Binary Mesh Partitioning for Cache-Efficient Visualization

Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin

Abstract—One important bottleneck when visualizing large data sets is the data transfer between processor and memory. Cache-aware (CA) and cache-oblivious (CO) algorithms take into consideration the memory hierarchy to design cache efficient algorithms. CO approaches have the advantage to adapt to unknown and varying memory hierarchies. Recent CA and CO algorithms developed for 3D mesh layouts significantly improve performance of previous approaches, but they lack of theoretical performance guarantees. We present in this paper a $\boldsymbol{O}(N \log N)$ algorithm to compute a CO layout for unstructured but well shaped meshes. We prove that a coherent traversal of a N-size mesh in dimension d induces less than $N/B + \boldsymbol{O}(N/M^{1/d})$ cache-misses where B and M are the block size and the cache size, respectively. Experiments show that our layout computation is faster and significantly less memory consuming than the best known CO algorithm. Performance is comparable to this algorithm for classical visualization algorithm access patterns, or better when the BSP tree produced while computing the layout is used as an acceleration data structure adjusted to the layout. We also show that cache oblivious approaches lead to significant performance increases on recent GPU architectures.

Index Terms—Cache-aware, cache-oblivious, mesh layouts, data locality, unstructured mesh, isosurface extraction.

1 Introduction

Many visualization related processing steps, like isosurface extraction, rely on read-only and memory intensive algorithms. Adequately combining data layout and access patterns can significantly improve performance. Since classical processor architectures cache blocks of adjacent data, storing data accessed consecutively nearby in memory enables to reduce cache-misses. Enforcing locality is also relevant for some GPU architectures that coalesce parallel memory accesses to save clock cycles when the target data are close in memory. For instance, the Nvidia G80 and G200 [1] can coalesce concurrent threads data accesses, while the Intel Larrabee supports vector level coalesced loads and stores for its VPUs.

For regular data structures, data layouts based on space filling curves, like the Z curve, are common [2]. They provide a cache-efficient layout for access patterns showing a strong spatial locality. For irregular data structures, computing cache-efficient layouts is significantly more difficult.

We can distinguish two classes of cache-efficient algorithms: Cache-aware (CA) and cache-oblivious (CO) algorithms. CA algorithms are based on the external-memory (EM) model [3]. The memory hierarchy consists of two levels, a main memory of size M called cache and an infinite size secondary memory. The data are transferred between these two levels in blocks of B consecutive elements. CA algorithms can be very efficient but require the layouts to be recomputed for each memory architecture. It makes it difficult to efficiently share the same layout between heterogeneous processing units mixing CPUs and GPUs for instance. CO approaches [4] intend to overcome this limitation by proposing layouts that are independent from the cache size M and the block size B (Fig. 1). The Z curve is an example of a CO layout (Fig. 2). For irregular data structures, the most significant and recent work is probably the CO mesh layout proposed by Yoon et al. [5] (OpenCCL algorithm). In comparaison to other layouts, experiments show speedups ranging from 2 for in-core computations, up to 20 for out-of-core computations. This algorithm is experimentally efficient for a wide range of meshes. However, this algorithm is based on heuristics, without theoretical performance guarantees, neither on the layout computation complexity nor on the quality of the resulting layout.

In this paper, we introduce a new CO layout algorithm for irregular but well shaped meshes with a theoretical performance guarantee (Fig. 3). It relies on a recursive mesh partitioning using a specific BSP (Binary Space Partitioning) algorithm introduced by Miller et al. [6]. This algorithm cuts the mesh guaranteeing a good tradeoff between minimizing the number of cut elements and having two partitions of similar size. When applied recursively, it ensures that spatially close and strongly connected data tend to be partitioned deeper in the BSP tree. The CO layout is obtained by storing the data linearly in memory from the first leaf of the BSP tree to the last one. The data loaded in a cache block are thus contiguous leaves of the BSP tree. It is cache oblivious as to any block and cache size corresponds a BSP tree depth level ensuring a strong locality and connectivity.

- *M. Tchiboukdjian is with CNRS and CEA/DAM, DIF, ENSIMAG—Antenne de Montbonnot, Zirst 51, Avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France. E-mail: marc.tchiboukdjian@imag.fr.*
- *V. Danjean is with Grenoble Universités and Laboratoire Informatique de Grenoble (LIG) (UMR 5217), ENSIMAG—Antenne de Montbonnot, Zirst 51, Avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France. E-mail: vincent.danjean@imag.fr.*
- *B. Raffin is with INRIA and Laboratoire Informatique de Grenoble (LIG) (UMR 5217), ENSIMAG—Antenne de Montbonnot, Zirst 51, Avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France. E-mail: bruno.raffin@imag.fr.*

Manuscript received 7 Sept. 2009; revised 13 Dec. 2009; accepted 30 Dec. 2009; published online 11 Jan. 2010.
Recommended for acceptance by H. Pfister.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2009-09-0206.
Digital Object Identifier no. 10.1109/TVCG.2010.19.

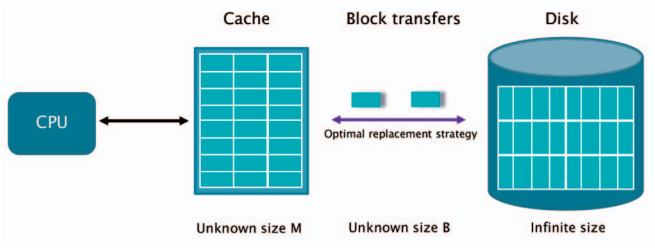

Fig. 1. The cache-oblivious memory model. The data are transferred by block of B consecutive elements into a cache of size M. Both parameters are unknown to the algorithm.

Classical BSP algorithms or space filling curves could be used in a similar way for building layouts. But these space partitioning techniques only take into account geometric information and not connectivity. Performance is not guaranteed.

Our CO layout algorithm has several benefits. The layout computation has a $O(N \log N)$ complexity. It also guarantees that a coherent traversal of a N-size mesh in dimension d induces less than $N/B + O(N/M^{1/d})$ cache-misses where B and M are the block and cache size. Experiments show that the layout computation is about two to three times faster than for the OpenCCL algorithm while requiring significantly less memory (only 2 percent of the memory used by OpenCCL on the biggest meshes). At execution, perfomance is comparable with the OpenCCL algorithm for classical access patterns. The BSP tree computed for the layout can also be used as an internal, layout consistent, acceleration data structure. Experiments reveal that using it as a min-max tree for accelerating an isosurface extraction brings significant additional performance improvements (from 12 percent to 55 percent for in-core computations) compared to using an external kd-tree not necessarily consistent with the layout.

We also show that CO layouts can lead to significant performance improvements on recent NVIDIA GPUs (speedups ranging from 1.52 to 4.09), even if no cache mechanism is involved. Because CO algorithms enforce data locality, they favor coalesced data accesses. To our knowledge, this is the first time such benefits of CO layouts on GPUs are highlighted.

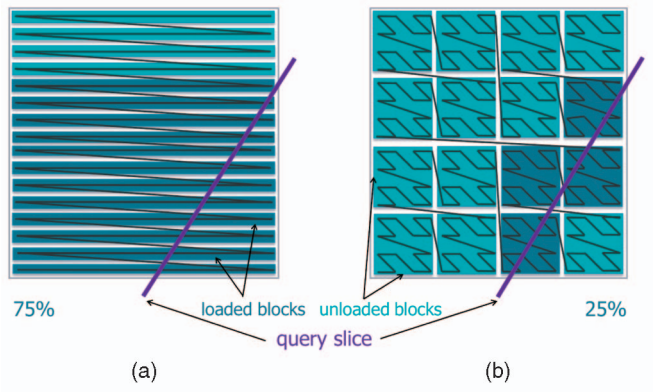

Fig. 2. Good layouts can significantly reduce the number of block transfers. (a) 75 percent of the data must be loaded to access the queried slice (each line corresponds to a cache line), while the CO layout used on (b) (Z curve) enables to reduce this amount to only 25 percent of the data (each block fits into a cache line).

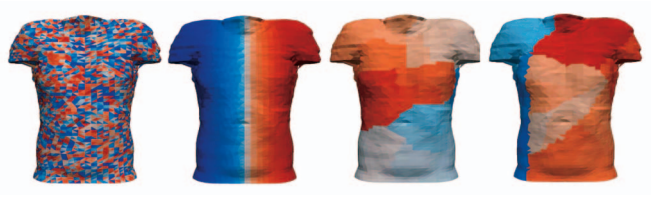

Fig. 3. Visual illustration of different layouts for the torso mesh. Successive cells in memory are colored from blue to red. From left to right, the original, geometric (sorted by x, y, and z-coordinates), OpenCCL (cache-coherent layout from [5]) and FastCOL (our approach) layouts. For spatially close tetrahedra, color discrepancy decreases from the left to right layouts. It denotes an improved memory locality, less likely to generate cache-misses for spatially coherent access patterns.

Related work is discussed in Section 2. We introduce our framework and review common mesh access patterns in Section 3. Overlap graphs, the class of meshes our algorithm applies to, and the graph separator algorithm are presented in Section 4. The CO algorithm and its implementation are described in Section 5. Experimental results are presented in Section 6 before the conclusions.

2 Related Work

2.1 Cache-Efficient Algorithms

Today, many algorithms have their CA or CO versions [7] where computations and data are reordered for an efficient cache use. A widely used technique is blocking or tiling: elements are mapped in memory and accessed by blocks of size B to fit in a cache line. For instance, regular search trees are made CA by grouping B keys in a single node. Such trees are called B-trees. Another example is matrix multiplication. Instead of rows or columns, the ATLAS library [8] traverses the matrix by blocks such that all involved blocks for a partial computation fit in the cache.

It is often possible to obtain the same result while being oblivious to the block size. For example, by carefully storing a binary tree in memory with the vam Emde Boas layout [7], a search can match the I/O bound of the CA B-tree. In this layout, the tree is divided at the middle of its height into a top tree and several bottom trees. The same layout is recursively applied to each of these trees, which are then stored sequentially in memory. The same recursive blocking technique is applied within the divide and conquer matrix multiplication. Indeed, this is a CO algorithm [7] with the same theoretical complexity as its CA counterpart.

Another CO alternative to blocking is the use of space filling curves. These layouts have been used efficiently for regular mesh traversals [2] and matrix multiplication algorithms [9].

CO algorithms for regular structures are not always as efficient as their CA counterparts. The access pattern to a CO layout is more complex, leading to a significant overhead that limits the benefit of a CO approach. For instance, the CO matrix multiplication is not competitive with the CA version [10]. We do not face this problem here as an unstructured mesh has already a complex access pattern.

2.2 Mesh Layouts

The problem of reordering mesh elements for efficient cache use was first encountered when a vertex cache was

introduced in graphic cards. To maximize the efficiency of the hardware vertex cache, triangles needed to be reordered before being sent to the graphic card. The algorithm developed by [11] reorders triangles to form triangle strips. They assume the cache has a FIFO policy and the cache size is known to the algorithm. Algorithms not based on strips that work for all cache sizes have been introduced in [12]. The layout quality has been improved in [13], and the overdraw rate reduced in [14]. When the geometry and the topology of the mesh can be modified, the method of [15] generates a single strip representing all the mesh while improving the efficiency of back-face culling. In [16], the authors propose a mesh compression scheme that is also cache efficient. However, as they target graphic cards, all these approaches only reorder mesh cells and not points. They consider the graphics card cache model (no cache line, independent vertices fetching, etc.) which is very different from the CPU cache models. Only the temporal locality on mesh points is taken into account and not the spatial locality. Moreover, the application must access the mesh in the exact same order as given by the cell layout (especially for triangle strips). Finally, work in this area mostly deals with surface meshes.

Processing sequences [17] reorder the points and the cells of a mesh, but this approach focuses on streaming computations. The goal is to minimize the maximum amount of memory used during the computation. The application should again follow the mesh layout.

OpenCCL [18] presented in [5] casts the mesh layout problem as a graph optimization problem. To describe the access pattern of the application using the mesh, the user must provide a graph where vertices represent data and edges link data that are likely to be accessed in sequence at runtime. A good mesh layout is a permutation of the graph vertices that results in a more efficient layout of the mesh in memory. They developed a local metric to decide if a swap of some vertices improves the layout. They optimize this measure thanks to a multilevel optimization scheme. In [19], two global cache-oblivious metrics are introduced to quantify the quality of a mesh layout. These two metrics involve edge lengths. If two mesh elements i and j likely to be accessed sequentially are stored in the layout at position x_i and x_j then the edge length l_{ij} is $|x_i - x_j|$. The first metric proposed (COM_a) is the arithmetic mean of edge lengths and the second (COM_g) is the geometric mean of edge lengths. While both metrics yield a good correlation with measured cache misses, COM_g seems to perform better. All previously proposed mesh layout optimization algorithms [5], [19] are based on heuristics. No bound on the quality of the layout, i.e., number of cache-misses, is provided. Our algorithm, called FastCOL, guarantees an upper bound on the number of cache-misses for the class of meshes it applies to. This bound is closely related to edge lengths, like the COM_a and COM_g metrics introduced in [19]. FastCOL is based on the mesh geometry, a data often available for the meshes considered in scientific visualization. OpenCCL is more general on that aspect as it only uses the mesh topology and thus can be applied to graphs not embedded in space.

Aforementioned approaches mainly focus on optimizing mesh layouts when the application accesses the mesh without the help of any additional data structure. That is, the application only traverses the mesh with the help of the cells and points arrays or with the cell-to-points or point-to-cells pointers of the mesh. Another approach [20] optimizes the layout for applications accessing the mesh through bounding volume hierarchies (BVH) trees. To generate an efficient layout, they use the OpenCCL algorithm and provide two kind of links in the access graph: links representing spatial locality in the mesh and links representing parent-child locality in the BVH tree. Our algorithm also handles these two kinds of locality. During the layout computation, we build a BSP tree that is used to reorder the mesh. This tree is tailored to efficiently use our mesh layout. Contrary to the approach in [20] where the layout algorithm takes a mesh and a BVH tree as input to produce a layout, we only take the mesh as input and produce both a layout and a BSP tree consistent with this layout. This BSP tree can be used as an acceleration structure, for isosurface extraction for instance.

2.3 Isosurface Extraction

The marching tetrahedra algorithm can be accelerated with various data structures allowing to efficiently search for the cells intersected by the isosurface. One such data structure is the min-max tree [21]. An octree where each node stores the minimum and maximum values of its subtree permits to quickly discard parts of the mesh that do not contain any intersected cell. The search is thus improved from $O(n)$ to $O(k + k \log n/k)$ where n is the number of cells and k the size of the isosurface (usually $k \ll n$). If the scalar field is spatially coherent, the performance is actually improved on the theoretical bound.

An optimal data structure for this problem is the interval tree storing for each cell c the interval whose extremes are the minimum and maximum value of the points of c [22]. The query time is improved to $O(\log n + k)$ whatever the spatial repartition of the scalar field is. The interval tree has been made I/O-efficient allowing a query with complexity $O(\log_B n + k/B)$, where B is the block size. This bound is optimal [23]. However, this approach is not space-efficient since the vertex information is duplicated many times. The two-level indexing scheme based on the meta-cells technique introduced in [24], [25] is both practical and space-efficient as there is no duplicated information. Spatially close cells are grouped into meta-cells, which are then used in the I/O-efficient interval tree.

The approach we developed with the consistent BSP tree is a CO alternative to the meta-cells technique, but we use the min-max tree instead of the interval tree, which may not be as efficient on a scalar field with high spatial variations.

3 Framework

3.1 Common Mesh Access Patterns

A mesh data structure usually consists of two multi-dimensional arrays: an array storing point attributes (e.g., coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g., the nature of the cell, scalar values, etc.). When the mesh is composed of cells of different nature (using various number of points), an additional array allows random access to cells (Fig. 4). As many visualization filters also need to access neighbors of a point or a cell (e.g., the gradient filter), additional structures

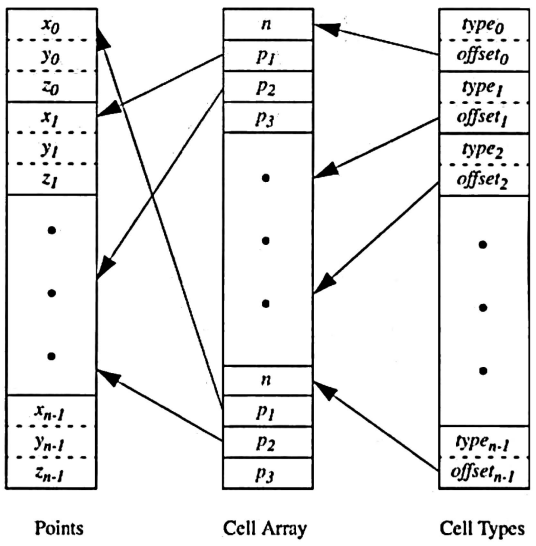

Fig. 4. The vtkUnstructuredGrid data structure (from the VTK Textbook [26]). The Points array contains points coordinates and the Cells array contains the indices of cell points. The Cell Types array contains the nature of each cell and provides $O(1)$ random access to cells.

storing the connectivity permit efficient access to point and cell neighbors. Finally, accelerating structures can be used to efficiently select cells or points verifying a certain property (e.g., select cells intersecting an isosurface).

A mesh can be traversed using the following strategies:

- **layout order traversal:** traverse all points or all cells in the order given by the corresponding array (e.g., the marching tetrahedra algorithm);
- **connectivity traversal:** traverse all points or all cells using the connectivity information (e.g., the ray casting algorithm or the VTK connectivity filter);
- **data structure traversal:** traverse points or cells in the order given by an external data structure (e.g., isosurface extraction with a min-max tree).

While traversing the mesh, several local operations are commonly used to process a mesh element:

- **neighborhood operation:** get all points/cells connected to the current point/cell (e.g., the VTK gradient filter);
- **attributes operation:** get attributes from points composing the current cell or get attributes from cells using the current point (e.g., marching cube).

Multiple local operations can be used at the same time.

3.2 Layout Influence on Cache Performance

The cache performance of visualization filters is greatly influenced by the underlying mesh data structure and specifically the indices of points and cells: the mesh layout. These indices can be modified without affecting the intrinsic characteristics of the mesh like the geometry or the topology and without any modification on the visualization filters. Depending on the access pattern, the layout can impact the cache performance in various ways.

- Cache performance is optimal with **layout order traversals** as they lead to sequential memory accesses.
- A layout improves cache performance of **connectivity traversals** and **neighborhood operations** if the elements that are connected in the mesh topology are stored nearby. Spatial locality is increased as two consecutive cells in the traversal could be in the same memory block. It also enhances temporal locality since the current cell in the traversal has a good probability to still be in cache if its memory block has been accessed recently.
- A layout improves cache performance of **data structure traversals** if the elements that are accessed consecutively by the data structure are stored nearby. Two consecutive elements could be in the same memory block, increasing spatial locality.
- A layout improves cache performances of **attributes operations** if points corresponding to a same cell (or cells using a common point) are stored nearby. Points may share memory blocks, which increases spatial locality.

A layout order traversal with attributes operations can be further optimized if the layout stores nearby the cells that share common points. The memory blocks containing the common points have a higher chance to still be in cache, which enhances temporal locality. The marching tetrahedra algorithm is an example of such access pattern.

3.3 The Access Graph Model

To optimize a layout for a specific access pattern, we need to model the data accesses. As in [5], we use a graph $G = (V, E)$ where vertices are mesh elements (point or cell) and edges represent consecutive data accesses. However, we constrain the topology of the access graph to forbid edges between elements that are 'far' from each other, as detailed in Section 3.5.

We now model a visualization filter as a function f applied once to each element of the mesh. As we are interested in the cache performance, we do not consider the processing part of f. We restrict its memory accesses so that they are compatible with the access graph: only the neighbors of the element i in the access graph can be accessed to compute $f(i)$.

Fig. 5 presents examples of access graphs for the neighborhood and attributes operations. A point neighborhood operation is represented in Fig. 5a. This kind of local access scheme is used by the VTK gradient filter. To compute the gradient value at a point, values of the scalar field at the neighborhood points are needed, thus edges link neighbor points in the access graph. Likewise, a point attributes operation is represented in Fig. 5c. The marching cube algorithm is an example of such a scheme. Indeed, to compute the isosurface within a cell, the coordinates and scalar value for each point composing the cell are needed. Therefore, edges link each cell to its points in the access graph.

3.4 Access Graph for Layout Order Traversals

Visualization filters do not always rely on intrinsic mesh characteristics such as topology or geometry when accessing the mesh. They sometimes rely on the layout itself. For example, the Seed Set isosurface extraction algorithm processes the mesh with a connectivity traversal. The access graph does not change when the layout changes, provided

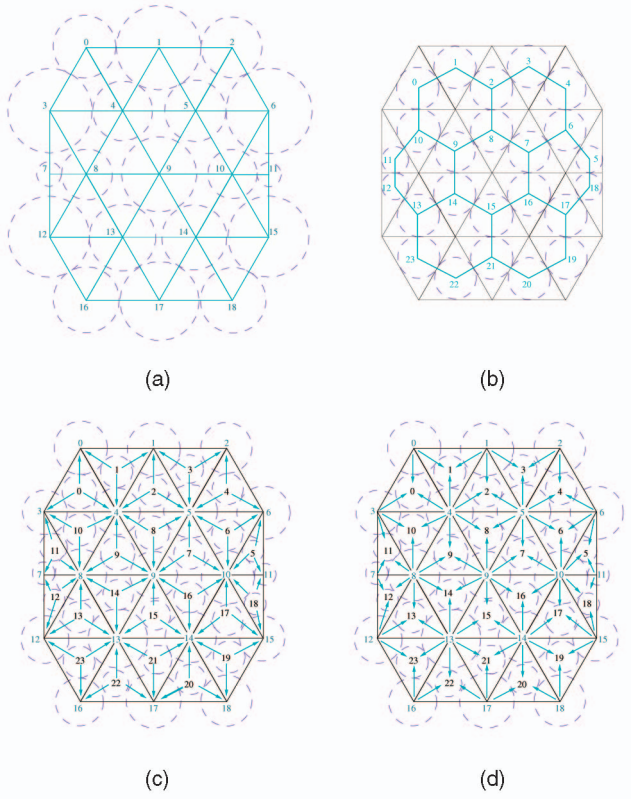

Fig. 5. Example of access graphs for the neighborhood and attributes operations defined in Section 3.1. Numbers represent points (a), cells (b), or points and cells (c, d) indices in the layout. Data accesses are represented in blue: neighbor points in (a), neighbor cells in (b), points of each cell in (c), and cells of each point in (d). Dashed circles show that all these graphs are overlap graphs (cf. Section 4.1). (a) Points neighborhood, (b) cells neighborhood, (c) points attributes, and (d) cells attributes.

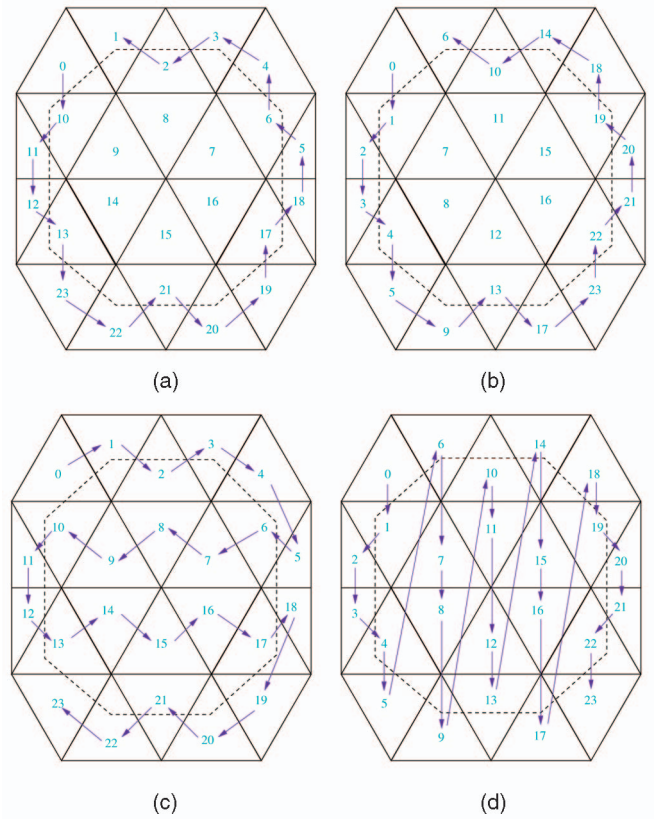

Fig. 6. Example of access patterns for two isosurface extraction algorithms: Marching Cube (layout dependent) and Seed Set (layout independent). Arrows represent the access pattern followed to extract the isosurface (dashed line). Numbers represent cell indices in the layout. (a) Seed set (layout 1), (b) Seed set (layout 2), (c) Marching Cube (layout 1), and (d) Marching Cube (layout 2).

that the initial seeds stay the same (Fig. 6a and 6b). On the contrary, the Marching Cube algorithm processes the mesh with a layout order traversal, and thus, the access graph depends on the layout (Fig. 6c and 6d).

The access graph of Fig. 5c properly models the local operations of the marching cube algorithm. However, to optimize the global traversal strategy, the edges of the access graph of Fig. 5b should be added instead of using the access graphs of Fig. 6c or 6d. First, because the resulting access graph does not depend on the layout. Second, because it enables temporal locality optimization: cells that share common points should be stored nearby.

3.5 Restriction to Overlap Graphs

Access graphs can model a large range of access patterns, even ones with a weak spatial coherency where edges connect distant elements. To be able to build efficient CO layouts with a provable quality, we restrict access graphs to be overlap graphs. These graphs model spatially coherent access patterns, i.e., where edges connect spatially close elements. They are formally defined in the next section.

In contrast to OpenCCL [5], we add geometric information to the access graph and use it to constrain its topology. We add to each vertex the coordinates of the corresponding element of the mesh and we restrict the graph to be an overlap graph. This assumption forbids consecutive access of mesh elements that are too 'far' from each other. This restriction is satisfied by most visualization filters and allows us to devise an efficient separator-based algorithm with a theoretical guarantee on the quality of the mesh layout generated.

Meshes are often composed of elements that are well shaped in some sense, such as having a bounded aspect ratio or angles that are not too small or too large. Provided that the underlying mesh is constrained by such geometric features, the access graphs for connectivity traversals, neighborhood, and attributes operations are overlap graphs [6] (Fig. 5). We have seen in the previous section that the layout order traversals should be handled differently depending on the visualization filter as they are based on the layout.

The only remaining mesh access pattern is data structure traversal. Unfortunately, access graphs for data structure traversals may not always be overlap graphs. For instance, the interval tree used in the isosurface extraction of [22] does not traverse cells intersected by the isosurface in an order based on the geometry or topology of the mesh. However, in this case, the access graph depends on the value of the isosurface. It is not practical to compute a layout and reorder the mesh for each isosurface extraction. We develop an alternative with our consistent BSP tree (cf. Section 5.6). This tree is tailored to efficiently use our layout as the induced mesh traversal is a layout order traversal with good cache performance.

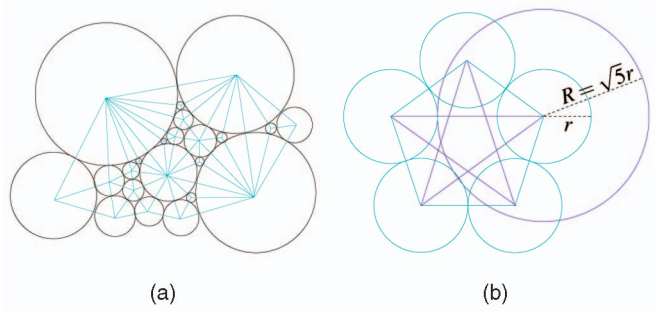

Fig. 7. (a) Example of overlap graph for $\alpha = 1$. There is an edge (solid blue line) between two points if their corresponding circles are tangential. (b) Example of a nonoverlap graph for $\alpha = 1$ but overlap for $\alpha = \sqrt{5}$ (K_5 complete graph).

4 Overlap Graphs Partitioning

In this section, we review the work of Miller et al. [6] on overlap graphs, which we use to restrain the topology of the access graph.

4.1 Overlap Graphs

We associate to each vertex v_i of the access graph the coordinate p_i in $\mathbb{R}^d$ of the corresponding mesh element (point or cell). A graph is α-overlap if:

- It is possible to associate to each vertex v_i a ball B_i centered at p_i such that the two balls of any pair intersect in at most one point (Fig. 7a);
- Edges can connect two vertices only if expanding the smaller of their two balls by a factor of α make them intersect (Fig. 7b).

The α factor constrains the topology of the graph to follow the geometry of the mesh: two elements that are too far away from each other cannot be edge connected (Fig. 7b).

As detailed in Section 3.5, most access patterns of visualization filters can be modeled with overlap graphs. Fig. 5 shows how balls can be added to the neighborhood and attributes access graphs so that all edges respect the overlap graph constraint.

4.2 Geometric Separator Algorithm

Given their geometrical properties, overlap graphs can be partitioned efficiently in two parts of approximately equal size, while minimizing the number of edges cut.

The following randomized algorithm introduced by Miller et al. [6] computes in linear time and with a high probability an optimal geometric separator (Algorithm 1). It starts by randomly sampling a constant number of points V_s from the input graph. Next it projects these V_s points onto the surface of the unit sphere centered at the origin in $\mathbb{R}^{d+1}$, using a stereographic projection. It produces V_p points. Then it finds a centerpoint c of this random sample V_p in linear time relative to the sample size. A point is a centerpoint if every hyperplane passing through it divides the sample set V_p approximately evenly, at most in a ratio $d+1 : 1$. With good probability, this centerpoint is a centerpoint of the projection of the original set of points V [27]. Finally, we randomly choose a hyperplane $(c, \mathbf{n})$ passing through this centerpoint. This hyperplane splits the graph into two partitions, each one consisting of the points of V that project on the same side of the hyperplane in $\mathbb{R}^{d+1}$. Repeating this process and selecting the separator cutting the smallest number of edges gives a small separator with high probability:

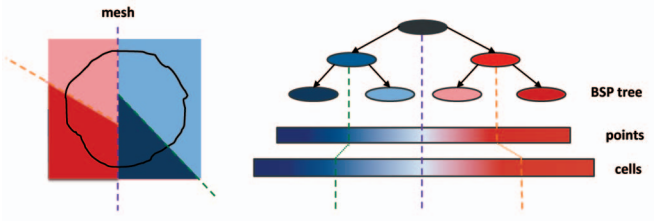

Fig. 8. Illustration of the correspondence between mesh regions, BSP tree branches, and data arrays.

Algorithm 1. Geometric separator algorithm
Input: Graph $G =$ (Vertices V, Edges E)
Output: A separator ϕ
1: **repeat** n_c **times**
2: $V_s \leftarrow$ sample of $(d+3)^4$ points of V
3: $V_p \leftarrow$ project V_s to the unit sphere in $\mathbb{R}^{d+1}$
4: $c \leftarrow$ find a centerpoint of V_p
5: **repeat** n_h **times**
6: $\mathbf{n} \leftarrow$ random normal vector
7: $\phi \leftarrow$ separator defined by $(c, \mathbf{n})$
8: compute the number of edges cut by ϕ
9: **end**
10: **end**
11: **return** the best ϕ

The most time consuming part of the algorithm is the quality evaluation of the separator (Algorithm 1 line 8). The other operations involve only a small number of points.

The quality of the obtained separator is guaranteed by the following theorem:

Theorem 1 (Geometric separator [6]). *Let G be an n-vertex α-overlap graph in d dimensions. With high probability, the previous algorithm (Algorithm 1) partitions the vertices of G into two sets A and B such that $|A|, |B| \leq \frac{d+1}{d+2} n$ and the number of edges between A and B is $O(\alpha n^{1-1/d})$.*

Such a separator is asymptotically optimal for the class of overlap graphs. Indeed, we cannot find a smaller separator for a regular d dimensional grid [6].

5 Recursive Mesh Layout

Applying the separator algorithm recursively for a given overlap graph corresponding to the mesh access pattern enables us to define a CO layout. In this section, we present the CO layout computation algorithm, prove its performance, and discuss some implementation details.

5.1 Mesh Layout Algorithm

The recursive application of the separator gives a BSP tree where each node is a separator (Fig. 8). Leaves of this tree correspond to small subparts of the mesh that are stored consecutively to provide the layout (Algorithm 2).

Algorithm 2. Layout algorithm
1: **function** COLAYOUT(G,layout,i,j)
2: **if** size$(G) > 1$ **then**

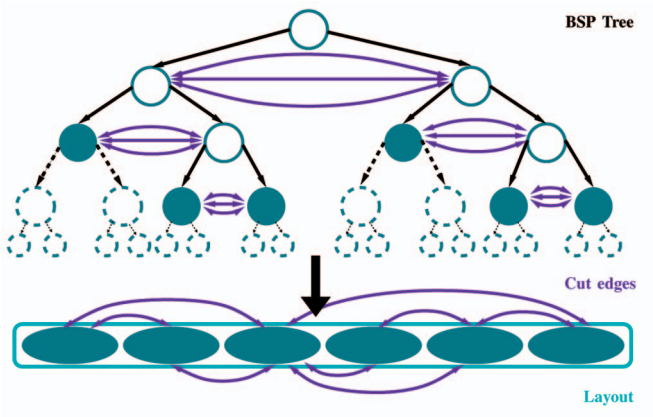

Fig. 9. A full tree generated by the Algorithm 2. The subtree (solid lines) represents T_m and the purple arrows the edges cut for this subtree. The leaves of T_m (green filled) all have less than m vertices. Below, the green ellipses identify the leaves of T_m in the layout.

3: FINDSEPARATOR(G,G_{left},G_{right})
4: $n_{left} \leftarrow \text{size}(G_{left})$
5: COLAYOUT(G_{left},layout,i,$i + n_{left}$)
6: COLAYOUT(G_{right},layout,$i + n_{left}$,j)
7: **end if**
8: **end function**

Given the linear complexity of the geometric separator, our layout algorithm has a complexity of:

$$W(N) = \max_{1/2 \leq \lambda \leq \delta} [W(\lambda N) + W((1-\lambda)N)] + \boldsymbol{O}(N)$$
$$= \boldsymbol{O}(N \log N),$$

where $\delta = \frac{d+1}{d+2}$. The only requirement to obtain the claimed complexity is to have a point sampler of linear complexity and an iterator on edges of linear complexity too.

5.2 Layout Quality

The Algorithm 2 generates a BSP tree that can be used to create a partition of the access graph such that each subpart fits in cache. When processing the mesh, the edges of the access graph linking elements in the same subpart do not generate extra cache misses as the whole subpart fits in cache. Conversely, edges of the access graph linking elements in different subparts (cut edges) may generate extra cache misses. Because the number of such cut edges is known, we can exhibit an upper bound on the number of cache misses for the layout (Theorem 3). The number of cut edges is equal to the total number of edges cut by all separators down to the largest nodes fitting in cache (Lemma 2 and Fig. 9).

Lemma 2 (Cut edges). *Let G be a N-vertex α-overlap graph in d dimensions. Let T be the BSP tree obtained by recursively applying the geometric separator. Let T_m be the tree corresponding to T after removing all nodes that have a father node with less than m vertices. The leaves of T_m verify* $\text{size}(\text{father}(x)) > m$ *and* $\text{size}(x) \leq m$ *(Fig. 9). The total number of edges cut by all separators in T_m is bounded by $k_m = \boldsymbol{O}(\frac{N}{m^{1/d}})$.*

Proof. We sum the number of edges cut by all separators from the root of T_m to its leaves. The separator theorem (Theorem 1) ensures that the number of cut edges is less than $\alpha c r^{1-1/d}$ for a r-vertex graph (c is a constant). It provides two subgraphs of size λr and $(1-\lambda)r$ with $1/2 \leq \lambda \leq \frac{d+1}{d+2}$.

The total number of cut edges in a subtree rooted at a node $v \in T_m$ representing an r-vertex graph is thus:

$$K(r) \leq \max_{1/2 \leq \lambda \leq \frac{d+1}{d+2}} [K(\lambda r) + K((1-\lambda)r)] + c\alpha r^{1-1/d}.$$

The $K(\lambda r)$ and $K((1-\lambda r))$ terms are due to the edges cut by all the separators in the left and right subtrees. The $c\alpha r^{1-1/d}$ term corresponds to the edges cut by the separator of node v. By induction on r, we show that

$$K(r) \leq c' \left(\frac{r}{m^{1/d}} - r^{1-1/d} \right)$$

taking

$$c' \geq \frac{\alpha c}{2^{1/d} - 1}.$$

And thus,

$$k_m = K(N) = \boldsymbol{O}\left(\frac{N}{m^{1/d}} \right).$$

□

We now assume that the mesh is traversed by chunks of m elements, i.e., each chunk contains m consecutive elements in the layout that should all be processed (in any order) before accessing another chunk anywhere in the layout. The size of a chunk m expresses how much the filter access pattern respects the layout locality. As spatially close elements in the mesh tend to be close in the layout, filters with spatially coherent access patterns use big chunks.

Theorem 3 (Chunk traversal). *The CO layout guarantees that a traversal by chunks of size $m \leq M$ of an N-size mesh induces less than $N/B + \boldsymbol{O}(N/m^{1/d})$ cache misses where B and M are the block and cache size, respectively.*

Proof. Assume first that there is no edge cut in T_m, i.e., processing an element in a chunk only accesses elements in the same chunk. Processing a chunk would not induce any cache miss beside the m/B compulsory ones to read the chunk as the entire chunk fits in cache. This sums up to N/B cache misses for processing all the mesh. However, processing an element may require data that are not in the same chunk, causing $\boldsymbol{O}(1)$ extra cache-misses per edges linking elements in different chunks: the cut edges. Accessing an element in another chunk induces one cache miss to read the element and may generate another one as it can evict a block of the current chunk that may still be needed. The total number of these extra cache misses is proportional to the number of cut edges: $\boldsymbol{O}(\frac{N}{m^{1/d}})$. We thus obtain the claimed bound. □

For the sake of simplicity, we assume in the proof that the chunks are perfectly aligned with the leaves of T_m. One can easily show that there are still $\boldsymbol{O}(\frac{N}{m^{1/d}})$ cut edges when the chunks are not aligned with the leaves. Consider leaves of size $2m$ and add edges within a leaf that link different chunks. This only modifies the number of cut edges by a constant factor.

Corollary 4 (Layout order traversal). *The CO layout guarantees that a layout order traversal of an N-size mesh induces less than $N/B + \boldsymbol{O}(N/M^{1/d})$ cache misses where B and M are the block and cache size, respectively.*

Proof. A layout order traversal is a traversal by chunks of size M. □

With our layout, a visualization filter still needs to traverse the mesh in an order coherent with the layout, but the assumption is strongly relaxed compared to a layout order traversal. We believe that we could obtain the same performance guarantee slackening the traversal by chunks assumption to rely only on the characteristics of the mesh itself. We are however not able to prove it yet. Experiments (cf. Section 6) use visualization filters that traverse the mesh in the layout order (e.g., gradient, vtkiso, cpuiso, etc.), filters that traverse the mesh by connectivity (e.g., connectivity, RC), and filters that traverse the mesh through another data structure (e.g., CpuTree). All of them yield speed up, which indicates that in practice the chunk traversal assumption is usually verified for some m.

At this point, we cannot directly compare this algorithm with OpenCCL. OpenCCL is based on a meta-heuristic and no upper bound on the quality of the resulting layout is given.

5.3 Layout Computation

We implemented the geometric separator algorithm in C++. We first randomly generate all the $n_h n_c$ separators (Algorithm 1). We then traverse all the cells of the mesh and for each of them we check that its points are on the same side of the separator. If not, the cell is cut by the separator and we increment the cut size by 1. Using cells instead of edges to select the best separator produces a very close result and allows us not to compute the edges of the graph, a task that can be computationally expensive. The bound of Theorem 3 still applies as at most a constant number of edges correspond to a cell. All separators are checked against a cell before going to the next one. This allows us to dereference each cell index only once for the entire separator computation.

To keep the memory usage low, we do not project all the points before evaluating a separator but project them on the fly. This induces duplicate computation as a point is used in several cells but keep memory overhead close to zero. That way we do not need to store an entire copy of the points in memory.

Once we found the best separator, the points of the mesh are reordered according to this separator. All points laying to the left of the separator are moved to the left part of the array and points laying to the right are moved to the right part of the array. The same partitioning is done on cells. When a cell is cut by the separator, we choose a side according to the center of gravity of the cell. We then recurse on the left and right mesh generated. This algorithm is very similar to a quicksort and could be efficiently parallelized.

We stop when a submesh has a size lower than 8. We choose $n_c = 2$ and $n_h = 30$ for the experiments (Algorithm 1). As the randomized centerpoint algorithm is quite good, we can keep n_c low. During our experiments, we noticed that even substantial changes of all these parameters did not impact significantly the quality of the generated layout.

5.4 Choosing the Access Graph

The algorithm described in Section 5 can be applied to any access pattern as long as the corresponding graph is an overlap graph (actually the algorithm still works if it is not the case but the bound on cache-misses does not hold). However, to generate a new layout for each application is not practical. For instance to compute a volume rendering by ray casting of the mesh, one might want to optimize the mesh layout according to the rays direction. Both our algorithm and OpenCCL are too slow to generate a layout before each image generation.

In practice it is better to compute only once a layout that will be efficient in general. We choose in the implementation to only consider the graph where vertices are points of the mesh and edges link two points sharing a cell (Fig. 5a). Using this access graph produces an efficient layout for most access patterns as access graphs for connectivity traversal, neighborhood and attributes operations look alike (Fig. 5). Following on our volume rendering example, this layout will be reasonably good for any ray direction. One ray traversing c cells of the mesh induces $c/B^{1/3}$ cache-misses while a layout optimized for this specific direction may induce only c/B cache-misses (but as bad as one cache-miss per cell for an orthogonal direction). In this specific case, packing rays should also improve performance of the more general layout to c/B.

5.5 Cells Layout

A mesh layout tries to optimize both points and cells ordering. As points and cells are usually accessed in a similar way, a consistent ordering for points and cells is better. For instance in an isosurface extraction, points composing a cell are often accessed immediately after the cell itself. Thanks to our geometric approach, the same geometric separators can be applied for both points and cells. A separator cutting few edges for the points graph (Fig. 5a) also cut few edges for the cells graph (Fig. 5b). It is not possible to do the same with OpenCCL for example as their separators are combinatorial and not geometric.

Computing points and cells layouts independently often leads to a larger computation time and a lower runtime efficiency. For OpenCCL, computing the cell layout is around three times slower than computing the points layout on our meshes. A consistent cells layout (min-vertex), can be deduced from the points layout by sorting the cells using the minimal index of their points.

Having consistent points and cells layouts can also improve runtime performance. The min-vertex approach enforces the consistency. For instance, min-vertex with OpenCCL leads to a 20 percent runtime performance increase compared to applying OpenCCL to both points and cells. The FastCOL layout further enforces this consistency. On large meshes the consistent layout for points and cells with the BSP tree is up to 10 percent faster than the min-vertex layout applied to FastCOL points layout on isosurface extraction.

5.6 Consistent BSP Tree

The BSP tree defining the partitioning of the mesh can be used as an acceleration structure that has the advantage of being consistent with the layout. In this paper, we show how

it can be used as a min-max tree for isosurface extraction. At each node of the BSP tree, we store the minimum and maximum value of the scalar field in the corresponding region of the mesh. A region that do not contain any cell intersected by the isosurface can be quickly discarded.

An interesting property of this BSP-tree is that each region corresponds to a small part of the mesh stored sequentially in memory (Fig. 8). When traversing the BSP-tree in prefix order and examining the mesh cells that might contain a part of the isosurface, mesh cells are accessed sequentially. The sequence of cells can jump part of the mesh but it never goes back. This leads to a layout order traversal of the mesh that induces fewer cache-misses (see experimental results in Section 6.3).

6 Experiments

We compare the performance of the initial, geometric, OpenCCL, and FastCOL layouts on various meshes and access patterns. For sake of conciseness, we present only some representative results. Full results are provided in our research report [28].

6.1 Architectures, Filters, and Meshes

We took nine different meshes,[1] processed to generate several instances of various sizes. We used tetgen[2] to refine the meshes by adding a volume constraint to each tetrahedron.[3] For each mesh and each size (100 k, 1 M, 10 M, and 50 M cells) we generated two finer meshes. In the first one, all tetrahedra have approximatively the same volume. In the second one, we used a volume constraint proportional to the inverse of the gradient of the scalar field to mimic an adaptive mesh refinement. It leads to a set of 50 meshes that can be divided by their size into four groups: 5 meshes of about 100 k cells, 10 meshes of about 1 M cells, 17 meshes of about 10 M cells, and 18 meshes of about 50 M cells.

The experiments were conducted on three different architectures, two classical CPU architectures with two cache levels (AMD Opteron875 @ 2.2 Ghz, cache L1 8 KB, cache L2 1 MB, and Intel Core2 E6750 @ 2.66 Ghz, cache L1 32 KB, cache L2 4 MB), and one GPU architecture (NVIDIA GTX280 with 1 GB of memory) tested to probe the influence of the layout on the number of coalesced memory accesses.

Ten filters were tested on each layout, using VTK filters [26], homemade CPU codes or Cuda (version 1.3) codes for the GPU tests:

- **Gradient.** The VTK gradient filter computes the gradient of the mesh scalar field. Each gradient value is computed from the local scalar value and the values of neighbor points. Data are processed in the order given by the point layout. Using the terms introduced in Section 3 this is a point layout order traversal with point neighborhood operations.
- **Connect.** The VTK connectivity filter applies a breadth first search on the mesh to compute the connected region each cell lies in. This filter uses a connectivity traversal.
- **RC.** A mesh volume rendering computed by the VTK Bunyk Ray Cast filter [29]. Each ray traverses the mesh cell by cell and then accesses points attributes to compute the contribution of the cell to the pixel color. This is a connectivity traversal with point attributes operations.
- **PT.** A mesh volume rendering computed by the VTK Projected Tetrahedra filter [30]. Tetrahedra are sorted by their centroid according to the viewing direction and then sent to the GPU for projection. During the sorting phase, each tetrahedron accesses its points and tetrahedra are processed in the order given by the cell layout. This is a cell layout order traversal with points attributes operations. Both CPU and GPU computations are included in the time measure but only the CPU part is included in the number of cache misses.
- **HAVS.** A mesh volume rendering computed by the VTK HAVS filter [31]. Data accesses are similar to PT. Again, both CPU and GPU computations are included in the time measure but only the CPU part is included in the number of cache misses.
- **VtkIso.** The VTK isosurface extraction filter implements the marching tetrahedra algorithm. Each cell accesses to its points. Cells are processed in the cell layout order. This is a cell layout order traversal with points attributes operations.
- **CpuIso and GpuIso.** One CPU and one GPU homemade implementation of the marching tetrahedra isosurface extraction algorithm.
- **CpuTree and GpuTree**. The CpuIso and GpuIso code extended to include a min-max tree acceleration structure. For the OpenCCL layout a kd-tree is used. For the FastCOL layout two versions are tested: one based on a kd-tree and one relying on the BSP tree built when computing the layout. Only some cells are processed, in a min-max tree driven order. This is a data structure traversal with points attributes operations. We only time the processing of the cells intersected by the isosurface and not the tree traversal (the kd-tree is the same for both layout and the code for its traversal is not optimized).

The bigger meshes (50 M cells) have not been tested with the volume rendering filters due to the very large execution time, nor on the GPU that has only 1 GB of memory.

For our GPU implementation, we only measure the time to compute the kernel and not the memory transfers between CPU and GPU, which take the same amount of time for all layouts.

6.2 Layout Algorithm Performance

All layouts have been prepared on an Opteron875 @ 2.2 Ghz with 32 GB of memory and 64 GB of swap. Table 1 shows the execution time and memory needs for computing the OpenCCL and FastCOL layouts. Our FastCOL program is about three times faster than the OpenCCL one. It requires far less memory. The bigger meshes with 50 M cells have not been processed with OpenCCL because their computation would have required more than 96 GB of memory. The

1. Blunt fin, buckyball, langley fighter, liquid oxygen post, plasma64, san fernando, and spx models are provided by the AIM@SHAPE Shape Repository (http://shapes.aim-at-shape.net/). Torso is courtesy of SCI and the last one is not published.
2. Available at http://tetgen.berlios.de/.
3. We used the command `tetgen – raq`.

TABLE 1
Layout Computation (on Opteron)

Mesh size		OpenCCL		FastCOL	
#cells	Bytes	Time	Mem. space	Time	Mem. space
100k	3.7 MB	4.5 s	123 MB	1.2 s	22 MB
1M	43 MB	54 s	1.24 GB	17 s	81 MB
10M	370 MB	9 min 24 s	9.96 GB	3 min 50 s	0.56 GB
50M	1.8 GB	NA	$>$ 96 GB	26 min 44 s	2.72 GB

multilevel heuristic used in OpenCCL may explain such memory consumption. Space is needed at each level to store the coarsened access graph and additional information to undo the coarsening operation.

Computing the geometric layout, a coordinate sort by the x, y, and z-axes, is very fast (less than 40 s for the biggest meshes) and compact in memory.

6.3 Mesh Layout Performance

We measured the execution time, the number of L1 and L2 cache-misses using the PAPI software [32] for the CPU tests, and the number of uncoalesced parallel accesses for the GPU ones. For each experiment (architecture, layout, and algorithm fixed), the execution time, the numbers of cache-misses and uncoalesced accesses are very stable.

Tables 2 and 3 show the means of the speedup ("Speedup"), ratio of saved L2 cache-misses on CPU ("L2") and ratio of coalesced memory accesses on GPU ("Coal.") for the geometric, OpenCCL and FastCOL layouts. These ratios are relative to the performance obtained with the initial layout. In all cases, higher values are better. Table 2[4] gathers the results for the tests visiting the entire mesh, while Table 3 displays the performance results for the min-max tree accelerated isosurface extraction using a kd-tree for the OpenCCL and FastCOL layouts, and the BSP tree computed for the FastCOL layout ("FastCOL (bsp)").

TABLE 2
CPU and GPU Performance Ratios Relative to the Original Layout (on Core2)

	Mesh	Geometric		OpenCCL		FastCOL	
	size	Speedup	L2	Speedup	L2	Speedup	L2
Gradient	100k	1.02	1.51	1.01	1.49	1.02	1.52
	1M	1.06	3.53	1.07	4.03	1.08	3.94
	10M	1.07	2.36	1.15	8.22	1.15	7.81
	50M	1.1	1.34			1.36	10.53
Connect	100k	0.95	0.94	1.12	1.17	1.11	1.21
	1M	0.97	0.95	1.16	1.19	1.19	1.19
	10M	1.09	1.08	1.45	1.49	1.46	1.49
	50M	0.89	0.87			1.66	1.9
RC	100k	0.98	1.08	1.05	1.4	1.06	1.36
	1M	1.01	1.07	1.2	1.8	1.2	1.79
	10M	0.76	0.72	3.28	5.02	3.2	4.89
PT	100k	1.06	0.82	0.93	1.18	1.15	1.18
	1M	0.91	0.64	1.09	1.51	1.1	1.52
	10M	0.97	0.9	1.37	2.66	1.37	2.65
HAVS	100k	1.02	2.04	1.01	2	1.08	2
	1M	1.14	3.43	1.06	4.04	1.09	4.03
	10M	1.2	1.9	1.33	5.96	1.32	5.77
VtkIso	100k	0.99	1.04	1.04	1.06	1.04	1.09
	1M	1.1	1.22	1.15	1.24	1.15	1.24
	10M	1.32	1.71	1.44	1.79	1.44	1.78
CpuIso	100k	1.06	1	1.16	1.02	1.17	1.02
	1M	1.71	2.85	2.34	2.8	2.35	2.78
	10M	2.28	5.4	4.08	5.78	3.99	5.68
	50M	0.97	0.79			4.87	6.84
		Time	Coal.[4]	Time	Coal.[4]	Time	Coal.[4]
GpuIso	100k	0.96	1.18	1.56	3.08	1.52	2.97
	1M	1.26	1.04	2.2	2.59	2.11	2.38
	10M	1.83	1.25	4.09	3.85	3.8	3.39

TABLE 3
CPU and GPU Performance Ratios Relative to the Original Layout for Tree Accelerated Isosurface Extraction (on Core2)

	Mesh	OpenCCL		FastCOL		FastCOL (bsp)	
	size	Speedup	L2	Speedup	L2	Speedup	L2
CpuTree	100k	1.23	1.31	1.21	1.30	1.37	1.23
	1M	1.46	1.8	1.45	1.74	1.75	1.72
	10M	2.37	3.14	2.31	3.02	2.75	3.06
	50M			2.92	3.47	4.55	4.85
			Coal.[4]		Coal.[4]		Coal.[4]
GpuTree	100k	1.20	1.85	1.18	1.75	1.33	1.90
	1M	1.63	1.81	1.57	1.67	1.84	1.89
	10M	2.50	2.14	2.34	1.86	2.79	2.14

6.3.1 CO Layouts on CPU

Table 2 shows higher performance ratios with larger meshes where cache effects are predictably more important. Indeed, with smaller meshes, a bigger part of the mesh can be loaded in the cache, whatever the layout is. Both CO layouts, OpenCCL, and FastCOL, lead to speedup ratios from 1.01 to 4.87, all tests being in-core. It shows the benefits of CO layouts that can bring significant performance increases without any change to the application. The geometric layout is significantly less efficient for most of the tests, a result analyzed in Section 6.3.2.

The FastCOL layout reaches similar performance compared to OpenCCL, while providing a theoretical performance guarantee.

Some important differences are observed between the L2 cache-miss ratio and the speedups for the Gradient and HAVS tests. Gradient is computationally intensive and HAVS extensively uses the GPU, making the cache-miss overhead a small fraction of the overall computation time.

We can also observe that measured speedups are generally smaller with VTK filters than with homemade ones. This clearly appears for the isosurface filter that is implemented with VTK (VtkIso) compared to the homemade code (CpuIso). The VTK implementation shows a maximum speedup of 1.44 whereas our implementation goes up to 4 (with a smaller global execution time). The VTK library is not fully optimized and performs several other computations. For instance, after the extraction of the isosurface, the VtkIso filter merges the identical points to provide a mesh (instead of a triangle soup) as a result.

6.3.2 Edge Lengths and Layout Quality

To better analyze the properties of the different layouts, we analytically relate the performance improvements to the

4. Ratio of coalesced parallel memory accesses on GPU.

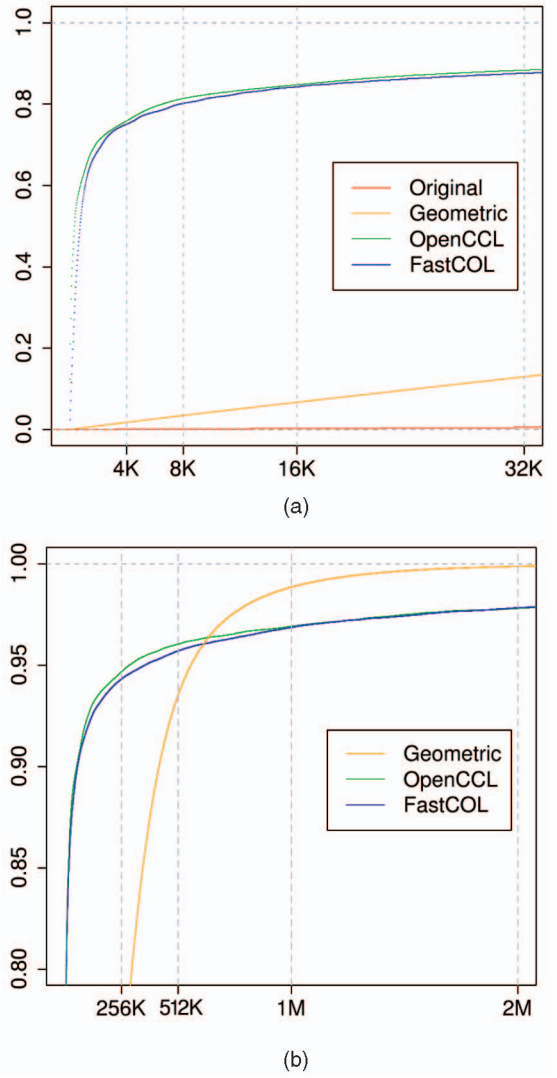

Fig. 10. Cumulative distribution function of edge lengths for various layouts applied to the torso mesh (10 M resolution). The CO layouts (OpenCCL and FastCOL) favor small edges: 80 percent of their edges have a length below 8 K (a) and 95 percent below 256 K (b). The original layout does not appear on graphic (b) as the cumulative distribution is too small: only 40 percent of its edges have a length smaller than 2 M. (a) Zoom on L1 cache sizes and (b) zoom on L2 cache sizes.

better data locality in memory. We call "edge length" the memory gap between two vertices of the same edge in the vertex array loaded in memory. If a mesh has shorter edges, more of them will fit in cache and a better performance should be observed. Other analysis could also be conducted with similar metrics. For example, instead of considering the length of edges, we can consider the "size of a cell," which would be either the maximum memory gap between all vertices of the cell in the vertex array, or the maximum memory gap between all adjacent cells in the cell array.

Fig. 10 shows the cumulative distribution of edge lengths for the 10 M cells torso mesh.[5] The two graphs are focused around the L1 and L2 cache sizes of the tested architectures. CO layouts appear to favor small edge lengths.

The geometric layout behaves differently. The amount of small edges is reduced compared to CO layouts, but almost all edges have a length shorter than 2 M. Actually, by construction, the edge lengths are shorter than the size of two entire slices of the mesh in the x direction. The layout thus leads to a good performance when two slices in the x direction can fit in the L2 caches. This is visible in the results where the geometric layout performs well for small meshes while it is outperformed by the CO layouts for the bigger ones. For small meshes, the geometric layout is often slightly less efficient due to its low efficiency with respect to the L1 cache (L1 cache-miss ratios omitted for sake of conciseness).

We now estimate the number of cache misses using an edge length based metric, and show that there is a strong correlation with the actual number of observed cache misses. Let N be the size of a mesh (in bytes), E the set of all edges of the mesh, B the cache line size, and M the cache size, we estimate the number of cache-misses by:

$$ExpectedCM \approx \frac{N}{B} + \sum_{e \in E} \mathbb{1}_{\lambda_e > M},$$

where λ_e is the length of the edge e. We count the number of cache-misses for a linear full read of the data arrays, and we add one cache-miss per edge whose length is bigger than the cache size M.

The theoretical upper bound $\frac{N}{B} + O(k_M)$ for our FastCOL algorithm is larger because we count one cache miss for each cut edge in the T_M BSP tree (Theorem 3). Some cut edges counted in our theoretical bound can in fact have an edge length shorter than M.

In Fig. 11, we display the correlation between the expected cache-misses for the considered mesh layouts and the cache misses observed on both CPU architectures. The $\frac{N}{B}$ factor has been subtracted from this measure as it does not depend on the layout. The correlation between expected cache misses and actual ones is very high with a calculated r^2 of 0.98.

Notice that the layout quality is not only influenced by the edge lengths (directly linked with the number of edges cut), but also by the dispersion of cut edges. The number of cache misses is smaller than the number of edges cut by a separator if successive cut edges point toward the same memory block (Fig. 12).

6.3.3 CO Layouts on GPU

The last test of Table 2 (GpuIso) evaluates the benefits of CO layouts on a Nvidia GPU. We measure the execution time and the number of coalesced accesses. All data are stored in the global GPU memory. There is no cache mechanism involved. The only block-based data transfer that occurs is related to coalesced parallel memory accesses. The concurrent global memory access performed by all threads of a half-warp (16 threads) is coalesced into a single memory block transfer as soon as the data accessed lie in the same 128 Bytes segment for 32, 64, and 128 bit data. The context is very different from cache based CPUs. We only have a single small block $M = B = 128$ Bytes. CO layouts lead to speed-ups ranging from 1.52 to 4.09, which is significant knowing that only the layout is modified. It shows they efficiently minimize the edge lengths even for very small sizes (128 Bytes). OpenCCL slightly outperforms the FastCOL layout. The geometric layout suffers from too long edges.

5. The other meshes produce similar graphs.

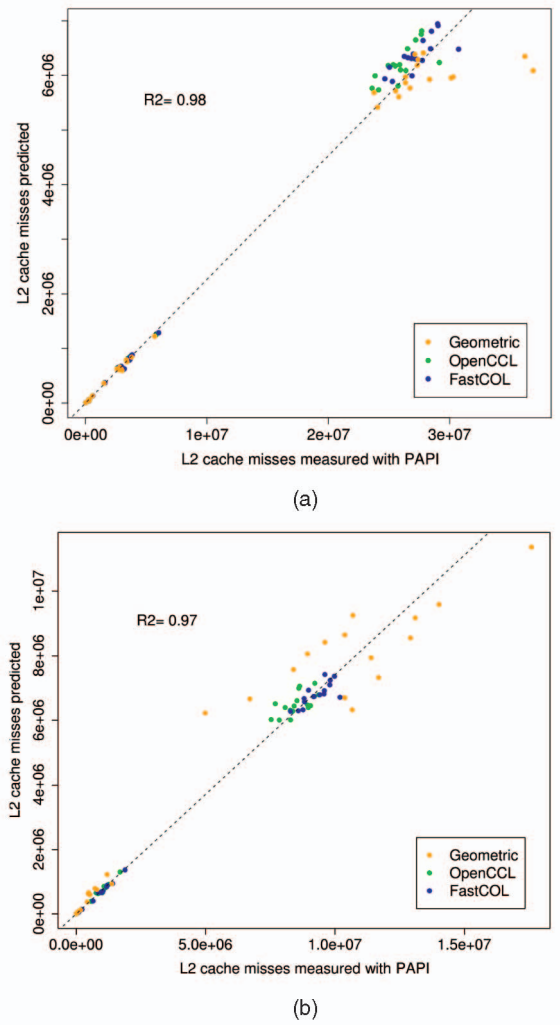

Fig. 11. Correlation between edge lengths and measured L2 cache-misses on CpuIso. Each point corresponds to a mesh with geometric, OpenCCL or FastCOL layouts. (a) Core2 and (b) Opteron.

Various applications can share work between the CPU and the GPU. The same CO layout can thus be shared between the CPU and the GPU to reduce both cache-misses and noncoalesced accesses.

6.3.4 Layout Consistent Min-Max Tree

In all tests, the OpenCCL and FastCOL layouts show similar results. However, the FastCOL layout is computed from a BSP tree that can be used as an internal, layout consistent, acceleration data structure to further take advantage of this layout. Experiments of Table 3 reveal that using it as a min-max tree for accelerating an isosurface extraction brings significant additional performance improvements. Compared to OpenCCL or the FastCOL layout that both use an external min-max kd-tree, the min-max BSP tree provides a performance improvement of 11 percent to 55 percent on CPU and of 11 percent on GPU. The leaves to be processed being M-size leaves of the BSP tree, they are less likely to trigger cache-misses than the leaves of the kd-tree computed independently from the layout. The speedup is smaller on the GPU because we could not use the biggest meshes (50 M cells) due to memory constraints.

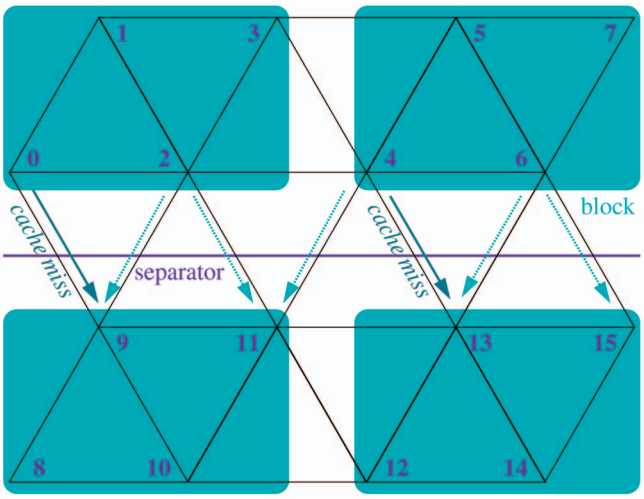

Fig. 12. Mesh layout using the solid purple line as a separator. We assume that neighbor points are needed to process each mesh point (e.g., the VTK gradient filter). Even if the separator cuts seven edges of the access graph, only two induce a cache miss if the cache can hold at least three blocks (in light green).

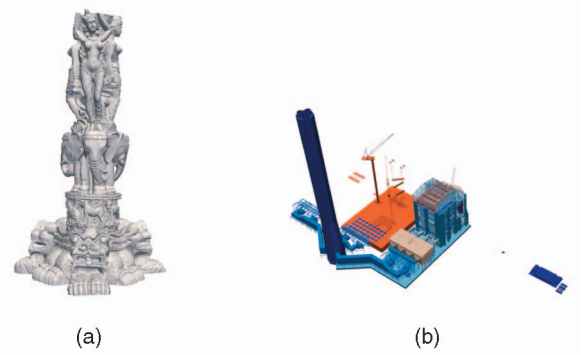

Fig. 13. Two triangle meshes. (a) Thai statue and (b) UNC Powerplant.

6.3.5 Comparison on a Scanned Model

We compare OpenCCL and FastCOL on the Thai statue.[6] This is a triangle mesh with 5 M vertices and 10 M triangles (Fig. 13a). To build the layout, OpenCCL needs 912 s and FastCOL 311 s, which is comparable with the tetrahedral meshes. We compared these two layouts on two VTK filters, the connectivity filter previously used and the depth sort filter that sorts triangles with respect to a view direction. We cannot use all previous filters as they require a tetrahedral mesh. On these two filters, the performances of both layouts are comparable, between 10 and 20 percent faster than the original layout, OpenCCL being slightly better (Table 4).

6.3.6 Comparison on a CAD Mesh

We now compare OpenCCL and FastCOL on the UNC Powerplant mesh.[7] This is a triangle mesh with 12.7 M triangles and 11 M points and a complex geometry and topology (Fig. 13b). It consists of several totally disconnected parts (1,083,733). We reorder each of those parts independently with OpenCCL and FastCOL. OpenCCL reorder points and then use min-vertex to find the cell

6. Available at http://graphics.stanford.edu/data/3Dscanrep/.
7. Available at http://www.cs.unc.edu/geom/Powerplant/.

TABLE 4
Comparison of OpenCCL and FastCOL on Two Triangle Meshes (Thai Statue and UNC Powerplant) with the VTK Connectivity Filter and the VTK Depth Sort Filter (Median of 30 Runs on Core2)

Mesh	Filter	OpenCCL		FastCOL	
		Speedup	L2	Speedup	L2
Thai Statue	Connectivity	1.09	1.26	1.08	1.23
	Depth Sort	1.19	1.40	1.15	1.31
Power plant	Connectivity	0.84	0.71	0.89	0.90
	Depth Sort	0.88	0.76	0.94	0.92

order. For the whole mesh, OpenCCL needs 671 s and FastCOL only 223 s.

We again compare those two layouts on the connectivity filter and the depth sort filter. Both layouts perform worse than the original that is already well optimized (Table 4). In the original layout, each connected part is stored contiguously and each of those parts is then well organized. No big improvement was expected due to this already good layout (Fig. 14). Previous work on this mesh led to improvements over the original layout using experiments much more based on the geometry than the filters we used: view dependent rendering in [19] and collision detection in [20].

6.3.7 Comparison with a Space Filling Curve Approach

We now compare our layout to a space filling curve approach. We use the Z-curve as in [2]. To compute the layout efficiently, we decompose the space using a kd-tree[8] until there is only one point for each leaf and then order the leaves in the order of the Z-curve. This algorithm is very similar to FastCOL except that, instead of looking for an efficient separator at each step of the recursion, we use planes parallel to the x-, y-, z-axes cutting exactly in half the set of points.

The space-filling curve approach is faster but does not take into account the topology of the mesh. The kd-tree does not provide an upper bound on the number of cells cut by the plane separator. However, this approach performs almost as well as FastCOL and OpenCCL on most of our meshes and very well on regular meshes.

As the space-filling curve does not take into account the topology of the mesh, it can perform badly on specific meshes. We created a mesh with a high density of points and cells where the kd-tree cut the mesh. To do so, we first generated a set of points in $[-1,1]^3$ with a high density around the planes $x=0$, $y=0$, $z=0$, $x=\pm 0.5$, $y=\pm 0.5$, $z=\pm 0.5$. We tetrahedralized them with tetgen. We mapped the scalar field of one of our meshes using a linear interpolation. On CpuIso the FastCOL layout is about 1.4 time faster than the space-filling curve layout.

7 Conclusion

We introduced FastCOL, an algorithm relying on Miller et al. [6] geometric separator for computing CO mesh layouts. To our knowledge, this is the first CO layout algorithm for unstructured meshes with a guaranteed theoretical upper bound of $N/B + O(N/M^{1/d})$ cache-misses.

8. We used the VTK implementation of the kd-tree.

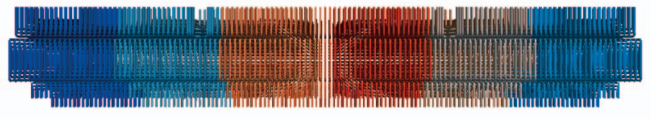

Fig. 14. Visual illustration of the original cell layout of the section 01 part a of the UNC powerplant model. Successive cells in memory are colored from blue to red.

Experiments show that this algorithm requires significantly less computation time and memory than OpenCCL, the best known CO mesh layout algorithm [5]. Without modifying the visualization algorithms, both CO layouts can bring comparable performance improvements on CPUs where they reduce the number of cache-misses, as well as on GPU architectures where they favor parallel coalesced data accesses. FastCOL improves its performance by more than 10 percent when using the layout consistent BSP tree produced by the algorithm as an acceleration data structure instead of an external one.

Acknowledgments

This work is partly funded by CEA/DIF/DSSI, Bruyères le Châtel, France, and by the Agence Nationale de la Recherche contract ANR-07-CIS7-003.

References

[1] NVIDIA, "Nvidia Cuda Programming Guide 2.3.1," 2009.
[2] V. Pascucci and R. Frank, "Global Static Indexing for Real-Time Exploration of Very Large Regular Grids," *Proc. Supercomputing*, p. 45, 2001.
[3] A. Aggarwal and J.S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Comm. ACM*, vol. 31, no. 9, p. 1116, 1988.
[4] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *Proc. Ann. Symp. Foundations of Computer Science (FOCS '99)*, p. 285, 1999.
[5] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-Oblivious Mesh Layouts," *Proc. ACM SIGGRAPH*, p. 886, 2005.
[6] G. Miller, S.-H. Teng, W. Thurston, and S. Vavasis, "Geometric Separators for Finite-Element Meshes," *J. Scientific Computing*, vol. 19, no. 2, pp. 364-386, 1998.
[7] *Algorithms for Memory Hierarchies, Advanced Lectures*, U. Meyer, P. Sanders, and J. Sibeyn, eds., Springer, 2003.
[8] R.C. Whaley and A. Petitet, "Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101-121, 2005.
[9] M. Bader and C. Zenger, "Cache Oblivious Matrix Multiplication Using an Element Ordering Based on a Peano Curve," *Linear Algebra and Its Applications*, vol. 417, nos. 2-3, p. 301, 2006.
[10] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An Experimental Comparison of Cache-Oblivious and Cache-Conscious Programs," *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA '07)*, pp. 93-104, 2007.
[11] H. Hoppe, "Optimization of Mesh Locality for Transparent Vertex Caching," *Proc. ACM SIGGRAPH*, pp. 269-276, 1999.
[12] A. Bogomjakov and C. Gotsman, "Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes," *Proc. Graphics Interface (GRIN '01)*, pp. 81-90, 2001.
[13] G. Lin and T.P.-Y. Yu, "An Improved Vertex Caching Scheme for 3D Mesh Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 12, no. 4, pp. 640-648, July/Aug. 2006.
[14] P. Sander, D. Nehab, and J. Barczak, "Fast Triangle Reordering for Vertex Locality and Reduced Overdraw," *ACM Trans. Graphics*, vol. 26, no. 3, 2007.

[15] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajarola, "Single-Strips for Fast Interactive Rendering," *The Visual Computer: Int'l J. Computer Graphics,* vol. 22, no. 6, pp. 372-386, 2006.
[16] J. Chhugani and S. Kumar, "Geometry Engine Optimization: Cache Friendly Compressed Representation of Geometry," *Proc. Symp. Interactive 3D Graphics and Games (I3D '07),* pp. 9-16, 2007.
[17] M. Isenburg and P. Lindstrom, "Streaming Meshes," *Proc. Visualization Conf.,* pp. 231-238, 2005.
[18] "OpenCCL: Cache-Coherent Layouts," http://www.cs.unc.edu/geom/COL/OpenCCL/, 2009.
[19] S.-E. Yoon and P. Lindstrom, "Mesh Layouts for Block-Based Caches," *IEEE Trans. Visualization and Computer Graphics,* vol. 12, no. 5, pp. 1213-1220, Sept./Oct. 2006.
[20] S.-E. Yoon and D. Manocha, "Cache-Efficient Layouts of Bounding Volume Hierarchies," *Computer Graphics Forum,* vol. 25, no. 3, pp. 507-516, 2006.
[21] J. Wilhelms and A. Van Gelder, "Octrees for Faster Isosurface Generation," *ACM Trans. Graphics,* vol. 11, no. 3, pp. 201-227, 1992.
[22] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Optimal Isosurface Extraction from Irregular Volume Data," *Proc. Symp. Volume Visualization (VVS '96),* pp. 31-38, 1996.
[23] Y.-J. Chiang and C. Silva, "I/O Optimal Isosurface Extraction," *Proc. Visualization Conf.,* p. 293, 1997.
[24] Y.-J. Chiang, C. Silva, and W. Schroeder, "Interactive Out-of-Core Isosurface Extraction," *Proc. Visualization Conf.,* pp. 167-174, 1998.
[25] Y.-J. Chiang and C. Silva, "External Memory Techniques for Isosurface Extraction in Scientific Visualization," *Proc. External Memory Algorithms Conf.,* pp. 247-277, 1999.
[26] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics,* 3rd ed., Kitware Inc., 2004.
[27] K. Clarkson, D. Eppstein, G. Miller, C. Sturtivant, and S.-H. Teng, "Approximating Center Points with Iterated Radon Points," *Proc. Ann. Symp. Computational Geometry (SoCG '93),* pp. 91-98, 1993.
[28] M. Tchiboukdjian, V. Danjean, and B. Raffin, "Binary Mesh Partitioning for Cache-Efficient Processing," INRIA, technical report, 2009.
[29] P. Bunyk, A. Kaufman, and C. Silva, "Simple, Fast, and Robust Ray Casting of Irregular Grids," *Proc. Visualization Conf.,* pp. 30-36, 1997.
[30] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Proc. ACM SIGGRAPH,* vol. 24, no. 5, p. 63, 1990.
[31] S. Callahan, M. Ikits, J. Comba, and C. Silva, "Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering," *IEEE Trans. Visualization and Computer Graphics,* vol. 11, no. 3, pp. 285-295, May-June 2005.
[32] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *The Int'l J. High Performance Computing Applications,* vol. 14, pp. 189-204, 2000.

Marc Tchiboukdjian received an engineering degree in computer science at ENSIMAG in France in 2007. He is currently working toward the PhD degree in the MOAIS team of the Laboratoire d'Informatique de Grenoble where he is studying efficient scientific visualization algorithms. His research interests include scientific visualization, cache-aware and cache-oblivious algorithms, parallel computing, and gpu programming.

Vincent Danjean received the PhD degree on parallel computing from the École normale supérieure de Lyon in 2004. He is a research scientist at University Joseph-Fourier at Grenoble. After a one year postdoc in French CEA institute working on large parallel computers, he has been hired in the MOAIS INRIA team to work on middleware for large and efficient parallel computing. Today, his research interests include high performance parallel computing on large scale machine, multicore machines, and on embedded hardware such as GPU. He develops the KAAPI software that wins several times the GRIDS@WORK international challenge, being able to deploy and efficiently run applications on several thousands of cores.

Bruno Raffin received the PhD degree on parallel computing from the Université d'Orléans in 1997. He is a research scientist at INRIA Rhône-Alpes Grenoble. After a 2 year postdoc in USA working on large parallel computers, he returned to France to work on the association of virtual reality, scientific visualization, and parallel computing. Today his research interests include high performance interactive computing. He develops the FlowVR software suite and manages the real-time multi-camera 3D modeling platform called Grimage. He has co-chaired the 2004 and 2006 Eurographics Symposium on Parallel Graphics and Visualization and participated to the program committee of several international conferences.