that possibly needs to be communicated among processors, which we want minimized. This is important for the many-multiprocessors-plus-one-global-memory architectures (as is the case of GPUs, for example) or for partitioning a big tree among many processors in a distributed parallel computer (such as a multi-GPU environment).

The following lemmas concerns the number of skip links between partitions of the compressed octrees, each partition being assigned to a different processor (or multiprocessor).

Lemma 1. *There is at most one skip link from a given level crossing two consecutive partitions of the Compressed implicit tree.*

Proof. By way of contradiction, suppose there are two internals nodes, A_1 and A_2 at the same level in partition A, and that both A_1 and A_2 point to a node b_{next} in partition B. Suppose, without loss of generality, that A_1 comes before A_2 in the *preorder*. In this case we must have either:

a) A_1 and A_2 have the same father (illustrated in figure 3.6). This implies that A_1 is a sibling of A_2. If there exists more siblings between A_1 and A_2 then A_1 points to a sibling; otherwise, A_1 points to A_2 with either an implicit or an explicit link. Thus, A_1 cannot point to b_{next}, a contradiction.

b) A_1 and A_2 have a least common ancestor, but not the same father (illustrated in figure 3.7). In this case, as A_2's father precedes A_2 in the preorder, but A_2's father cannot precede A_1 by our hypothesis, we conclude that A_1 points to some node between A_1 and A_2. Therefore, A_1 does *not* point to b_{next}, which is a contradiction.

□

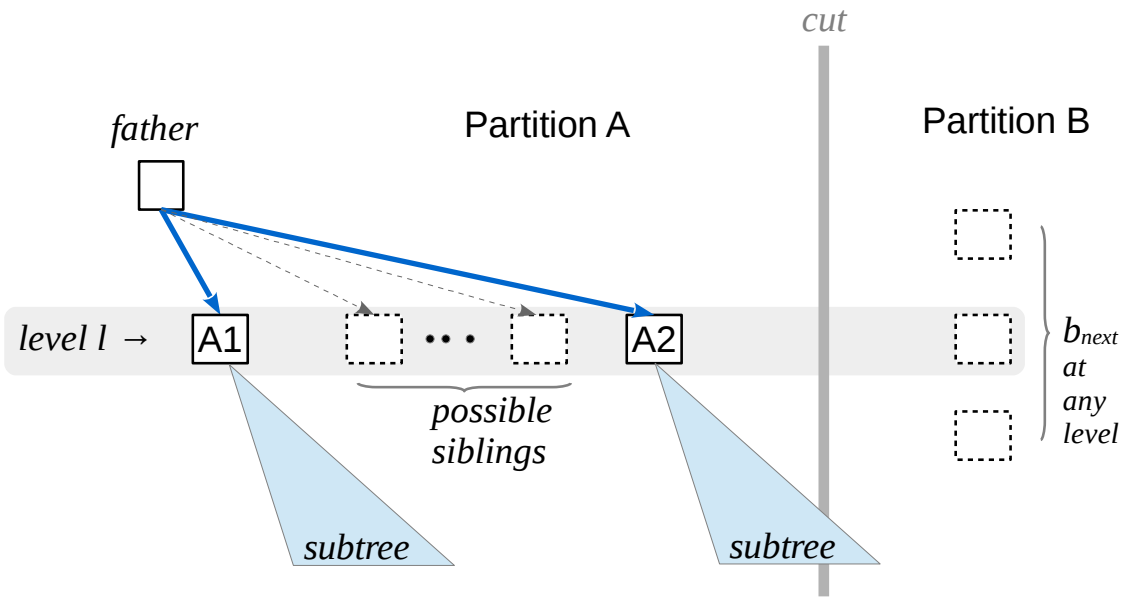


Figure 3.6: Illustrates the proof of Lemma 1: Case A

Lemma 2. *There are at most l skip links crossing two consecutive partitions of the Compressed implicit tree, where l is the number of levels of the Compressed implicit tree.*

Proof. Since there are at most l levels in the tree, and Lemma 1 excludes multiple skip links from the same level to point to a given node in the next partition, it follows that there are at most l skip links crossing partitions, coming from distinct levels.

□

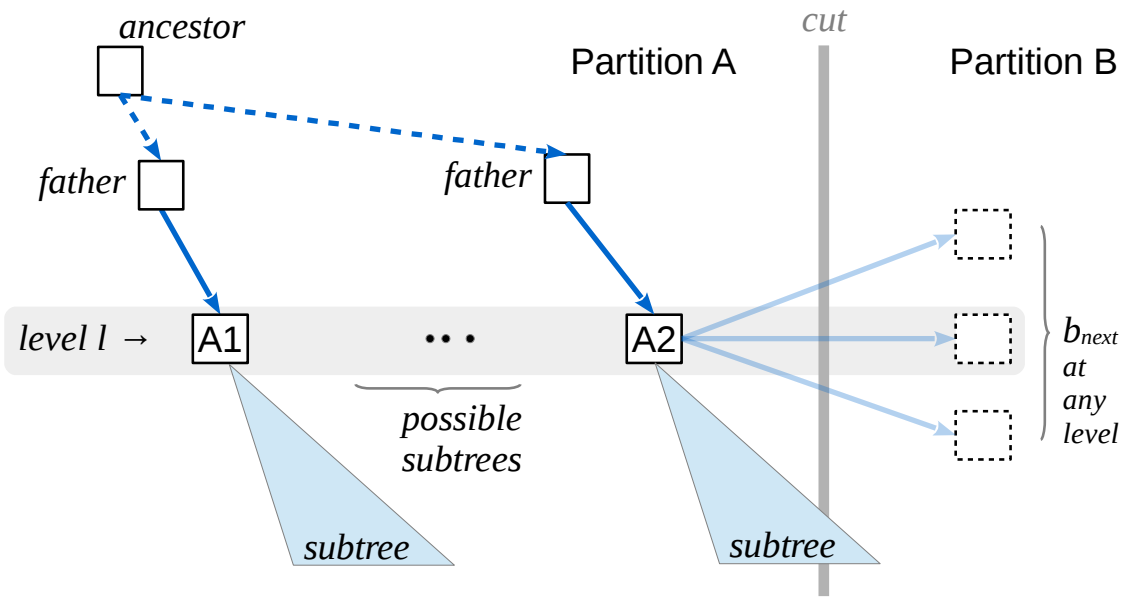


Figure 3.7: Illustrates the proof of Lemma 1: Case B

3.4 Implicit pre-order layout versus Hariharan-Aluro and derived Compressed Octree generation Algorithms

The work in (Hariharan and Aluru, 2005) presents a parallel algorithm for generation of compressed octree. This algorithm generates an internal node for every pair of consecutive leaves, sorted in *"post-order"* using Morton keys. This process generates duplicates for internal nodes, as stated in Lemma of reference (Hariharan and Aluru, 2005): "there may be duplicates – each internal node is generated at least once but at most seven times (2^d-1 times in d dimensions)". These internal nodes are generated on each processor, and *out of order* nodes generated at a processor must be communicated to other processors using Many-to-Many communication. The consequence of the possible duplication of generated internal nodes is that "The total number of nodes received by a processor is at most $7l$ " in a tree of l levels (Lemma 3, of reference (Hariharan and Aluru, 2005)).

Our algorithm does not generate duplicate internal nodes, as a consequence of identifying the correct place for internal nodes in the pre-order, which is done in the "level differences" phase of our algorithm. In the aftermath each processor needs to commu-

nicate only at most l "transfer cells", to a "next neighbour processor", in a global tree of l levels. This way, we communicate considerable less information between processors, and as one processor only sends to its neighbour processor we don't need Many-to-Many communication.

Another consequence of duplicate internal nodes in (Hariharan and Aluru, 2005) is that "the received nodes are merged with the local postorder traversal array and their positions are communicated back to the sending processors", as stated in the reference.

A recent, still unpublished work (Goradia, 2012), that is based on (Aluru and Sevilgen, 1999), (Hariharan and Aluru, 2005) and (Ajmera et al., 2008) proposes a bottom up GPU-based adaptive octree construction algorithm. This algorithm, described in figures 2.9 and 2.10 also uses Morton space filling curve (SFC) to sort tree leaves, and constructs the tree in *postorder* as in its seminal work , with one difference that there is a step (3d and 3e) in the algorithm to eliminate duplicates of internal tree nodes. Nonetheless, this algorithm uses *four* sorting steps based on SFC indices: step (1c) sorts N leaves, step (2c) sorts $2N$ leaves and internal nodes, step (2f) sorts N leaves and $\mathcal{O}(N)$ internal nodes, and finally step (3d) sorts up to $(4N - 2)$ internal and/or leaf nodes, as can be verified in section 2.6 or in the reference. By comparison, our direct parallel GPU tree generation algorithm of section 6.3 only uses *one* sorting step for N pairs (m, i) where m is a Morton code of a leaf and i is the array index of the leaf. As we can verify in figure 6.5 of the experimental section this takes roughly half of the tree generation time. We conclude that our massively parallel algorithm for *direct* generation of the compressed implicit octrees presented in chapter 6 is at least around 2.5 times faster.

3.5 Implicit Pre-order Layout versus Left-Child Right-Sibling Representation

The implicit pre-order layout can be naivelly understood as a direct implicit layout of the Left-Child Right-Sibling (LCRS) representation (Knuth, 2006) (Cormen et al., 2009) of a tree, but in fact this is not the case. The source of this erroneous interpretation comes

from the fact that each internal node in the implicit layout has one implicit pointer and one explicit pointer, which resembles a node in the LCRS representation.

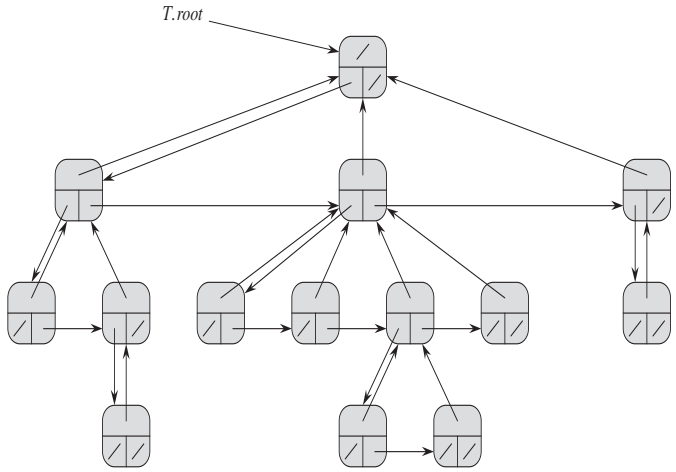


Figure 3.8: A pointer based Left-Child Right-Sibling representation of a tree, figure from reference (Cormen et al., 2009)

Cormen et al. (2009) present a pointer based Left-Child Right-Sibling representation of a tree (Figure 3.8). This version of the representation uses 3 pointers per tree node, besides application data. The pointers are named $x.p$ (top pointer), $x.lc$ (lower left pointer), and $x.rs$ (lower right pointer), pointing respectively to the *parent*, *left-child* and *right-sibling* of a node.

The pointer to the *parent* of a node is only necessary if the application needs bottom-up traversals of the tree. For direct comparison with our implicit *preorder* layout we consider a simplified version for nodes, using only pointers to a *left-child* and to *right-sibling* of each internal node.

In section 3.6 we have presented a parallel algorithm to transform any pointer based tree in the equivalent Compressed implicit layout. Any pointer based tree can also be transformed to its equivalent LCRS representation (Knuth, 2006), and vice-versa. As such, it would be also conceptually possible to devise an algorithm to transform from the LCRS representation to an equivalent Compressed implicit layout. Although, it is not clear if such method would be amenable as an efficient GPU parallel transformation, we would most likely need to maintain a stack per thread in kernels, and/or resort to locking nodes or counting and busy waiting. Counting and busy waiting were used in

our transformation kernels of section 3.6, and also in the GPU octree construction for the Barnes-Hut problem in reference (Burtscher and Pingali, 2011). Besides, our direct parallel algorithm for the Compressed implicit layout as presented in chapter 6 uses no locking primitives nor busy waiting loops, and all steps are inherently parallel.

One possible direct simplification of LCRS would be to dispose all tree nodes per level. In such linearization, we would have a simple mapping: all *right-sibling* can be made implicit, and *left-child* pointers need to be explicit, pointing to nodes ahead in the linear order. Our *pre-order* disposal that can be used in "top-down" traversals clearly differ from this natural "per level" representation of LCRS, that would more likely be used in breadth first traversals. In such case it is not clear what would be the general purpose of *left-child* pointers.

Nonetheless, with the following argument, we can see that the implicit layout is not directly mapped from the LCRS representation:

i) implicit pointers in the *Compressed implicit layout* are not directly equivalent to any pointers in the LCRS representation. Taking examples from figure 3.4, implicit pointers are either to a "left-child" of a node (e.g. implicit link $o \rightarrow p$), or to a "next-sibling" (e.g. implicit link $q \rightarrow r$), or yet to a "next-in-preorder" node as in link $r \rightarrow s$.

ii) explicit pointer in the *Compressed implicit layout* do not correspond directly to a "next-sibling" pointer. For example, also in figure 3.4, explicit link $b \rightarrow m$ correspond to a "next-sibling" pointer whereas explicit link $h \rightarrow m$ correspond to a "next-in-preorder" pointer.

iii) implicit or explicit pointers in the *Compressed implicit layout* can be pointing to a "next node" in a level far up from the pointer origin. Examples in figure 3.4 are: implicit pointer $k \rightarrow m$ points to a cell that is 3 levels above, and explicit pointer $h \rightarrow m$ points to a *next-in-preorder successor* that is 2 levels above. On the other hand, *any* pointer in LCRS points to a next node at most one level away from the

pointer origin (either to the right-sibling in same level, to the father in the upper level, or to a left-child, which is one level down).

Another aspect of the implicit layout is the economy of memory space, as it uses only one explicit pointer per internal node. In the best scenario with LCRS, we could avoid using pointer "to-father", and only adopt two pointers: LS and RS. Yet, this simplified LCRS representation would take two pointers per node (either internal or leaf nodes). A simplified LCRS representation of a tree with i internal nodes and L leaf nodes, has $2 * (i + L)$ pointers, whereas the Compressed implicit layout for the same tree uses only i pointers.

3.6 A Parallel Tree Layout Transformation Algorithm

In this section we present a parallel algorithm to transform any pointer based tree to its correspondent compressed implicit octree layout. Algorithm 2 shows this *Parallel Tree Layout Transformation Kernel*.

We first inductively present the central idea that permits execution of this algorithm not employing any global synchronization nor any atomic primitive in figure 3.9.

Suppose a pointer based tree $\mathcal{T}_p$ is to be transformed to its equivalent tree $\mathcal{T}_i$ with implicit layout. Let's assume the that final disposal of nodes reflects a *preorder* traversal of $\mathcal{T}_p$. Also, assume that each internal node n of the tree stores variables nl and ni, representing respectively the *number of leaf nodes* and *number of internal nodes* of the subtree. These values are implicit for leaf nodes: $nl = 1$ and $ni = 0$. Values ni and nl of each internal node of $\mathcal{T}_i$ are computed by a *Summarization Kernel*. Each thread in the algorithm picks a target node number tnn representing the order number of a node in the implicit layout. The thread needs to compute the word address of node tnn in $\mathcal{T}_i$. To do this, the thread performs a top-down traversal of $\mathcal{T}_p$, determining the exact path to follow to tnn, computing the total number of internal and leaf nodes in the $prefix$ of node tnn in the implicit layout. The total number of internal and leaf nodes in a prefix is computed respectively with variables ni_t and nl_t.

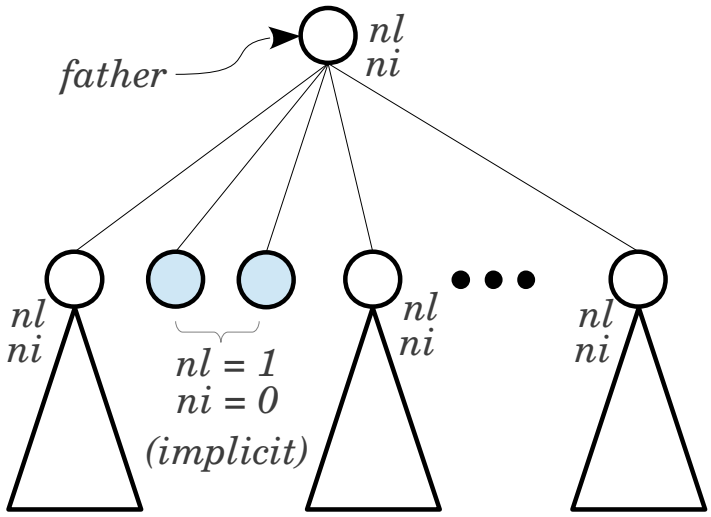


Figure 3.9: A step of the parallel tree conversion from scattered to implicit tree layout, not using atomic instructions nor any synchronization.

The number *tnn* is unique in the implicit order of nodes, formed by the thread global index . The special case of $tnn = 0$ is dealt by a unique thread generating the root node in $\mathcal{T}_i$.

The exact path to node *tnn* is found by a thread as follows: suppose we have traversed the tree up to a current node, let's call it the $father$ node as in figure 3.9 , with variable $prefix$ reflecting the number of nodes (either internal or leaves) in the prefix. If $tnn = prefix + 1$ the current node is the target node, and we have ni_t and nl_t reflecting the correct number of nodes of each type in the prefix. Otherwise, we visit the child nodes from left to right, updating the prefix quantities and applying a similar test to decide whether any of the child nodes is the target node, or if we need to enter a subtree (i.e. move down the current node).

Once the target node is found we can compute its word address in the linear layout by simply adding the number of leaves in the prefix times the size of a leaf, and the number of internal nodes in the prefix times the size of each internal node. The algorithm can then proceed to copy the node from $\mathcal{T}_p$ to $\mathcal{T}_i$ in the correct address.

In the octree, the number of nodes visited by a thread searching for *tnn* is $\mathcal{O}(h)$, where h is the height of the tree, since the maximum number of nodes visited per level is constant (i.e. up to eight nodes). For the octree, this is $\mathcal{O}(logN)$ where N is the number

of leaves. Considering p processing cores and N leaf nodes we conclude that the parallel transformation algorithm is $\mathcal{O}((NlogN)/p)$.

Algorithm 2 Parallel Transform Tree to Implicit Layout.

```
1:  procedure ParGenImplicitTreeKernel()
2:      [precompute values that depend only on tree level.]
3:      __syncthreads();                                   ▷ threads barrier
4:      level ← 0
5:      #define thGlobalIdx \
                (threadIdx.x+blockIdx.x*blockDim.x)
6:      #define thGlobalStep (blockDim.x*gridDim.x)
7:      if (threadIdx.x = 0) && (blockIdx.x = 0) then
8:          [generate root node]
9:      end if
10:     tnn ← 1 + thGlobalIdx                              ▷ target node number
11:     while tnn < nnodes do
12:         // iterate over a number of nodes assigned to thread
13:         parent ← rootIndex                             ▷ root is the first parent
14:         bn ← 0                                         ▷ branch index
15:         nl_t ← ni_t ← prefix ← 0
16:         level ← 0
17:         do
18:             // follow path to node to find its prefix
19:             cnn ← child[parent * 8 + bn]               ▷ curr. node idx
20:             if tnn = (prefix + 1) then
21:                 prefix ← prefix + 1
22:             else if is_a_leaf(cnn) then
23:                 nl_t ← nl_t + 1
24:                 prefix ← prefix + 1
25:                 bn ← bn + 1
26:             else if tnn > (prefix + ni[cnn] + nl[cnn]) then
27:                 ni_t ← ni_t + ni[cnn]
28:                 nl_t ← nl_t + nl[cnn]
29:                 prefix ← prefix + ni[cnn] + nl[cnn]
30:                 bn ← bn + 1
31:             else
32:                 parent ← cnn
33:                 bn ← 0
34:                 ni_t ← ni_t + 1
35:                 prefix ← prefix + 1
36:                 level ← level + 1
37:             end if
38:         while tnn ≠ prefix
39:         address ← (ni_t + 1) * nwords(internal_node_t)
                    + nl_t * nwords(leaf_node_t)
40:         // store tnn node at calculated word address
41:         // in AoS layout, add precomputed level information
42:         copyNodeFields(tnn, address, level)

43:         tnn ← tnn + thGlobalStep                       ▷ thread step
44:     end while
45: end procedure
```

Chapter

4

Barnes-Hut Simulation and Octree Traversal Techniques

In this chapter we propose efficient methods for traversing the implicit octree on GPUs with application in Barnes-Hut simulations.

Our GPU parallel N-Body simulator executes entirely on the GPU if all data fits in the GPU memory. In fact, the results of a simulation step can be streamed to CPU memory via asynchronous data transfers from GPU to CPU memory. These transfers can occur simultaneously with GPU processing of the next step. Modern versions of the PCI-express BUS possess enough bandwidth to transfer the entire set of reference bodies and related information in less time than a simulation step, so that not much overhead is added to the simulation loop when streaming data to the CPU.

Figure 4.1(a) depicts one step on the simulation loop. We use a set of parallel kernels that execute in sequence in each simulation step. The *Build Kernel* takes the set of bodies and related information in SoA (Struct-of-Arrays) layout and produces a pointer based sparse octree representation. For this kernel we used the fast GPU algorithms for sparse octree construction of Burtsher and Pingali (Burtscher and Pingali, 2011). The next kernel, not shown in the picture, summarizes information about subtrees and stores them in the internal nodes of the sparse tree. This step also exposes irregular computation

patterns, and we applied the algorithms described in (Burtscher and Pingali, 2011) and (Nasre et al., 2013). The data produced in internal nodes are used by our *Transformation Kernel* and by the *Tree Traversal Kernel* . Our *Transformation Kernel*, described in section 3.6, is invoked to generate the implicit octree layout in AoS (Array-of-Structs) format, to be used by our *Tree Traversal Kernel* (section 4.2).

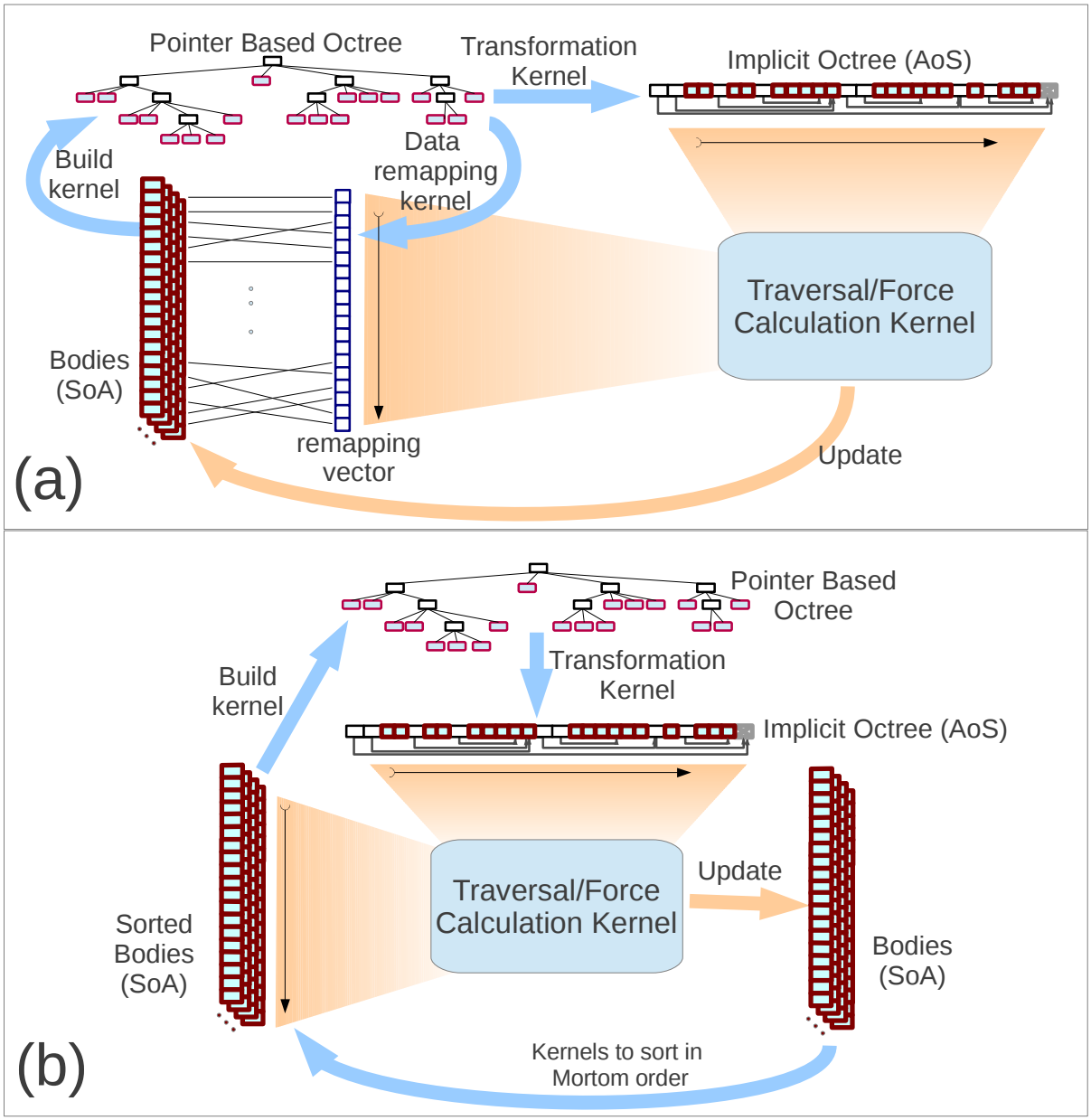


Figure 4.1: N-Body Simulation Step

4.1 Data Remapping Kernel versus Sorting in Morton order

We can observe in figure 4.1(a) or in 4.1(b) that the *Tree Traversal Kernel* accesses data in the array of bodies and traverses the implicit tree structure. The implicit octree structure is arranged *preorder*, which is equivalent to a linear displacement of nodes in Morton order. Since the reference bodies array is disposed in a given initial order, and as the wide warps of threads (section 4.3) coherently traverse the tree in groups, it would be efficient to layout the reference bodies in the same (Morton) order as the implicit tree is disposed. This way we exploit data locality in the Morton order, less divergent paths are followed in the tree during traversals, and less extra work is performed as the groups visit the same subtrees. One way to do this order reconciliation is shown in figure 4.1(b), by using an extra kernel to sort the bodies in Morton order at each simulation step. Another way, viewed in figure 4.1(a), is to use a *Data remapping kernel* to generate a remapping vector to be used by the *Tree Traversal Kernel*. This way the *Tree Traversal Kernel* consults this vector to index the reference bodies in the proper order. We have experimented with both methods, and found that the remapping vector scheme is faster. The reason is that the *Data remapping kernel* is much faster than sorting all reference body structures. On the other hand, the entire data remapping vector is only referenced in the outer loop of the simulation (line 7 of algorithm 3), so the extra cost is only n references in the simulation step, where n is the number of bodies.

4.2 Parallel Tree Traversal and Forces Calculation

4.2.1 Parallel Tree Traversal Algorithm

The GPU parallel Barnes-Hut N-Body forces calculation method is shown in algorithm 3. The implicit octree is traversed to evaluate forces on reference bodies applied by particles (tree leaves) or particle clusters (internal cells). Our GPU kernel uses the persistent

threads (Gupta et al., 2012) model. Normally each thread loads a reference body and traverses the tree computing the forces applied by other particles. We are able to vary the thread granularity in the method, configuring each thread to work with sets of reference bodies per traversal. The Barnes-Hut tree traversal generates irregular execution patterns even for the unit work per thread configuration, as the paths followed by the code and data accessed in the tree depend on the reference particles and on the particle distribution. We try to minimize these effects by constraining execution flow in the code and tree traversal in large groups of threads called wide warps. This software simulated wide warp (SWW) (Nunan Zola et al., 2014) technique is described in section 4.3. Essentially, each physical warp simulates a wide virtual warp simply by making each thread simulate the action of a set of virtual thread steps. Common steps that execute with scalar shared data can be unified inside a thread simulating SISD (Single Instruction Single Data) phases. Steps can be also replicated to simulate wide SIMD (Single Instruction Multiple Data) phases.

Algorithm 3 uses a simplified notation for compactness. We denote the reference to a data field F in the internal node n of the tree as $node_g(n, F)$. Variables stored in global memory are subscripted by letter g. Non subscripted variables are local scalar variables, generally stored in registers. Wide warp sections of the code are denoted by *#pragma unroll* $\$wwarp$ as detailed in section 4.3. Variables indexed by $[\$w]$ are vector variables maintained in registers or shared memory, representing variables that replicate per virtual thread of the simulated wide warp.

Algorithm 3 N-Body Forces Calculation with Implicit Tree using SWW

```
1:  procedure ParTraverseImplicitOctreeKernel()
2:      k ← thGlobalIdx * $WWidth                          ▷ using Simulated Wide Warps
3:      while k ≤ nbodies_g do
4:          #pragma unroll
5:          $wwarp {                                       ▷ get data remapping "wide" index
6:              // fill the i[$w] (an index vector) formed from k
7:              i[$w] ← dataRemappingIndex_g[k + $w]
8:              refX[$w] ← posx_g[i[$w]]                   ▷ load ref bodies
9:              refY[$w] ← posy_g[i[$w]]                   ▷ using index vector
10:             refZ[$w] ← posz_g[i[$w]]
11:             ax[$w] ← ay[$w] ← az[$w] ← 0.0f             ▷ initialize
12:         }
13:         n ← nwords(internal_node_t)      ▷ node 0 is root (of entire tree), we will skip it
14:         while n ≤ end_of_tree do                       ▷ all threads traverse entire tree
15:             mass ← node_g(n, MASS)
16:             #pragma unroll                             ▷ compute distance squared
17:             $wwarp {                                   ▷ ... plus softening
18:                 dx[$w] ← node_g(n, X) − refX[$w]
19:                 dy[$w] ← node_g(n, Y) − refY[$w]
20:                 dz[$w] ← node_g(n, Z) − refZ[$w]
21:                 dist_sq[$w] ← dx[$w] * dx[$w] + dy[$w] * dy[$w] + dz[$w] * dz[$w] + epssq_g
22:             }                                          ▷ COV is a pre-calculated value
23:             COV ← node_g(n, COV)        ▷ speculative read, COV is Cell Opening Value
24:             if is_internal_node(n) then
25:                 skip ← node_g(n, SKIP_LINK)
26:             end if
27:             if is_leaf_node(n) ||
                    __all_$wwide( dist_sq[$w] ≥ COV )
28:             then                                       ▷ all threads agree in skipping subtree
29:                 #pragma unroll $wwarp {                ▷ OR the node is a body
30:                     tmp ← rsqrtf(dist_sq[$w])          ▷ Calculate forces
31:                     tmp ← mass * tmp * tmp * tmp
32:                     ax[$w] ← ax[$w] + dx[$w] * tmp
33:                     ay[$w] ← ay[$w] + dy[$w] * tmp
34:                     az[$w] ← az[$w] + dz[$w] * tmp
35:                 }                                      ▷ OR the node is a
36:             else                                       ▷ enter subtree
37:                 skip ← n + nwords(internal_node_t)
38:             end if
39:             if is_internal_node(n) then
40:                 n = skip                               ▷ OR override skip
41:             else
42:                 n ← n + nwords(leaf_node_t)            ▷ implict skip
43:             end if
44:         end while
45:         #pragma unroll $wwarp {
46:           [ ...and using i[$w] index vector and "wwarped" variables, update ... ]
47:           [ ...velocity and save computed acceleration in global mem. ] }
48:     end while
49:     k ← k + thGlobalStep * $WWidth                     ▷ "wide" step
50: end procedure
```

4.2.2 Cell Opening Criterion

Particle clusters are represented by subtrees. The *Cell Opening Criterion* is evaluated during traversals and determines if a subtree will be visited or if its effects in the reference bodies are approximated using summarized values stored in the subtree's root node.

We used the standard Barnes-Hut (Barnes and Hut, 1986) *Cell Opening Criterion:* $d > \frac{l}{\theta}$, plus a softening parameter, where d is distance from a body to the cell's center of mass, l is the cell diameter (i.e. octant's side length) and θ is the opening angle parameter, where $0.0 \leq \theta \leq 1.0$, typically $\theta = 0.5$. To simplify this calculation we resort to the standard procedure of pre-computing most of the expression, avoiding use of square root and division in each *opening test*, making comparisons of *squared* values. Pre-computed *Cell Opening Values (COV)* are usually held in a table, indexed per level, instead we store a corresponding *COV* at each internal node in the implicit octree. By doing so, we avoid the need to store the internal node's *tree level*, this way incurring in no extra cost in terms of occupied memory space and memory accesses per node. The cost of accessing a table is eluded.

4.3 Simulated Wide-Warp

Thread divergence occurs when control flow reaches a conditional branch where threads take different execution paths. Warps are implemented with SIMT instructions that automatically disables and enables threads to execute each path at a time. This serialization of divergent execution flows causes underutilization of SIMT hardware as we can see in figure 4.2(a). Points of execution re-convergence are automatically determined an optimizing code generator.

A uniform branch takes place when execution flow reaches a SIMT branch instruction where all threads follow the same path. As seen in figure 4.2(b) warps execute faster when performing uniform branches, as either one of the alternative blocks of instructions are executed.

Wide thread warps can be easily simulated for a block of instructions by simple repetition of the SIMD instructions and using vector variables, if the block contains no branches or if the execution of branches within a the block is always uniform. Figure 4.2(c) shows this situation. This can be done in software simply by using the compiler provided *#pragma unroll* directive and by providing an extra *for loop* preceding a command block. By annotating scalar variables as vectors and declaring these variables as register arrays they can hold the different values needed by each virtual warp thread. CUDA compilers can transform indexed variables to use registers if register arrays are used and if indexes are constants. By using preprocessor defined macros to hide the *for loop* needed by the unroll directive the programmer can define uniform wide warp simulation with little extra annotation in the program.

4.3.1 Wide Warp Directives

Using preprocessor defined macros permits little extra annotations on a GPU kernel that uses simulated wide warps. Figure 4.3 depicts the *#pragma unroll $wwarp* directive necessary to determine that a *command sequence* in the kernel is to be executed with SWW. The unrolling simulates a sequence of wide SIMD instructions. We need to *#define* a macro value $*wwidth* that is used by the unroll. The simulated large warp has $*wwidth*$*$ *warpSize* thread lanes, where *warpSize* is the physical warp width. We use the term *wwarp N* to mean a simulated wide warp with N times the physical warp width. For example, *#define $wwidth 5* defines that subsequent $*wwarp* unrolls simulate $wwarp = 5$ execution. This way we can have a different virtual warp width per kernel.

Although it could be considered over-annotation, the need for the *#pragma unroll* keywords at each *$wwarp* block is a reminder of the underlying unrolling mechanism.

We can also see in figure 4.3 that SISD (Single Instruction Single Data) execution can be obtained by selecting a single thread in CUDA. Evidently, all statements not annotated in the ways described are executed with a normal hardware warp of warpSize lanes.

4.3.2 Warp Wide Variables, Scalar Variables, Loading and Storing

Algorithm 3 uses simulated wide warps, with block annotations in various parts. Taking the command block in lines 34-38 for example, variable *tmp* can be scalar as its value is only needed locally. In fact it could have been declared inside the block to enforce this case. Variable *mass* is also scalar, although used in a larger scope, as its value can be shared by all wide warp threads. Sharing variables between threads in the wide warp is possible when their values are unified across the wide warp. Some scalar variables can have unified values which can be identified by the programmer mainly when no divergence occur in a warp. For example, scalar variable n has a unified value across a unified wide warp. Scalar variable *tmp* does not have a unified value, but can still be scalar because the unrolling replication mechanism permits, and because of its local scope. The use of scalar variables simplify programming in some situations, e.g. on load/store phases and on conditional expressions like $is\_leaf\_node(n)$ test on line 27. Besides, the use of scalars reduces register pressure (section 4.3.4) and is efficient, as the value can be defined once in a register and used by all virtual warp threads.

Warp wide variables are annotated with [$\$w$], and must be declared as register arrays of $\$wwidth$ size. In the referred block, variables $dist\_sq, ax, ay, az, dx, dy, dz$ are warp wide variables (declarations not shown in the algorithm). Inside a thread these variables are normally operated as scalars, the [$\$w$] can be just regarded as a type annotation, meaning that each virtual thread in the wide warp has a private copy of the variable. The simulated virtual wide SIMD instructions operate on all private copies. Scalar variables could have been located in shared memory but we have chosen to maintain all SWW related variables in registers for various reasons: addressing is simplified, for efficiency, and because the register file in current GPUs are much larger than shared memory space. Scalar variables occupy one copy per physical lane, warp wide variables ocuppy $\$wwidth$ copies per physical lane.

Small programming difficulties remain with warp wide variables as they are in fact vectors, these are related to loading or storing from memory, as these instructions will move $\$wwidth$ data elements at a time. Some programming frameworks provide templates

for these warp-centric operations. We have chosen to explicitly expose these details in the program. In line 7, for instance, on loading $i[\$w]$ from vector $dataRemappingIndex_g$ at global memory, we need to add the global index k to the $\$wwarp$ index w, to address the global vector. This loads contiguous values from memory to each $i[\$w]$ scalar in the physical thread. For the small $\$wwidth$ values we experimented with, contiguous loading performed better than a strided version, as we used cached GPUs which are less sensitive to uncoalesced accesses. Likewise, lines 51-53 describes in pseudo code the storing of warp wide variables using $i[\$w]$ as an index. This will in fact store a vector of width $\$wwidth$ of contiguous values at each $i[\$w]$ destination.

Another issue related to loading vector variables with unrolling directives is associated with the possibility of invalid indexing of global arrays. We have dealt with this problem by adding extra leaf cells or bodies (*ghost cells* with null mass) in global arrays, so that the total number of cells per global vector is a multiple of $\$wwarp$. This way we incur in very little extra work during traversals, but we avoid boundary condition tests for indexing the global arrays and possible thread divergence caused by these tests. Using these *ghost* leaf cells introduces no final effects on the simulation results.

4.3.3 Thread Divergence Control and Work Efficiency

Fully convergent wide warp execution of threads (i.e. no divergence, as in fig 4.2-c) must be guaranteed by kernel algorithms. In the case of the BH tree traversal/forces calculation kernel, divergence can occur as each thread must decide whether to visit a the current subtree for calculations or use the summarized values in the internal node (root of the current subtree). To constrain for a uniform execution all threads in the simulated wide warp must agree to take the the same traversal path on the tree. Each physical thread evaluates the Cell Opening Criterion (line 23 in algorithm 3) for all virtual threads it is responsible for, using a wide variable. The final agreement is unified by all threads in the physical warp using our $\_\_all\$wwide()$ macro/template, which is a wide expansion of the $\_\_all()$ SIMT primitive. If any virtual thread in the wide warp decides to enter a subtree, all others will follow.

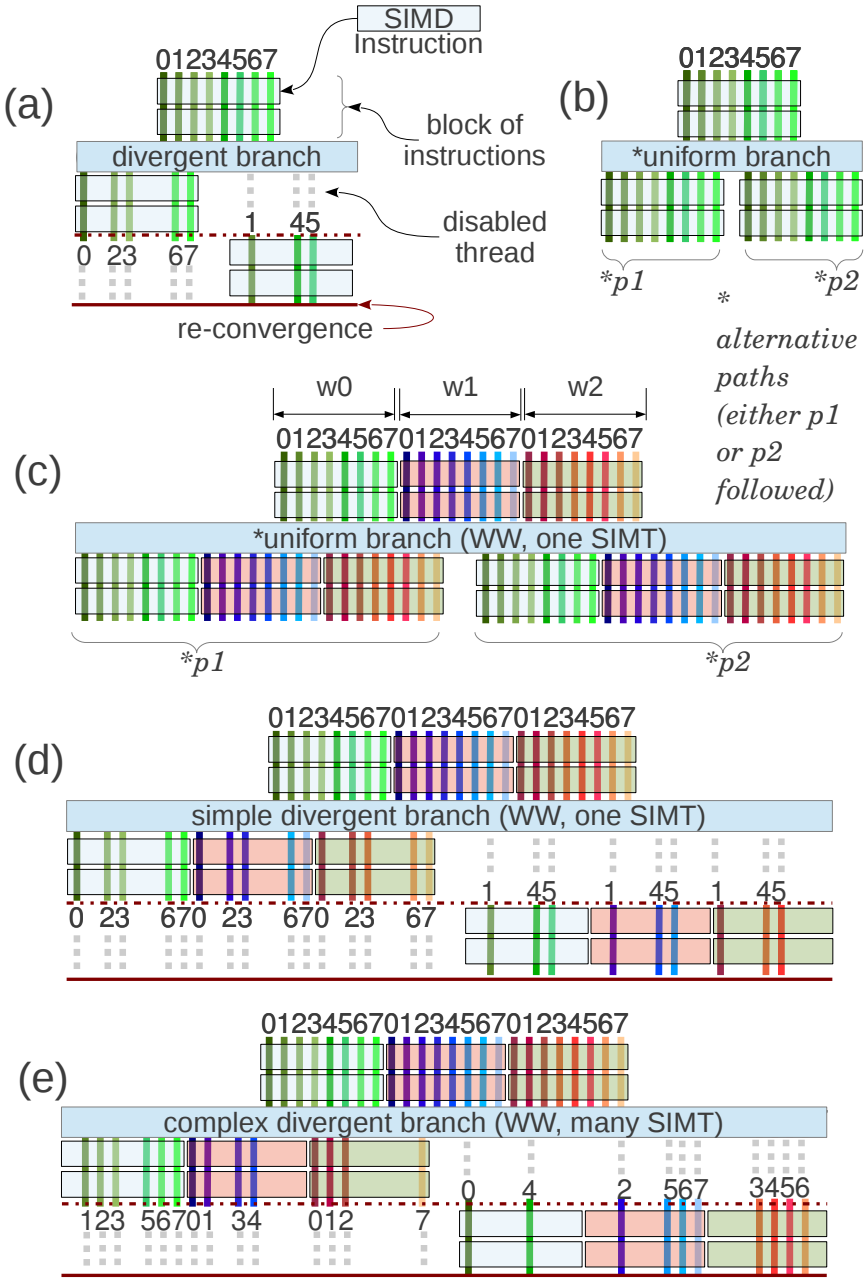
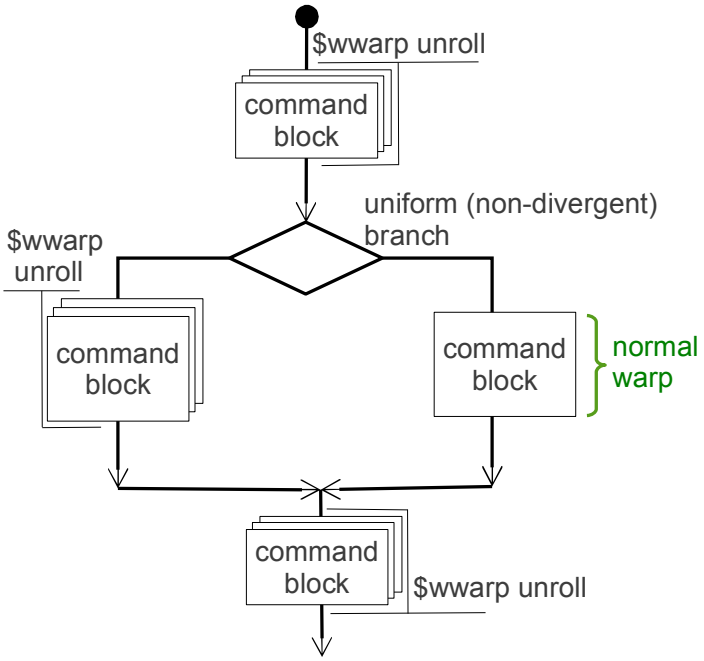


Figure 4.2: Hardware warps and Simulated Triple Wide Warps (WW): (a) hardware divergent warp; (b) hardware fully convergent warp; (c) fully convergent simulated wide warp; (d) simple divergent simulated wide warp; (e) fully divergent simulated wide warp.

Generally, predicates reduced by warp primitives $\_\_all()$ or $\_\_any()$ that unify thread execution in physical warps need to be expanded when applied in a wide warp block of code or expression. Assume that a predicate p using a wide variable v is denoted by $p|v$. The wide warp expansion of the operation $\_\_all\$wwide(p|v)$ is $\_\_all(\ p|v[0]\ \&\&\ p|v[1]\ \&\&\ \ldots\ \&\&\ p|v[\$width-1]\ )$, as in



Annotations

Simulated Wide Warp (wide SIMD)	**#pragma unroll $wwarp** { <*command sequence*> }
Single Thread Execution (SISD)	If(threadIdx.x == N) { <command sequence> }
Normal Warp (hardware SIMD)	<command sequence>

Figure 4.3: (Left) mixed regular warps and $wwarp unroll applied to different paths and code regions. (Right) program annotations

line 27 of algorithm 3. Operation $\_\_any\$wwide(p|v)$ needs to be wide expanded as $\_\_any(\ p|v[0]\ \|\ p|v[1]\ \|\ ...\ \|\ p|v[\$width-1]\ )$. These expressions are evaluated per physical thread. An interesting situation happens in line 27 of algorithm 3. The first part of the expression is executed as a regular warp, because this part of the predicate uses the unified scalar value n, while the second part uses an $\_\_all\$wwide()$ primitive and the predicate uses the wide warp variable $dist\_sq$.

While the fully uniform wide warp traversal incur in extra work as all virtual threads are forced o traverse the tree in a convoy pattern, the effects are benign as this produces more precise calculations, as more subtrees are visited. On the other hand, for all threads in the wide warp most global memory accesses to the implicit tree structure produces a hit in the data cache and are mostly coalesced. This way the overhead is mostly absorbed in processing elements that would otherwise be idle (disabled) if divergent execution is allowed. Figure 5.2 compares $wwarp = 1$ and $wwarp = 4$ showing an increase in the average number of particle-particle and particle-cells interactions for the problem sizes analysed. A minimum number of interactions is obtained if we allow divergent execution, but at a cost in performance. Our experimental evaluation (section 5.4) showed that

$wwarp = 5$ produced higher speedups on Kepler GPUs while $wwarp = 4$ was fastest in Fermi cores.

4.3.4 Register Pressure and Maximum Number of Resident Threads

Typical GPU kernels use many CUDA threads to hide memory access latency, determining cost in terms of register usage. GPU multiprocessors have a large register file that are equally divided by the threads of an active (resident) thread block. Table 5.1 shows the hardware properties and limits of current GPU multiprocessors.

In SWW we have an increase in register pressure, if we want to maintain variables in registers during the lifetime of a thread in a resident thread block. On the other hand, the register pressure with t virtual threads in the simulated wide warps model is lower than with t physical threads, because virtual threads can share register variables. Temporary registers used by the compiler, and some program variables, are naturally shared by virtual threads by the mechanism of unrolling. As an example, consider lines 30-34 in algorithm 3. The *tmp* variable has been declared as a single scalar taking less register space. On the contrary, using a wide (vector) variable produces less data dependency, allowing more instructions to be issued in parallel. This is a trade-off between register space usage, instruction level parallelism, and latency hiding by the the maximum number resident threads per multiprocessor (table 5.1), which can be analysed in SWW kernels. Recent GPUs allow more registers per thread and higher IPC, bringing a wider experimentation spectrum for GPU kernels using SWW.

In the traversal kernel we use the persistent threads model (Gupta et al., 2012) where the maximum number of threads is equal to the maximum number of resident threads supported by the hardware. The situation is similar if applying SWW with general *many-small-threads* model, since any given GPU model will support a maximum number of resident threads per multiprocessor (MP). By using SWW virtual threads we accomplish the effect equivalent to a higher number of physical resident threads and consequent hidden latency. For example, consider our traversal kernel, in figure 5.3 using $wwarp = 5$:

in current GPUs the maximum number of physical threads per block is 1024, and and we can maintain up to 2 resident thread blocks, for a total of 2048 resident threads in the multiprocessor (table 5.1). If allowed by the size of register space, the number of simulated threads can be up to $wwarp = 5$ times higher. On the other hand, if register usage dictates lower thread occupancy, an equivalent latency hiding effect of maximum number of virtual threads in the MP can be earned, using SWW and $(1/\$wwidth)$ physical threads.

Traversal Kernel	$wwarp = w$					
	1	2	3	4	5	6
w virtual threads (SWW)	18	29	40	51	62	73
w physical threads	18	36	54	72	90	108

Table 4.1: R_d register demand for w threads

Traversal Kernel						
Resident physical warps	32	64	96	128	160	192
Physical threads	1024	2048	3072	4096	5120	6144
Rd	18432	36864	55296	73728	92160	110592

Table 4.2: R_d Register Demand without SWW

Traversal Kernel	$wwarp = w$					
	1	2	3	4	5	6
Virtual threads	1024	2048	3072	4096	5120	6144
R_d	18432	29696	40960	52224	63488	74752

Table 4.3: R_d Register Demand for Traversal Kernel using SWW with 1024 physical threads (32 resident physical warps)

By raising wide warp $\$wwitdh$ we save registers per virtual thread. Suppose a kernel with t physical threads with $wwarp = w$, each thread using s scalar variables and n \$warp-wide (vector) variables. The total register demand R_d is given by: $R_d = t*(s+w*n)$. Not using SWW, with $t*w$ physical threads, de register demand builds up to: $R_d = t*w*(s+n)$. Taking our traversal kernel in figure 3, it uses $s = 7$ scalar variables and $n = 11$ \$wwarp variables. Table 4.1 depicts register demand for w threads with or without using SWW.

As we can see, $wwarp = 6$ can not be used in current GPUs unless we allow register spilling, as we reach the maximum registers per thread (table 5.1). We can visualize register demand for the traversal/forces calculation kernel in table 4.3 for various *wwarp* widths using 1024 physical resident threads (maximum on Fermi GPUs as seen in table 5.1). The register demand for $wwarp = 5$ can be accommodated in Kepler GPUs, almost reaching the maximum number of registers in the MP, while in Fermi GPUs we have to lower the number of physical threads, since the register file is smaller. In table 4.2 we can see that only using regular warps (no SWW) we underutilize register space, as we reach a maximum number of physical threads per MP with a low utilization of the register file. We can also observe on these tables that the number of virtual threads that can be obtained using SWW is much higher than only using regular physical warps.

Chapter

5

Experimental Results on tree traversals and techniques in the context of Barnes-Hut simulations

In this chapter we present experimental results on the application of implicit octree layouts and our SWW techniques for tree traversals in the context of Barnes-Hut simulations.

Section 5.1 discusses related literature in this context, section 5.2 presents our experimental methodology, section 5.3 describes the computing systems and software used in our implementation and experiments. Finally section 5.4 present the results of our experiments.

5.1 Related Work

An efficient implementation of the quadratic algorithm on GPUs (Nyland et al., 2007) for N-Body gravitational forces calculation was presented by Nyland *et al.* As we pointed out earlier, this direct method is faster for small number of particles. In section 5.4 we have compared the running times of this method with our Barnes-Hut version.

Burtscher and Pingali developed efficient methods (Burtscher and Pingali, 2011) for generation and traversal of pointer based octrees on GPUs in Barnes-Hut gravitational

simulations. Recently they have incorporated newer techniques (Nasre et al., 2013) in their GPU parallel Barnes-Hut implementation, some applicable to other irregular algorithms. Our present work builds upon these techniques, as we have used their efficient GPU kernels to construct pointer based octrees, and to summarize information about subtrees in internal nodes. Our parallel algorithm presented in section 3.6 can be efficiently applied to any pointer based tree, including trees with dynamically allocated (scattered) nodes, to generate the implicit layout described in section 3. We also show that, the extra time taken in the layout transformation is amortized in the forces-calculation phase (equivalent to do N traversals of the tree, in parallel), as our implicit layout allows for the tree traversal to be done without using a stack data structure. Further speedup can be obtained in the traversal phase, by application of our *simulated wide warp* technique described in section 4.3.

Another recent N-body gravitational simulator (Bédorf et al., 2012) that runs entirely on GPUs was presented by Bedorf *et al.* This code also uses sparse pointer based octrees but with a different structure. Their trees are generated and traversed top down, per tree level. The tree layout process needs to sort all leaf (body) nodes in Morton order. This phase, together with subsequent data movement necessary to lay out the nodes per level takes roughly 66% of tree generation time. One likely advantage of sorting leaf cells in Morton order is that it possibly improves cache efficiency during tree traversals. Our approach avoids the sorting phase using a permutation index, as does the base method (Burtscher and Pingali, 2011) of Burtsher and Pingali. Our three transformation phase also includes the data movement to the implicit layout. The tree transformation lays out the nodes in "pre-order", using the previously constructed pointer based octree. This order is equivalent to the Morton order of all tree nodes. This way our traversal phase also has improved cache efficiency. One might think that there is a possible downside, as using the permutation index to access reference (body) cells takes two memory accesses, one to the index and another to the cells, instead of just one access in the sorted method. Nevertheless, the cost of sorting body cells is much higher, as accessing the permutation index is $\mathcal{O}(N)$, done in the outer loop as can be seen in the traversal algorithm 3, whether

the sorting is $\mathcal{O}(NlogN)$. As shown in section 5.4, generating the scattered tree and transforming to our implicit layout is up to 4.55 times faster. Bedorf *et al.* also compared the performance of their implementation to the baseline version 1.0 work of Burtsher and Pingali, with fairness reservations to the fact that their gravitational tree-code a uses a different *multipole acceptance criterion (MAC)*, which is the condition tested to decide whether to enter a subtree during traversals. Making these same reservations on MAC differences we also present, in section 5.4, a comparison of our methods running on recent GPU models.

The applicability of large warps (Narasiman et al., 2011) to GPU hardware microarchitectures was investigated by Narasiman *et al.* using a GPU simulator. The proposed architecture can dinamically assign threads to form sub-warps, which are grouped into large warps (LW). The average speedup achieved in various simulated parallel benchmark applications was 19.1%, Barnes-Hut not among them, with a maximum LW width of 256 threads. Higher widths showed not advantageous. Our software simulated wide warp (SWW) technique used static assignment of threads to warps, and also static formation of the wide warp from regular SIMT instructions. The best performance was obtained with a SWW equivalent of 160 threads on Kepler GPUs (i.e. grouping 5 SIMT instructions) and 128 threads on Fermi GPUs (grouping 4 SIMT instructions). The SWW Barnes-Hut implementation shows a maximum speedup of arround 200% over the same parallel implementation using a regular warp. We credit this result to two factors: traversing the tree with a larger working set, formed by grouping small working sets; and all threads in the SWW walking in a coherent memory access pattern.

The work in (Zhang et al., 2010) and (Zhang et al., 2011) proposed a framework to eliminate thread divergence at runtime and the effects of non coalesced accesses to data by automatically performing data remapping. Analysed benchmark applications shows higher speedups of up to 2X on two analysed applications on GPUs C1060, which have compute capability 1.3 which are more vulnerable to irregular accesses, since the data caches are not present in this GPU architecture. The maximum obtained speedup on a cache based GPU GTX480 was 1.21x for the applications studied. This GPU which has

compute capability 2.0, meaning the presence of hardware L1 and L2 caches, and less problems with non coalesced data accesses.

5.2 Experimental methodology

Running times were obtained by instrumenting the source code with CUDA toolkit time measurement functions. All input was generated with a Plummer distribution of particles for the different input sizes shown. For each experiment we measured 100 simulation steps to take the average per step. For compatibility or ease on comparisons, we also used log scales in some graphs, as in the references.

5.3 Systems and compilers

We have executed our experiments on two different systems with different cache based GPU architectures, namely Fermi and Kepler. Table 5.1 summarizes GPU characteristics which are important to our developments. We have used an NVIDIA GPU GTX480 (Fermi) on an Intel i5-3330 CPU @ 3.00GHz, running a GNU/Linux 3.2 i686 operating system kernel, and Ubuntu 12.04 LTS distribution. Another system, using NVIDIA GPU GTX680 (Kepler) on an Intel i7-3820 CPU @ 3.60GHz, running a GNU/Linux 3.2 x86_64 operating system kernel, and Ubuntu 12.04 LTS distribution. Both systems used NVIDIA Cuda compilation tools, release 5.0, specific for each operating system configuration.

GPU Model	GTX 480 (Fermi)	GTX 680 (Kepler)
Device Capability	2.0	3.0
No. of Multiprocessors (MP)	15	8
CUDA Cores/MP	32	192
Total CUDA Cores	480	1536
Max. threads per MP	1536	2048
Max. threads per block	1024	1024
Max. registers per MP	32768	65536
Max. registers per thread	63	63
Max. resident warps per MP	48	64
Peak IPC	2	4
L2 cache size	768 KB	512 KB

Table 5.1: GPU Models

5.4 Experimental Results

The upper chart in figure 5.1 shows the execution times of each kernel in the simulation, running on GTX480, with wide warp 4, which performed best on this architecture. The tree traversal phase with forces calculations dominates the time in a simulation step, typically taking at least one order of magnitude more time than any other kernel, even after applying our acceleration techniques and obtaining speedups discussed bellow for this phase. We conclude that the tree building/transformation phases are still not the bottleneck on a simulation step and, according to Amdahl's law, the exploration space for any further optimization would still aim to accelerate the tree traversal/forces calculations phase. Also, the time spent on the data transformation phase to the implicit layout is amortized by the time reduction in the tree traversal kernel due to our optimization techniques.

The lower graph in figure 5.1 compares execution times (left axis in log scale) and speedup (right axis, linear scale) of the traversal kernel with wwarp=4 over GPU parallel Barnes-Hut implementations in references (Burtscher and Pingali, 2011) and the Bonsai BH code (Bédorf et al., 2012). Using our implict layout and SWW we achieve a maximum

speedup of 224% compared with (Burtscher and Pingali, 2011) and 458% with respect to parallel version of reference (Bédorf et al., 2012).

Figure 5.2 compares the amount of particle-particle and and particle-cell interactions with reference (Bédorf et al., 2012) and shows that the amount of work increases with the wide warp width as explained in section 4.3. Nonetheless, this extra work is benign as it enhances precision in calculations. On the other hand, this extra work is executed by the exceeding computing power by using extra cores present in SIMT warps using SWW. These cores would otherwise be unused due to SIMT divergence or latencies of memory access. This way, SWW and the use of the implicit tree seems to match the algorithm characteristics to the full flops (floating point operations per second) capability of throughput hardware.

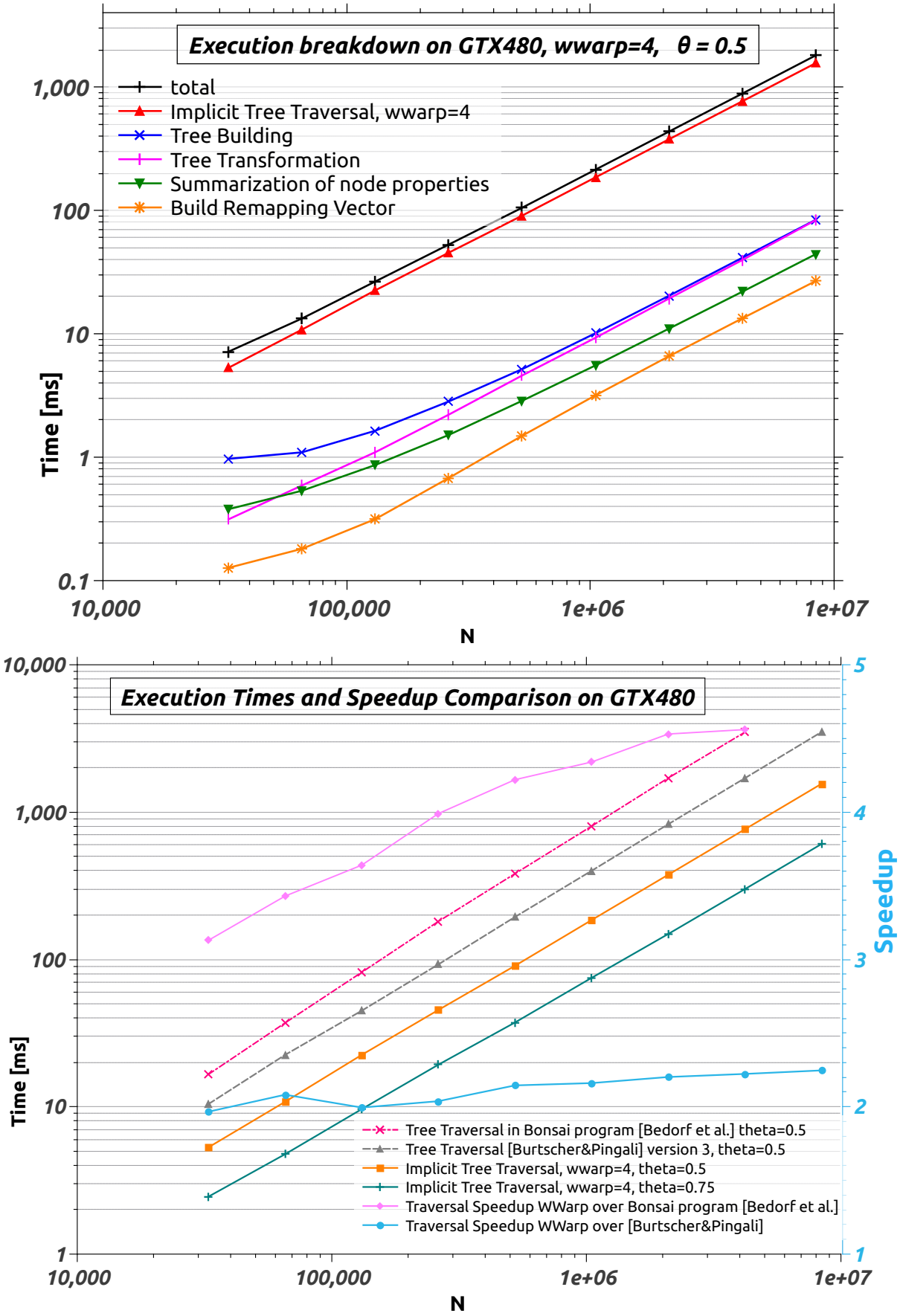


Figure 5.1: (Upper) execution times of each kernel in the simulation, running on GTX480 with wide warp 4. (Lower) execution times (left scale) and speedup (right scale) of the traversal kernel with wwarp=4 compared to references (Burtscher and Pingali, 2011) and (Bédorf et al., 2012)

A comparison with the GPU parallel quadratic algorithm (Nyland et al., 2007), from NVIDIA SDK, and with Barnes-Hut version 3.0 (lonestar GPU suite v1.0) of Burtscher and Pingali (Burtscher and Pingali, 2011) on kepler GPU (GTX 680) architecture is shown in the upper graph of figure 5.3. The direct method [15] is faster for small number of particles, which is expected.

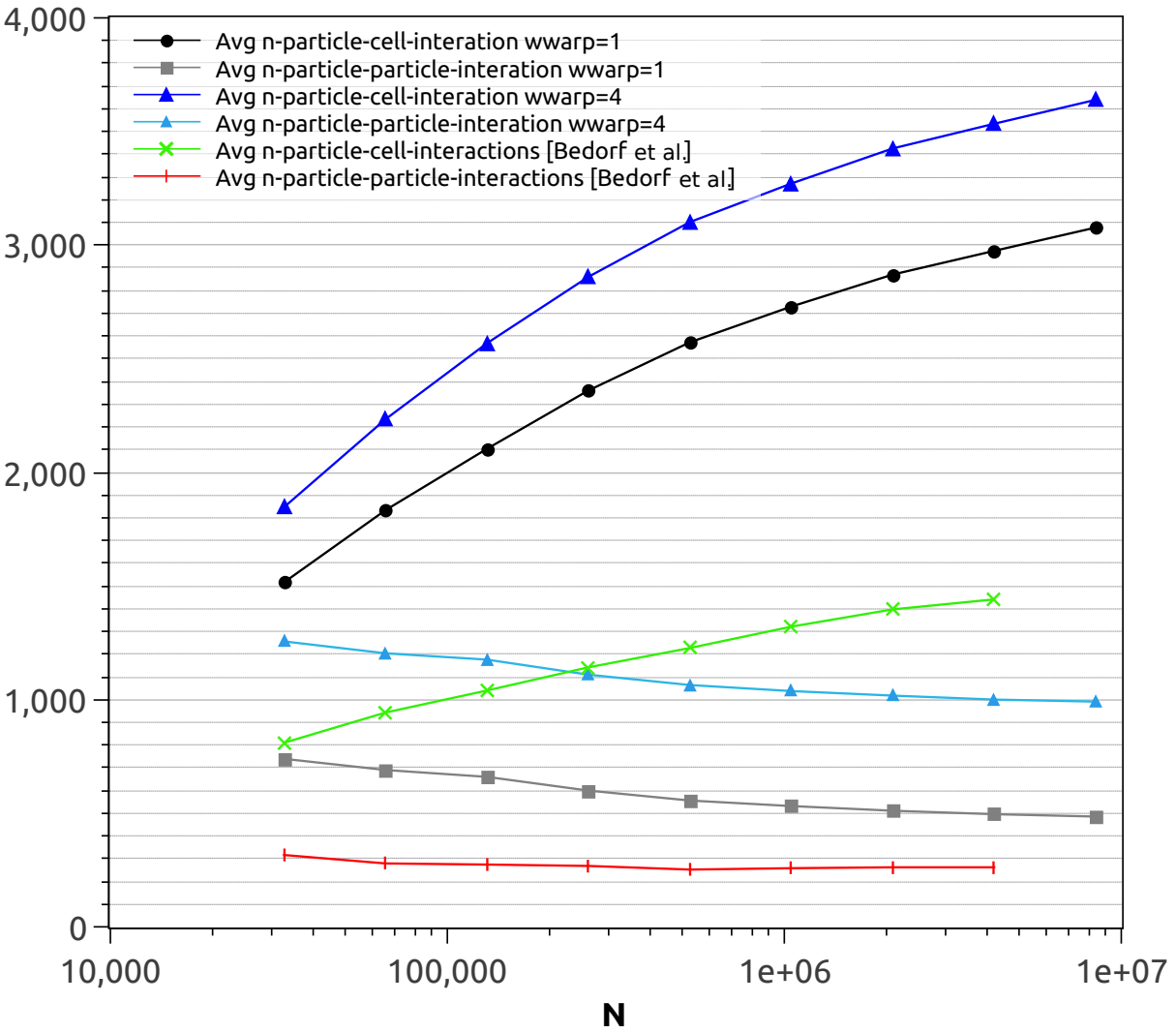


Figure 5.2: Amount of particle-particle and and particle-cell interactions increases with the wide warp. Also shown, amount of work with the implementation in reference (Bédorf et al., 2012)

We have experimented with different SWW widths to tune the Barnes-Hut traversal kernel for a given GPU. For example, we have these experiments documented for GPU GTX680 in the lower chart of Fig. 5.3. These results confirm our expectation that that higher speedups are obtained with wider SWW versions. Using wwarp=1 we can see

the effect of the implicit layout alone (wwarp=1 means using regular physical warp of 32 cores or threads per warp in current hardware). The use of our acceleration structure (the implicit octree in AoS layout) produces around 40% of improvement in the traversal kernel, not using SWW, when compared to the version 3.0 of the algorithm in reference (Burtscher and Pingali, 2011) which uses sparse octrees in SoA layouts, both running on this Kepler architecture. Using the implicit layout and applying simulated wide warps from virtual width 1 to 4 raises the speedup proportionally up to 268% over the parallel GPU version of (Burtscher and Pingali, 2011). We obtained a maximum speedup on Kepler architecture with wwarp=5, higher SWW widths were not beneficial. This can be explained by the fact that the amount of work also raises when we use higher SWW widths (experiment of figure 5.2), and we approach the peak performance (of GFLOPS or instructions per second) of the throughput processor.

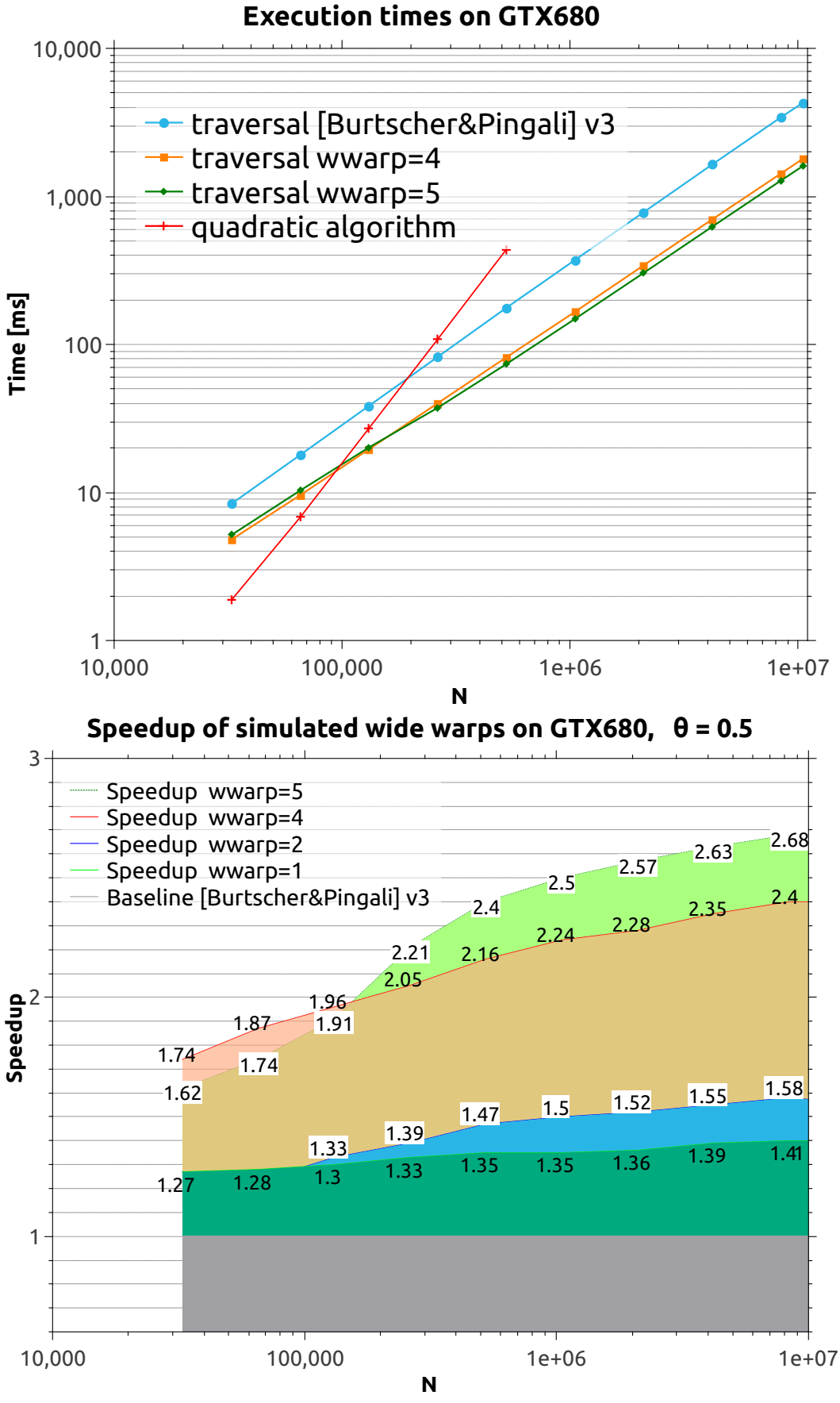


Figure 5.3: (Upper) Comparison of the quadratic algorithm with Barnes-Hut version 3.0 of Burtscher and Pingali (Burtscher and Pingali, 2011) on kepler GPU. (Lower) speedups for various wide warp widths on Kepler architecture

Chapter

6

A GPU Parallel Algorithm for Direct Generation of Compressed Implicit Octrees

In this chapter we present a new efficient parallel algorithm that directly generates entire octrees at once, as a massively parallel method. This algorithm differs from previous ones as it utilizes massively parallel primitives such as prefix-sums (Blelloch, 1989), stream/vector compaction, sorting, and scattering, as opposed to traditional methods that incrementally traverse and insert nodes on partially constructed trees using many threads. We believe that, while efficient, this methodof processing data is in conformance with enormous parallel processing power present in modern *throughput computing systems*.

We present parallel primitives used in building the algorithms of this chapter in section 6.1. Section 6.2 describes the general ideas on building an algorithm that permits producing an entire octree at once using massively parallel steps. Our new algorithm for direct generation of compressed implicit octrees is defined in section 6.3. We present in section 6.4 the experimental results obtained with the use of our algorithm.

6.1 Massively Parallel Primitives

Construction of parallel algorithms, or even providing efficient parallel implementations of existing sequential programs, is a challenging task. The use of parallel primitives have played central role to facilitate building and understanding parallel algorithms. In this section we present some of these important building blocks.

Some of our kernels are implemented in an ad-hoc manner, in general this chosen for efficiency, or if the problem is well adapted for programing in an array of threads style. This was the case in the definition of algorithm 2, or in the tree traversal kernels of section 4.

In the implementation of the parallel algorithm of this chapter we have used both programming styles, depending on the specific part of the algorithm. The parallel primitives for *all-prefix-sums*, *stream compaction* tasks and *sorting* are used in the definition of the algorithms in this chapter. We present a brief description of these primitives in the next sections 6.1.1 , 6.1.2 and 6.1.3 respectively.

6.1.1 All-prefix-sums

Operation all-prefix-sums is one of the most important primitives in parallel algorithms. Many algorithms are efficiently built using this operation, for example radix sorting, minimum spanning tree, quicksort, etc. Many other important basic parallel operations can be implemented with all-prefix-sums such as stream compaction and load balancing phases on algorithms.

Prefix-sums primitives are also called scan. (Blelloch, 1989) defines scan as an operation that takes a binary associative operator $\oplus$ with identity i, and an ordered set $[a_0, a_1, ..., a_{n-1}]$ of n elements, and returns the ordered set $[i, a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$. A common operator used in scan is the addition. An equivalent definition of scan is presented by Harris et al. (2007).

The *thrust* library defines operation *exclusive_scan*, with the following definition from (Bell and Hoberock, 2011):

"The term 'exclusive' means that each result does not include the corresponding input operand in the partial sum. More precisely, an *init value* is assigned to $*result$ and the sum of *init* and $*first$ is assigned to $*(result+1)$, and so on". This version of $exclusive\_scan$ assumes plus as the associative operator but requires an initial value *init*. When the input and output sequences are the same, the scan is performed in-place.

For example given input vector V bellow, and $init = -2$, results in vector PS by application of this latter definition of the $exclusive\_scan$ is as follows:

V:	[	8	2	5	2	5	2	2	2	2	8	2]
PS:	[	-2	6	8	13	15	20	22	24	26	28	36]

Since we use this implementation we present a more appropriate definition for the *exclusive all-prefix-sums operation*:

Definition 6.1.1 (Exclusive all-prefix-sums operation)**.** *The all-prefix-sums operation takes a binary associative operator* $\oplus$ *with initial value* I*, and an array of* n *elements* $[a_0, a_1, ..., a_{n-1}]$ *and returns the array* $[I, (I \oplus a_0), (I \oplus a_0 \oplus a_1), ..., (I \oplus a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$.

In this work we use binary operator plus (+) on the *exclusive all-prefix-sums operation*.

The complexity of all-prefix-sums operation is $O(\frac{n}{p})$, for n elements and p processors. The first implementation on GPUs to show this complexity was (Sengupta et al., 2006).

6.1.2 Stream compaction

Stream compaction is also an important operation used in many algorithms. Operation $copy\_if$ is an example of stream compaction. This operation works as a copy operation, selecting elements according to the result of application of a function. Given an input vector A, and a predicate operation $P()$, that operates on elements of A, $copy\_if$ copies to output vector B the elements e for which $P(e)$ returns true. For example, suppose a predicate operator $greater\_0()$ that returns true if an input element is greater than 0. The output of application of $copy\_if$ on vector A, with predicate $greater\_0()$ is shown in vector B bellow:

A:	[	8	-2	5	-3	5	2	0	0	-1	8	2]
B:	[	8	5	5	2	8	2	]				

Another useful version of this operation called *copy_if_stencil* uses a different *stencil* vector S to select elements of input A. For example, applying *copy_if_stencil* on input A, with *greater_0*() predicate and stencil S produces output B bellow:

A: [8 7 5 1 5 2 0 0 1 8 -2]

S: [1 -1 1 -1 1 1 1 -1 -1 1 1]

B: [8 5 5 2 0 8 -2]

Stream compaction operations can use *all-prefix-sums* in their implementation. The complexity of *copy_if* and *copy_if_stencil* is also linear with the number of elements in the input vector, i.e. $O(\frac{n}{p})$, for n elements and p processors.

6.1.3 Sorting

Sorting is a general tool, used as an important step in many parallel algorithms. We use parallel radix-sort of $(key, value)$ pairs of 64 bit integers in this work. We discuss the complexity of the radix sort step used in our algorithms in section 6.3.2.1.

6.2 General Ideas

In this section we initially describe, by using one example case, how we can produce the implicit tree layout in preorder. Before tackling all steps of the algorithm which are described in section 6.3, we would like to entail a smaller problem, which is part of the problem to be solved by the whole algorithm of section 6.3.

Problem 6.2.1. *Suppose we have a set of leaf nodes sorted in preorder and our problem is to produce all nodes of the tree in a linear preorder sequence. For this subproblem, we don't want to discuss how to fill the information on internal nodes, but to find the proper place and reserve the space necessary for all internal nodes in the preorder. We want also to copy all leaf nodes from the contiguous preorder to their proper place in the final linear tree. We want to solve this problem in parallel, and we know the size of each internal node and the size of each leaf node. All internal nodes have the same size, and all leaf nodes also have the same size, but internal and leaf nodes can have different sizes. And finally*

suppose we can use a function $LCA.level(leaf_i, leaf_{i+1})$ *that takes two adjacent leaves in the preorder and returns the level of their LCA (Least Common Ancestor, definition 6.2.1) node.*

Definition 6.2.1 (LCA, Least Common Ancestor)**.** *The LCA means the closest possible common ancestor of both leaves in their path to the root node.*

We will show our solution to this problem in two steps. First we solve for the case where all internal and leaf nodes have unit size, let us call this problem 6.2.1.(a). Next we modify our solution considering any size for each type of node, thus solving the general problem 6.2.1.

Our solution to problem 6.2.1.(a) is algorithm 4 and illustrated in figure 6.1 with an example case. The figure shows the tree at the top to help us keep track of what is being processed and generated. The *input* array of leaves sorted in preorder is shown in blue, at the bottom of the tree. Each value in the vectors shown, are aligned horizontally with the corresponding leaf of the tree, to help with visualization of what is being processed, in parallel, for each leaf. Bellow the tree after the *Scatter leaves* step, the *output* linear Tree array is shown. Each step of the parallel algorithm 4 correspond to a brief description shown in black, at the right of figure 6.1. The three big vectors of values shown in white bellow the leaves represent the state of vector V at each step, corresponding to processing steps 1-3 in algorithm 4. The vector shown in blue, represents vector PS of the algorithm, and contains the result of step 4: parallel exclusive all-prefix-sums of vector V with initial value -1. Finally the *Scatter leaves* step in the figure correspond to the execution of step 5 of the algorithm. We can observe that the final destination of each internal and leaf node is correctly annotated above the *output* linear Tree array, and correspond to values of PS vector for leaf nodes.

There is an implicit value -1 shown in the figure, which is used in step 2 of the algorithm, that evaluates differences from two adjacent positions of vector V. Step 2 takes the differences of one position to the immediate previous position. For this reason we need a particular case for the calculation in position 0, and value -1 is implicitly used as the value stored immediately before position 0. The value -1 allows reservation of

space for the root node. Setting it to -1 is equivalent to taking into account level 0, in the level differences step. [1]

Algorithm 4 Parallel solution to problem 6.2.1.(a)

Input : *Contiguous array of leaves in preorder*
Output: *Linear Tree array, with leaves in place and gaps to fit internal nodes in the preorder.*

1: Produce in array V, the LCA.level for every two adjacent leaves, in parallel:
$V_i \leftarrow LCA.level(leaf_i, leaf_{i+1})$, $LCA.level$ of last node and the following is 0 (root)

2: Evaluate adjacent differences of values in vector V, in parallel, producing LCA.level differences of adjacent leaves: $V_i \leftarrow V_i - V_{i-1}$;
Note that V_0 operates with implicit value $= -1$ (to generate space for root).

3: Transform array V values in parallel, producing space needed at each position:
$V_i \leftarrow (V_i > 0)\ ?\ (V_i + 1)\ : 1$; positive values means: space needed for V_i internal nodes plus 1 leaf; Negative values or 0 means: space is needed only for 1 leaf

4: Perform a parallel exclusive all-prefix-sums of vector V with initial value = -1 obtaining vector PS.

5: Perform a parallel scatter of leaves to PS addresses (i.e. use prefix value PS_{i+1} as destination address to copy $leaf_i$ in final linear Tree array).

Our algorithms for generation of the compressed implicit octree of this chapter, as well as algorithms 4 and 5 evaluate and use the the value of the *LCA.level adjacent difference* of adjacent leaves in the input to take decisions on how much space needs to be allocated between these leaves (*i.e. the gap size*), to make room for insertion of internal nodes in the correct preorder in the final linear tree. For example, algorithms 4 and 5 evaluate this value at step 2 and use the value in step 3 to obtain the size of the *gaps*. Lemma 3 shows that the use of the *LCA.level adjacent difference* is correct to predict the size of each gap.

After obtaining the size of each gap, we need to perform a parallel *exclusive all-prefix-sums* with these values, to obtain the number PS_i (*i.e. number of words in the prefix of* $leaf_i$) for each leaf node in the final preorder. This is done in step 4 in algorithms 4 and

[1] This can be can be further explained by lemma 3 case (c), that explains how the differences are used to allocate space. Since case (c) is the case that is used to allocate space between two leaves, and since there is no leaf before the first, we have to explicitly provide the exception to allocate space for the root node.

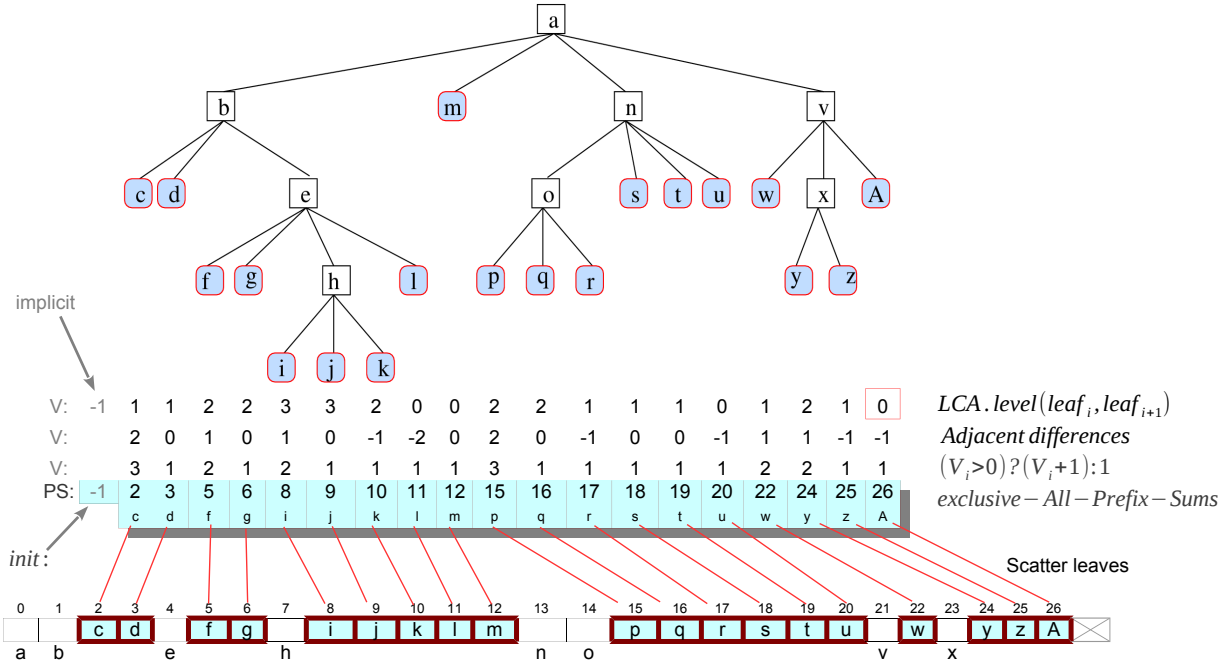


Figure 6.1: Ideia of an algorithm for problem 6.2.1.(a), with unit node sizes

5. The array PS is used afterwards in the algorithms (in the next step) to copy leaves (*i.e. scatter*) to their final positions in the output linear tree. In fact, vector PS is of size $N+1$ for N leaves. And the destination address of $leaf_i$ is PS_{i+1}.

The initial value to be used in the *exclusive all-prefix-sums* of step 4 of the algorithms is $-(size_{leaf})$, i.e. the negative value of the size of a leaf. This value is -1 for algorithm 4. The reason for this value will be explained for the general case algorithm 5.

We are now in condition to enunciate and prove lemma 3, afterwards we show how to slightly change algorithm 4 to allocate space for different leaf and internal node sizes. Then, the lemma can also be used in section 6.3 in the complete algorithm for direct generation of compressed implicit octrees.

Lemma 3. *Given a leaf node at position i, denoted as $leaf_i$, if $LCA.levelDifference(i) = LCA.level(leaf_i, leaf_{i+1}) - LCA.level(leaf_{i-1}, leaf_i) = k$, with $k > 0$, then k internal nodes are to be inserted (allocated) in the final linear preorder sequence at the position of $leaf_i$, else no extra space is needed for internal nodes in this position.*

Proof. By definition, an ancestor node is in the path from the cell towards the root. In a tree any path of a cell to the root is unique. Let us denote $\{a, b\}$, as the $LCA(a, b)$, i.e.

the *least common ancestor* node of any two adjacent leaf cells a and b. Suppose there are three adjacent leaves a, b and c in the preorder sequence and, without loss of generality, a, b and c are in positions $i-1$, i and $i+1$. That is, they are in sequence. Nodes $\{a, b\}$ and $\{b, c\}$ are in the unique path from the root to b. In such situation, we have two possibilities:

(i) either $\{a, b\}$ and $\{b, c\}$ are in the same level, or

(ii) they are in different levels.

In case (i), from the uniqueness of the path in the tree, we conclude that $\{a, b\}$ and $\{b, c\}$ are the same node, i.e., $\{a, b, c\}$ which is *LCA(a, b)* and *LCA(b, c)*.These two situations of case (i) are depicted at the top of Figure 6.2 with the only possible case labeled as *Case(a)*.

Case (ii) is separated in two cases: either $\{a, b\}$ is at a higher level than $\{b, c\}$ (labeled *Case(c)* in Figure 6.2) or $\{a, b\}$ is at a lower level than $\{b, c\}$ (labeled *Case(b)* in Figure 6.2).

We will now analyze the number of *internal nodes* to be inserted between leaves a and b in the preorder, depicted as Cases (a), (b) and (c) in the figure.

- **Case (a):** Suppose $\{a, b, c\}$ is at level n. This implies that *LCA.level(a, b)* $= n$ and *LCA.level(b, c)* $= n$. The *LCA.level adjacent difference* of b is $n - n = 0$. Note that this is the same as the number of internal nodes between a and b in the preorder, i.e. none, since $\{a, b, c\} = \{a, b\}$ in this case, and it was placed before a in the preorder, and thus we insert no internal cells between a and b (annotated none in the figure).

- **Case (b):** As established before for this case, $\{a, b\}$ is at a lower level (farther from the root) than $\{b, c\}$. Suppose $\{a, b\}$ is at level n and $\{b, c\}$ is at level $(n - k)$, for some $k > 0$. Since both $\{a, b\}$ and $\{b, c\}$ are in the path of b, and the path is unique, $\{a, b\}$ is below $\{b, c\}$ in this path. From the fact that $\{b, c\}$ is the *least* common ancestor of c, and it is in the path of a, we conclude that it is in fact the *least* common ancestor $\{a, b, c\}$ which is at level $(n - k)$ in this case. All the ancestors in the path of a from $\{a, b\}$ to $\{a, b, c\}$ come before a in the preorder. Since $\{a, b\}$

was placed before a, we insert no internal cells between a and b (annotated *none* in the figure). Note also that this is the only case where the *LCA.level adjacent difference* is negative, as $LCA.level(b,c) - LCA.level(a,b) = LCA.level(a,b,c) - LCA.level(a,b) = (n-k) - n = -k$.

- **Case (c):** As settled for this case, $\{a,b\}$ is at a level above $\{b,c\}$ in the tree. Suppose $\{a,b\}$ is at level n and $\{b,c\}$ is at level $(n+k)$, for some $k > 0$. Since both $\{a,b\}$ and $\{b,c\}$ are in the path of b, and the path is unique, $\{a,b\}$ is higher (closer to the root) in this path. From the fact that $\{a,b\}$ is the least common ancestor of a and it is also an ancestor of $\{b,c\}$ then $\{a,b\}$ is the *least* common ancestor of a, b and c, i.e., $\{a,b\} = \{a,b,c\}$, which is at level n in this case. All ancestors in the path from $\{b,c\}$ to $\{a,b,c\}$, not including $\{a,b,c\}$, need to be placed between a and b in the preorder (k internal nodes indicated in the figure) and $\{a,b,c\}$ must be placed before a in the preorder. Note that this is the only case where the *LCA.level adjacent difference* is positive, as $LCA.level(b,c) - LCA.level(a,b) = LCA.level(b,c) - LCA.level(a,b,c) = (n+k) - n = +k$.

So we have only three distinct cases based on the *LCA.level adjacent difference* of a leaf at position i, and we can use this number to obtain the number of internal nodes to be placed (reserved space) before a node, as shown in the table below.

Case	Result of *LCA.level difference* of $leaf_i$	Number of internal nodes to allocate for $leaf_i$
Case(a)	0	0
Case(b)	< 0	0
Case(c)	$n > 0$	n

Table 6.1: *LCA.level difference* cases of lemma 3 and implications on the number of internal nodes at position i of the final preorder

□

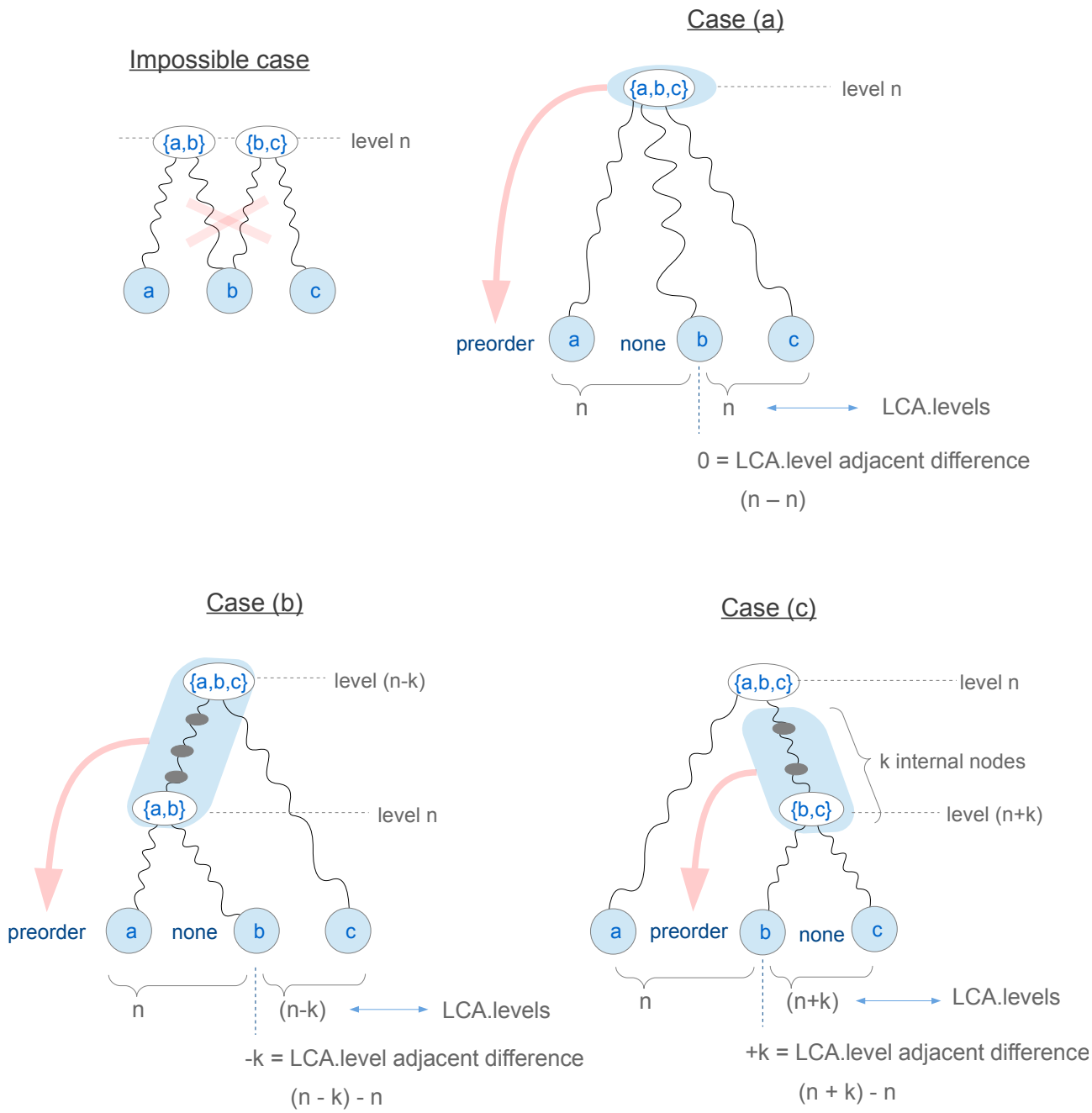


Figure 6.2: Illustrates LCA.level differences cases in the preorder, which determines the number of internal nodes to be inserted before a given leaf

We can now change algorithm 4 to solve for the general case and allocate gap spaces when internal node sizes are different from leaf node sizes. The modification is reflected in algorithm 5, with slight changes shown in red, in step 3 and in step 4. According to table 6.1 of lemma 3 if the value of the $LCA.levelDifference = 0$, with $n > 0$, we need to reserve space for n internal nodes in the position of the leaf, otherwise 0 internal nodes are

inserted. In either case we also have to allocate space for the leaves, i.e. positive values means: space needed for V_i internal nodes plus 1 leaf; Negative values or 0 means space is needed only for 1 leaf. The conditional statement $V_i \leftarrow (V_i > 0)\ ?\ (V_i * size_{int} + size_{leaf})\ :\ size_{leaf}$; executed in parallel in step 3 of algorithm 5 assigns the correct values depending on the value of expression $(V_i > 0)$, with size adjustments in either cases.

The initial value to be used in the *exclusive all-prefix-sums* of step 4 of the algorithm is $-(size_{leaf})$, i.e. the negative value of the size of a leaf. This is because step 3 allocates the space for a gap (the space for the internal nodes in it) plus a leaf at each position of array V_i, before the prefix-sum. Without the correction in the initial value, the exclusive prefix sum will produce the address of the *last* word of each leaf in the final preorder. Subtracting the value of the $leafsize$ in the initial value produces the address of the *first* word of each leaf in the final preorder. Note that the prefix-sum vector PS is of size $N+1$ for N leaves. And the destination address of $leaf_i$ is PS_{i+1}.

Algorithm 5 Parallel solution to problem 6.2.1 with different sizes for leaf and internal nodes

▷ *changes in red*

Input : *Contiguous array of leaves in preorder*
Output: *Linear Tree array, with leaves in place and gaps to fit internal nodes in the preorder.*

1: Produce in array V, the LCA.level for every two adjacent leaves, in parallel:
$V_i \leftarrow LCA.level(leaf_i, leaf_{i+1})$, $LCA.level$ of last node and the following is 0 (root)

2: Evaluate adjacent differences of values in vector V, in parallel, producing LCA.level differences of adjacent leaves: $V_i \leftarrow V_i - V_{i-1}$;
Note that V_0 operates with implicit value $= -1$ (to generate space for root).

3: Transform array V values in parallel, producing space needed at each position:
$V_i \leftarrow (V_i > 0)\ ?\ (V_i * size_{int} + size_{leaf})\ :\ size_{leaf}$;
Positive values means: space needed for V_i internal nodes plus 1 leaf;
Negative values or 0 means: space is needed only for 1 leaf

4: Perform a parallel exclusive all-prefix-sums of vector V with
initial value $= -(size_{int})$ obtaining vector PS.

5: Perform a parallel scatter of leaves to PS addresses (i.e. use prefix value PS_{i+1} as destination address to copy $leaf_i$ in final linear Tree array).

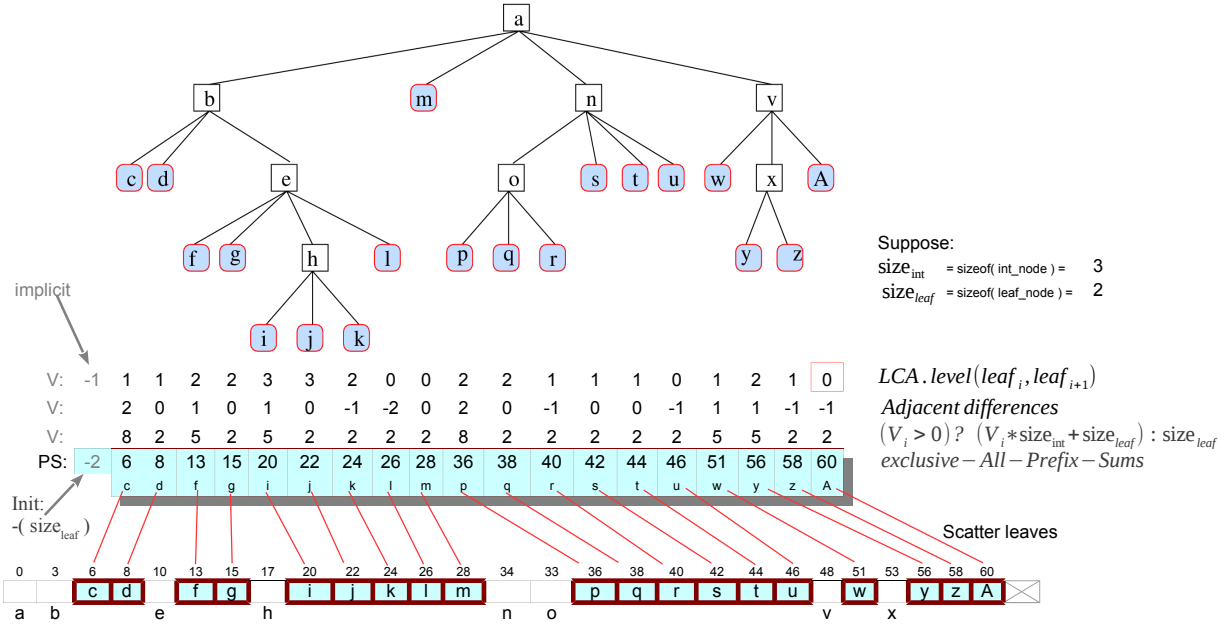


Figure 6.3: Example execution of algorithm 5 with differentiated sizes of *internal* and *leaf* nodes, example sizes described in the picture

6.3 The Algorithm

Our massivelly parallel algorithm for direct generation of compressed implicit octrees is described in this section as algorithm 6. The algorithm takes as input an array of leaves with $3D$ coordinates (x, y, z) and *application specific* fields of each leaf. As a result, the algorithm produces a linear array in *Compressed Implicit Tree* (CIT) preorder layout, with leaves in place, internal nodes in place and filled with *skip links*. If needed by the application other information can be filled in internal nodes such as coordinates of each centroid, level of each internal node, or a list of internal nodes per level. Any partial set of application specific fields can be copied to the final tree. The input array of leaves can be in any order, and in general not all application fields need to be inserted in the tree, but can be left in the original vector. For this reason, another by-product to this algorithm is a *data remapping vector* (DMV), that can be used to efficiently access application specific fields in the original input vector. The data remapping vector allows accessing data fields of each leaf in the input vector, in the same sequence (preorder in our case) that the leaves are laid out in the final tree, producing more efficient memory access.

Another difference from algorithms 4 and 5 is that we don't use the leaves to build the Compressed Implicit Tree but we employ only Morton keys generated from each leaf coordinate to build it, up to the final steps of the algorithm where the leaves must be copied to the tree. The *skip links* of the Implicit Tree are also generated from a sequence of Morton keys sorted in ascending order, which correspond to the preorder sequence of leaf nodes.

A constant declared in the algorithm defines the maximum height of the trees produced in compressed implicit layout. This constant can be considered as a parameter that configures the algorithm. Suppose this number is k. In the experiments of this chapter k is set to 21. The algorithm produces the compressed implicit tree with all the leaves from the input, i.e. irrespective of k no leaves are eliminated. In this case the *tree height limit* can produce a number of leaves in the lowest tree levels, which is allowed in our octree definition 2.1.4. The maximum number of nodes on a octree with k levels is 8^k, if the *tree height limit* is not reached. This number can be higher if the *tree height limit* is reached, as more than 8 leaves can be inserted in the lowest level. For example, if $k = 20$ and the *tree height limit* is not reached, we would have a tree with potentially $8^{20} = 2^{60}$ leaves, and this would be enough to fit a variety of leaf distributions for even the largest n-body simulations, as stated in reference (Hariharan and Aluru, 2005). In terms of the implementation, this constant k only have implications on the number of bits used for Morton keys. The number of bits in Morton keys to reach k levels in the tree is $3k$. If we set $k = 21$ as in our experiments, Morton keys have $3 \times 21 = 63$ bits and can fit in 64 bit integers. The next available key size would be using 128 bit integers, and could be used to produce trees with up to 42 levels. The size of Morton keys has implications in running times, due to step 4 of algorithm 6, that uses radix sorting of Morton keys. This step clearly dominates the tree generation time, as shown in the experiments.

Besides presenting the parallel algorithm for direct generation of Compressed octrees as algorithm 6 we show two other views of the implementation, in algorithm 7 and in figure 6.4. Algorithm 7 is offers a different format that closely follows the parallel kernels of the implementation, showing inputs and outputs of each kernel. The purpose of each

kernel closely follows the steps described in algorithm 6, which are labelled with mnemonic names on the right side of the figure. These mnemonics can be used to identify the running times of each kernel in the graphs of section 6.4. Another graphical view of inputs and outputs of each kernel is presented in figure 6.4.

In the next section we further describe the workings of each step of algorithm 6.

6.3.1 Algorithm steps

Algorithm 6 is very similar to algorithm 5, but incorporates extra steps. For example, algorithm 5 used function $LCA.level$, we need to show how this information can be obtained since the tree is yet to be built. Elimination of linear chains in the octree is another step to be incorporated in algorithm 6.

After performing initialization (I) and evaluating the bounding box (BB) of all N leaves, algorithm 6 produces in vector $Paths$ all Morton codes corresponding to the leaves. Morton keys represent paths from the root to a leaf. Producing all Morton keys is denominated leaves classification (LC). Vector $Order$ was initialized with numbers in sequence from 0 to N. The set of pairs formed by $(Paths_i, Order_i)$ are sorted in ascending order using *radix sort pairs*, where keys used for sorting are the Morton keys in $Paths$. This step is labelled (Srt). After this step, each pair $(Paths_i, Order_i)$ correspond to the set of Morton keys in Morton SFC order, where $Paths_i$ is the Morton key of a leaf at position $Order_i$ in the input vector. We then proceed to step 5, labelled (LDc), to produce in vector DV the $LCA.level$ of each adjacent Morton keys. The $LCA.level$ function works as follows in definition 6.3.1.

Definition 6.3.1 (LCA.level function)**.** *Given two Morton keys k_a and k_b in Morton 3D SFC, the Morton key of $LCA(a, b)$ as well as its level, can be obtained using the result of operation $k_a \oplus k_b$, where $\oplus$ is the exclusive-or operation. For 63 bit Morton keys represented in 64 bit numbers the high order bit is not used. In such case $LCA.level(k_a, k_b) = (clz(k_a \oplus k_b) - 1)/3$ where operation clz on a 64 bit number returns the number of leading zeros and the division by 3 is an integer division.*

Operation clz is common in modern instruction sets. In CUDA GPUs clz executes in the same number of clocks as integer operations. Thus, $LCA.level$ of adjacent leaves is $O(1)$. The reasoning behind this computation is that the exclusive-or operation turns off the common prefix bits of the operated Morton keys. In this case, function clz counts the number of common bits in the prefix. We subtract 1 from this number as a correction due to the fact that we don't use the high order bit. Finally, the integer division by 3 counts the number of common levels (each group of 3 bits represents one of 8 octants in $3D$ space, a changing bit in a group of 3 determines the level of the LCA).

In order to generate space for internal nodes between leaves, as we did for algorithm 5, we evaluate adjacent differences of values in vector V, which is done in step 6, denominated (ADmin0). Note that here in algorithm 6 we slightly depart from what we did before, by evaluating the difference in adjacent cells and applying function $min0$, which eliminates negative values. This still reflects the properties of lemma 3 that we used before to predict the number of internal nodes needed between leaves. Let us call this difference the *gap count*.

With the *gap count* of each gap in vector DV, step 7 produces vector $AllNodesToSearch$, that will contain the Morton key of *possible next nodes in sequence*. These keys will be used in step 8 to find the *skip links* of all internal nodes. Step 7 performs four kernels: (CpIf), (PSG1), (PSG2) and (GNTS), that can be better visualized in algorithm 7. The first two kernels (CpIf) and (PSG1) are stream compaction steps, performed in vector DV of n integers and function $greater\_than\_0()$, with output in vector $GapIndexes$ and $Gap\_prfxsum$. This stream compaction selects only nonempty gaps. Kernel (PSG2) performs an exclusive all-prefix-sums in vector $Gap\_prfxsum$. After these 3 kernels executions, vectors $GapIndexes$ points the positions on vector DV where the gaps were taken from, and vector $Gap\_prfxsum$ contains the positions of each gap in vector $AllNodesToSearch$. As a last task of step 7, kernel (GNTS) produces in vector $AllNodesToSearch$ the Morton key of *possible next nodes in sequence*. These keys will be used in step 8 to find the *skip links* of all internal nodes.

Note that step 7 is in fact a *load balancing phase* of algorithm 6, by identifying only nonempty gaps, and producing work in vector $AllNodesToSearch$, each gap with approximately equal number of elements of work.

To produce the Morton key that will be used in step 8 to find skip link values we resort to a new trick. We don't know the Morton code of the node the skip link will point to, and we don't even know if the skip link will point to a leaf or to an internal node. But we can calculate the Morton code of the first Morton key in the next octant of the SFC. To do this we only need the Morton codes of the key before the gap and the key after the gap, and we used function $LCA.level$ of definition 6.3.1. With these two Morton keys, we find what we have called the Morton key of *possible next nodes in sequence*, which is done with definition 6.3.2.

Definition 6.3.2 (possible next nodes in sequence)**.** *Using Morton 3D SFC, given two adjacent Morton keys k_a and k_b in the sorted order we define k_{next} as the Morton key of possible next node in sequence, which is in fact the first Morton key in the next octant of the 3D SFC. The value of k_{next} for the first internal node at the left of k_a in the gap is calculated with the following code sequence, for 64 bit Morton keys (high order bit unused):*

```
level = LCA.level( Ka, Kb );
levelmask_shift = 63 - (level * 3);
one  = 0x0000000000000001;
ffff = 0xffffffffffffffff;
knext = ( ka + (one << levelmask_shift) ) &
        ( ffff << levelmask_shift );
```

The definition above produces k_{next} for the first internal node in the gap to the left of k_a, that is, this generates k_{next} for LCA of k_a. If there are more than one internal nodes in the gap, i.e. if $gap\_size > 1$ we have to iterate and produce one key k_{next} for each ancestor internal node in the gap. Suppose a thread needs to generate these keys at position p of $AllNodesToSeach$ vector, and variable gap_size contains the number of internal nodes in the gap. The following piece of code can be used:

```
level = LCA.level( Ka, Kb );
level = level - gap_size;
levelmask_shift = 63 - (level * 3);
one  = 0x0000000000000001;
ffff = 0xffffffffffffffff;
for( i=0; i<gap_size; i++ ) {

     knext = ( ka + (one << levelmask_shift) ) &
             ( ffff << levelmask_shift );
     AllNodesToSeach[ p+i ] = knext;    // store knext

     levelmask_shift -= 3;
}
```

Step 8 uses these K_{next} keys produced in the last step to find the skip links of each internal node. Note that, each k_{next} represents a key that may or may not be in the sorted sequence. A binary search for lower bound finds the insertion positions of these keys, i.e. the lowest positions where this key should be inserted in the sorted order, this is equivalent to the position of the leaf node that the skip link will be pointing to. In fact, as we displace these leaf nodes with the insertion of internal nodes in gaps, the skip link remains pointing at the same position in vector DV. But the final prefix-sums array (in vector DV) will show the address that the skip link needs to point to. Step 8 has mnemonic (BSlb), and produces the skip links in vector $BsResults$.

In step 8 (ELC), each thread in parallel, picks the ith segment of $BsResults$ values, each segment of size DV_i, corresponding to a gap. The thread then eliminates adjacent duplicate contiguous values on its segment and adjusts DVi accordingly. This corresponds to eliminating linear chains in the tree. Figure 6.5 shows a plot obtained from a small generated tree, with indexes of skip links drawn above each internal node (internal nodes in white). These values were in vector $bsResults$ (labeled bs_out in the graph). The contents of each vector identifying each segment are also shown. Note that duplicate values of skip links appeared in two segments. They correspond to linear chains, as can be observed in the plot, because they represent the same set of nodes.

Steps 10-11 are presented in algorithm 6. These steps work as discussed for algorithm 5.

Step 12 fills internal nodes in gap_i in the final linear array. Skip links are obtained from the ith segment of $BsResults$. The initial address of gap_i is PS_{idx}, where $idx = GapIndexes_i$, obtained in step 7. If needed by the application other information can be filled in internal nodes such as coordinates of each centroid or level of each internal node. Another possibility in this step (not shown in the algorithm), is to insert indexes of each internal node in a bucket list per level, i.e. preparing a list of internal nodes per level, if demanded by the application code.

Step 13 copies the leaves to the correct position in the linear array. The correct vectors used and indexes for this operation are presented in the algorithm.

6.3.2 Complexity of the Algorithm for Direct Generation of the Compressed Implicit Tree

In this section we analize the complexity of algorithm 6. Steps 1-2, 5-6, 9-10 and 12-13 are clearly $O(\frac{n}{p})$, where n is the number of leaves and p, the number of processors (cores). This follows from the fact that, at each of these steps, a thread (processor) is responsible for processing one item, and there are $O(n)$ items on each case.

Step 3 is also $O(\frac{n}{p})$ as producing a Morton key from a coordinate (x, y, z) of a point is $O(1)$. In fact, in our implementation, we take $O(k)$ steps to produce a Morton key of width $w = 3k$ bits, and k is a constant defined in the program that determines the maximum number of tree levels.

The complexity of steps 4 and 8 are discussed in section 6.3.2.2 and they are both $O(\frac{n \log n}{p})$. We argue in the section that the running time of these steps are linear with n, that is $O(\frac{n}{p})$. Finally, step 11 executes an *all-prefix-sums* operation which is $O(\frac{n}{p})$, and the first implementation on GPUs to show this complexity was (Sengupta et al., 2006).

Step 7 performs four kernels: (CpIf), (PSG1), (PSG2) and (GNTS). The first two kernels (CpIf) and (PSG1) are stream compaction steps, performed in vector DV of n integers, with output in another vector. As stated before in section 6.1.2, stream

compaction is $O(\frac{n}{p})$. Kernel (PSG2) executes a exclusive all-prefix-sums operation also on n elements, which is $O(\frac{n}{p})$ as discussed before. Kernel (GNTS) is a massively parallel step in which each thread picks up to k elements from a given gap. For each element a Morton key of possible next nodes in sequence is generated (this function is $O(1)$)s. We conclude that (GNTS) is $O(\frac{n\,k}{p})$, which is $O(\frac{n}{p})$ considering that k is constant. Since all parts of step 7 are $O(\frac{n}{p})$ we conclude that it has this complexity.

Since all steps have complexity up to $O(\frac{n\ log\ n}{p})$ we conclude that algorithm 6 has this complexity.

Our experiments in section 6.4 show that the running time of the algorithm is linear with respect to the number of leaves, and we conjecture on section 6.3.2.2 that the algorithm 6 is in fact $O(\frac{n}{p})$.

6.3.2.1 Complexity of the radix sorting steps

We use radix sort of (key, value) pairs in step 4 of algorithm 6. It is possible to consider the complexity of radix sort to search k bit keys as $O(k\ n)$ or $O(n\ log\ n)$, as k is $O(log\ n)$. In our algorithm we have constant k limiting the number of levels in the generated compressed octree, and this is not a limitation of the program, but it also determines the number of bits of Morton keys used in the algorithm. Configuring constant k, we can produce deeper trees, at the cost of taking more time in the sorting step, and with consequences on tree traversals. We have tested the tree generation for different N, where N is the number of leaves, and the sorting step behaves as linear with respect to N, as shown in the experiments of section 6.4 and graph 6.6. For this reason, we will consider the interpretation of k constant as the correct predictor of performance of our algorithm, but we will offer the complexity at the worst interpretation of $O(\frac{n\ log\ n}{p})$ for the radix sorting step, where p is the number of processors (cores).

6.3.2.2 Complexity of *linear chain elimination* and binary search steps

The number of nodes between leaves is at most k (constant limiting the tree height) in the worst case. In practice, the average is much lower, as the internal cells in paths of the tree

are distributed between various leaves in the preorder. This has implications on the size of the vector *AllNodesToSearch*, which needs to be $k.n$ in this implementation. Step 7 then generates up to $k.n$ nodes, each thread generating up to k nodes. For p processors (cores) we conclude that this step is $O(\frac{n\ k}{p})$. Considering k constant, we have $O(\frac{n}{p})$. The number of Morton keys to search by the binary search step is, consequently, $O(n\ k)$. Considering the binary search on the n Morton keys vector, we have a complexity of $O(k\ n\ log\ n)$ and using p processors (cores) we have $O(\frac{n\ log\ n}{p})$ considering k as a constant.

Vectorized binary searches of s sorted keys to search on an array of e elements also sorted is actually linear $O(s + e)$. Our search for possible next nodes to search does not behave exactly as sorted, but nearly sorted. Suppose two items a_i and b_j representing next nodes to search (i.e. skip link values) and i, j represent the position of the items in the search vector. Suppose also that a_i occurs before b_j in the search vector, that is, $i < j$. If the result of the searches return $k = search(a_i)$ and $l = search(b_j)$, we don't have in this case the transitivity i.e., the implication $(i < j) \Rightarrow (k \leq l)$ is not always true. As a result of lemma 2, we conjecture that this step is $O(n)$, or we could have a special binary search algorithm for this case that is $O(n)$: as s and e are sorted, they can be partitioned between multiprocessors (or processors), each processor performing a local search for ocurrences of a subset of s on a partition of e. Local searches can be performed on local memory of each processor (e.g scratchpad memory or shared memory). If the local searches are not satisfied on a local partition, lemma 2 guarantees that only up to k searches need to be propagated to the next processor.

We have tested the direct compact implicit tree generation algorithm for different number of leaf nodes N, and the binary search running time is linear with respect to N in practice, as shown in graph 6.9 of section 6.4. As stated before, we conjecture that this linearity is a consequence of lemma 2 and we intend to look for a formal proof of this assumption in the future.

This way, we will be using the complexity of the binary search step in the tree generation algorithm as $O(\frac{n\ log\ n}{p})$, for the time being.

Algorithm 6 Massively Parallel Algorithm for Direct Generation of Compressed Implicit Octrees

Input: *An array of leaves with 3D coordinates* (x, y, z) *and application specific fields of each leaf*

Outputs:
- *Linear array in Compressed Implicit Tree preorder layout*
- *A data remapping vector to be used in traversals*

1: Initialize variables, parallel initialize: $Indexes_i \leftarrow i$; $Order_i \leftarrow i$; ▷ (I)

2: For each leaf, in parallel, evaluate BB: the bounding box of all points (coordinates);

3: For each leaf, in parallel, produce Morton codes of each leaf in vector $Paths$:
$Paths_i \leftarrow MortonKey(leaf_i, BB)$; ▷ leaves classification (LC)

4: Parallel radix sort pairs $(MortonKey_i, Order_i)$, output to $Paths$ and $Order$ vectors
$Paths, Order \leftarrow radixSortPairs(Paths, Order)$; ▷ (Srt)

5: Produce in array DV, the LCA.level for every two adjacent Morton keys, in parallel:
$DV_i \leftarrow LCA.level(Path_i, Path_{i+1})$; ▷ (LDc)
LCA.level of last node and the following is 0 (root)

6: Evaluate adjacent differences min_0 of values in vector DV, in parallel, producing *positive* LCA.level differences of adjacent leaves: $DV_i \leftarrow min(DV_i - DV_{i-1}, 0)$;
Note that DV_0 operates with implicit value $= -1$, ▷ (ADmin0)
to generate space for root: i.e. $DV_0 \leftarrow min(DV_0 - (-1), 0)$;

7: Evaluate the Morton keys of *possible next nodes in sequence* ▷ (CpIf)
in parallel for all gaps, and internal nodes in gaps, ▷ (PSG1)
based on $Paths$ and DV vectors, producing ▷ (PSG2)
vector $AllNodesToSearch$ (MortonKeys of all internal nodes) ▷ (GNTS)

8: Perform a parallel binary search for lower bound positions of all Morton keys in vector $AllNodesToSearch$, representing *possible next nodes in sequence*. Produce results in vector $BsResults$. ▷ (BSlb)

9: Each thread in parallel, picks the ith segment of $BsResults$ values, ▷ (ELC)
each segment of size DV_i, corresponding to a gap. Eliminating adjacent duplicate contiguous values on this segment and adjusting DVi accordingly. This corresponds to eliminating linear chains in the tree.

10: Transform array DV values in parallel, producing space needed at each position: in this version, DV_i has number of internal nodes in gap. We adjust size to fit the number of internal nodes plus one $leaf$. 0 means: space is needed only for 1 leaf
$DV_i \leftarrow (DV_i > 0)\ ?\ (DV_i * size_{int} + size_{leaf})\ :\ size_{leaf}$; ▷ (PS_1)

11: Perform a parallel exclusive all-prefix-sums of vector DV with ▷ (PS_2)
initial value $= -(size_{int})$ obtaining vector PS.

12: Each thread in parallel, picks the ith segment of $BsResults$, each ▷ (FINs)
segment of size DV_i contains skip links, that are copied to internal nodes in gap_i in the final linear array. Initial address of gap_i is PS_{idx}, where $idx = GapIndexes_i$ (obtained in step 7). Centroids of internal nodes can also be copied.

13: Perform a parallel scatter of leaves *in preoder*, i.e. via $Order$ vector. ▷ $(LScatt)$
Each $leaf_o$, where $o = Order_i$, is copied to addres d, where
$d = PS_{i+1} + gapsize(GapIndexes_i)$ in the final linear Tree array.

Algorithm 7 Kernels of the Massively Parallel Algorithm for Direct Generation of Compressed Implicit Octrees

```
0: procedure GenerateCompressedImplicitTreeKernel()

1:    Order, Indexes ← InitializationKernel();                                  ▷ I
2:    BB ← BoundingBoxKernel( X, Y, Z );                                        ▷ BB
3:    Paths ← leaves_classification( X, Y, Z, BB );                             ▷ LC
4:    Paths, Order ← RadixSort_Pairs( Paths, Order );                           ▷ Srt
5:    DV ← LCA_LevelDiffs( Paths );                                             ▷ LDc
6:    DV ← AdjacentDifferences_min_0( DV );                                     ▷ ADmin0

7:    ngaps, GapIndexes ← copy_if_stencil(                                      ▷ CpIf
                  /* from = */ Indexes,
                  /* use stencil = */ DV,
                  /* select with predicate = */ greater_than_0() );
8:    Gap_prfxsum ← copy_if( /* from = */ DV,                                   ▷ PSG 1
                  /* select with predicate = */ greater_than_0() );
9:    Gap_prfxsum ← exclusive_PrefixSum( Gap_prfxsum );                         ▷ PSG 2
10:   nnodes_tosearch, Allnodes_tosearch ← generate_allnodes_tosearch(
                  /* Paths = */ Paths,
                  /* gap_sizes = */ DV,
                  /* gap_indexes = */ GapIndexes,                              ▷ GNTS
                  /* gap_prfxsum = */ Gap_prfxsum,
                  /* ngaps = */ ngaps );

11:   Bs_results ← BinarySearch_lower_bound( Paths,                             ▷ BSlb
                  Allnodes_tosearch, nnodes_tosearch );

12:   Bs_results, DV ← eliminate_linear_chains(                                 ▷ ELC
                  /* skip_links = */ Bs_results,
                  /* gap_sizes = */ DV,
                  /* gap_indexes = */ GapIndexes,
                  /* gap_prfxsum = */ Gap_prfxsum, );
                  /* ngaps = */ ngaps );

13:   DV ← adjust_all_sizes( /* gap_sizes = */ DV, nLeaves );                    ▷ PS_ 1
14:   PS ← exclusive_PrefixSum( DV );                  // in-place scan        ▷ PS_ 2

15:   compact_tree_array ← fill_internal_nodes(                                 ▷ FINs
                  /* skip_links = */ Bs_results,          // after ELC
                  /* final_pos_PS = */ PS,          // final positions prefixSum
                  /* gap_indexes = */ GapIndexes,
                  /* gap_prfxsum = */ Gap_prfxsum,
                  /* ngaps = */ ngaps );

16:   compact_tree_array ← leaves_Scatter( X, Y, Z, M, Order                    ▷ LScatt
                              /* final_pos_PS = */ PS );
17: end procedure
```

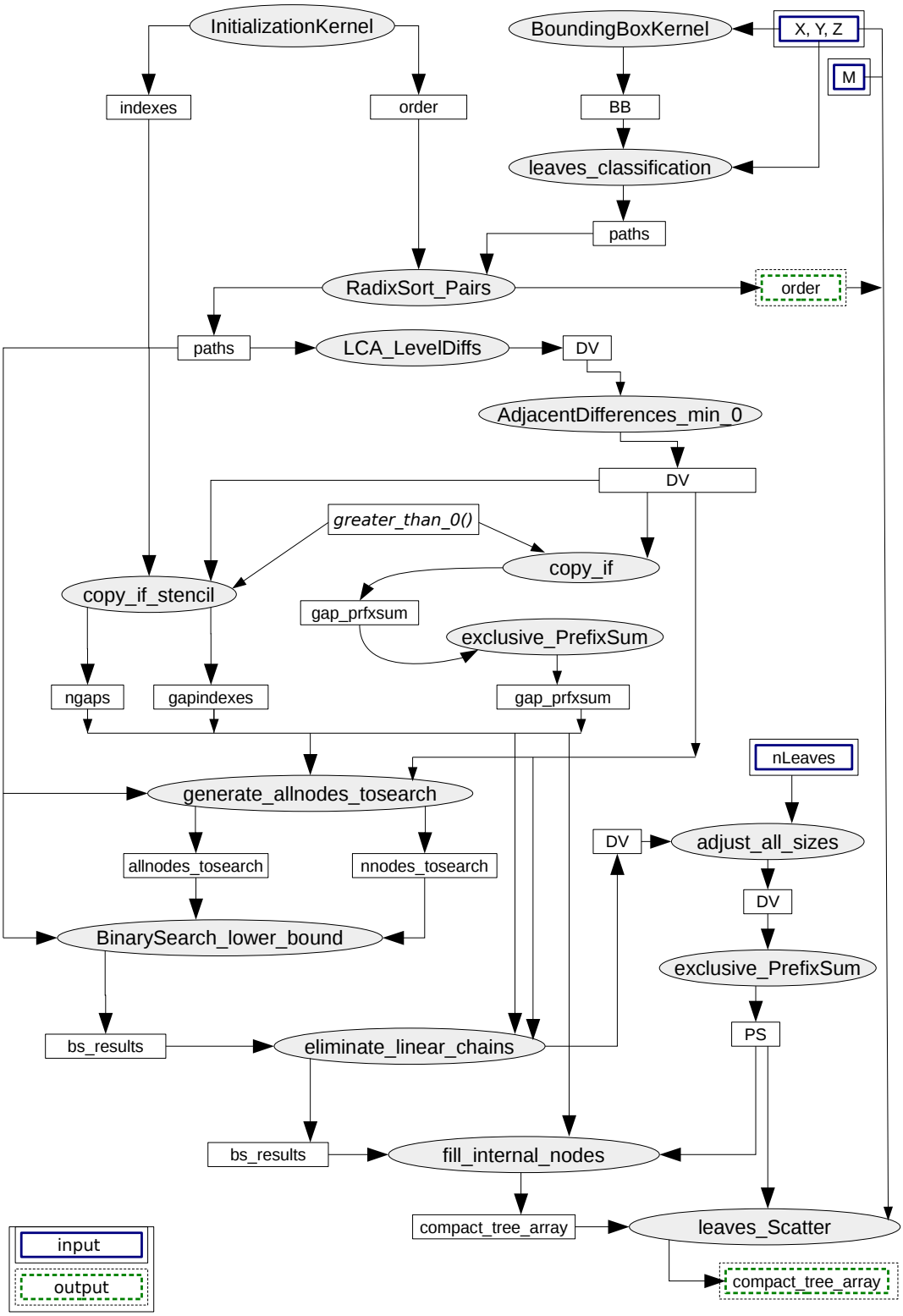


Figure 6.4: Graphical view of inputs and outputs of each kernel of the Massively Parallel Algorithm for Direct Generation of Compressed Implicit Octrees

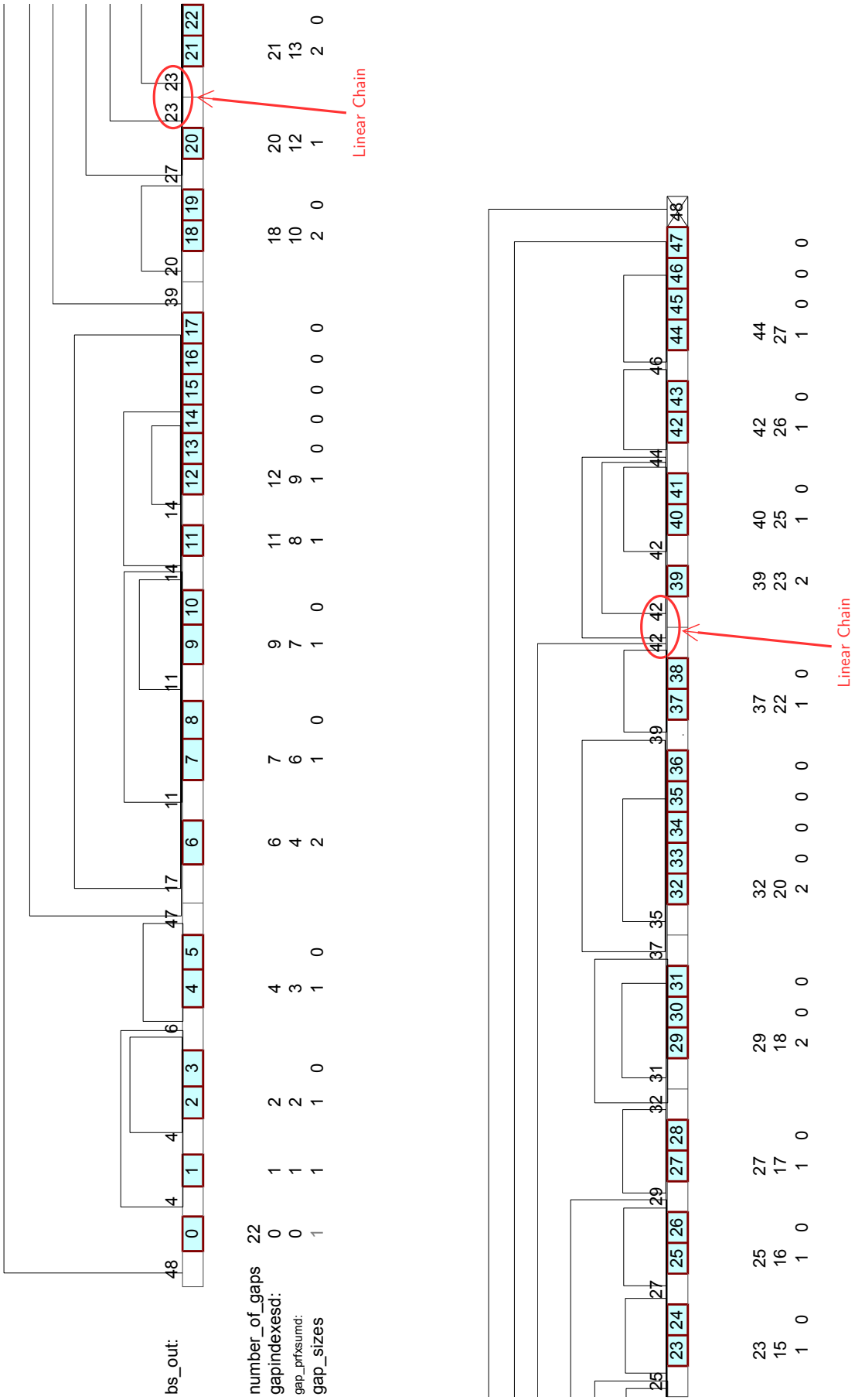


Figure 6.5: Graphical view of the output of step 8 of algorithm 6 showing duplicate skip link numbers which are equivalent to linear chains in the tree, to be eliminated in step 9.

6.4 Experimental Results on the Parallel Direct Generation of the Compressed Implicit Octree Layouts

In this section we present the experimental results relative to the direct generation of Compressed Implicit Octrees using our massively parallel algorithm 6. In section 6.4.3 we also compare our algorithm with version 3.0 of octree generation proposed by (Burtscher and Pingali, 2011), as well as the effectiveness of using algorithm 6 in Barnes-Hut N-body simulations in conjunction with the application of SWW (presented in section 4.3) in octree traversals.

6.4.1 Experimental methodology

Running times were obtained by instrumenting the source code with CUDA toolkit time measurement functions. All input was randomly generated with Plummer distribution of leaf nodes (particle nodes of Barnes-Hut simulations) for the different input sizes shown. For each experiment we measured 100 executions of the massively parallel direct implicit octree generation algorithm and report the average value per step. Since the experiments have shown linearity with respect to input size, we used linear scales in the graphs. All times shown in the charts are in ms (milliseconds).

6.4.2 Systems and compilers

We have executed our experiments on a system with a modern cache based GPU architecture, namely Kepler architecture. The GPU utilized on these experiments was an NVIDIA GPU GTX Titan (Kepler) on an Intel i5-3330 CPU @ 3.00GHz, running a GNU/Linux 3.2 i686 operating system kernel, and Ubuntu 12.04 LTS distribution. Programs were compiled with NVIDIA Cuda compilation tools, release 5.5, specific for the operating system configuration.

For the parallel building blocks library we have used Thrust (Bell and Hoberock, 2011) (Hoberock and Bell, 2015) with recent version 1.8.1.

Table 5.1 summarizes GPU characteristics which are important to our developments.

GPU Model	GTX Titan
Architecture	Kepler
Device Capability	3.5
No. of Multiprocessors (MP)	14
CUDA Cores/MP	192
Total CUDA Cores	2688
Max. threads per MP	2048
Max. threads per block	1024
Max. registers per MP	65536
Max. registers per thread	256
Max. resident warps per MP	64
Peak IPC	4
L2 cache size	1.5 MB
Shared memory/MP	48 KB
Memory Bandwidth	288.4 GB/s

Table 6.2: Characteristics of GPU Model GTX Titan

6.4.3 Experimental Results

We have tested the running times of algorithm 6, the parallel direct generation algorithm for *Compressed Implicit Octrees* measuring the total execution time of all kernels (steps), with different input sizes. For example, we have these experiments documented for GPU GTX Titan in the chart of figure 6.6. We have generated the tree for input sizes that are powers of 2, starting at 1Mi leaves and up to 16Mi leaves (Mi stands for $2^{20} = 1,048,576$). The times presented include execution of the radix sort step. It can be observed, that the algorithm presents linear running times with respect to the input size and can generate octrees with throughput of up to 202 million leaves/s (not including internal nodes, which in such case approximately doubles the throughput). The resulting running times confirm our expectation based on the complexity theoretical analysis.

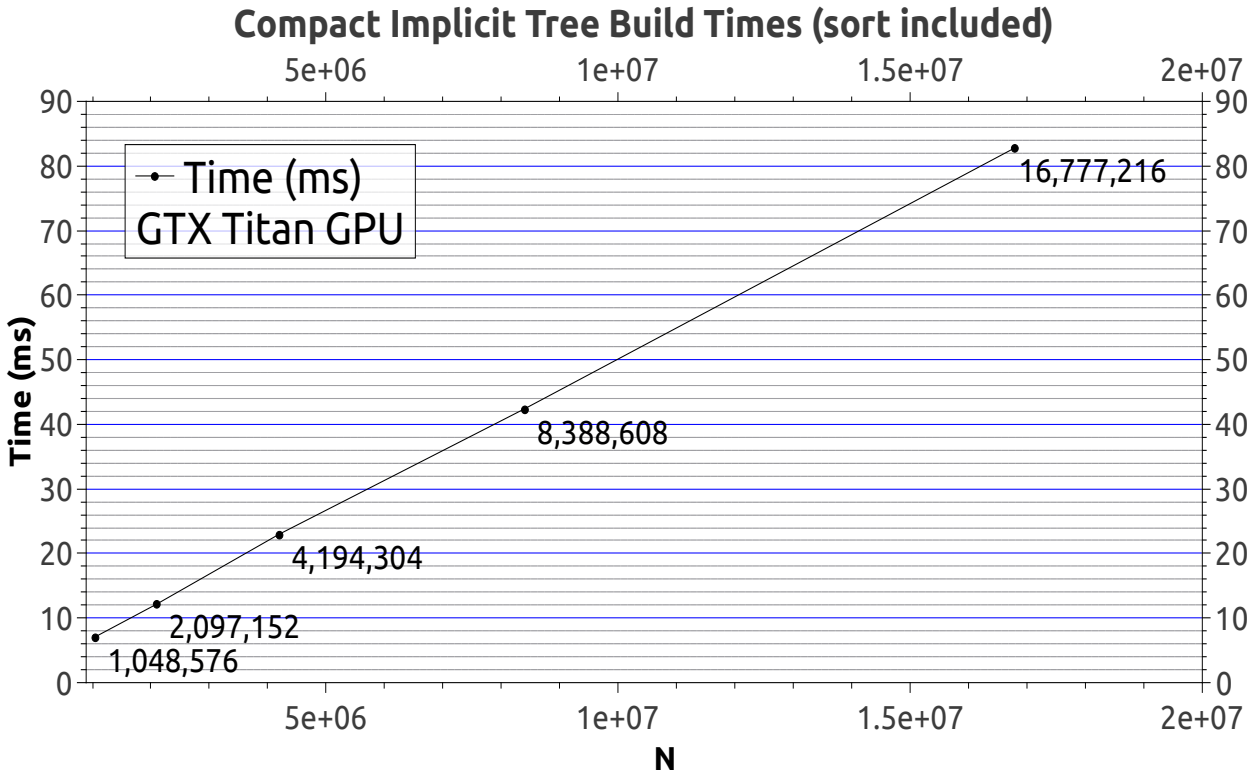


Figure 6.6: Running times of the parallel direct generation algorithm for *Compressed Implicit Octrees* (algorithm 6) on GTX Titan GPU. N refers to the number of leaves in the tree.

Figure 6.7 shows the breakdown of execution time for each kernel, that correspond closely to the steps of algorithm 6. The mnemonic name of each kernel is annotated above each measurement. For this test, we have chosen the input size of 8 million leaves (where million = 10^3). The experiment shows that the radix sorting of (key, value) pairs dominate the running time, with 64bit Morton keys, which allows trees of up to 21 levels. In fact, the expected time for sorting 32bit keys, is half of what is presented, in such case the maximum height of the tree lowers to 10 levels. We intend to also measure the algorithm times for 32bit and 128bit Morton keys, in this case the times will also change in other kernels. As we can observe in the chart, the sorting step, kernel (Srt), takes approximately half of the total time. The second kernel to take considerable time is kernel (LScatt) that scatters leaves. This is expected as this operation incurs many uncoalesced memory transactions. Surprisingly, the binary search step (BSlb) is quite fast compared to other operations.

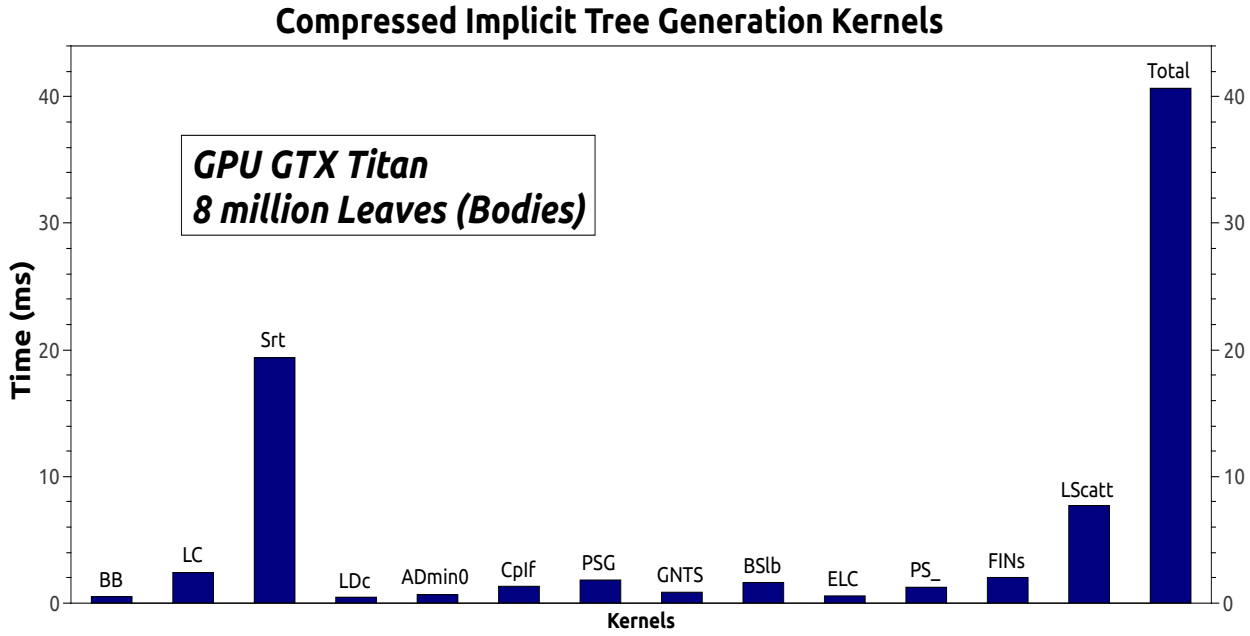


Figure 6.7: Breakdown of execution time for each kernel in algorithm 6 on GTX Titan GPU. The number of leaves in the tree for this experiment is 8 million leaves (where million = 10^3).

To verify the behavior of the parallel radix sort from Thrust version 1.8.1 we have tested the tree generation for different N, where N is the number of leaves. The radix sort of 64bit (key, value) pairs is used by the algorithm, and the sorting step behaves as linear with respect to N, as shown in graph 6.8. This version of radix sort implemented in the Thrust library is, to the best of our knowledge, the fastest GPU sort nowadays. This experiment confirms our expectations about running times of the parallel radix sort step, considering the 64bit width of the keys.

We have tested the direct compact implicit tree generation algorithm for different number of leaf nodes N, and instrumented the measurement of kernel (BSlb) which is step 8 of algorithm 6. Graph 6.9 shows these results, where we can observe that the binary search running time is linear with respect to N in practice. As stated before, since the input keys are sorted, but the results don't come out sorted, this linearity can not be explained only by the complexity of the vectorized binary search algorithm, as discussed in section 6.3.2.2. We conjecture that this linearity is a consequence of lemma 2 and we intend to look for a formal proof of this assumption in the future.

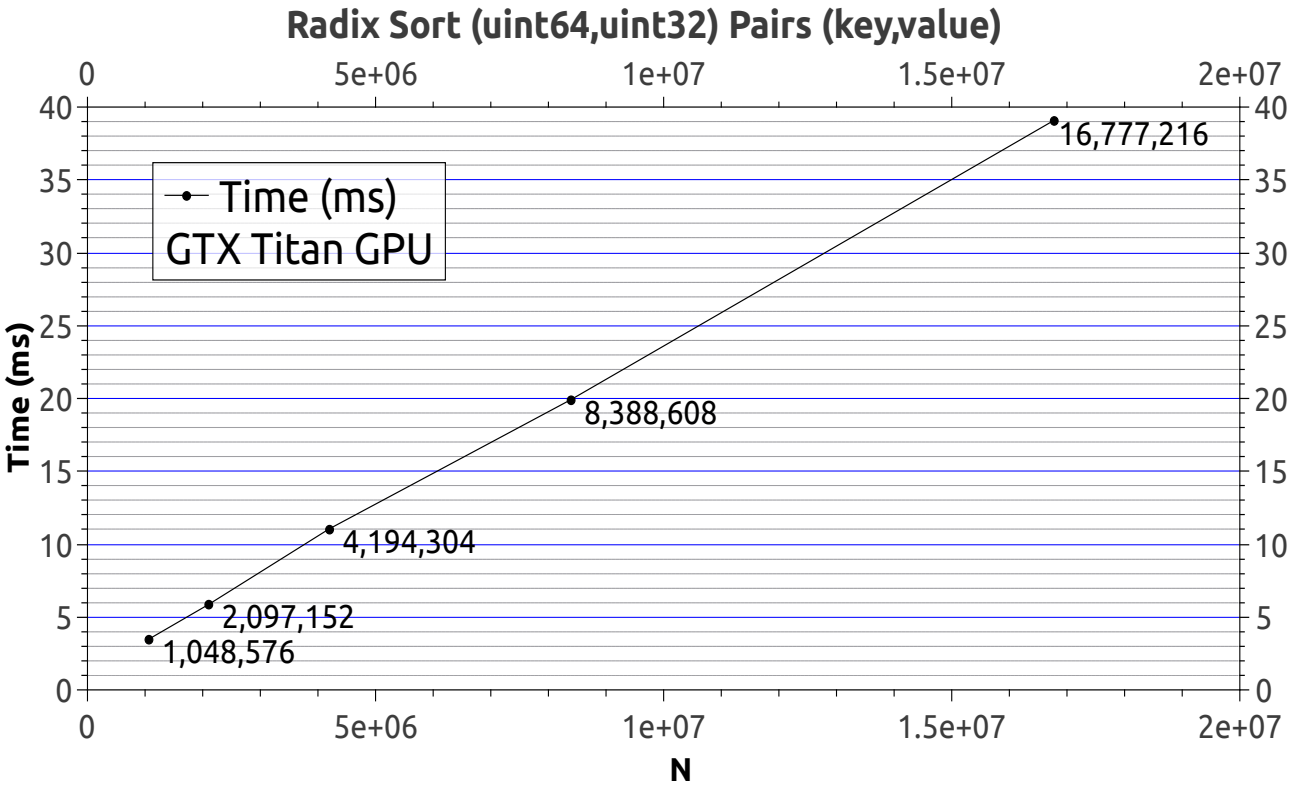


Figure 6.8: Running times of the parallel radix sort of 64bit (key, value) pairs on GTX Titan GPU, during compressed implicit octree build algorithm execution. N refers to the number of leaves in the tree, which is the same as the number of Morton keys sorted.

Our last experiment compares the parallel direct compressed implicit tree generation algorithm with the GPU implementations of the Barnes-Hut simulation kernels in package Lonestar GPU (Burtscher and Pingali, 2011), version 3.0. This is one of the fastest GPU implementations of Barnes-Hut simulation, and has efficient GPU kernels for the generation of pointer based octrees, which is a step of the simulation process. This comparison is presented in figure 6.10. As discussed in chapter 5 the use of the compressed implicit layout accelerates traversal steps of BH simulations, and the octree traversal/forces calculations phase dominates the simulation time. This way, to use the tree building kernels of (Burtscher and Pingali, 2011) we need to transform the pointer based octrees to the implicit octree layout. We have presented a parallel algorithm for this task in chapter 3. As the octree generation has to be used in conjunction with the transformation kernel (algorithm 2) we add the running times of this algorithm is our comparison.

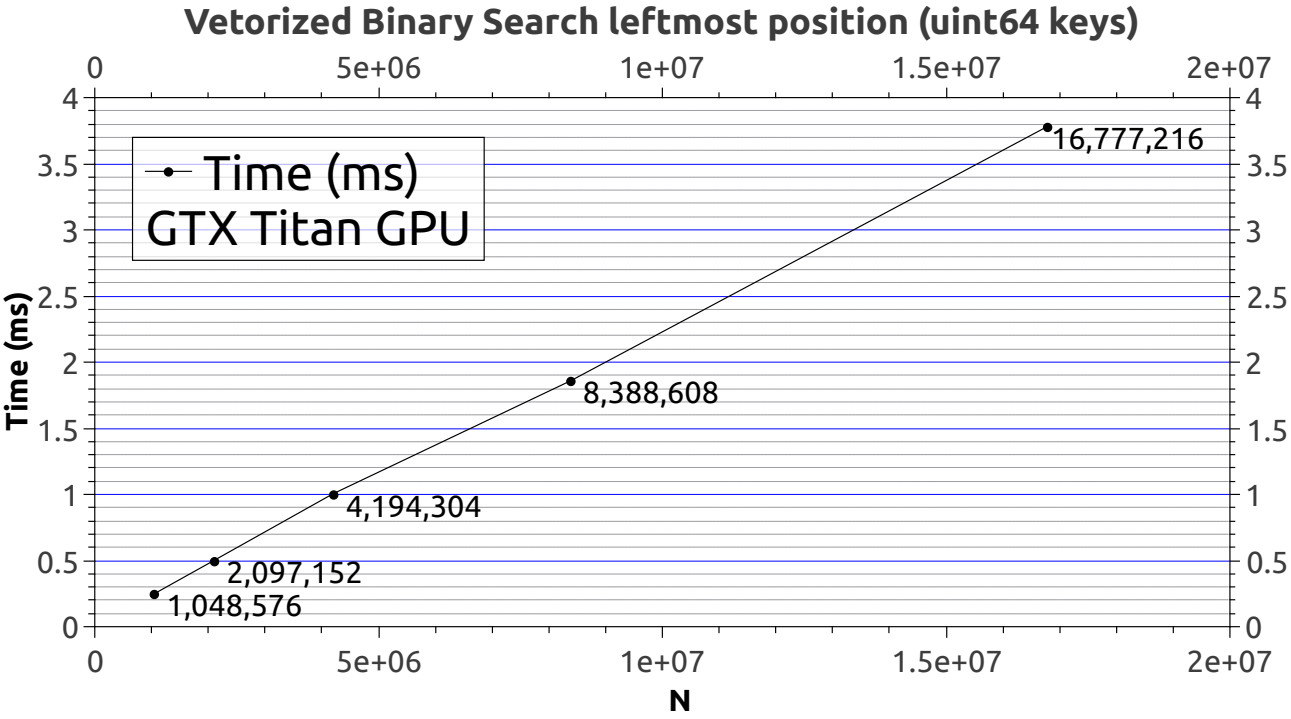


Figure 6.9: Running times of kernel (BSlb), setp 8 of algorithm 6 on GTX Titan GPU. Keys searched are 64bit Morton keys, during compressed implicit octree build algorithm execution. N refers to the number of leaves in the tree, which is the same as the number of Morton keys in the search space. The number of keys searched is O(N).

In graph 6.10 we compare algorithm 6 with the octree tree building kernel of reference (Burtscher and Pingali, 2011) alone (labelled B&P build), and tree building (Burtscher and Pingali, 2011) combined with our transformation algorithm 2 (ParTransform2CIT). Let us call this latter kernels the *combined* B&Pbuild + ParTransform2CIT. We observe that our algorithm 6 is up to 23% faster than the pointer based octree building kernel of (Burtscher and Pingali, 2011), and up to 312% faster when compared to the *combined* B&Pbuild + ParTransform2CIT. We can also observe that, algorithm 6 is always faster than combined B&P. Generation of octrees with B&P build is slightly faster of small workloads. One possible explanation is that B&P build executes only two GPU kernels, whereas algorithm 6 executes 15 distinct GPU kernels. There is an overhead in the spawning of GPU kernels that proportionally shows in the construction of smaller trees, as the overhead is constant and for smaller trees the total time is small. One possible solution to lower the kernel starting overhead could be by fusing some kernels of algorithm 6.

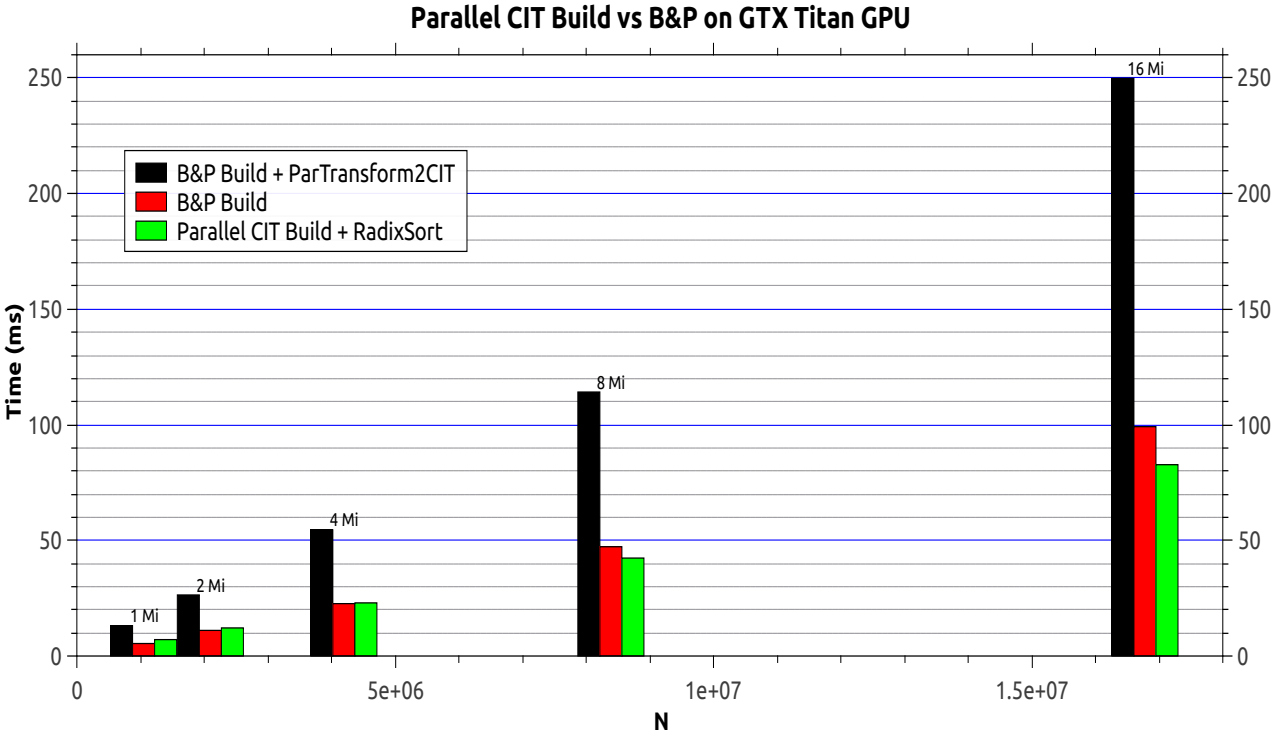


Figure 6.10: Comparison of the parallel direct generation algorithm for *Compressed Implicit Octrees* (algorithm 6) with octree generation of reference (Burtscher and Pingali, 2011) on GTX Titan GPU. N refers to the number of leaves in the tree (where Mi = 1,048,576).

In table 6.3 we present the running times of these kernels when used to generate octrees with 16Mi leaves. We show the tree building of (Burtscher and Pingali, 2011) combined with our transformation algorithm 2 (ParTransform2CIT) compared with our algorithm 6 (Parallel CIT build) including radix sort time. Our parallel algorithm achieves a speedup of 2.99 in this comparison.

Kernels	Time (ms)	Throughput (Million leaves/s)
B&Pbuild + ParTransform2CIT	249.8	67.2
Parallel CIT build + RadixSort	83.3	201.4
Speedup	2.99	

Table 6.3: Octree generation times, throughput and speedup for 16Mi leaves on GTX Titan GPU.

Next, in table 6.4 we present the running times of these kernels in Barnes-Hut simulations with 16Mi particles. The time measured in this table is the average for one step

of the simulation process. We show the simulation time of regular (Burtscher and Pingali, 2011) BH version 3.0 compared with our BH simulator using SWW and producing implicit octrees with B&P build with transformation of ParTransform2CIT. Our parallel algorithms achieve a maximum speedup of 2.21 in this comparison, running on GTX Titan GPU.

Kernels	Time (ms)
B&Pbuild + BHv3.0	3720.2
B&Pbuild + ParTransform2CIT + BH.SWW	1685.8
Speedup	2.21

Table 6.4: Average running times and speedup for one step of Barnes-Hut simulation, simulated wide-warps and B&P octree-build with 16Mi particles on GTX Titan GPU.

As a last experiment, in table 6.5 we also show the running times in Barnes-Hut simulations with 16Mi particles, in GTX Titan GPUs. The time measured in this table is the average for one step of the simulation process. We show the simulation time of regular (Burtscher and Pingali, 2011) BH version 3.0 compared with our BH simulator using SWW and *directly* producing compressed implicit octrees with our algorithm 6. Our parallel algorithms achieve the best speedup of 2.46 in Barnes-Hut simulations.

Kernels	Time (ms)
B&Pbuild + BHv3.0	3720.2
Parallel CIT build + RadixSort + BH.SWW	1510.4
Speedup	2.46

Table 6.5: Average running times and speedup for one step of Barnes-Hut simulation, simulated wide-warps and building compressed implicit octrees with our algorithm 6. Simulation performed with 16Mi particles on GTX Titan GPU.

Chapter

7

Conclusions

We have proposed an implicit octree data layout that accelerates GPU computation of forces in Barnes-Hut (BH) N-body simulations. The data layout presented can also be used with wide warps to produce more regular traversals on the tree and consequently minimize the effects on uncoalesced data accesses.

The tree traversal phase with forces calculations typically takes at least one order of magnitude more time than any other kernel in the simulation loop. The time spent on the data transformation phase to the implicit layout is amortized by the time reduction in the tree traversal kernel due to our optimization techniques.

The use of the implicit layout produces around 40% of improvement in the traversal kernel, not using SWW, when compared to the version 3.0 of the algorithm proposed by (Burtscher and Pingali, 2011) on Kepler architecture. Using the implicit layout and applying simulated wide warps raises the speedup to 268%, on traversals/forces calculations in Barnes-Hut simulations.

Our BH algorithm using implicit octree traversal is likely to accelerate on a GPU microarchitecture that implements large warps.

Wide warp execution can be effectively simulated in software, and be applied to carefully chosen regions of GPU kernels, where thread divergence is algorithmically constrained. Extra work is sometimes performed but the final execution time can be lower due to caching effects.

Regular algorithms are primary candidates to use SWW. Irregular algorithms where precision of calculations can be enhanced by performing extra work are potential candidates for acceleration using the SWW techniques presented in this work. We intend to investigate the applicability of SWW to other algorithms. We are also interested in extending our Barnes-Hut simulation code to work on multi-GPU clusters.

We have presented two algorithms for generating implicit octrees. The first algorithm transforms any existing pointer based octree to the equivalent tree in implicit layout. The use of the transformation algorithm and SWW techniques produces speedup of 2.21 in Barnes-Hut simulations. We have also presented a new massively parallel algorithm that directly generates compressed implicit octrees. Our new algorithm further accelerates N-body simulations, and raises the speedup to 2.46, also using SWW in traversals, when compared to the fastest GPU Barnes-Hut simulator from the literature. The massively parallel algorithm alone is approximately 3 times faster than producing a pointer based tree and transforming to implicit layout.

7.1 On the General Applicability of the Implicit Octree and Algorithms

Octrees are data structures that efficiently represent spatial data in many fields such as scientific computing, computer graphics and image processing, among others. In our study we have designed a new set of techniques to efficiently deal with the irregular nature of traversing octrees and the irregularity in producing sparse octrees. Our novel algorithm for direct generation of compressed implicit octrees is considerably regular. The proposed algorithm showed immediate application in our GPU parallel Barnes-Hut implementation. In fact, in certain cases even techniques or general principles applied for regular algorithms can be applied to irregular algorithms. As an example, we have previously worked on GPU AES encryption applications (Nunan Zola and Bona, 2012) which is fundamentally a regular algorithm. Techniques such as using thread Ids to avoid synchronizations between threads were developed, and similarly applied in algorithm 2.

Conversely, in this previous work we have studied ways to efficiently use registers to avoid shared or global memory latencies. This is, in some sense, the same principle applied to efficiently implement software wide warps (SWW).

For future work we would like to investigate the applicability of the compressed implicit octree generation and traversal in a multi-GPU Barnes-Hut simulator. A recent work that investigated this subject is (Zhang et al., 2015). We believe that we can obtain good efficiency of message exchange as a consequence of lemma 1. We also expect applicability of the techniques and algorithms in other contexts, such as in supporting "K Nearest Neighbour" queries (Li et al., 2012) (Leite et al., 2009) in compressed implicit octrees.

Bibliography

Ajmera, P.; Goradia, R.; Chandran, S.; Aluru, S. Fast, parallel, gpu-based construction of space filling curves and octrees. In: *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, New York, NY, USA: ACM, 2008, p. 10:1–10:1 (*I3D '08*,).

Aluru, S.; Sevilgen, F. Dynamic compressed hyperoctrees with application to the n-body problem. In: Rangan, C.; Raman, V.; Ramanujam, R., eds. *Foundations of Software Technology and Theoretical Computer Science*, v. 1738 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 21–33, 1999.

Asanovic, K.; Bodik, R.; Catanzaro, B. C.; Gebis, J. J.; Husbands, P.; Keutzer, K.; Patterson, D. A.; Plishker, W. L.; Shalf, J.; Williams, S. W.; Yelick, K. A. *The landscape of parallel computing research: A view from berkeley.* Relatório Técnico, EECS Department, University of California, Berkeley, 2006.

Barnes, J.; Hut, P. A hierarchical O(N log N) force-calculation algorithm. *Nature*, v. 324, p. 446–449, 1986.

Bédorf, J.; Gaburov, E.; Portegies Zwart, S. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *J. Comput. Phys.*, v. 231, n. 7, p. 2825–2839, 2012.

Bell, N.; Hoberock, J. Thrust: A productivity-oriented library for CUDA. In: Hwu, W.-m. W., ed. *GPU Computing Gems Jade Edition*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

BLELLOCH, G. Scans as primitive parallel operations. *Computers, IEEE Transactions on*, v. 38, n. 11, p. 1526–1538, 1989.

BURTSCHER, M.; PINGALI, K. Chapter 6 - an efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In: MEI W. HWU, W., ed. *GPU Computing Gems Emerald Edition*, Boston: Morgan Kaufmann, p. 75 – 92, 2011.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to algorithms, third edition.* 3rd ed. The MIT Press, 2009.

COUTINHO, B.; SAMPAIO, D.; PEREIRA, F. M. Q.; MEIRA JR., W. Divergence analysis and optimizations. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011, p. 320–329 (*PACT '11*,).

FOLEY, T.; SUGERMAN, J. KD-tree acceleration structures for a GPU raytracer. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, New York, NY, USA: ACM, 2005, p. 15–22 (*HWWS '05*,).

GORADIA, R. Gpu-based adaptive octree construction algorithms, unpublished paper, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, 2012.

GROEN, D.; PORTEGIES ZWART, S.; MCMILLAN, S.; MAKINO, J. Simulating n-body systems on the grid using dedicated hardware. In: BUBAK, M.; ALBADA, G.; DONGARRA, J.; SLOOT, P., eds. *Computational Science - ICCS 2008*, v. 5101 de *Lecture Notes in Computer Science*, 2008.

GUPTA, K.; STUART, J.; OWENS, J. A study of persistent threads style gpu programming for gpgpu workloads. In: *Innovative Parallel Computing (InPar), 2012*, 2012, p. 1–14.

HARIHARAN, B.; ALURU, S. Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods. *Parallel Computing*, v. 31, n. 3-4, p. 311 – 331, 2005.

Harris, M.; Sengupta, S.; Owens, J. D. Parallel prefix sum (scan) with cuda. In: Nguyen, H., ed. *GPU Gems 3*, Addison Wesley, 2007.

Hennessy, J. L.; Patterson, D. A. Chapter 4: Data-Level Parallelism in Vector, SIMD, and GPU Architectures. In: *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed, cáp. 4, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

Hoberock, J.; Bell, N. Thrust: A parallel template library. Version 1.8.1 release (18 Mar 2015), http://thrust.github.io/, 2015.

Ishiyama, T.; Nitadori, K.; Makino, J. 4.45 Pflops astrophysical n-body simulation on K computer: the gravitational trillion-body problem. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012, p. 5:1–5:10 (*SC '12*,).

Jenkins, J.; Arkatkar, I.; Owens, J. D.; Choudhary, A.; Samatova, N. F. Lessons learned from exploring the backtracking paradigm on the GPU. In: *Euro-Par 2011: Proceedings of the 17th International European Conference on Parallel and Distributed Computing*, 2011.

Jetley, P.; Wesolowski, L.; Gioachin, F.; Kalé, L. V.; Quinn, T. R. Scaling hierarchical n-body simulations on GPU clusters. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA: IEEE Computer Society, 2010 (*SC '10*,).

Knuth, D. E. *The art of computer programming, volume 4, fascicle 4: Generating all trees–history of combinatorial generation (art of computer programming).* Addison-Wesley Professional, 2006.

Lange, B.; Fortin, P. Astrophysical N-body simulations on multi-core and many-core architectures. In: *Euro-Par 2014: Proceedings of the 20th International European Conference on Parallel and Distributed Computing*, 2014.

Leite, P.; Teixeira, J.; de Farias, T.; Teichrieb, V.; Kelner, J. Massively parallel nearest neighbor queries for dynamic point clouds on the gpu. In: *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, 2009, p. 19–25.

Li, S.; Simons, L.; Pakaravoor, J. B.; Abbasinejad, F.; Owens, J. D.; Amenta, N. kANN on the GPU with shifted sorting. In: *High Performance Graphics*, 2012, p. 39–47.

Morton, G. *A computer oriented geodetic data base and a new technique in file sequencing.* International Business Machines Company, 1966.

Narasiman, V.; Shebanow, M.; Lee, C. J.; Miftakhutdinov, R.; Mutlu, O.; Patt, Y. N. Improving GPU performance via large warps and two-level warp scheduling. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, 2011, p. 308–317 (*MICRO-44 '11*,).

Nasre, R.; Burtscher, M.; Pingali, K. Data-driven versus topology-driven irregular computations on GPUs. In: *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*, IEEE, 2013.

Nunan Zola, W.; Bona, L. Parallel speculative encryption of multiple AES contexts on GPUs. In: *Innovative Parallel Computing (InPar), 2012*, 2012, p. 1–9.

Nunan Zola, W. M.; Bona, L. C.; Silva, F. Fast GPU parallel N-Body tree traversal with Simulated Wide-Warp. In: *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, 2014, p. 718–725.

NVIDIA Corporation *NVIDIA CUDA C Programming Guide.* 2013.

Nyland, L.; Harris, M.; Prins, J. Fast N-Body Simulation with CUDA. In: Nguyen, H., ed. *GPU Gems 3*, cáp. 31, Addison Wesley Professional, 2007.

Popov, S.; Günther, J.; Seidel, H.-P.; Slusallek, P. Stackless kd-tree traversal for high performance GPU ray tracing. In: *Computer Graphics Forum (Proc. EURO-*

GRAPHICS), European Association for Computer Graphics, Prague, Czech Republic: Blackwell, 2007, p. 415–424.

dos Santos, A.; Teixeira, J.; de Farias, T.; Teichrieb, V.; Kelner, J. kd-tree traversal implementations for ray tracing on massive multiprocessors: A comparative study. In: *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, 2009, p. 41–48.

Sengupta, S.; Lefohn, A. E.; Owens, J. D. A work-efficient step-efficient prefix sum algorithm, in: Workshop on edge computing using new commodity architectures. 2006.

Tanikawa, A.; Yoshikawa, K.; Nitadori, K.; Okamoto, T. Phantom-GRAPE: Numerical software library to accelerate collisionless n-body simulation with SIMD instruction set on x86 architecture. *New Astronomy*, v. 19, n. 0, p. 74 – 88, 2013.

Warren, M. S.; Salmon, J. K. A parallel hashed oct-tree n-body algorithm. In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, New York, NY, USA: ACM, 1993, p. 12–21 (*Supercomputing '93*,).

Zhang, E. Z.; Jiang, Y.; Guo, Z.; Shen, X. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, 2010 (*ICS '10*,).

Zhang, E. Z.; Jiang, Y.; Guo, Z.; Tian, K.; Shen, X. On-the-fly elimination of dynamic irregularities for GPU computing. In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, 2011 (*ASPLOS XVI*,).

Zhang, J.; Behzad, B.; Snir, M. Design of a multithreaded barnes-hut algorithm for multicore clusters. *IEEE Trans. Parallel Distrib. Syst.*, v. 26, n. 7, p. 1861–1873, 2015.

Appendices

Chapter A

Apendix: Another view of the Parallel solution to problem 6.2.1.(a)

Algorithm 8 Parallel solution to problem 6.2.1.(a)

Input : *Contiguous array of leaves in preorder*
Output: *Linear Tree array, with leaves in place and gaps to fit internal nodes in the preorder.*

1: allocate n threads, with thread numbers starting at 0;
2: **for** each thread i in parallel **do**
3: **if** $i = (n-1)$ **then**
4: $V_i \leftarrow 0$; ▷ LCA of the last leaf is root
5: **else**
6: $V_i \leftarrow LCA.level(leaf_i, leaf_{i+1})$;
7: **end if**
8: **end for**
9: $V_0 = V_0 - (-1)$; ▷ Note that V_0 operates with value $= -1$ (to generate space for root).
10: **for** each thread i in parallel **do**
11: **if** $i \neq 0$ **then**
12: $V_i \leftarrow V_i - V_{i-1}$; ▷ Adjacent differences, V_0 operates with implicit value -1
13: **end if**
14: **end for**
15: **for** each thread i in parallel **do** ▷ V_i has level differences, means size of gaps
16: $V_i \leftarrow (V_i > 0)\ ?\ (V_i + 1)\ : 1$; ▷ positive values, space for V_i internal nodes plus 1 leaf
17: **end for** ▷ negative or 0, space needed only for 1 leaf
18: $PS \leftarrow exclusiveAllPrefixSums(-1, V)$; ▷ prefix-sum of vector V with initial value -1
19: $linearTree \leftarrow scatterLeaves(PS, leaves)$; ▷ use prefix value PS_{i+1} as ▷ destination address to copy $leaf_i$ to final Tree array