A Fast Cache-Oblivious Mesh Layout with Theoretical Guarantees

Marc Tchiboukdjian, Vincent Danjean, Bruno Raffin

A Fast Cache Oblivious Mesh Layout with Theoretical Guarantees

Marc Tchiboukdjian *
Grenoble Universités
CNRS - CEA/DIF/DSSI

Vincent Danjean †
Grenoble Universités

Bruno Raffin ‡
INRIA

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. However, these algorithms are based on heuristics. We propose in this paper a new CO algorithm for meshes that has both a low theoretical complexity and proven quality. We guarantee that a coherent traversal of an $O(N)$-size mesh in dimension d will induce less than $O(N/B+N/M^{1/d})$ cache misses where B and M are the block size and the cache size. We compare our layout with previous ones on several 3D meshes.

Keywords: Visualization of large data sets ; Cache-aware ; Cache-oblivious ; Mesh layouts ; Heterogeneous platforms.

1 Introduction

As the size of data generated by massively parallel simulations reaches terabyte scale, interactive scientific visualization, even on clusters, becomes challenging. One important bottleneck when visualizing large data sets is the data transfer between processor and memory. This issue tends to worsen with today multi-core processors where cache and bandwidth are shared among cores.

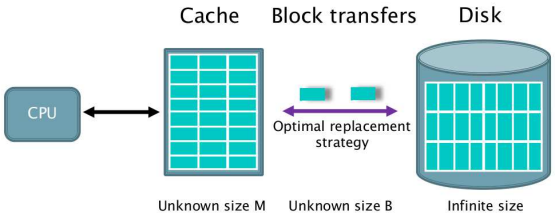

Figure 1: The cache-oblivious memory model

Cache-aware (CA) and cache-oblivious (CO) algorithms take into consideration the memory hierarchy to design cache efficient algorithms. CA algorithms are based on the external-memory (EM) model [1]. The memory hierarchy consists of two levels, a main memory of size M called cache and an infinite size secondary memory called disk. Data is transferred between these two levels in blocks of B consecutive elements. The I/O complexity of an algorithm is Q the number of blocks transfers. The CO memory model [6] is essentially the EM model where cache and block sizes are unknown (Fig 1). CO approaches have the advantage to adapt to unknown and varying memory hierarchies. This is well adapted to data visualization applications that must face complex memory hierarchies: simulation and rendering usually take place on different machines with different memory hierarchies, and platforms can be heterogeneous, mixing multi-core CPUs and GPUs. In this field the most significant and recent work is probably the CO mesh layout proposed by Yoon et Al. [18]. Performance is not theoretically guaranteed, but experiments show a significant performance improvement compared to other approaches.

As memory is accessed by blocks on modern com-

*Marc.Tchiboukdjian@imag.fr
†Vincent.Danjean@imag.fr
‡Bruno.Raffin@imag.fr

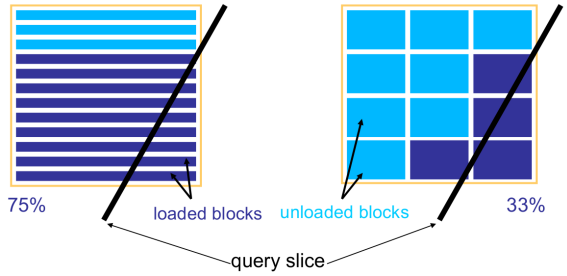

Figure 2: Good layouts can significantly reduce the number of block transfers. On the left hand side, 75% of the data must be loaded to access the query slice (each line corresponds to a cache line), while the layout used on the right hand side enables to reduce this amount to only 33% of the data (each block fits into a cache line).

puters, elements accessed sequentially during computations should be stored nearby in memory to increase cache performance (Fig. 2). In this article we propose a CO data layout for meshes that is cache efficient for access patterns showing a good spatial locality. An assumption on the access pattern is mandatory to build an efficient layout: no layout can provide a performance gain for a random access pattern.

Given a mesh, we model the access pattern as a graph where each node represents a data (a vertex, a triangle or a tetrahedron for instance) and each edge connects data that are likely to be accessed sequentially. The goal is to compute a mesh layout, i.e. a permutation of the mesh data, that minimizes the number of cache misses in the CO model when we access all the mesh respecting the spatial locality defined by the graph.

In the general case where there is no restriction on the graph, the problem is very likely to be NP-complete. Even deciding if we can have a layout with less than N/B cache misses is NP-complete (it is the bandwidth reduction problem).

In this article, we propose an algorithm to compute a CO mesh layout with a proven quality. We prove that a coherent traversal of a $O(N)$-size mesh in dimension d leads to less than $O(N/B+N/M^{1/d})$ cache misses in the CO model whatever B, the block size, and M, the cache size, are. The cost of running this algorithm on a mesh is also reduced ($O(N \log N)$) and the amount of memory allocated is about the memory size required to store the mesh.

We tested the algorithm with several 3D meshes. We visualize the mesh using VTK. Once the mesh layout is computed, it is stored as a VTK unstructured mesh. No modification of the VTK visualization pipe-line is required. We compare the performance results with the one obtained using the original layout and using the layout computed from the CO algorithm, named OpenCCL, developed by Yoon et Al. [18]. Both CO layouts significantly outperform the original layout for unstructured meshes. The two CO layouts show similar performance when visualizing the data, except that our algorithm computes the layout significantly faster and with less memory.

Related works are discussed in section 2. The algorithm is presented and analyzed in section 3, before to introduce experimental results in section 4.

2 Related Work

Cache efficient algorithms and layout for structured meshes. There is a large body of work on CA and CO algorithms for regular data structures, see for example the surveys [15] and [2]. For a focus on scientific visualization and computer graphics see [14]. The area where we can observe the biggest improvement over standard (non cache efficient) counterpart is dense linear algebra. The basic technique used by BLAS libraries (e.g. for dense matrix multiplication) is blocking [16]. A CO alternative to blocking (where cache sizes are needed) is space filling curves [3]. Layout for regular meshes uses similar techniques as dense linear algebra: blocking and space filling curves [12].

Layout for unstructured meshes. The first work in the area of unstructured meshes is [8] which assumes a FIFO cache and known cache parameters. [4] extends this work, still requiring a FIFO cache but without knowing the cache parameters. [10, 13] give fast greedy algorithms but requires the cache parameters and policy. [18, 17] do not assume any particular cache policy and developed both CA and CO approaches. The 3D mesh layout algorithms of [18, 17] significantly improve performance of previous approaches. However, these algorithms are based on heuristics that do not guarantee any

bound on the time needed to generate the layout nor on the quality obtained (number of cache misses and processing time for rendering). They are both based on a multi-level optimization scheme which gives relatively slow algorithms.

3 Cache Oblivious Layout

Our goal is to compute a layout for meshes that shows a good behavior in the CO memory model. The access pattern on the mesh data is modeled by a graph where each vertex represents a data and each edge connects data that are likely to be accessed sequentially. Our algorithm require the graph to be an overlap graph, which is define in the following. It included most of spatially coherent access patterns on meshes. For instance an important example of overlap graphs is the graph whose vertices are mesh nodes and edges connect vertices sharing a face in the mesh.

Starting from the graph of sequential accesses, we compute the CO layout by recursively splitting the graph in 2 parts. The layout is obtained by storing sequentially the leaves resulting from the recursive graph splitting. The intuitive idea is that each recursive level corresponds to a block size. The initial level correspond to a block size able to store the full graph. The second level corresponds to a block size able to store half the graph, etc. Thus, at visualization time, there will always be a level that will correspond to the actual block sizes of the machine.

The separator algorithm we use was developed by [11]. It splits the graph in 2 parts of about the same size while attempting to minimize the number of edges cut. We will later see that the number of edges cut is closely related to the number of cache misses induced by a mesh traversal. Notice that we are not limited to 3D meshes. It can apply to any d-dimensional mesh.

3.1 Overlap graphs

Overlap graphs is the class of graphs embedded in a fixed dimension that have a small separator, which is a relatively small subset of edges whose removal divides the rest of the graph into two disconnected pieces of approximately equal size. By taking advantage of the geometric structure, partitioning can be performed efficiently. Computational meshes are often composed of elements that are well-shaped in some sense, such as having bounded aspect ratio or having angles that are not too small or too large. Overlap graphs model this kind of geometric constraint.

An overlap graph starts with a neighborhood system, which is a set of closed balls in $\mathbb{R}^d$ that restrains how they can intersect.

Definition 1 (Neighborhood system [11]) *A neighborhood system in d dimensions is a set $B_1, \ldots, B_n$ of closed balls in $\mathbb{R}^d$, such that no point in $\mathbb{R}^d$ is strictly interior to more than one ball.*

A neighborhood system and the parameter α define an overlap graph where there is one vertex per ball. An edge joins two vertices if expanding the smaller of their two balls by a factor of α would make them intersect.

Definition 2 (Overlap graph [11]) *Let $\alpha \geq 1$ and let $B_1, \ldots, B_n$ be a neighborhood system. The α-overlap graph for the neighborhood system is the graph with vertex set $\{1, \ldots, n\}$ and edge set*

$$\{(i,j) \quad | \quad B_i \cap (\alpha.B_j) \neq \emptyset \text{ and } B_j \cap (\alpha.B_i) \neq \emptyset\}.$$

Overlap graphs are good models of computational meshes because every mesh of bounded aspect- ratio elements in two or three dimensions is contained in some overlap graph (for a good choice of the parameter α) [11].

The separator algorithm we detail in next section is based on the following theorem:

Theorem 3 (Geometric separator [11]) *Let G be an n-vertex α-overlap graph in d dimensions. Then the vertices of G can be partitioned 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 $O\left(\alpha n^{1-1/d}\right)$. Such a separator can be computed with high probability by a randomized linear time algorithm.*

Such a separator is optimal for this class of graphs. Indeed we cannot find a smaller separator for a regular d dimensional grid.

3.2 Geometric separator algorithm

The geometric separator algorithm (Algo. 1) from [11] starts by randomly sampling a constant number of points from the input mesh and project them

onto the surface of the unit sphere in $\mathbb{R}^{d+1}$. Then it finds a centerpoint of this random sample in linear time relative to the sample size. A point is a centerpoint if every hyperplane passing through it divides the sample set approximately evenly, at most in the ratio $d+1:1$. With good probability, this centerpoint is a centerpoint of the original set of points [5]. Finally we randomly choose a hyperplane through the centerpoint and expect that it will not cut too many edges and will be a good separator.

Algorithm 1: Geometric separator algorithm

Input: Mesh $M = (\text{Nodes } N, \text{Edges } E)$
Output: A separator ϕ

1 **repeat** n_c times
2 $N_s \leftarrow$ sample of $(d+3)^4$ points of N
3 $P \leftarrow$ project N_s to the unit sphere in $\mathbb{R}^{d+1}$
4 $c \leftarrow$ centerpoint of P
5 $(\mathbf{u}, r) \leftarrow$ rotation and scaling factor to move c at the origin
6 **repeat** n_h times
7 $\mathbf{n} \leftarrow$ random normal vector
8 $\phi \leftarrow$ separator defined by $(\mathbf{u}, r, \mathbf{n})$
9 compute the number of edges cut by ϕ
10 **end**
11 **end**
12 **return** the best ϕ

The execution time is dominated by the quality evaluation of the separator (Algo. 1 line 8). Both the time complexity and the I/O complexity of this quality evaluation are linear in the number of edges. This is the non coherent accesses of the nodes through the edges that induce the high I/O complexity. The remaining steps of the algorithm only lead to a time complexity linear in n_c and s the sample size and very small I/O complexity ($Q(s) = O(n_c s/B)$).

The quality evaluation could be improved using a random sample on the edges or a special data structure to store the edges taking into account some geometrical characteristics of the data.

3.3 Layout algorithm

The layout algorithm (Algo. 2) is the recursive application of the separator algorithm. It leads to a complete partition tree.

Algorithm 2: Layout algorithm

Input: Graph G
Output: Permutation of the nodes π

1 $\mathcal{T} \leftarrow$ complete partition tree of the graph G
2 $\pi \leftarrow$ left to right order of the leaves of $\mathcal{T}$
3 **return** π

Definition 4 (Partition tree) *The m-partition tree $\mathcal{T}$ of a graph G is a binary tree with G at the root and the children of a node H are the subgraphs H_0 and H_1 obtained by separating H using the separator algorithm. We apply this recursive scheme until each leaf contains less than m nodes. A complete partition tree is a partition tree with 1-node leaves.*

Its complexity directly derives from the recursive calls of the separator algorithm.

Theorem 5 (Layout computation) *The number of computational steps of the layout algorithm is $W(n) = O(n \log n)$.*

Proof. The separator algorithm 1 has linear complexity. Splitting a subgraph decreases the size of its children by at least a factor $\frac{d+1}{d+2}$. Thus the complete partition tree has depth $O(\log n)$ and the overall complexity is $O(n \log n)$. □

Different data structures can be used to store the data when computing the layout. The only requirement to obtain the claimed complexity is to have a node sampler of linear complexity and an iterator on edges of linear complexity too. The approach we adopt in this paper uses two arrays: one for the vertex coordinates and one for unordered edges (pair of vertex indices). We partition the nodes and the edges according to the computed separator. Nodes at the left of the separator are moved at the left of the array and nodes at the right of the separator are moved to the right of the array. For edges we use a 3-way quicksort approach. Inner edges of H_0 are stored at the left, inner edges of H_1 at the right and cut edges between H_0 and H_1 in the middle.

3.4 Layout quality

Before analyzing the number of block transfers related to our layout, we define the cut, inner and outer edges of a partition tree (Fig. 3).

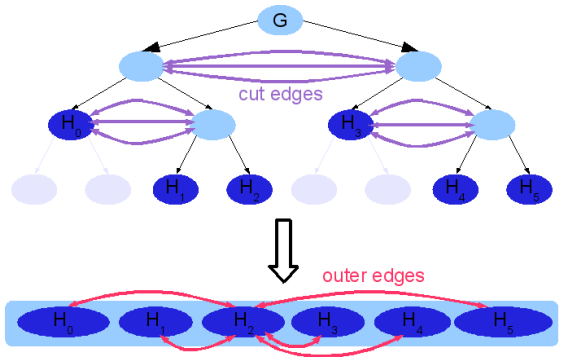

Figure 3: Cut edges of $\mathcal{T}$ and outer edges of H_2.

Definition 6 (Cut edge) *In a partition tree $\mathcal{T}$, a cut edge of a non-leaf subgraph H is an edge with one endpoint in each of the children of H.*

Definition 7 (Inner and outer edges) *In a partition tree $\mathcal{T}$, an inner (resp. outer) edge of G is an edge whose endpoints are in the same leaf subgraph (resp. different leaf subgraphs).*

We then compute an upper bound for the number of outer edges. These edges are related to the cache misses induced by a graph traversal.

Lemma 8 (Number of outer edges) *Let $\mathcal{T}$ a m-partition tree of an n-node graph. Then the total number of outer edges is $O\left(\frac{n}{m^{1/d}}\right)$.*

Proof. The number of outer edges in $\mathcal{T}$ is equal to the sum of all cut edges for all non-leaf subgraphs of $\mathcal{T}$. Let $K(r)$ be the maximum number of cut edges in a subtree $\mathcal{T}'$ of $\mathcal{T}$ rooted at a subgraph of size r. If $r \leq m$ then $\mathcal{T}'$ is just a leaf and $K(r) = 0$. If $r \geq m$ we count the cut edges of the left and right children and the cut edges of the root of $\mathcal{T}'$:

$$K(r) \leq \max_{1/2 \leq \lambda \leq \delta} [K(\lambda r) + K((1-\lambda)r)] + cr^{1-1/d}$$

By induction, we can prove that

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

as long as

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

Moreover the number of outer edges is less than $K(n)$. Thus we obtain the claimed bound. □

We now can compute the number of block transfers.

Theorem 9 (Layout quality) *The cache-oblivious layout of section 3.3 guarantees that a traversal of the graph induces less than $O(n/B + n/M^{1/d})$ block transfers.*

Proof. Let $\mathcal{T}$ the complete partition tree of G computed in algorithm 2. Let $\mathcal{T}' \subset \mathcal{T}$ the M-partition tree of G and $H_1, \ldots, H_k$ its leaves. $H_1, \ldots, H_k$ is a partition of G that is stored sequentially in memory. A traversal of G is a traversal of each of the H_i. As the size $|H_i|$ of one H_i is less than M, we can store it completely in cache for a cost of $O(|H_i|/B)$ and a traversal of the inner edges of H_i does not cause any additional block transfer. Only outer edges cause additional block transfers of 1 per edge. From lemma 8, the number of outer edges of $\mathcal{T}'$ is $O(n/M^{1/d})$. Thus the total number of block transfers needed to traverse all the graph is

$$\sum_i^k O\left(\frac{|H_i|}{B}\right) + O\left(\frac{n}{M^{1/d}}\right).$$

As $\sum_i^k |H_i| = n$, we get

$$Q(n) = O\left(\frac{n}{B} + \frac{n}{M^{1/d}}\right)$$

which complete the proof. □

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 nor a β-approximation.

4 Experiments

In this section we detail experiments run with 3 different 3D meshes: a structured mesh, an AMR mesh and an unstructured one (Table 1). For each of these meshes we compare the performance results of the original layout (as provided), the layout computed by the OpenCCL algorithm of Yoon et Al. [18], and our layout.

Experiments ran on 8 processor PC equiped with opteron 875 dual-core processors running at 2,2Ghz. Only one core was used. Each core as its own L1 and L2 caches: a 2-way associative L1 cache of 64K and a 16-way associative L2 cache of 1024K, both caches using lines of 64 bytes.

Mesh	Points	Cells	Mesh type
Plasma	274k	1,310k tetra	structured
Reactor	84k	78k hexa	AMR
Skull	37k	156k tetra	unstructured

Table 1: Characteristics of the 3 meshes tested.

	Our layout		OpenCCL	
Mesh	time	mem	time	mem
Plasma	107.4	124	282.4	6843
Reactor	8.8	15	27.6	458
Skull	10.6	16	26.9	814

Table 2: Time (in seconds) and maximal memory (in MB) for the layout computation.

We first focus on the computation of the layout and compare our algorithm with OpenCCL. Our algorithm was developed in C++ following the Matlab implementation of [7]. Its parameters were set to $n_c = 2$ and $n_h = 50$ to compute 5-partition trees (graph of less than 5 nodes are not further split). The OpenCCL algorithm was used as provided at [9]. For the 3 meshes tested our algorithm significantly outperform OpenCCL regarding the computation time and the memory used (Table 2).

Once the layouts computed, we compare the resulting performance when the meshes are read for a visualization. We use VTK 5.0.4 and measure the execution time when moving a cut plan in diagonal on the mesh from bottom left to top right. Each experiment is repeated 30 times. In all cases OpenCCL and our algorithm show very similar performance (Table 3). For the structured and AMR meshes, the CO layouts do not provide a performance improvement over the original layout. Two reasons explain this result:

- Being structured meshes, the original layouts already have a cache efficient data structure;
- Both meshes are small, making it difficult to measure any cache improvement that the CO layouts could provide.

In opposite, the CO layouts both significantly outperform the original layout for the Skull unstructured mesh. These results are confirmed when counting the number of L1 cache misses using the Cachegrind utility (Table 4).

Time (s)	Original	OpenCCL	Our layout
Plasma	32.38 (0.25)	32.49 (0.22)	32.38 (0.22)
Reactor	3.34 (0.05)	3.32 (0.1)	3.36 (0.09)
Skull	5.11 (0.04)	4.95 (0.02)	4.96 (0.02)

Table 3: Average execution time and standard deviation (in parenthesis) when visualizing the different meshes: a cut plane is moved through the whole mesh.

Mesh	Original	OpenCCL	Our Layout
Plasma	146.6	150.7	151.2
Reactor	21.5	21.8	22.0
Skull	26.5	21.7	21.8

Table 4: Number of L1 cache misses (in millions) when visualizing the different meshes.

We also compute the density of edges according to their length for each Skull layout (Fig. 5). The edge length is computed from the layout. The length of an edge is the number of nodes separating the storage location of each extremity. Large edges tend to produce more cache misses that small ones as the cache lines storing both extremities are more likely to store data required before being evicted. The edge length distribution for the original layout shows various peaks while CO layouts tend to have a smoother distribution with high density of small edges. This edge length distribution is coherent with the experienced execution times and number of cache misses.

We further investigate the effect of the cache size on the Skull mesh using Cachegrind (Fig. 4). The smaller the cache size is, the more efficient CO layouts are compared to the original layout. It confirms what we observed about edge lengths: favoring small edge lengths leads to more robust layouts.

5 Conclusion

Memory transfers are one of the main bottleneck for the visualization of large meshes. This bottleneck tends to worsen with new multi-core architectures showing complex memory hierarchies. In this context, it is important to control the mesh memory layout in an attempt to reduce the number of useless data transfers. In this paper we presented an algorithm to compute cache oblivious mesh layouts.

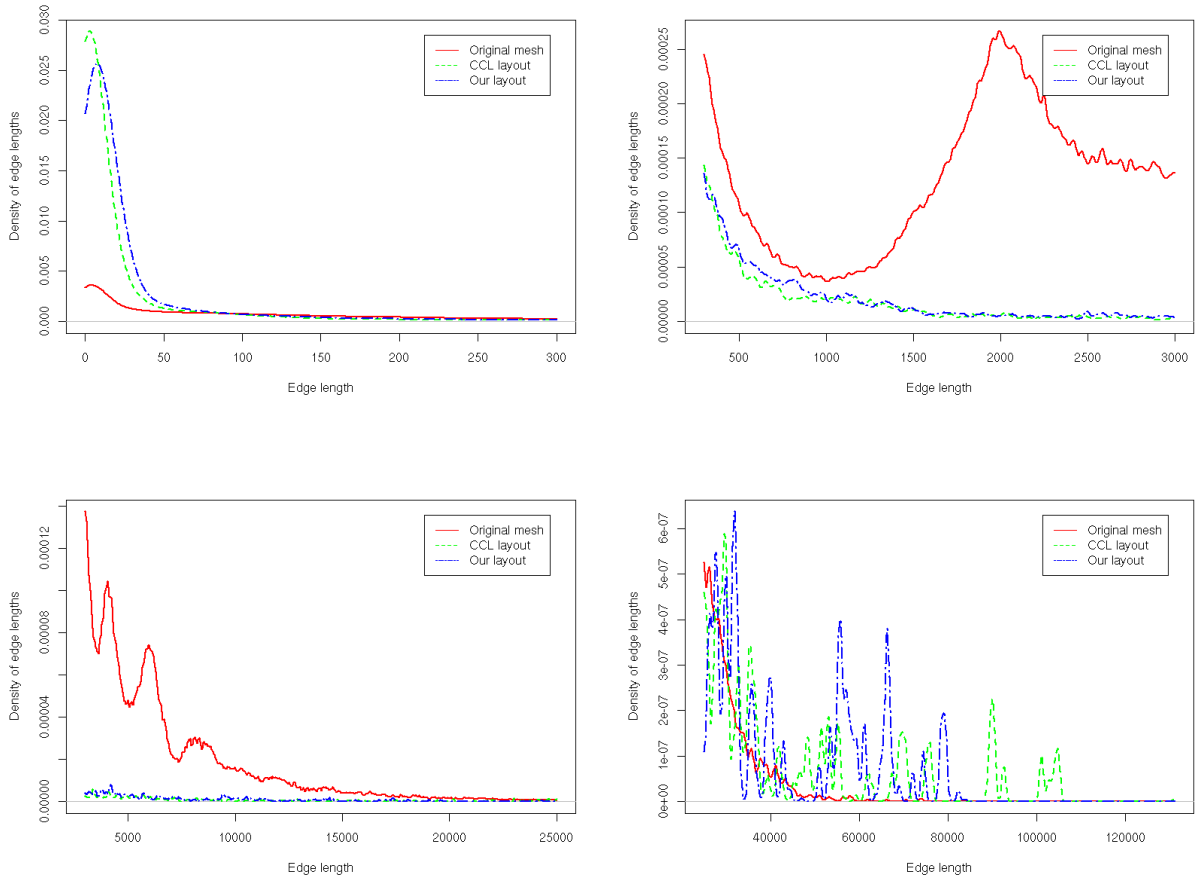

Figure 5: Density of edges depending on their size for the different Skull mesh layouts.

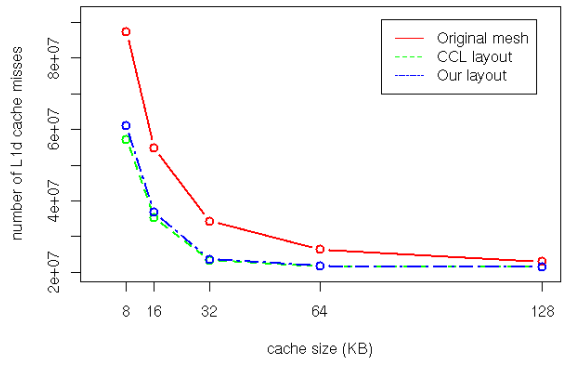

Figure 4: Number of cache misses for the Skull mesh with L1 cache size varying from 8KB to 128KB. Cachegrind was used to simulate the various cache sizes.

Cache oblivious layouts have the advantage not to depend on a given block and cache size. The layout does not have to be recomputed when changing of machine or if the cache size available dynamically changes during execution for instance.

Experiments shows that our layout performance match the layouts computed from Yoon et Al. [18] OpenCCL algorithm. But in opposite to this algorithm, our algorithm performance is theoretically guaranteed on the number of block transfers. It also significantly outperform OpenCCL regarding execution time and memory allocation when computing the layout, an important criteria as we focus on large meshes.

Future work will focus on validating our algorithm on very large meshes and a more important

variety of access patterns. We are also investigating the benefits of this algorithm on GPUs and in the context of parallel mesh visualization.

Acknowledgements

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.

Plasma [1] and Skull [2] models are provided by the AIM@SHAPE Shape Repository [3].

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. of ACM*, 31(9):1116–1127, 1988.

[2] L. Arge, G. Brodal, and R. Fagerberg. Cache oblivious data structures, 2004.

[3] M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on a peano curve. *Linear Algebra and Its Applications*, 417(2–3):301—-313, 2006.

[4] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. In *GRIN'01*, pages 81–90, 2001.

[5] K. L. Clarkson, D. Eppstein, G. L. Miller, C. Sturtivant, and S.-H. Teng. Approximating center points with iterated radon points. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, pages 91–98, 1993.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, 1999.

[7] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: implementation and experiments. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 418–427, 1995.

[8] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 269–276, 1999.

[9] OpenCCL: Cache-Coherent Layouts. http://www.cs.unc.edu/ geom/col/openccl/.

[10] G. Lin and T. P.-Y. Yu. An improved vertex caching scheme for 3d mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):640–648, 2006.

[11] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite-element meshes. *SIAM J. Sci. Comput.*, 19(2):364–386, 1998.

[12] R. Niedermeier, K. Reinhardt, and P. Sanders. Towards optimal locality in mesh-indexings. In *FCT '97: Proceedings of the 11th International Symposium on Fundamentals of Computation Theory*, pages 364–375, 1997.

[13] P. V. Sander, D. Nehab, and J. Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.*, 26(3):89, 2007.

[14] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics, 2002.

[15] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[16] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, 2005.

[17] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, 2006.

[18] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. In *ACM SIGGRAPH*, pages 886–893, 2005.

[1] courtesy of ISTI-CNR

[2] courtesy of Pierre Alliez

[3] http://shapes.aim-at-shape.net/