Compact Data Structure and Scalable Algorithms for the Sparse Grid Technique

Alin Murarașu Josef Weidendorfer Gerrit Buse Daniel Butnaru Dirk Pflüger

Technische Universität München

{murarasu,weidendo,buse,butnaru,pflueged}@in.tum.de

Abstract

The sparse grid discretization technique enables a compressed representation of higher-dimensional functions. In its original form, it relies heavily on recursion and complex data structures, thus being far from well-suited for GPUs. In this paper, we describe optimizations that enable us to implement compression and decompression, the crucial sparse grid algorithms for our application, on Nvidia GPUs. The main idea consists of a bijective mapping between the set of points in a multi-dimensional sparse grid and a set of consecutive natural numbers. The resulting data structure consumes a minimum amount of memory. For a 10-dimensional sparse grid with approximately 127 million points, it consumes up to 30 times less memory than trees or hash tables which are typically used. Compared to a sequential CPU implementation, the speedups achieved on GPU are up to 17 for compression and up to 70 for decompression, respectively. We show that the optimizations are also applicable to multicore CPUs.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming - Parallel Programming; G.1.2 [*Mathematics of Computing*]: NUMERICAL ANALYSIS - Approximation

General Terms Algorithms, Performance

Keywords Sparse grids, GPU, Performance optimization

1. Introduction

The numerical representation and treatment of functions in higher-dimensional settings suffer the so-called *curse of dimensionality*, the exponential dependency on the number of dimensions. Consider a piecewise d-linear interpolation of a d-dimensional function based on a spatial discretization of the domain of interest, e.g.: spending $\tilde{N}$ grid points in each dimension leads to a full grid with $\tilde{N}^d$ grid points. Thus, the treatment of functions in more than four variables is practically impossible for reasonable discretizations. In applications requiring efficient numerical techniques for multi-dimensional functions – as they occur in computational steering –, this is clearly an obstacle.

Sparse grids enable one to mitigate the curse of dimensionality to some extent, allowing to tackle dimensionalities that are of interest in engineering settings where models depend on a moderate number of variables. They significantly reduce the number of grid points while maintaining similar approximation accuracies as obtained for full grids. Thus, they are well-suited for the representation of higher-dimensional functions, as they provide an efficient compression scheme. Sparse grids, as introduced by Zenger for the solution of partial differential equations in 1990 [1], have meanwhile been employed to a whole range of different applications from fields such as astrophysics, finance, molecular dynamics, or data mining [2, 3].

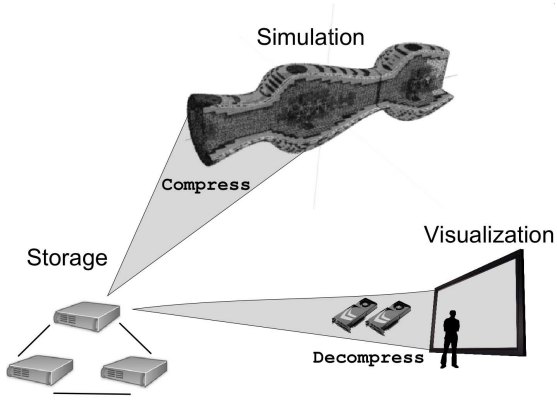


Figure 1: Interactive exploration of multi-dimensional data generated by a previous simulation run, and stored in compressed format.

Our application for sparse grids is the visual and interactive exploration of multi-dimensional data. The idea is that by browsing through the data, new insight into complex phenomena can be gained. However, the sheer size of the data generated by the multi-dimensional and multi-physics simulation under investigation does not only inhibit a smooth interaction with the visualization, but also poses data management challenges.

In order to take advantage of sparse grids in this scenario, two core algorithms need to be implemented efficiently: *hierarchization* and *evaluation*. These methods correspond to a compression step (pre-processing) and a decompression step (online) respectively (see Fig. 1). The more compact data format resulting from the compression step strongly enhances a fast and fluent visualization experience, as bandwidth requirements decrease considerably when forwarding subsets of the data to the visualization system. At the same time visualization algorithms rely heavily on the decompression scheme. The high resolution demands of a smoothly-running visual data exploration application make it a critical component of the whole system. Thus, an extremely efficient and most of all highly scalable implementation becomes crucial.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

A first study revealed that the original sparse grid algorithms for compression and decompression, using the usual data structures and parallel implementations for modern multicore CPUs, did not meet the requirements of our application regarding physical size and financial costs. Recent experiences [4] with the potential of GPGPU[1] (general-purpose computation on Graphics Processing Units) suggested that this may allow us to reach our goals by developing alternative data structures and porting the corresponding algorithms.

In this paper, we describe our implementation of compression and decompression for Nvidia GPUs. These are the crucial algorithms of the sparse grid technique for our application. In the later sections, we will focus on *hierarchization* as main part of compression (see Sec. 3), and use *evaluation* as more exact term for decompression. To our knowledge, this is the first space and time efficient implementation of the direct sparse grid technique on GPU architectures.

For such an implementation, the data structure used for storing the sparse grid is of high importance. Usually, trees or hash tables are employed for storing the sparse grid values attributed to grid points at specific coordinates. Typically, much space is needed both for the coordinates in direct or indirect form, and for internal management (e.g. pointers into parts of the data structure). In contrast to this, our approach uses a bijection (*gp2idx*) from the set of points in a sparse grid to a set of consecutive natural numbers (see Sec. 4). Consequently, we are able to store the sparse grid values in a contiguous 1d array without the need for any further strucural information as required in conventional tree- or hash-based approaches. This minimizes the space used, allowing for large sparse grids to be stored in the GPU device memory. But even more importantly, it improves data locality and thus further enhances the scalability of the parallelized implementations of compression and decompression on the GPU. To sum up, the main contributions of this paper are as follows:

- We propose a space efficient data structure for sparse grids (Sec. 4). We compare the memory requirements of our data structure with other data structures typically used for sparse grids. For a 10-dimensional sparse grid with approximately 127 million points, e.g., our data structure consumes up to 30 times less space.
- We present highly efficient and scalable implementations for the compression and decompression algorithms for sparse grids on Nvidia GPUs. The implementations use static workload distribution for parallelization. The compression is up to 17 times faster than the sequential version running on one Intel Nehalem core, the decompression is up to 70 times faster (Sec. 6).

In addition, we measured the performance gain we can get with our compact data structure on standard CPUs with OpenMP parallelization. On a 32-core Opteron machine, a comparison between 1 and 32 cores gives us speedups of 24 for compression and 31 for decompression.

2. The Sparse Grid Technique

Sparse grids help to overcome the curse of dimensionality by reducing the number of grid points from $\mathcal{O}(\tilde{N}^d)$ to $\mathcal{O}(\tilde{N}(\log \tilde{N})^{d-1})$ with only a slightly deteriorated accuracy if the underlying function f is sufficiently smooth. In this section, we briefly describe the technique of sparse grids and introduce the two main principles they are based on, a hierarchical representation of the one-dimensional basis and the extension to the d-dimensional setting via a tensor product approach.

[1] See `http://www.gpgpu.org`

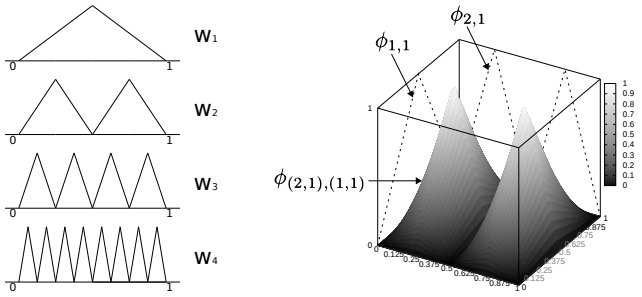


Figure 2: Left: 1d basis functions up to level 4. $V_4 = W_1 \bigoplus \cdots \bigoplus W_4$. Right: 2d basis functions, constructed from two 1d basis functions: $\phi_{(2,1),(1,1)}(x,y) = \phi_{2,1}(x) \cdot \phi_{1,1}(y)$.

2.1 Basis Functions

We consider the representation of a piecewise d-linear function $f_s : \Omega \to \mathbb{R}$ for a certain mesh-width $h_n := 2^{-n}$ with some discretization level n. For reasons of simplicity, we restrict ourselves to the domain $\Omega := [0,1]^d$. To obtain the approximation f_s of some function f, we discretize Ω and employ basis functions ϕ_i which are centered at the grid points stemming from the discretization. f_s is thus provided as a weighted sum of N basis functions, $f_s := \sum_{j=1}^{N} \alpha_j \phi_j$, with coefficients α_j.

We use the standard one-dimensional hat function, $\phi(x) = \max(1-|x|, 0)$, from which we derive one-dimensional hat basis functions by dilatation and translation,

$$\phi_{l,i}(x) := \phi(2^l x - i)\,,$$

which depend on a level l and an index i, $0 < i < 2^l$. The basis functions are centered at grid points $x_{l,i} = 2^{-l} i$ at which we interpolate f and have local support. Introducing the hierarchical index sets

$$I_l := \{i \in \mathbb{N} : 1 \le i \le 2^l - 1, i \text{ odd}\}$$

we obtain a set of hierarchical subspaces W_l spanned by the corresponding basis $\Phi_l := \{\phi_{l,i}(x), i \in I_l\}$. See Fig. 2 (left) for the basis functions up to level 4. Note that we restrict ourselves to functions that are zero on the boundary of Ω to keep the descriptions as simple as possible; adding the two basis function $\phi_{0,0}$ and $\phi_{0,1}$ on level 0 would allow to treat non-zero boundary values.

The hierarchical basis functions are then extended to d dimensions via a tensor product approach and are defined as

$$\phi_{\underline{l},\underline{i}} := \prod_{t=1}^{d} \phi_{l_t,i_t}(x_t),$$

where $\underline{l}$ and $\underline{i}$ are multi-indices, uniquely indicating level and index of the underlying one-dimensional hat functions for each dimension; see Fig. 2 (right). The basis

$$\Phi_{W_{\underline{l}}} := \left\{\phi_{\underline{l},\underline{i}}(\underline{x}) :\ i_j = 1, \ldots, 2^{l_j} - 1,\ i_j \text{ odd},\ j = 1, \ldots, d\right\}$$

span subspaces $W_{\underline{l}}$. As in the one-dimensional setting, all basis functions for a certain $\underline{l}$ belong to a regular grid, have pairwise disjoint, equally sized supports, and cover the whole domain.

2.2 Full and Sparse Grids

We can now formulate the space of piecewise linear functions V_n on a full grid with mesh-width h_n for a given level n as a direct sum of $W_{\underline{l}}$,

$$V_n = \bigoplus_{|\underline{l}|_\infty = n} W_{\underline{l}}, \qquad |\underline{l}|_\infty := \max_{1 \le t \le d} l_t\,.$$

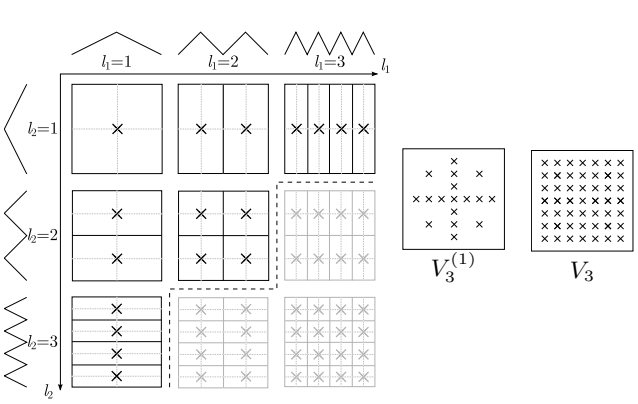


Figure 3: The two-dimensional subspaces $W_{\underline{l}}$ up to $l = 3$ ($h_3 = 1/8$) in each dimension. The optimal a priori selection of subspaces is shown in black (left), leading to a sparse grid of level $n = 3$ for the sparse grid space $V_3^{(1)}$ (middle). In comparison to the corresponding full grid V_3 (right) using the grey subspaces as well, the benefit of using sparse grids can be clearly seen.

The hierarchical subspace splitting allows to select those subspaces (or subgrids, respectively) that contribute most to the approximation. This can be done by an a priori *selection* [2], resulting in the sparse grid space $V_n^{(1)}$,

$$V_n^{(1)} = \bigoplus_{|\underline{l}|_1 \leq n+d-1} W_{\underline{l}}\,, \qquad |\underline{l}|_1 := \sum_{t=1}^{d} l_t\,.$$

Figure 3 shows the selection of subspaces and the resulting regular (i.e. non-adaptive) sparse grid for $n = 3$, i.e. the sparse grid space $V_3^{(1)}$, as well as the corresponding full grid for V_3.

2.3 Data Structures

Typical data structures for sparse grids are either hash-based or tree-based. The former ones map a grid point to an index which is then used to access a vector of coefficients, as this reduces the memory requirements and especially simplifies implementations considerably; for references, see the section on related work.

The latter ones aim to replicate the hierarchical structure of sparse grids, storing a pointer-based tree-like structure. If the full parent-child relationship is realized, this requires $\mathcal{O}(dN)$ storage, as each grid point has $2d$ child nodes in general. A more memory efficient data structure is considering each dimension d in a fixed order and storing binary trees of $d-1$-dimensional trees, which reduces the memory requirements. Figure 4 shows this concept for a regular sparse grid of level $n = 3$ in three dimensions, replacing the one-dimensional binary trees by arrays. Thus, the data structure is essentially a prefix tree or trie, storing the common prefix for multiple grid point coordinates only once. Of course, pointer-based data structures are not well-suited for GPUs.

3. The Sparse Grid Operations

To compress a general function represented on a full grid, we select only the function values at grid points also contained in a sparse grid. We can then express the problem of representing a function on a sparse grid as the problem of computing the hierarchical coefficients $\alpha_{\underline{l},\underline{i}}$, which is called hierarchization. Decompression (interpolation) refers to evaluating f_s anywhere inside the domain. In this section, we present algorithms for both hierarchization and interpolation as they are usually used. They reflect the recursive nature of the sparse grid's structure, clearly illustrating the difficulties of porting them to GPUs.

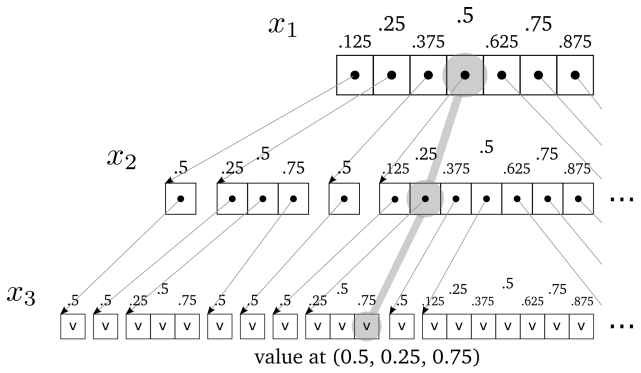


Figure 4: The prefix tree data structure for a regular sparse grid of level $n = 3$ representing grid points by coordinates. The arrays contain the one-dimensional substructures (essentially binary trees), each level corresponding to one dimension. The access to grid point with $\underline{l} = (1, 2, 2)$ and $\underline{i} = (1, 1, 3)$ or equivalent coordinates $(0.5, 0.25, 0.75)$ is indicated in gray.

3.1 *Hierarchization* – Computing Hierarchical Coefficients

Algorithm 1 shows the one-dimensional implementation of the hierarchization operation. This is the basic building block. To perform the full multi-dimensional hierarchization two levels of complexity are added. First, for a fixed dimension d the one-dimensional hierarchization is performed starting from all grid points with $l_d = 1$ and $i_d = 1$ in dimension d. Second, the previous procedure is applied unchanged to all the remaining dimensions, one after another, working on updated values from the previous steps.

Algorithm 1 1d recursive hierarchization

```
1: func hierarchize1d(gp, leftVal, rightVal, level) :
2: if level < maxLevel then
3:     hierarchize1d(gp.leftChild, leftVal, gp.value, level + 1)
4:     hierarchize1d(gp.rightChild, gp.value, rightVal, level + 1)
5: end if
6: gp.value ← gp.value − (leftVal + rightVal)/2
```

This procedure can be illustrated with the help of Fig. 5 (left): if the horizontal dimension (x_1) is first picked for hierarchization, *hierarchize1d* starts from all points on the main vertical sparse grid axis. From each of those, Alg. 1 is executed in the horizontal direction. Next, we hierarchize in the vertical dimension (x_2), starting from all grid points on the horizontal main axis. One important observation is the lack of locality of the *hierarchize1d* function (Fig. 5 (right)) with negative impact on cache efficiency.

3.2 *Evaluation* – Interpolating Between Grid Points

Interpolation directly applies to the evaluation of the sparse grid function. For visualization, we will have to be able to evaluate our sparse grid function at arbitrary locations in our domain: we have to perform d-linear interpolations.

As introduced in Sec. 2, each point of the sparse grid has a corresponding hierarchical coefficient with an associated multi-dimensional basis function. Evaluating the sparse grid function at an arbitrary point $x \in [0, 1]^d$ implies retrieving a subset of the grid's hierarchical coefficients, multiplying them with the corresponding basis function values and adding up the products. This is

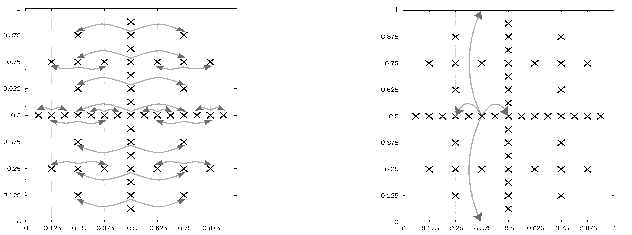


Figure 5: Left: Sparse grid traversal during hierarchization in the x_1-direction. Right: Shows the hierarchical parents (dependencies) for a grid point.

Algorithm 2 1d recursive evaluation

```
1: func evaluate1d(gp, x, level) :
2: res ← basis(gp, x) · gp.value
3: if level < maxLevel then
4:     if x left of gp then
5:         res ← res + evaluate1d(gp.leftChild, x, level + 1)
6:     else
7:         res ← res + evaluate1d(gp.rightChild, x, level + 1)
8:     end if
9: end if
10: return res
```

shown for the one-dimensional case in Alg. 2. After the contribution of the current point is calculated in line 2, the algorithm descends recursively and collects the contributions of all other points in the same dimension. In line 4 an optimization is made based on the fact that not all points contribute to the interpolation (too far away from the desired interpolation point). For a multi-dimensional representation the algorithm becomes more complicated due to the basis function evaluation. A recursion also in dimensions is necessary to first evaluate the multi-dimensional basis functions. Only afterwards, the resulting value can be multiplied with the hierarchical coefficient and summed up.

4. A Compact Data Structure

In this section we present a space efficient data structure for regular sparse grids. In contrast to the inherent hierarchical nature of sparse grids, our data structure is flat which makes it more suitable for iterative algorithms and thus the GPU architecture. Its key component is a bijection called *gp2idx* that perfectly maps level-index-vector pairs (i.e. $(\underline{l}, \underline{i})$) to consecutive integer indices. Using *gp2idx*, all hierarchical coefficients of a regular sparse grid can be efficiently stored and accessed in a 1d array. We explain how to modify the classic *hierarchization* and *evaluation* algorithms in order to fully benefit from our data structure, also in terms of parallel usage. We can overcome the simplifying assumptions we make in our data structure, mainly the representation of zero boundary multi-dimensional functions, by extending our approach in a very natural way. This aspect is discussed at the end of the section.

Note that for the remainder of this work we are not following the common notation and start counting levels and level vector components from 0.

4.1 Storage Scheme

When interpolating functions on a regular grid it is obvious not to store the grid points' coordinates explicitly but only the function values and mesh widths. Retrieving data from a multi-dimensional array as well as recalculating coordinates via a multi-index are simple and cheap tasks. Due to their structure, this is not as straightforward for sparse grids. Many applications thus need to store those multi-indices together with the data, the hierarchical coefficients, as key-value pairs (e.g. in a hash map). With our composite map *gp2idx* that bijectively maps a sparse grid's points to consecutive integer indices we are able to eliminate the need for storing any keys.

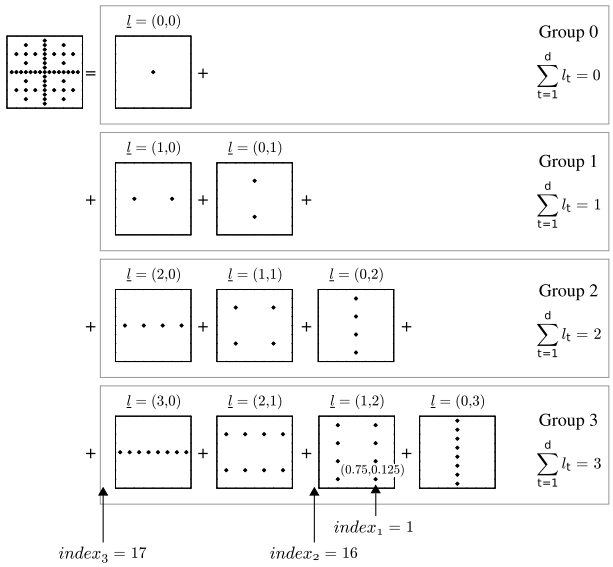


Figure 6: All subspaces of a 2d sparse grid are stored consecutively in memory from top to bottom, left to right. The value at grid point $\underline{l} = (1, 2)$, $\underline{i} = (3, 1)$ (coordinates $(0.75, 0.125)$) can be found at position 34 ($= index_1 + index_2 + index_3$) in a plain 1d array.

Figure 6 illustrates how we can decompose a sparse grid into a set of regular grids, the so-called subspaces, and group them with respect to their levels. The reason behind is that all subspaces $\underline{l}$ on level $n = |\underline{l}|_1$ contain the same number of hierarchical coefficients. With our modified level vector notation this number can directly be computed as $2^{|\underline{l}|_1}$.

The image also demonstrates that finding the hierarchical coefficient associated with level-index-vector pair $(\underline{l}, \underline{i})$ can be divided into computing three separate indices:

- $index_3$: number of coefficients associated with levels $l' < |\underline{l}|_1$
- $index_2$: number of coefficients in the subspaces that precede subspace $\underline{l}$ in its corresponding level group
- $index_1$: index of the coefficient in subspace $\underline{l}$ identified by $\underline{i}$.

Computing $index_1$ is simple as it comes down to identifying the position of one element in a regular grid. But before we can determine the offsets $index_2$ and $index_3$ we need to discuss the number of subspaces on level n. This is done in the next subsection.

4.2 An Optimal Index Map, *gp2idx*

The challenging part about defining a map from level-index-vector pairs to integer indices without gaps is counting the subspaces of a level. Consider the set

$$\mathcal{L}_n^d = \{\underline{l} \in \mathbb{N}_0^d \,\Big|\, |\underline{l}|_1 = n\} \tag{1}$$

of level vectors of level n in d dimensions. Determining its cardinality $\mathcal{S}_n^d$ is known as the problem of integer partitioning, where a positive integer is divided into d positive integer summands. Using

Algorithm 3 Recursive level vector enumeration, *enumerate*$(d,\ n)$

```
1: if d = 1 then
2:     output n
3: else
4:     for k = 0 to n do
5:         output concatenate(enumerate(d − 1, n − k), k)
6:     end for
7: end if
```

combinatorial mathematics the number can be written as

$$\mathcal{S}_n^d = C_{d-1+n}^{d-1} = \binom{d-1+n}{d-1} = \binom{d-1+n}{n}. \quad (2)$$

Concerning $index_3$, we can now compute it via the formula

$$index_3 = \sum_{j=0}^{n-1} \mathcal{S}_j^d \cdot 2^j$$

which counts the total number of hierarchical coefficients stored for all levels $l' < n$.

To be able to determine $index_2$ we introduce a recursive enumeration scheme (see Alg. 3) that induces an order on the elements of $\mathcal{L}_n^d$. For simplicity we now generally assume $d \geq 2$. Our algorithm starts with the last component l_d of a level vector $\underline{l}$ and sets it one after another to values $k \in \{0, \ldots, n\}$. For each of these values it recursively descends into *enumerate*$(d-1, n-k)$ in order to enumerate the elements of $\mathcal{L}_{n-k}^{d-1}$ in the first $d-1$ components of $\underline{l}$. The recursion stops once only the first component is left. It is clear that *enumerate* only returns valid level vectors and that no combination is omitted. Next, we observe that the enumeration starts with *first*(d,n) and ends with *last*(d,n) defined as

$$\textit{first}(d,n) := (n, 0, \ldots, 0)^T, \quad \textit{last}(d,n) := (0, \ldots, 0, n)^T. \quad (3)$$

Recursion is not supported on the GPU so we change the enumeration to an iterator scheme, in which we compute a unique successor *next*$(\underline{l}) = \underline{r}$ for each $\underline{l} \in \mathcal{L}_n^d \setminus \{\textit{last}(d,n)\}$. Take the smallest index t with component $m = l_t \neq 0$. It holds that we currently see *last*$(t+1, m)$ in the first $t+1$ components of $\underline{l}$. From the recursive definition it is then clear that the next change must happen on the recursion level above. Here we increase the loop variable k in line 4 of Alg. 3 by one and return the value *concatenate*(*enumerate*$(t+1,\ m-1),\ k)$ in line 5. This has two effects:

1. By increasing k we set $r_{t+1} := l_{t+1} + 1$.
2. We recursively initialize the first $t+1$ components of $\underline{r}$ with *first*$(t+1, m-1)$. Compared to $\underline{l}$ this only requires two changes in $\underline{r}$: Setting $r_t := 0$ and $r_0 := m-1$ as for all components $r_j, 0 < j < t$ we have $r_t = l_t = 0$.

Algorithm 4 implements these steps in an iterative function *next*. Note finally that all trailing components $r_j, j > t+1$ remain unchanged, and that the special case of $t = 0$ is treated automatically if lines 6 and 7 of the algorithm are executed in this order.

Based on this order induced on the level vectors we now define the function *subspaceidx* as follows

$$subspaceidx(\underline{l}) = \sum_{t=1}^{d-1} \binom{t + \sum_{j=0}^{t} l_j}{t} - \binom{t + \sum_{j=0}^{t-1} l_j}{t}. \quad (4)$$

It is constructed such that it maps all elements $\underline{l} \in \mathcal{L}_n^d$ to a consecutive integer index starting from 0. We prove this by showing that $b - a = 1$ holds for images $a =$ *subspaceidx*$(\underline{l})$ and $b =$ *subspaceidx*(*next*$(\underline{l})$) with $\underline{l} \neq \textit{last}(d,n)$. Still we assume $d \geq 2$.

Algorithm 4 Iterator increment function, $next(\underline{l})$

```
1: r ← l
2: t ← 0
3: while l[t] = 0 do
4:     t ← t + 1
5: end while
6: r[t] ← 0
7: r[0] ← l[t] − 1
8: r[t + 1] ← l[t + 1] + 1
9: return r
```

Proof. In order to understand the effect of *next* on the index a we focus on the differing components of $\underline{l}$ and $\underline{r} = \textit{next}(\underline{l})$. As we can see clearly in lines 6–8 of Alg. 4 these are at indices $\{0, t, t+1\}$ with $t = min_{j \in \{0,\ldots,d-1\}}\{l_j \neq 0\}$.

We note that the index of the sum in *subspaceidx* starts at 1. Therefore only the changes in the vector components t and $t+1$ are important for *subspaceidx*. About these we know $r_t = 0$ and $r_{t+1} = l_{t+1} + 1$ while their contributions to $b =$ *subspaceidx*$(\underline{r})$ have the form

$$b_{\underline{r},\{t,t+1\}} = \underbrace{\left[\binom{(t+1) + (l_t + l_{t+1})}{t+1} - \binom{(t+1) + (l_t - 1)}{(t+1)}\right]}_{\text{summand for } t+1}.$$

For a the corresponding summands look like this

$$a_{\underline{l},\{t,t+1\}} = \underbrace{\left[\binom{(t+1) + (l_t + l_{t+1})}{(t+1)} - \binom{(t+1) + l_t}{(t+1)}\right]}_{\text{summand for } t+1} + \underbrace{\left[\binom{t + l_t}{t} - \binom{t+0}{t}\right]}_{\text{summand for } t}.$$

Computing the difference and using $\binom{t+l_t}{t} + \binom{t+l_t}{(t+1)} = \binom{(t+1)+l_t}{(t+1)}$ gives us

$$b - a = b_{\underline{r},\{t,t+1\}} - a_{\underline{l},\{t,t+1\}} = \binom{t}{t} = 1.$$

In the special case $t = 0$ the "summand for t" does not exist. $b - a$ then degenerates to $\binom{l_t}{1} - \binom{l_t - 1}{1} = l_t - (l_t - 1) = 1$.

With the fact that *subspaceidx*(*first*$(d,n)) = 0$ and the knowledge that the *next* function returns a unique successor for any $\underline{l} \in \mathcal{L}_n^d \setminus \{\textit{last}(d,n)\}$ the proof is complete. □

Now we have the means to also compute $index_2$ and integrate this part into the composite index map *gp2idx* that maps each grid point to an integer index. The final algorithm for *gp2idx* is shown in Alg. 5. It runs in $\mathcal{O}(d)$ time if we apply two optimizations. First, the repeated expensive calculation of the binomial coefficient via *binomial* can be avoided. We use a small $n \times d$ lookup matrix called *binmat* to do so, n being the maximum level in the grid. *binmat* can be initialized in $\mathcal{O}(n \cdot d)$ time and because of the binomial coefficient's symmetry property we can even reduce it to half its size. Second, the computation of $index_3$ in lines 13 to 16 of the algorithm would also rather be implemented as an $\mathcal{O}(1)$ lookup operation than as an $\mathcal{O}(|\underline{l}|_1)$ loop.

Using basic rules of combinatorial mathematics we can furthermore reduce the number of lookups in *binmat* in lines 8–10 to one per iteration.

4.3 Impact on Initial Algorithms

The *gp2idx* bijection eliminates the need to store coordinates, or equivalent $(\underline{l}, \underline{i})$, for matching hierarchical coefficients to their

Algorithm 5 The optimal index map, *gp2idx*($\underline{l}, \underline{i}$)

```
1: index_1 ← 0
2: for t = 0 to d − 1 do
3:     index_1 ← index_1 · 2^{l[t]} + (i[t] − 1)/2
4: end for
5: sum ← l[0]
6: index_2 ← 0
7: for t = 1 to d − 1 do
8:     index_2 ← index_2 − binomial(t + sum, t)
9:     sum ← sum + l[t]
10:    index_2 ← index_2 + binomial(t + sum, t)
11: end for
12: index_2 ← index_2 · 2^{sum}
13: index_3 ← 0
14: for s = 0 to sum − 1 do
15:    index_3 ← index_3 + binomial(d − 1 + s, d − 1) · 2^s
16: end for
17: return index_1 + index_2 + index_3
```

corresponding basis functions. Besides the minimal memory consumption, there are also other benefits from using the bijective function *gp2idx*, its inverse *idx2gp* and the *next* iterator. We modified the algorithms for hierarchization and evaluation presented in Sec. 3 in order to fully benefit from our data structure.

In Alg. 6 the hierarchical coefficients are stored in a 1d array, *rawStorage*. The coefficients in this array can be traversed in the order of their dependencies. The groups of subspaces or regular grids are updated in descending order of the refinement level, starting with the largest $|\underline{l}|_1$ and finishing with the smallest. This ensures that updating a coefficient does not break the dependencies of other coefficients. It is worth mentioning that, compared to Alg. 1, the new hierarchization algorithm is no longer recursive, thus becoming better suited for GPUs.

Algorithm 6 Multi-dimensional hierarchization based on *gp2idx*

```
1: Initialize rawStorage with corresponding values from the full grid
2: for t = 0 to d − 1 do
3:     for j = numOfGridPoints − 1 downto 0 do
4:         (l, i) ← idx2gp(j)
5:         val_1 ← rawStorage[gp2idx(leftParent(l, i, t))]
6:         val_2 ← rawStorage[gp2idx(rightParent(l, i, t))]
7:         rawStorage[j] ← rawStorage[j] − (val_1 + val_2)/2
8:     end for
9: end for
```

Another advantage lies in a reduced number of cache misses caused when accessing hierarchical coefficients. Since *gp2idx* and *idx2gp* operate on a small matrix of size $d \cdot n$, *binmat*, the number of cache misses triggered by their execution can be considered 0. Accordingly, cache misses only occur when referencing coefficients in the *rawStorage* array, and we therefore expect to have at most one miss per coefficient access. This even applies for the worst case scenario of random access.

The j loop can be parallelized using static decomposition which represents another positive aspect for GPUs. However, minor modifications must be applied to Alg. 6 so that the groups of subspaces are updated correctly, without destroying the semantics of sequential execution. Specifically, a global barrier must be executed after each group of subspaces is updated in descending order, from the largest $|\underline{l}|_1$ to the smallest.

In its new form, *evaluation* (Alg. 7) is also iterative, thus GPU compatible. We can see that neither *gp2idx* nor *idx2gp* are used. Traversing all valid combinations for $\underline{l}$ is done using the *next* function and the definitions for the *first* and *last* subspaces in a group. The number of cache misses is minimal, generated by line 15. Cache exploitation can be improved by observing that it is advantageous to execute the loops from lines 3 and 5 on multiple evaluation points, i.e. blocking is performed on the set of evaluation points and each block is processed after the j and $\underline{l}$ loops. The optimization is based on the fact that a subspace containing the coefficients with the same $\underline{l}$ is needed by all the evaluations and is already present in cache.

Algorithm 7 Multi-dimensional evaluation based on *next* iterator

```
1: res ← 0
2: index_2 ← 0
3: for j = 0 to n − 1 do
4:     l ← first(d, j)
5:     for k = 1 to C^j_{d−1+j} do
6:         prod ← 1
7:         index_1 ← 0
8:         for t = 0 to d − 1 do
9:             div ← 2^{−l[t]}
10:            index_1 ← index_1 · 2^{l[t]} + ⌊coords[t]/div⌋
11:            left ← ⌊coords[t]/div⌋ · div
12:            right ← left + div
13:            prod ← prod · basis(left, right, coords[t])
14:        end for
15:        prod ← prod · rawStorage[index_1 + index_2]
16:        res ← res + prod
17:        l ← next(l)
18:        index_2 ← index_2 + 2^j
19:    end for
20: end for
21: return res
```

The evaluation operation is embarrassingly parallel provided that each thread performs evaluation for its private set of multi-dimensional points in the domain.

4.4 An Extendable Context

One of our assumptions was that the functions to be represented using the sparse grid technique are zero-boundary functions. The application can be easily modified in order to cope with non-zero boundaries whereas our data structure requires a more complex extension. This extension is based on the observation that the boundary of a d-dimensional sparse grid is composed of lower-dimensional, zero-boundary sparse grids for which we already have an efficient data structure.

Let C_n^k be the number of k-combinations from a set of n elements. The number of $d-j$-dimensional sparse grids in the boundary is $2^j \cdot C_d^{d-j}$. One can verify this formula by simply counting the projections of a sparse grid in lower-dimensional planes as depicted in Fig. 7. Storing the boundary as a 1d contiguous array implies determining for any point on the boundary the first index of its corresponding sparse grid in the 1d array. This can be achieved by grouping the sparse grids according to their dimensionality. The number of sparse grids in the group corresponding to dimensionality j is $2^{d-j} \cdot C_d^j$. An ordering function has to be defined in order to find the correct sparse grid within a group. Next, *gp2idx* can be used.

5. Sparse Grids on GPUs

In this section we describe the architecture of Nvidia GPUs and the CUDA programming model. We continue with our implementation of sparse grid compression and decompression on GPUs. Since it has a minimal memory footprint, our data structure is a good fit for GPUs which have a relatively small amount of RAM. Moreover, our implementation is optimized for the best exploitation of computational resources on GPUs.

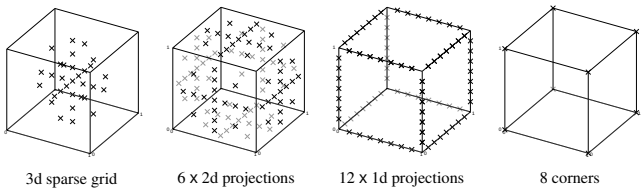


Figure 7: 3d sparse grid with non-zero boundary. The boundary of a 3d sparse grid is composed of lower-dimensional sparse grids.

5.1 GPU Architecture

In contrast to CPUs which use the die space for complex control logic and large caches, GPUs devote a higher percentage of transistors to floating point units. GPUs provide massive parallelism and deliver better performance than CPUs especially for applications with regular access patterns, e.g. dense matrix operations [5].

In the following we focus on the C1060 model of Nvidia Tesla [6]. This is a high-end GPU which contains 30 8-way SIMD units called by Nvidia Streaming Multiprocessors (SM). In Nvidia terminology, each way is called a Scalar Processor (SP). C1060 supports up to 1024 thread contexts in hardware per SM [7]. Each thread is generally executed on one of the 8 ways of the SIMD unit. One of the main characteristics of Nvidia GPUs is multithreading which offers the possibility to hide the latency originating from various instructions, especially the latency caused by loads from and stores to RAM, by executing a large number of threads (up to 30720 on a Tesla) concurrently with a low cost per context switch.

Nvidia GPUs are SIMD based architectures. Each SM control unit creates, manages and executes synchronously the threads in groups of 32 called warps. Every instruction is broadcasted synchronously to all the active threads in a warp. Branching may cause threads in the same warp to follow different execution paths. This type of behavior of the threads inside a warp is called diverge. It has the potential to severely reduce the performance of a GPU application, up to a factor of 32.

GPUs have their own dedicated RAM, global memory, which is in the order of several Gigabytes, e.g. C1060 has 4 GB of DDR3 memory. For performance, GPUs are also equipped with multiple fast memories on the chip: constant cache, texture cache and shared memory [8]. The properties of these fast memories vary in terms of latency, bandwidth and usage. The constant cache is a read-only cache and, according to [9], its level-1 has the lowest latency among the memories on the GPU. The texture cache is a read-only memory used for optimizing the bandwidth rather than latency, i.e. its latency is comparable with the one of the global memory. Finally, the shared memory is a low latency, read-write memory, is private per SM and controlled explicitly by the programmer.

5.2 GPU Programming

The Compute Unified Device Architecture (CUDA) is one of the available frameworks for programming Nvidia GPUs. From a programming point of view, a CUDA application has a CPU and a GPU part. The main responsibilities of the CPU part are: allocating memory on the GPU, transferring data to and from the GPU over PCI Express and launching the GPU program called kernel. A kernel cannot contain recursive functions. Each instance of a kernel in execution is a thread. Besides being packed in warps, the threads are also grouped in blocks of threads. This grouping is important as only the threads inside a block can synchronize via barriers (*__syncthreads*) and share data from the shared memory. Each thread has a 2d block identifier (*blockIdx*) and a 3d thread identifier (*threadIdx*). Combining the block and the thread identifiers offers the means to uniquely identify a thread on the GPU and to assign its part from the workload to be computed on the GPU.

A first optimization for GPU programs consists of the proper exploitation of multithreading. This is equivalent to maximizing the number of active threads and can be achieved by reducing the register file and shared memory consumption per thread. Second, efficient use of the memory hierarchy can provide a substantial speedup to GPU applications. This optimization includes at least: enabling coalesced accesses to global memory in order to reduce the number of memory transactions, using the fast memories on-chip and reducing the number of bank conflicts caused by accessing the shared memory. Third, divergent branches can serialize the execution of the threads composing a warp. To improve the performance, the number of divergent branches needs to be minimized. Note that these are the optimizations which proved to be relevant to our application and they represent only a subset of the optimizations applicable to CUDA applications. For a detailed list of optimizations we refer the reader to [7].

5.3 GPU Implementation of Sparse Grid Operations

The implementation of the hierarchization algorithm (Alg. 6) has minimal memory consumption. Moreover, the parallelization is based on statically decomposing the set of grid points for which the hierarchical coefficients are computed. Both minimal memory consumption and static workload distribution are factors that enable an efficient implementation of hierarchization on Nvidia GPUs. The decomposition is done such that each thread block is responsible for updating one subspace or regular grid of coefficients. In order to avoid breaking dependencies between coefficients, the subspace groups depicted in Fig. 6 are updated in descending order of $|\underline{l}|_1$. In other words, the update of group j must finish before group $j-1$ can be updated. The necessary barriers are enforced by the CPU program by launching the hierarchization kernel multiple times, each time for updating another group of subspaces. With respect to efficient use of fast memory, $\underline{l}$ and $\underline{i}$ are placed in shared memory. Accessing global memory for the subspaces to be updated are optimized for coalescing. However, accesses to dependencies or parents as shown in Fig. 5 (right) cannot be packed, thus representing the main source of uncoalesced accesses and branch divergence.

Finding the best implementation of hierarchization on GPU implies determining the fastest way to compute the bijection *gp2idx* which in turn relies heavily on the *binmat* matrix. Since *binmat* is a read-only matrix containing the binomial coefficients, it is compatible with all the fast memories available on the GPU. However, the texture cache is not a true candidate as our goal is to obtain low latency access to *binmat* rather than high bandwidth. In this context, three options were considered: computing the binomial coefficients on the fly in $\mathcal{O}(n)$, accessing *binmat* from shared memory and placing *binmat* in constant cache. Experiments revealed that computing on the fly makes hierarchization approximately 4 times slower than when shared memory or constant cache are used. If *binmat* is stored in constant cache, hierarchization is slightly faster than the version based on placing *binmat* in shared memory, confirming the GPU memory benchmarking results presented in [9].

For the evaluation algorithm (Alg. 7), the parallelization scheme relies on statically decomposing the set of multi-dimensional points for which function values are to be computed. This approach is embarrassingly parallel and makes sense since the number of interpolation points is typically around 10^5 or more, thus fully utilizing the GPU computational resources. More precisely, each thread performs interpolation for one multi-dimensional point in the domain. Shared memory is used for storing $\underline{l}$ and *coords*. Furthermore, copying *coords* from global memory to shared memory satisfies the coalescing requirements. Note also that divergent branching inside a warp is minimized in this approach.

Data Structure	Time	Non-seq. Refs.
Standard STL map	$\mathcal{O}(d \cdot log(N))$	$\mathcal{O}(log(N))$
Enhanced STL map	$\mathcal{O}(d + log(N))$	$\mathcal{O}(log(N))$
Enhanced STL hash_table	$\mathcal{O}(d)$	$\mathcal{O}(1)$
Prefix tree	$\mathcal{O}(d)$	$\mathcal{O}(d)$
Our data structure	$\mathcal{O}(d)$	$\mathcal{O}(1)$

Table 1: Time complexities and number of non-sequential memory references for accessing a value from a grid point. N is the total number of sparse grid points.

All the arrays from hierarchization and evaluation are private to each thread, have length d and are stored in shared memory. Consequently, the pressure on shared memory is linearly dependent on d and can decrease the number of active threads, i.e. it reduces the benefits of multithreading. In order to decrease the shared memory consumption per thread, we set $\underline{l}$ as an array shared between all threads inside the same thread block. Only the master thread (thread 0) from the thread block modifies $\underline{l}$, and all the threads in the block read $\underline{l}$. Although this adds synchronization via the *__syncthreads* device function, the improvements over the versions without block shared $\underline{l}$ cannot be ignored, i.e. this results in 1.62 times faster hierarchization and 1.59 times faster evaluation.

6. Results

For all evaluations in this section, we use sparse grids with refinement level 11.

6.1 Comparison of Data Structures

First, we compare the memory consumption of various data structures for sparse grids. Table 1 shows the structures taken into consideration and their access properties. The number for non-sequential references gives a hint regarding locality of accesses, i.e. the number of cache misses to be expected on standard CPUs.

The first three data structures use the C++ Standard Template Library (STL) [10]. "Standard STL map" consumes space for keys linearly to the number of dimensions. "Enhanced STL map" and "enhanced STL hash table" use our *gp2idx* function. Memory consumption is reduced by storing the result of *gp2idx* as key, i.e. it is constant with regard to dimensionality. The "prefix tree" data structure is the classical tree-based data structure using pointers, as described in Sec. 2.3. As it can be seen in Fig. 8, our data structure uses the smallest amount of main memory. Moreover, "prefix tree"

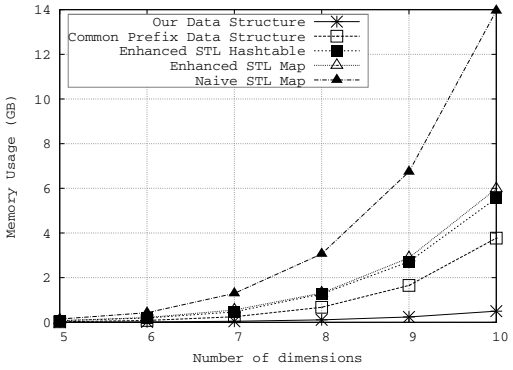
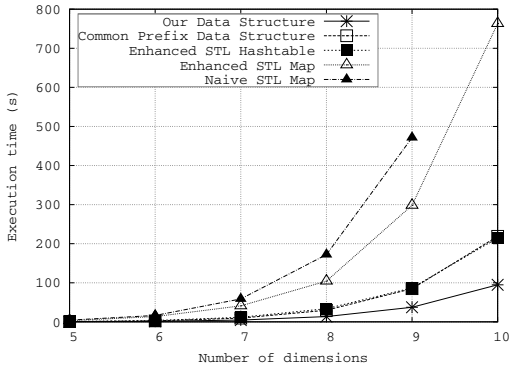
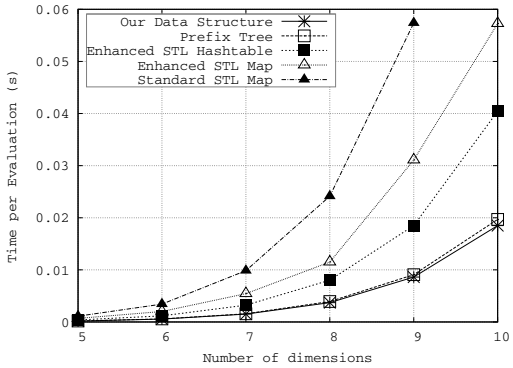


Figure 8: Memory consumption of a sparse grid.



(a) Runtime for sequential hierarchization.



(b) Runtime for sequential evaluation.

Figure 9: Sequential runtimes on i7-920.

follows, showing that its compression scheme is superior to using *gp2idx* in conjunction with an STL hash table or map.

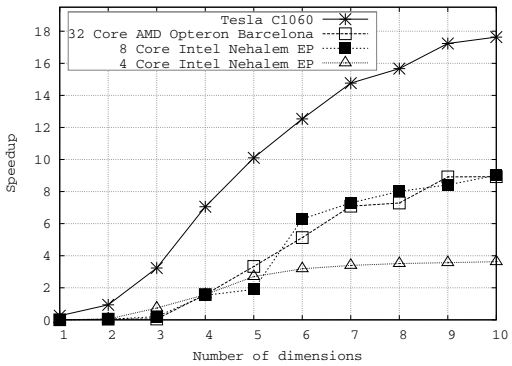
6.2 Performance Measurements

As experimental setup, the following hardware was used:

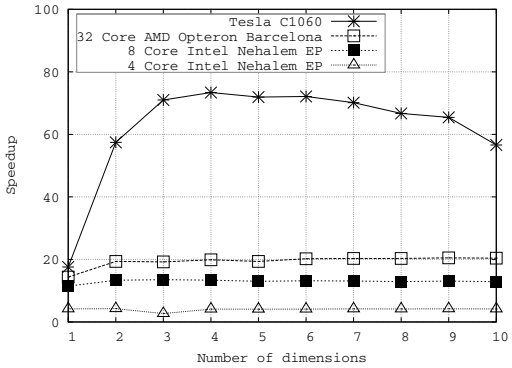
- a 4-core, single-socket Nehalem i7-920 with 24 GB of DDR3 1066 MHz memory,
- a Tesla C1060 with 4 GB of DDR3 800 MHz 512 bit memory,
- a 32-core, 8-socket AMD Opteron 8356 machine with 256 GB of DDR2 667 MHz main memory,
- an 8-core, dual-socket Nehalem E5540 supporting 16 simultaneous threads with 24 GB of DDR3 1066 MHz memory.

There are three implementations of our application that we considered in our tests: the sequential C++ version which provides the base for speedup numbers, the CUDA version implemented using the CUDA SDK [11], and a version for x86 multicore systems based on OpenMP 3.0. The CPU versions are optimized with respect to cache and SSE. The tasking concept was applied for parallelizing the recursive algorithms for both hierarchization and evaluation shown in Sec. 3. The number of points in the sparse grids used in our tests was in the range of $[2047, 127574017]$, corresponding to level 11 sparse grids with dimensionalities between 1 and 10.

First, we did runtime measurements with sequential C++ versions of the sparse grid operations using the different data structures on the i7-920 system. Our data structure gives the best execution times for both sequential hierarchization and evaluation as depicted in Fig. 9a and Fig. 9b. The prefix tree has similar execution time for hierarchization as the enhanced STL hash table. This is

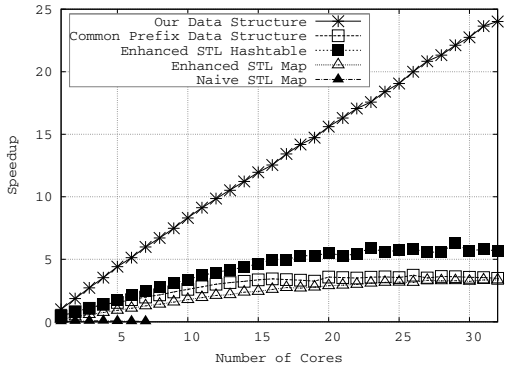


(a) Hierarchization on GPU and multicore CPU.

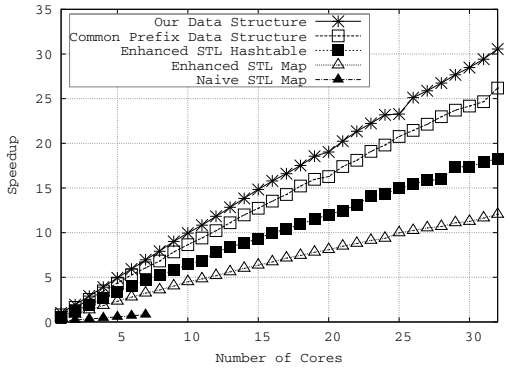


(b) Evaluation on GPU and multicore CPU.

Figure 10: Performance on GPU and CPU.



(a) Hierarchization scalability.



(b) Evaluation scalability.

Figure 11: Scalability on CPU.

due to good cache locality for accessing the children of grid points. For evaluation, its performance is very close to the performance obtained with our data structure. This also happens because the cache is exploited properly. Once a hierarchical coefficient is found, the next needed one – corresponding to a higher refinement level – resides in the same coefficient array at the bottom of the structure. Hence, the next hierarchical coefficient is already loaded in cache.

With regard to the sequential runtimes on the i7-920 system, we provide speedup numbers of implementations using our data structure in Fig. 10a and Fig. 10b. Most important for our application are the Tesla C1060 results corresponding to the GPU implementations. We see that these versions of the sparse grid operations are superior to the ones on x86 multicore architectures. The execution time of hierarchization is reduced by approximately a factor of 2 compared to the fastest multicore architectures considered in our tests whereas evaluation is roughly 3 times faster than the best performance on multicore CPUs. Evaluating a function represented as a sparse grid at 10^5 points leads to an efficient usage of the parallel resources as all the multicore architectures are able to offer constant speedup independent of the number of dimensions. The speedup on the GPU is expected to decrease when the number of dimensions is greater than 10. This is due to the increasing pressure on the shared memory inside an streaming multiprocessor. More exactly, each GPU thread requires shared memory space that depends linearly on the number of dimensions. If the number of dimensions grows over a certain value, this limits the number of threads that can be executed concurrently on one streaming processor, thus reducing any potential for latency hiding.

Finally, we measured the scalability achievable with our data structure on the Opteron system. Regarding parallel hierarchization, the tree and hash table data structures saturate the connection to main memory, thus limiting the scalability as shown in Fig. 11a when the number of processors is greater than 15. Another cause for the reduced scalability for these data structures can be linked to the use of tasks necessary for the dynamic decomposition of the workload. Evaluation is not memory bound and this can be seen in Fig. 11b. The memory connection is not saturated and does not block the scalability. The prefix tree provides the best speedup from all the tree and hash table data structures. This happens because of better cache locality when accessing the hierarchical coefficients required by the evaluation operation.

7. Related Work

The power of sparse grids has already been exploited for a whole range of different types of problems, see [2, 3]. Depending on the requirements of the respective application, different types of data structures have been considered. Typically, hash-based realizations of the data structure are employed (see [12, 13] for discussions). They provide a reasonable trade-of by reducing the memory requirements in contrast to pointer-based approaches. Yet, they keep the access structures as flexible as possible and suitable for adaptive refinement. To meet harsh requirements such as minimal storage space, flexibility can be traded for efficiency. Note that data structures for sparse grids are ongoing research [14, 15].

Parallelizations of sparse grid methods have typically been achieved by employing the so-called combination technique [16], which obtains an approximation of the sparse grid solution by a superposition of partial solutions obtained on several smaller, but anisotropic full grids. Obtaining the independent partial solutions can be parallelized trivially. Furthermore, due to the regular struc-

ture of the full partial grids, each partial solution can be vectorized in a straightforward manner [17]. The benefit of employing vectorization on GPUs is evident [18, 19]. However, grid points and corresponding function values have to be replicated across multiple full grids. Thus, higher memory requirements have to be met.

Our approach requires minimal storage at the cost of computing a cache-efficient mapping from a grid point to the position of its corresponding coefficient or function value in a linear ordering of the sparse grid points. To the best of our knowledge, this is the first time that a direct sparse grid implementation on a GPU has been achieved.

In general, there are multiple data structures, hash tables and trees, developed to cope with the problem of storing and retrieving efficiently multi-dimensional data [20]. Such data structures can also be used for storing the points required by the sparse grid technique. However, they rely on storing information about multi-dimensional coordinates. Therefore, our performance expectations for these data structures are similar to the results provided in Sec. 6. In our approach based on *gp2idx*, we do not store any coordinates. Nevertheless, we emphasize that our solution does not address the sparsity problem in all types of grids. In contrast to the work mentioned above and to the best of our knowledge, there is no other data structure in the context of the sparse grid technique with smaller memory footprint than our data structure.

An approach for enabling complex data structures on GPUs is presented in [21]. Although the proposed solution was published before the release of CUDA, some of its concepts are still valid. For instance, the authors describe the procedure for handling iterators in parallel on GPUs. Moreover, copying a pointer based data structure between different memory spaces is another issue addressed by the authors. The paper shows that it is feasible to operate with complex data structures on GPUs but the inefficiencies related to memory consumption and access time are still present.

8. Conclusion

In this paper we focus on identifying problems and offering solutions for successfully porting the sparse grid technique to Nvidia GPUs. Our motivation is a computational steering application in which compressing and decompressing multi-dimensional data plays a crucial role. In their original form, sparse grid compression and decompression are based on recursion and complex data structures like trees and hash tables. As GPUs do not support recursion and do not favor trees or hash tables, we propose a data structure that eliminates these limitations as the main optimization.

Our data structure is based on a bijection that maps a multi-dimensional sparse grid to a 1d continuous array. Therefore, it minimizes memory consumption. Furthermore, it enables us to replace the recursive compression and decompression operations by iterative algorithms, fully compatible with GPUs. We obtain impressive speedup numbers on the GPU. Consequently, our sparse grid implementation is both space and time efficient.

As a next step, we plan to tune our application for Nvidia GPUs based on the Fermi architecture [22]. We expect that the two-level cache, 64 KB level-1 per SM and 768 KB shared level-2 could be beneficial for both sparse grid operations. Another point of interest for us will be the integration of our data structure into other applications that rely on the sparse grid technique for representing higher-dimensional functions.

Acknowledgement

This publication is based on work supported by Award No. UK-C0020, made by King Abdullah University of Science and Technology (KAUST).

References

[1] C. Zenger. Sparse grids. In Wolfgang Hackbusch, editor, *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 241–251. Vieweg, 1991.

[2] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13(-1):147–269, 2004.

[3] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Dissertation, Institut für Informatik, Technische Universität München, February 2010.

[4] H. Hacker, C. Trinitis, J. Weidendorfer, and M. Brehm. Considering GPGPU for HPC Centers: Is it Worth the Effort? In *Proceedings of "Facing the Multicore-Challenge 2010"*, volume 6310 of *LNCS*. Springer, 2010.

[5] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC*, page 31, 2008.

[6] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[7] NVIDIA. *CUDA Programming Guide 2.2.1*, 2009.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPOPP*, pages 73–82, 2008.

[9] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246, 2010.

[10] Hewlett-Packard Company. *Standard Template Library Programmer's Guide*, 1994.

[11] NVIDIA. CUDA SDK. `http://developer.nvidia.com/object/cuda_2_2_downloads.html`.

[12] M. May and T. Schiekofer. An Abstract Data Type for Parallel Simulations based on Sparse Grids. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Proceedings of the Third European PVM Conference*, volume 1156 of *Lecture Notes in Computer Science*, pages 59–67. Springer Verlag, 1997.

[13] M. Griebel. Adaptive sparse grid multilevel methods for elliptic PDEs based on finite differences. *Computing*, 61(2):151–179, 1998.

[14] C. Feuersänger. Dünngitterverfahren für hochdimensionale elliptische partielle Differentialgleichungen. Diplomarbeit, Institut für Numerische Simulation, Universität Bonn, 2005.

[15] S. Dirnstorfer. Adaptive numerische Quadratur höherer Ordnung auf dünnen Gittern. Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2000.

[16] M. Griebel. The Combination Technique for the Sparse Grid Solution of PDE's on Multiprocessor Machines. *Parallel Processing Letters*, 2:61–70, 1992.

[17] M. Griebel. A parallelizable and vectorizable multi-level algorithm on sparse grids. In *Parallel algorithms for partial differential equations*, Notes Numer. Fluid Mech. 31, pages 94–100. Vieweg, Wiesbaden, 1991.

[18] C. Teitzel, M. Hopf, and T. Ertl. Scientific visualization on sparse grids. Technical report, Universitt Erlangen-Nrnberg, Lehrstuhl fr Graphische Datenverarbeitung (IMMD IX), 2000.

[19] A. Gaikwad and I. M. Toke. GPU based sparse grid technique for solving multidimensional options pricing PDEs. In *WHPCF '09: Proceedings of the 2nd Workshop on High Performance Computational Finance*, pages 1–9, New York, NY, USA, 2009. ACM.

[20] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[21] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[22] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. White paper, 2009.