Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 435–444

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Extending Perfect Spatial Hashing to Index Tuple-based Graphs Representing Super Carbon Nanotubes

Michael Burger[1,2], Giang Nam Nguyen[2], and Christian Bischof[2]

[1] Graduate School of Computational Engineering, Dolivostr. 15, Darmstadt, Germany
burger@gsc.tu-darmstadt.de
[2] Institute for Scientific Computing, Mornewegstr. 30, Darmstadt, Germany
{giang_nam.nguyen@stud., christian.bischof@sc.}tu-darmstadt.de

Abstract

In this paper, we demonstrate how to extend perfect spatial hashing (PSH) in order to hash multidimensional scientific data. As a use case we employ the problem domain of indexing nodes in a graph that represents Super Carbon Nanotubes (SCNTs). The goal of PSH is to hash multidimensional data without collisions. Since PSH results from the research on computer graphics, its principles and methods have only been tested on 2- and 3-dimensional problems. In our case, we need to hash up to 28 dimensions. In contrast to the original applications of PSH, we do not focus on GPUs as target hardware but on an efficient CPU implementation. Thus, this paper highlights the extensions to the original algorithm to make it suitable for higher dimensions. Comparing the compression and performance results of the new PSH based graphs and a structure-tailored custom data structure in our parallelized SCNT simulation software, we find that PSH in some cases achieves better compression by a factor of 1.7 while only increasing the total runtime by several percent. In particular, after our extension, PSH can also be employed to index sparse multidimensional scientific data from other domains where PSH can avoid additional index-structures like KD- or R-trees.

Keywords: perfect spatial hashing, data indexing, super carbon nanotubes, simulation

1 Introduction

Carbon nanotubes (CNTs) and super carbon nanotubes (SCNT) have drawn much attention because of their outstanding mechanical and electrical properties ([10], [4]). A CNT, which corresponds to an SCNT of order 0, can be imagined as a rolled up sheet of carbon atoms, arranged in a honeycomb grid. Following [4], to construct a super carbon nanotube of order 1 one replaces the single carbon atoms in the honeycomb sheet by Y-shaped junction-elements and each carbon-carbon-bond by a CNT. To construct an SCNT of order L one needs to replace the single atoms by appropriate Y-junctions and the bonds by order $L-1$ tubes. In our modeling approach, presented in [3], we employ sheets of level $L-1$ to construct Y-junctions with elongated arms of level $L-1$ used to replace the atoms and bonds to form a sheet of order L. This can either be used to form a tube of order L or to construct a junction of level L that can be employed to form higher order sheets.

We are working with directed graphs that represent our SCNT models. Each node corresponds to a carbon atom and each edge to a covalent C-C-bond. In contrast to many other graph applications, the distinct nodes are not labeled with numbers or letters but are identified by m-tuples. We call this type of graphs *tuple-based graphs*. Such a graph is shown in Figure 1, which represents the flat honeycomb grid of C-atoms that will result in a CNT when rolled up. Each node is identified by a tuple with four entries.

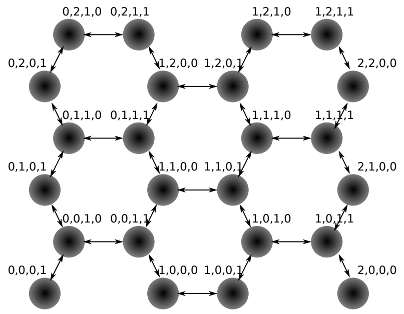

Figure 1: Graph with nodes labeled by tuples representing a honeycomb grid that can be rolled up to a tube.

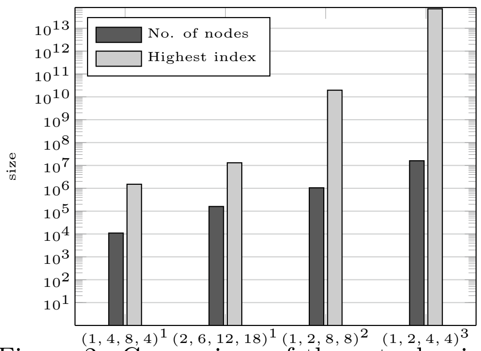

Figure 2: Comparison of the actual existing nodes in an SCNT model and the highest index calculated by tree-based flattening.

For efficient and compact storage as well as fast access to the node and edge data it is necessary to assign each tuple a unique index that we call the *serial index*. There exist some straightforward ways to serialize multidimensional data ([8], [3]) which allow a fast index-computation. There, an integer vector v of length m is mapped by $\phi(v) = \sum_{i=1}^{m} \left(v_i \prod_{j=i+1}^{m} (V_j + 1) \right)$ where V is an m-vector of the highest occurring values in each dimension of all v's and 0 is a possible value. This procedure is the reference for the comparison of the achieved index compression and is referenced as tree-based flattening in the following [3]. However, those schemes fail on large sparse data sets as they result from the tuple system. To give a first impression of this problem, Figure 2 compares, on a logarithmic scale, the actual number of nodes for several graphs with the highest index calculated by tree-based flattening. Four different tubes are shown with their configuration $(d_x, l_x, d_0, l_0)^L$ with d_x/l_x setting the diameter and the length of the Y-junction arms, d_0/l_0 determining the diameter and length of the tube and L setting the order of the tube. Details concerning the parameters can be found in [3].

The difference between the number of nodes and the occurring indices grows with the size and the order of the tube. For order 3 tubes the difference is already six orders of magnitude. Thus, there is the need for an alternative method to map each tuple to a serial index without occupying too large an index space for sparse tuple-data, while being able to calculate the transformation efficiently. One existing approach to cope with sparse, spatial data is the so called perfect spatial hashing (PSH) developed by Lefebvre and Hoppe [7] that has been employed for space-saving storage of 2D and 3D graphics data on graphic cards. In this paper, we demonstrate that this algorithm can be extended to the storage of high-dimensional sparse data and employ our tuple-based graphs as a use case. Other possible fields of application may arise from fields in which multidimensional measurement data needs to be managed as it is, for example, the case in genomics or cancer surveillance.

2 The Tuple System

Our SCNTs are modeled by directed graphs as shown in Figure 1. To uniquely identify the nodes within the graphs, they are labeled by m-tuples $t = (x_m, x_{m-1}, \dots, x_1)$ where x_m is called the *leading tuple entry* and x_1 the *lowest tuple entry*, respectively. The tuples code the hierarchy- and symmetry-relations between nodes as well as their spatial positions as described in detail in [3]. The distinct entries in a tuple can be accessed via subscript operator t_j with $j \in [1, \dots, m-1, m]$. Assume we have a set S of n tuples t_i with $i \in [1, 2, \dots, n]$, then the so called *tuple extent* is defined as the tuple $t^{\text{ext}} = (\max t_{i,m}, \max t_{i,m-1}, \dots, \max t_{i,1})$. The tuple extent determines an upper bound on all possible tuples that can be created.

In general, SCNTs of order L are formed by Y-junctions of level $L-1$, themselves formed by Y-junctions of level $L-2$, and so on, down to the level (-1) of single atoms with three bonds to their neighbors each. This hierarchy is also visible within the tuple system and a general tuple identifying a node in an order L SCNT is of the form:

$$t = \underbrace{x_m \;\; x_{m-1} \;\; x_{m-2} \;\; x_{m-3}}_{\text{tube order L}} \underbrace{x_{m-4} \;\; x_{m-5} \;\; \dots \;\; x_{m-10} \;\; x_{m-11}}_{\text{Y}-\text{junction level L}-1} \;\; \dots \;\; \underbrace{x_8 \;\; x_7 \;\; \dots \;\; x_2 \;\; x_1}_{\text{Y}-\text{junction level 0}} \qquad (1)$$

Equation 1 shows that always eight subsequent tuple entries code the information for a Y-junction of a certain level. The four highest tuple entries additionally contain the information of the diameter and the length of tube. They are called tube part and the remainder of the tuple junction part.

2.1 Theory of Perfect Spatial Hashing

PSH, as described in [7], works on a d-dimensional domain U where $d \in [2, 3]$ because 2D and 3D graphics are considered. The domain U contains all positions p in the original PSH. In each dimension the entries can assume $\overline{u}$ discrete entries from $\{0, 1, \dots, \overline{u} - 1\}$. Thus, the domain U can have in total $u = \overline{u}^d$ entries/data points when fully occupied. For SCNTs we have m dimensions with $m \in [4, 12, 20, 28]$ because of the respective tuple lengths for orders 0 - 3. There, the cardinality of U is determined by the tuple extent t^{ext} with $u = \prod_{i=1}^{m} t_i^{ext}$.

A subset $S \subset U$ denotes all those positions p_i that are occupied in U e.g. in [7], all pixels with colors. Its cardinality is given by n. This subset S corresponds to all actually existing tuples t_i within a given graph. The density ρ of the data is given by the fraction $\rho = |S|/|U| = n/u$. Each position p_i is associated with a data record $D(p_i)$. For the picture example this could be the color information of the pixel. Accordingly, in the tuple-based graphs each tuple t_i is associated with the information of a node $D(t_i)$ like its position or adjacent edges.

PSH tries to map the sparsely defined data $D(p_i)$ with $p_i \in S$ to a record in a dense hash map H by a hash function $h : D(p_i) \to H\left[h(p_i)\right]$. *Perfect hashing* means that there are no collisions in the hash map, i.e. for all $i, j \in [1, 2, \dots, n]$ and $i \neq j$ it holds that $h(p_i) \neq h(p_j)$. This is achieved through the combination of two imperfect hash functions h_0 and h_1, combined with an offset table Φ with $\overline{r}^d = r$ entries. The resulting hash table H has $\overline{m}^d = m$ entries and the perfect hash function h is described for each dimension by: $h(p) = h_0(p) + \Phi\left[h_1(p)\right] \bmod \overline{m}$.

The procedure of fetching two tuples t_1 and t_2 with the help of h is visualized in Figure 3. It is assumed that both tuples collide in h_0. Therefore, Φ requires an entry for $h_1(t_1)$ and one for $h_1(t_2)$ to resolve the collision and avoid further collision with other tuples in H.

The two involved imperfect hash functions are defined as $h_0 = M_0 p \bmod \overline{m}$ and $h_1 = M_1 p \bmod \overline{r}$. Since Lefebvre and Hoppe [7] found that M_0 and M_1 can be simply set to $m \times m$

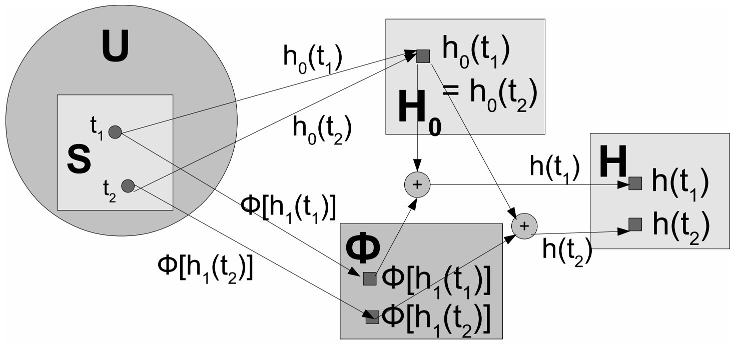

Figure 3: Fetching the data for two tuples t_1 and t_2 in the case of a collision in h_0. Figure based on Figure 2 of [7].

identity matrices, the hash function evaluation of h_0 and h_1 reduces to an application of the mod $\overline{m}$ and mod $\overline{r}$ calculation to each entry of the tuples with the additional recommendation that $\overline{r}$ and $\overline{m}$ do no have a common divisor.

2.2 Evaluation of Perfect Spatial Hashing for Indexing Tuples

Following [7], PSH has several properties that directly fit to the problem domain of tuple-based graphs: (1) It compresses spatial multidimensional data into small, dense tables, decreasing the tuple extent of the hashed tuples. (2) It increases the density of the hashed domain and allows to derive a serial index on the hashed domain quickly by tree-based flattening. (3) PSH avoids collisions and thus makes the calculated serial indices unique, which is a prerequisite for the SCNT simulation. (4) In contrast to most other perfect hash functions, PSH seeks also to create a minimal function with a fully occupied hash table H, resulting in minimal serial indices. (5) Another positive property of PSH is the uniform access time. A query always requires two search operations in two tables Φ and H. For other hashing schemes resolution of collisions leads to different times.

PSH also has some drawbacks compared to other hashing schemes but they are no severe problem for the tuple-based graphs. PSH only works on static data sets, but the structure of SCNTs is constant during simulation. The construction of the offset table Φ is complex and the process may fail, requiring several restarts. But this pre-computing step needs to be done once before the SCNT simulation and so its runtime is not crucial, particularly as the hashing scheme can be stored and reused.

In addition, compared to [7], some simplifications are possible. In contrast to spatial positions, tuples are discrete data with no need for discretization or rounding. Furthermore, no normalization of data is required since CPUs have direct integer support. There is also no problem with false positives since the solving algorithm never requests non-occupied tuples avoiding to store an additional integer per hash entry to identify false positives as done in [7].

But there are two main differences to the original work we have to cope with. First, PSH has only been applied to 2D and 3D problems, but we have up to 28 dimensions. Section 4 highlights the effect of this difference on the algorithm and the underlying data structures. Second, for all their tests in [7], the authors only employed spaces with quadratic or cubic shape which simplifies the problem and particularly leads to minimal hash functions more easily.

3 Related Work

Before deciding to extend PSH, we also considered several other spatial hashing schemes. Garcia et al. developed another coherent spatial hashing scheme [5] capable of a fast parallel hash table construction that succeeds in most cases with the first or at least very few trials. The authors state that they reach a higher memory coherence than [7]. The imbalance in the amount of memory accesses per entry is compensated by the usage of some additional memory. However, they gave up the concept that every resulting hashing scheme needs to be perfect.

Another non-perfect spatial hashing scheme has been proposed by Pozzer et al. [9] targeting CPUs in the application domain of collision detection. They employ fixed-size vectors and pivots, i.e. static data structures and compare the construction and access performance with two spatial hashing algorithms employing dynamic structures by Buckland [2] and Hastings [6]. In both cases they are at least three times faster. PSH is mentioned as related work but not taken into account for comparison which also employs static data structures.

Alcantara et al. [1] propose a parallel, spatial hashing scheme based on a two step procedure of FKS perfect hashing and cuckoo hashing targeted on GPU architectures. Their goal is to create a hash table in which each item can, even in the worst case, be accessed in constant time. Compared to PSH the access time of items is at the same level, while the construction of the hash structure is faster by some orders of magnitude. The memory overhead during construction is $1.42 * n$ compared to $1.12 * n$ of PSH. Normally, about 70% of the possible entries in the hash structure are used. However, their hash function requires integers as input. Their approach of encoding the 3D coordinates of a voxel by 10 bit of a 32 bit integer each, is not feasible for higher dimensions.

Ng and Ravishankar [8] present a scheme for indexing large data bases that can also be employed to calculate a unique serial index for multidimensional data as we have demonstrated in [3]. With a modified method of [8], a fast and straightforward index computation is possible with the drawback that the maximum index grows very fast for the sparse tuples spaces.

4 Implementation

This section describes the details of our C++ implementation and highlights our extensions to PSH for higher dimensional domains.

4.1 Creating the Offset Table

The construction of the offset table Φ is an NP-hard problem. For each $h_1(t)$ an appropriate offset must be found avoiding collisions in H. Thus, in principle, the choice of an entry is dependent on all other entries whose number can be several millions. Hence, the construction is realized with a trial and error approach like in [7] and based on several heuristics.

We change several things to optimize the speed of construction and the quality of the result. First of all, we do not stick to a constant $\overline{r}$ and $\overline{m}$ but allow vectors $\overrightarrow{r} = (\overline{r}_1, \overline{r}_2, \ldots, \overline{r}_d)$ and $\overrightarrow{m} = (\overline{m}_1, \overline{m}_2, \ldots, \overline{m}_d)$ with different entries $\overline{r}_k$ and $\overline{m}_k$ for each dimension to reduce the values of r and m which are defined by $r = \prod_{k=1}^{d} \overline{r}_k$ and $m = \prod_{k=1}^{d} \overline{m}_k$, respectively. High values of r lead to a large Φ since $h_1(t)$ will map each t to a distinct slot, while a large m results in high serial indices.

For the instantiation of the iterative construction, we apply two heuristics. Initially, all $\overline{m}_k$ are set to the smallest possible and uniform $\overline{m}$ satisfying $\overline{m} \geq \sqrt[d]{n}$. All entries $\overline{r}_k$ are set to the smallest possible $\overline{r}$ satisfying: $\overline{r}^d \geq \sigma n$ with $\sigma = 1/2d$ as proposed by [7]. Now, the value of m

is compared to n. If m is at least higher by a factor of 3, the entries within the respective tuple extent t^{ext} are searched that have the greatest distance to $\overline{m}$. At these positions the entry $\overline{m}_k$ is decremented by 1 as long as $\prod_{k=1}^{d} \overline{m}_k \geq n$.

In the second step, a first feasibility check is performed ensuring that there are no two tuples t_1, t_2 with $h_0(t_1) = h_0(t_2) \wedge h_1(t_1) = h_1(t_2)$ which makes the construction of a perfect hash function impossible. This check can be performed very efficiently with an additional table $T^{\text{col}-\text{h}_1}$ that is required for the next step anyways. It stores those sets of tuples that are mapped to the same hash value by h_1. For each tuple t_i the hash-value $h_1(t_i)$ is calculated. Now, a lookup on the table $T^{\text{col}-\text{h}_1}$ is performed that has $h_1(t_i)$ as its key value. If the key $h_1(t_i)$ does not exist, a new vector is inserted that has a reference to t_i as first entry. In the case that there already is an entry for $h_1(t_i)$, the reference to t_i is appended to the existing vector. In this way, just a small percentage of tuples will be left, divided in sets with a typical size $\ll 100$. Since only nodes within the same set can violate $h_0(t_1) = h_0(t_2) \wedge h_1(t_1) = h_1(t_2)$ it is sufficient to test every tuple t_i against the other tuples in this set for a collision in h_0. If there is only one collision, $\overline{r}_k$ and $\overline{m}_k$ need to be updated. Since the main goal is to assign minimal serial indices to the tuples, the algorithm primarily tries to adapt the $\overline{r}_k$ values by searching those entries with the highest distance to their respective entries in t^{ext} and incrementing them by one. Out of the tuples resulting in collisions, k are picked at random and the colliding positions in the $\overline{r}_k$'s are also incremented as long as all k collisions are resolved. The hope is that many more, ideally all, collisions are avoided due to these changes. The values presented here were obtained with $k = 1$ resulting in the most optimization steps but the lowest possible values for r and m. We also tested $k = 10$, resulting in a faster construction but slightly higher values for r. Afterward, the second step is repeated until it succeeds.

The third step tries to construct an offset table Φ for the given $\overrightarrow{r}$ resolving all collisions in h_0. To that end, we apply the heuristic of [7] and start to find offsets for the biggest colliding sets of U in $T^{\text{col}-\text{h}_1}$. From this auxiliary table it is easy to deduce the order in which the sets should be processed. For each set an offset is randomly generated and tested for feasibility. If an appropriate offset is found, i.e. it resolves existing collisions without creating new ones, it is inserted into Φ. Then, the algorithm proceeds with the next largest set until all collisions are covered. In the case that it is not possible to find an appropriate offset for a set c_l, i.e. the l's set that is processed, the algorithm applies backtracking. The chosen offset for the last set c_{l-1} is changed and afterward c_l is processed again. If this attempt does not succeed, the offset for c_{l-2} is changed. After several unsuccessful attempts the algorithm goes back to step two.

This three step procedure is repeated until a PSH is found or canceled when m and r become too high to achieve a sufficient reduction.

4.2 Dynamic Data Structures and 1-dimensional Keys

Since PSH was designed to work on GPUs, static multidimensional arrays are used for the tables Φ and H. The size of these tables is $\overline{m}^d$ and $\overline{r}^d$ and the hashed 2D and 3D coordinates are used to access the entries. This is feasible under the simplifying assumption that the problem domains are always quadratic or cubic. For the case of longer tuples with unequal dimensions this procedure wastes much memory. Table 1 visualizes this fact for several tubes. For example, for $(1, 6, 12, 18)$[1] nearly $1.6 * 10^7$ entries will be statically allocated for Φ while only $1.7 * 10^4$ entries are required. To cope with this problem, we employ the dynamic std::unordered_map which only stores existing values and the search for an entry is possible in constant time.

Another problem is the usage of the tuples and their hashes as keys for Φ and H, respectively, since a lot of memory is required when storing the vectors representing the tuples. Thus, we employ serial indices as keys that are calculated on the basis of the knowledge of the tuples and h_0, h_1, m and r. The keys for Φ are calculated by applying h_1 to the current tuple t and then this hashed tuple is serialized by tree-based flattening with $\overrightarrow{r}$ as extent. For order 3 tubes this means that instead of 28 16 bit-values only one 64 bit-value is required, which reduces the storage for $1*10^7$ entries in H for the keys by a factor of 7 from 530 MB to 75 MB. Additionally, long tuples cause high construction- and access-time. Generating a map with $1*10^7$ randomized 28-vectors as keys and values requires about 30 s on our test system, while the same test is done in about 10 s if the vectors are serialized and inserted in an integer based map. In this way, we can avoid the drawbacks of vectors as keys by exploiting our knowledge about the tuples.

4.3 Density Threshold and Different Compression Modes

The construction of a perfect hash function always succeeds if r or m are chosen large enough, since either Φ contains an entry for every t_i or H allocates so many slots that for each tuple a slot can be directly assigned. Then, it is possible that H reserves more slots than $|U|$ and the calculated serial indices are partly much higher than those calculated on the original domain.

An analysis of those cases reveals that they normally appear if the domain U is occupied relatively dense. This means that $|S|$ is not much smaller than $|U|$. To exclude those cases, we have empirically determined a density threshold $\rho_{\text{thres}} = |S|/|U| < 0.15$. For all domains with higher ρ_{thres} it is not tried to construct a full perfect hashing function because of the low probability of finding a sufficiently compact one. Instead, only a perfect hashing on the whole junction part i.e. the part of the tuple after the four highest entries $[x_m, \dots, x_{m-3}]$ is generated, since $[x_m, \dots, x_{m-3}]$ are usually relatively dense and thus can prevent a compact hash function. The resulting sub-tuple is appended to the original tube part and this new tuple is serialized by tree-based flattening. The compression of the whole domain is called *mode 1*, the compression of the junction part *mode 2*.

The reduction factor as a measurement of the efficiency of PSH is defined by $\psi = |U|/|H|$. For mode 2, ψ is equal to the reduction factor ψ^{juncs} achieved on the junction part of the tuple.

5 Results

Tests for the compression quality and the speed of calculation were performed on a dual socket machine (2 * Intel Sandy Bridge Xeon E5-2670 processor with 8 cores each) running CentOS 7 with 128 GB of shared DDR3 ECC RAM. The g++ compiler (version 4.8.5) with optimization level O3 and the GNU OpenMP implementation are employed.

5.1 Compression Results

We compare the reduction factor for the serial indices between general PSH to a structure-tailored index calculation scheme described in [3] that we call *IndexGraph*. Summarized briefly, the IndexGraphs are based on a hierarchy of maps that compress each junction level separately. For each junction level an internal serial index is calculated, consecutively numbered and the correlation stored in a table. This allows the assignment of the smallest possible indices to the junction part. The dense part of the tuples corresponding to the tube gets its index via tree-based flattening. Combining both leads to the serial index for the whole tuple.

Figure 4 summarizes the reduction factors for the different tested tubes, whose properties are listed in Table 1, on a logarithmic scale. Configurations that are marked by an asterisk over their bar use compression mode 2 because it reaches the better result while all others are hashed

Table 1: Summary of the tested tubes. For each configuration, the second column shows the number of tuples to hash. Columns $r = \overline{r}^d$ and $m = \overline{m}^d$ give the values achieved by the original PSH algorithm while r^* and m^* give the new values resulting from our extended procedure. $\%\Phi$ shows the fraction: (used slots in Φ) / (No. tuples), while the last column gives the time to construct PSH in seconds.

Configuration	No. tuples	r	r^*	m	m^*	$\%\Phi$	Constr.(s)
$(1,4,8,4)^{1*}$	$1.7*10^4$	$6.6*10^3$	$9.6*10^2$	$2.6*10^2$	$2.6*10^2$	99%	<1
$(1,4,8,14)^{1*}$	$4.5*10^4$	$6.6*10^3$	$9.6*10^2$	$2.6*10^2$	$2.6*10^2$	99%	<1
$(1,6,12,18)^1$	$1.3*10^5$	$1.6*10^7$	$9.4*10^5$	$5.3*10^5$	$1.6*10^5$	13%	<1
$(2,6,12,18)^1$	$1.7*10^5$	$1.6*10^7$	$9.4*10^5$	$5.3*10^5$	$2.4*10^5$	13%	<1
$(2,6,12,116)^1$	$1.0*10^6$	$1.3*10^{14}$	$2.7*10^6$	$1.7*10^7$	$1.0*10^6$	6%	4.7
$(2,3,6,12)^{2*}$	$5.6*10^6$	$4.3*10^7$	$3.3*10^5$	$6.6*10^4$	$6.6*10^4$	97%	<1
$(1,2,8,8)^2$	$1.0*10^6$	$9.5*10^{13}$	$2.0*10^7$	$1.0*10^6$	$1.0*10^6$	38%	11.8
$(1,2,4,4)^3$	$2.5*10^7$	$2.3*10^{13}$	$6.9*10^9$	$2.7*10^7$	$2.7*10^7$	75%	370

by mode 1. In all cases, our extended PSH algorithm achieves a reduction of the maximum index of the same order of magnitude as the IndexGraph, although the IndexGraphs exploit properties of SCNT models like repetitions in the tuple extent while PSH can be applied to generic data.

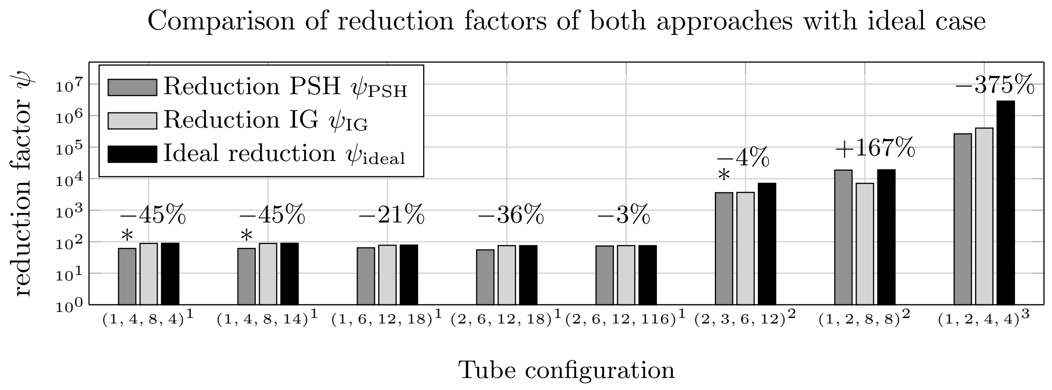

Figure 4: The reduction factors achieved by PSH (ψ_{PSH}) and by our structure-tailored IndexGraph (ψ_{IG}) for different tube configurations compared to the ideal reduction (ψ_{ideal}) calculated by the fraction of the number of tuples of the configuration and the highest index resulting from tree-based flattening. The numbers over the bars show the relative difference of both indexing procedures by: $\psi_{\text{PSH}}/\psi_{\text{ideal}} - \psi_{\text{IG}}/\psi_{\text{ideal}}$.

The order 1 tube $(2,6,12,116)^1$ demonstrates the efficiency of our algorithmic extensions compared to the standard PSH approach. The tube is characterized by highly varying entries within its tuple extent $t^{\text{ext}} = (59\,12\,2\,2\,|\,6\,3\,3\,8\,4\,4\,2\,2)$, which is a problem for the original PSH. The constraint of choosing a uniform $\overline{m}$ forces the algorithm to $\overline{m} = 4$ and $\overline{r} = 15$, as demonstrated in Table 1, in order not to violate the prerequisite $\nexists t_1, t_2 \in S \mid h_0(t_1) = h_0(t_2) \wedge h_1(t_1) = h_1(t_2)$. These values are actually higher than those for the order 2 and 3 tubes with longer tuples and more nodes. The resulting reduction factor would only be 5 in that case while the IndexGraph reaches the ideal value of 75. In our algorithm, the original $\overline{r}^{12} = 15^{12} = 1.3*10^{14}$

is replaced by the new vector $\overrightarrow{r} = (1\,5\,3\,3\,3\,|\,3\,3\,3\,3\,3\,3\,3\,3)$. But most important is that also m can be reduced by one order of magnitude by replacing the uniform $\overline{m} = 4$ with the vector $\overrightarrow{m} = (4\,4\,2\,2\,|\,4\,4\,4\,4\,4\,4\,2\,2)$.

For the remaining order 1 tubes the IndexGraph and the PSH achieve reduction factors in the range of the ideal value regardless of the chosen compression mode where for $(2, 6, 12, 18)^1$ m can be reduced by the factor 2.2 compared to the original PSH and even a factor of 3.3 for tube $(1, 6, 12, 18)^1$. r can be decreased by over one order of magnitude for these tubes.

For the tube $(1, 2, 8, 8)^2$ the reduction through PSH is even higher than that of the IndexGraphs and nearly reaches the optimum (18662 versus 19109). This tube is a kind of best-case scenario for PSH. It consists of $1,024,000$ nodes which is very close to $2^{20} = 1,048,576$ with 20 being the tuple length for an order 2 tube. In this case $\overline{m} = 2$ is chosen, resulting in a hash table H with $1,048,576$ available slots. In combination with an appropriate Φ all nodes can be assigned a slot, resulting in an occupation of 98% and hence nearly a minimal hash function.

As a last point, the size of the table Φ is investigated which is not crucial for performance but for the memory consumption of the PSH structure and is the main difference concerning the required storage compared to IndexGraphs whose small tables can be neglected. The second to last column of Table 1 shows how many slots are used in Φ in relation to the overall number of tuples in the respective tube. For the three mode 2 tubes nearly every tuple is hashed to a different slot. That is not an issue, since in these cases the number of tuples to hash is equal to the number of nodes within a junction, which is much smaller than the total number of nodes in the tube, e.g. 176 compared to $1.6 * 10^5$ for $(2, 6, 12, 18)^1$. As already mentioned, the size of r influences the size of Φ with a higher r leading to more entries. Hence, the optimization of r can also reduce the size of Φ. This can be demonstrated for $(2, 6, 12, 116)$ where the number of entries is reduced from $2.5 * 10^5$ to $6.5 * 10^4$ and thus from 25% to 6%. But also for $(1, 2, 8, 8)^2$ there is a reduction from $6.4 * 10^5$ to $3.8 * 10^5$ entries. The only instance where PSH does not perform that good is the order 3 tube which requires a relatively dense offset table, caused by the complexity of the 28-dimensional problem, requiring further optimizations.

5.2 Performance Results

In a first test, we performed a search of about $5 * 10^7$ random tuples on different tube configurations with PSH and with IndexGraphs. The tests show, that on these synthetic benchmarks, the search time for PSH is higher by a factor about 1.5 with values ranging from 1.26 to 1.67.

The influence of PSH on our overall solving routine is not that significant. For single-threaded execution, the performance difference lies below 10%. However, PSH has a negative influence on the scaling behavior of the solver, so the difference grows to about 20% when running with four threads and this difference remains when running with 8 and 16 threads.

The runtime for the construction of PSH plays, as explained, an subordinate role. But we also measured the time to construct the perfect hash function for all tubes, which is shown in the last column of Table 1. For nearly all tubes, the PSH can be generated within a few seconds. Only for the order 3 tube about 6 minutes are required. We are confident that there is potential to further optimize the runtime of the construction process.

6 Summary and Outlook

We presented a generalization of the perfect spatial hashing algorithm that is able to cope with high dimensional data as well as varying size per dimension. It was successfully applied to the problem of indexing nodes in tuple-based graph models of SCNTs. The resulting algorithm is competitive with a structure-tailored scheme exploiting special SCNT properties although PSH is a general scheme that has no assumptions about the data to hash. Required extensions to the original algorithm were proposed and an efficient implementation was outlined.

In the future, we want to apply PSH to other application domains as well by adapting the presented methodology to the respective problem domain and compare it to domain-specific solutions.

Furthermore, we plan to further improve the performance and quality in determining $\vec{r}$ and $\vec{m}$ because, for some cases, it seems to be reasonable to make a compromise between the size of r and $|\Phi|$ or to accept a slightly higher m to reduce collisions. We also plan to investigate ways of reducing the influence of PSH on the scalability of our solving routine.

Acknowledgement

The work of M. Burger is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt (TUD). The calculation have been performed on the Lichtenberg-Hochleistungsrechner at TUD. We thank M. Gösele for introducing PSH to us and the anonymous reviewers for their helpful comments.

References

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[2] M. Buckland. *Programming game AI by example*. Wordware Pub Co, first edition, 2004.

[3] M. Burger, C. Bischof, C. Schröppel, and J. Wackerfuß. Methods to model and simulate super carbon nanotubes of higher order. *Concurrency and Computation: Practice and Experience*, 2016.

[4] V. R. Coluci, D. S. Galvao, and A. Jorio. Geometric and electronic structure of carbon nanotube networks:'super'-carbon nanotubes. *Nanotechnology*, 17(3):617, 2006.

[5] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 161:1–161:8, New York, NY, USA, 2011. ACM.

[6] E. J. Hastings, J. Mesit, and R. K. Guha. Optimization of large-scale, real-time simulations by spatial hashing. In *Proc. 2005 Summer Computer Simulation Conference*, volume 37, pages 9–17, 2005.

[7] S. Lefebvre and H. Hoppe. Perfect spatial hashing. In John Finnegan and Julie Dorsey, editors, *ACM SIGGRAPH 2006 Papers*, page 579, 2006.

[8] W. K. Ng and C. V. Ravishankar. Block-oriented compression techniques for large statistical databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):314–328, 1997.

[9] C. T. Pozzer, C. A. de Lara Pahins, and I. Heldal. A hash table construction algorithm for spatial hashing based on linear memory. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, ACE '14, pages 35:1–35:4, New York, NY, USA, 2014. ACM.

[10] Z. Qin, X. Feng, J. Zou, Y. Yin, and S. Yu. Superior flexibility of super carbon nanotubes: Molecular dynamics simulations. *Applied Physics Letters*, 91(4):043108, 2007.