EXPLICIT MEMORY MANAGEMENT FOR MESH TRAVERSAL

by

Yuan Liu
B.Sc., Beijing University of Technology, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Yuan Liu 2013
SIMON FRASER UNIVERSITY
Spring 2013

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
APPROVAL

Name: Yuan Liu
Degree: Master of Science
Title of Thesis: Explicit Memory Management for Mesh Traversal

Examinig Committee: Dr. Kay Wiese
Chair

________________________________________
Dr. Hao Zhang, Senior Supervisor

________________________________________
Dr. Alexandra Fedorova, Supervisor

________________________________________
Dr. Arrvindh Shriraman, SFU Examiner

Date Approved: April 29, 2013
Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011
Abstract

Mesh traversal is a common and essential geometry processing problem in computer graphics. The traversal typically processes each face in a mesh in a systematic and consistent order for different applications such as mesh compression, rendering, curvature tracing, mesh simplification and surface smoothing. While cache-efficient mesh traversal methods where data and computations are reordered for good cache reuse have been well-studied, their runtime performances are limited by implicit (automatic) memory management. In this work we explore explicit optimizations on Explicitly Managed Memory (EMM) systems. Unlike conventional processors, EMM hardware has no automatic caching or prefetching. Instead, programmers decide what, when and where data are placed in the memory hierarchy, and then manually initiate the transfer and move the data by software. Programming such a system, however, is challenging since having programmers understand the hierarchies and manually move the data can be difficult and tedious. We address this challenge by introducing an access analyzer - distance profiler, associated with an interface construct. We demonstrate the effectiveness of the access analyzer and the runtime improvement obtained for several mesh traversal algorithms with different access patterns.

Keywords: geometry processing; mesh traversal; explicit managed memory; distance profiler
Acknowledgments

Special thanks to all those who made this thesis possible. First, I would like to express my heartfelt thanks and gratitude to my senior supervisor, Dr. Hao Zhang, and Dr. Alexandra Fedorova, my supervisor, for their patient guidance, encouragement and continuous support.

I would also like to thank my thesis examiner Dr. Arrvindh Shriramanand and chair Dr. Kay Wiese for taking time reviewing this thesis.

I would like to take this opportunity to thank my fellow students at the GrUVi lab. They have created a positive and motivating environment in the laboratory, and it is a great pleasure to work with them. I would like to thank Craig and Svetozar for their help and valuable discussions.

I would like to thank my dear friends for their companionship throughout my stay at Simon Fraser University, especially to Fei, Yu and Shaoyun. The time spent together with them is unforgettable.

Finally I would like to thank my parents for their patience and support. My deepest gratitude to them is beyond words.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Partial Copyright License</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Programs</td>
<td>xii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Related work</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Mesh traversal and access pattern</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Mesh compression and rendering</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Curvature tracing, symmetry detection and remeshing</td>
<td>9</td>
</tr>
<tr>
<td>2.1.3 Mesh streaming and out-of-core problems</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Cache-efficient algorithms and mesh layouts</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Explicit memory management</td>
<td>15</td>
</tr>
<tr>
<td>3 Background</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Mesh representation</td>
<td>17</td>
</tr>
</tbody>
</table>
3.2 Platform STHORM ............................................. 19
3.2.1 DMA overhead ........................................... 21
3.2.2 Minimal DMA transfer size ............................. 21

4 EMM optimizations ............................................. 22
4.1 EMM layouts and data structures ......................... 22
4.1.1 Grid ....................................................... 25
4.1.2 Interleaving Tiles ....................................... 27
4.1.3 Adaptive Quadtree ..................................... 28
4.2 EMM data fetching schemes ................................. 29
4.2.1 Convolution prefetching ................................. 32
4.2.2 Extrapolated prefetching ............................... 32
4.3 Mesh traversal applications ................................. 33
4.3.1 Curve tracing ............................................ 33
4.3.2 Region growing ......................................... 35
4.3.3 Random access .......................................... 36

5 Distance profiler and interface ............................. 38

6 Experiments and results .................................... 46
6.1 Experiment setup ........................................... 46
6.2 Performance metrics ...................................... 47
6.3 Results ..................................................... 47

7 Conclusion and future work ................................ 53

Bibliography ..................................................... 54
List of Tables

3.1 Memory access latency for STHORM Simulator. This table is from [53]. . . . 19

4.1 EMM strategies. The layout and the fetching scheme in a row do not suggest a binding. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5.1 Distance profile (DP) table. The DP with sudden jumps or drops indicates arbitrary accesses which have no spatial locality; the DP showing slow and linear increase or decrease implies good locality between accesses; An exponential increase in DP suggests a scanline or curve shape access pattern. A mix of any of these three gives us a hint about the changing of the programs access pattern. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
List of Figures

1.1 Illustrations of the coherence in the layout of a mesh. Left: the original layout; right: the layout after reordering the vertex and triangle arrays using spectral sequencing. The model is color-rendered based on triangle position in the array. The layout diagram connects triangles that share the same vertex with horizontal line segments (green) and vertices referenced by the same triangle with vertical line segments (gray); the farther the green and the grey line segments are from the diagonal, the less coherent is the layout. This figure is from [1].

2.1 (A) a triangle strip (B) a triangle fan (C) a generalized triangle strip. This figure is from [11].

2.2 (b) shows the vertex spanning tree of (a). Cutting through the vertex tree edges produces topological simply connected polygons (c) (d). Bounding loop (e) is the boundary of the polygon. The dual graph of the polygon is the triangle spanning tree (f). This figure is from [15].

2.3 A run of triangle conquest. The thick lines show the active lists. Conquered area is shown in dashed lines. This figure is from [16].

2.4 An illustration for layered traversal. The first three layers are generated from left to right in turn. This figure is from [20].

2.5 Lines of curvatures (following the principal directions) are traced given the direction field. This figure is from [21].

2.6 Cache-oblivious memory model. This figure is from [4].

2.7 (a) irregular space-filling curves (b) regular space-filling curves

2.8 Time skewing with a 3-point stencil. Dashed lines show the cuts. This figure is from [45].
3.1 Triangle mesh representation ........................................ 18

3.2 The left figure shows the performance of DMA, direct reads, and cache. DMA
overheads for different transfer sizes are shown on the right. This figure is
from [53]. ..................................................... 20

4.1 (a) shows a cell in Interleaving Tiles is surrounded by six other cells (b) shows
a cell in Grid is surrounded by eight. ................................. 23

4.2 Grid layout ...................................................... 25

4.3 An illustration of the triangle reindexing methodology. A triangle with ID
800 has a layout index 14812, where the first three digits 148 refers to the
cell ID and 12 implies its position in the cell. ......................... 26

4.4 Once mesh is partitioned into sub-meshes(from left to right), referencing ver-
tex via global index becomes inefficient; we address this by making references
as pointers .................................................... 26

4.5 Interleaving Tiles layout ......................................... 27

4.6 Octree layout. This figure is from [32]. ............................. 28

4.7 Triangle reindexing for Quadtree. A triangle with ID 800 has a layout index
22412, where the first three digits 224 shows the path from root to the leaf
containing that triangle and 12 implies its position in the leaf. .......... 30

4.8 Convolution prefetching. The red-color-shaded cells are the active, in-process
cells. We prefetch the yellow-color-shaded cells, (a) for Grid (b) for Interleav-
ing Tiles and (c) for Quadtree. ...................................... 31

4.9 The left figure shows the tensor field on the mesh and the right figure shows
the curves traced from the tensor field. This figure is from [37]. ...... 33

5.1 Relationship between access pattern, distance profile and EMM strategy .... 38

5.2 Z curves on different meshes have distinctly different access patterns ........ 39

5.3 Z curve and Gray code have different access pattern but if every four accesses
are grouped as one, the patterns become exact same hence should applied
with the same set of EMM optimizations. ................................. 40

5.4 Access i have good locality within its k predecessors and successors [i-k, i+k],
while the distance from the i th access to the first access for (b) is much
bigger than that for (a). .............................................. 41
5.5 A distance profile. The x-axis is the i th accesses, the y-axis shows its accumulative distance. 42
5.6 Programmer-separation interface 43

6.1 Short curves: performance of different EMM mesh layouts and fetching schemes 48
6.2 Short curves: performance of different transfer size 49
6.3 Short curves: performance comparison between direct read, cache and EMM 49
6.4 Long curves: performance of different EMM mesh layouts and fetching schemes 50
6.5 Long curves: performance comparison between direct read, cache and EMM 50
6.6 Region growing: performance of different EMM mesh layouts and fetching schemes 51
6.7 Region growing: performance of different transfer size 51
6.8 Region growing: performance comparison between direct read, cache and EMM 52
6.9 Random: performance between direct read, cache and EMM 52
List of Programs

4.1  Pseudo program for curve tracing .......................... 35
4.2  Pseudo program for region growing .......................... 37
Chapter 1

Introduction

Polygonal meshes are the most popular shape representation in computer graphics. A polyg- 
onal mesh is a collection of vertices, edges and faces that defines the shape and the topology 
of a 3D object. Traversing such a mesh is a common and crucial problem, which typically 
refers to visiting each face in the mesh in a systematic and consistent order. Mesh traversal is embedded in a series of geometry processing algorithms such as mesh compression, 
rendering, curvature tracing, mesh simplification, and surface smoothing.

While mesh traversal algorithms themselves are well-studied, they remain a challenge 
on the memory management level. More specifically, as the complexity and the size of mesh 
data sets have immensely increased in recent years, efficient algorithms for traversing huge 
meshes have been critically pursued. The efficiency issue lies in the delays in transmitting 
data between processor and memory, i.e., memory latency. The closer memory banks are 
located to processor, the lower latency they impose. Memory banks located further away 
from processor, such as main memory, are called remote memory(slow memory), while banks 
located closest to the processor such as processor cache are referred to as local memory(fast 
memory).

One solution to this issue, on hardware end, is to increase use of fast memory hierar- 
chies. On software end, enforcing coherence between mesh data layout and access pattern 
to obtain good spatial locality can also effectively improve performance. Spatial locality is a 
phenomenon when a particular memory location is referenced at a particular time, nearby 
memory locations are likely to be referenced in the near future. If we know program’s access 
pattern which describes how data items are requested by the program, we can rearrange 
data according to the access pattern to obtain good spatial locality. Good spatial locality
demonstrates to reduce cache-misses and consequently improves performance. In this thesis, access pattern refers to spatial access pattern, or 2D access pattern. For a mesh traversal application, the traversal order is seen as its access pattern.

Figure 1.1: Illustrations of the coherence in the layout of a mesh. Left: the original layout; right: the layout after reordering the vertex and triangle arrays using spectral sequencing. The model is color-rendered based on triangle position in the array. The layout diagram connects triangles that share the same vertex with horizontal line segments (green) and vertices referenced by the same triangle with vertical line segments (gray); the farther the green and the grey line segments are from the diagonal, the less coherent is the layout. This figure is from [1].

Input mesh data for geometry processing algorithms, however, are naturally unstructured, organized despite any particular pattern. As aforementioned, the lack of coherence in mesh layout will result in poor spatial locality hence more remote memory access. An illustration of the lack of coherence in the layout of the Lucy model[1] is given on the left in Figure 1.1. To solve this problem, many cache-efficient (cache-aware and cache-oblivious) approaches have been proposed, aiming at enforcing data locality through realignment of mesh data to reduce cache-misses. As hardware cache is subject to automatically (implicitly)-managed linear data fetching, where the asynchronous transfer (prefetch) can only fetch the adjacent line in 1D, we call cache-efficient methods implicit optimizations.

The realignment of mesh data refers to creating a coherent mesh layout for better cache reuse. Many layouts such as grid[2] have been examined. However, rearranging data into grid, as detailed in Section 2.2, is biased towards a particular traversal direction due to
the linear fetching of cache. Methods based on minimum linear arrangement\cite{3} are also biased towards a particular traversal direction. Yoon et al.\cite{4} proposed a method where the layout is locally optimized but it is time-consuming. In excess of the specific drawbacks or limitations, it is important to realize that even though data realignment enables data locality, prefetching data according to an application’s access pattern may not work due to cache’s automatic linear fetching behaviour. As a result, the performance of all cache-coherent approaches is constrained.

In contrast, there is an alternative where memories can be managed explicitly. Multi-core processors with explicitly managed memory (EMM) hierarchies, such as Cell and GPUs\cite{5, 6}, enable programmers to use optimal caching policies and multiple data buffers that allow overlap computation with communication\cite{7}. Unlike conventional processors, EMM hardware has no automatic caching or prefetching. Instead, data are transferred using software. EMM hierarchies let programmers decide what and when data are placed in local memory. As a result, with EMM, not only data can be reordered by locality, but data can also be moved according to application’s access pattern.

Having programmers understand the hierarchies and manage the data movement manually for a particular application, however, can be difficult and tedious. In an ideal world, no application programmer would ever have to manage the movement of data, even on EMM systems. Software should automatically arrange data closer to the processor that will need it. However, it can be difficult to perform analysis to enable enough understanding of a program’s data accesses. One approach to achieve this ideal is to develop programming models that make programmers more aware of data locality issues\cite{8, 9, 10}. However, programmers continue to face the challenge of explicitly managing the data across the various locality domains.

In this thesis we first explore EMM optimizations on three representative applications in terms of access pattern. Our goal is to orchestrate mesh layout and data movement/fetching according to the access pattern of each application in order to optimize their runtime. We present three EMM-efficient layouts: Grid, Quadtree and Interleaving tiles, all of which block spatially close data together to achieve good spatial locality. Several data fetching schemes are proposed along with each particular layout to adapt different requirements of the applications. They are compared with regard to runtime efficiency, applicability and scalability.

Furthermore, we explore the way of automatically communicating an application’s data
accesses to the underlying system such that programmers won’t need to explicitly/manually specify application’s access pattern. We propose a distance profiling approach that is able to analyze an application’s access pattern and map it to an appropriate EMM optimization strategy. The EMM system will then automatically build the mesh layout, define transfer size and orchestrate data movement according to the fetching scheme. Finally, we provide a set of interface functions which make EMM optimizations completely transparent.

The main contributions of this work include:

1. Providing several effective EMM optimizations for mesh traversal applications with different access patterns.

2. Introducing an access descriptor called distance profile that enables the system to automatically determine a right EMM strategy for an application based on its access pattern.

3. Providing library abstractions and interface functions for graphics applications with different access patterns. The interface triggers the distance profiler and data movement, and makes the EMM optimizations transparent to application programmers.

4. Appending quantifiable metrics that evaluate each individual performance factor to STHORM, a new EMM architecture, and providing insight on how to analyze memory access.

Our work provides several EMM optimization schemes tailored for three typical access patterns which cover a wide range of geometry processing applications. Thus in applications with any of the patterns, our EMM strategies would provide a better trade-off. New EMM strategies can be added in response to new application requirements, while the proposed distance profiling approach is general and is able to analyze new access patterns. With our interface, new architectures can be supported by replacing underlying implementations.

It is important to mention that our EMM layouts are based on spatial locality and are general enough such that stencil computations from diverse areas besides geometry processing, e.g., fluid dynamic, image processing and heat diffusion, would benefit from employing such layouts. Stencil computations sweep over a spatial grid, performing nearest neighbour computations using iterative finite-difference techniques.

Last but not the least, our EMM approaches would potentially work well as solutions to out-of-core problems since out-of-core algorithms aim to provide transparent access to arbitrary large dataset which can reduce costly loads of data from external memory. The portion containing the data needed for immediate computations needs to be loaded into the main memory effectively and efficiently, while our methods similarly optimize the shuffling
of data between main memory and local memory.

The remaining part of this work is organized as follows. In chapter 2, we survey some of the related researches on mesh traversal problems, cache-based mesh layout methods and explicitly managed memory. Chapter 3 describes the concepts used in this work and we will introduce the platform that carries all of our experiments. In chapter 4 we present the EMM optimizations in full detail including building the layouts, explicit memory management. Chapter 5 presents the distance profiling approach and our access-oriented interface design. Experimental results including the comparisons of runtime efficiency, applicability and scalability are then provided in chapter 6, and finally in chapter 7 we conclude and discuss the limitations of the current method and present the reader with some ideas for possible future work.
Chapter 2

Related work

In this Chapter we investigate some related previous works in three separate sections. First, we introduce several classic mesh traversal applications with varied access patterns. And then we look at the cache-efficient versions of these applications which improve performance implicitly by enforcing data locality. Finally, we survey explicit memory management techniques by which data can be rearranged by locality, and data can be moved according to access pattern as well.

2.1 Mesh traversal and access pattern

Mesh traversal algorithms have been extensively researched as a sub-problem embedded in many applications, such as mesh compression, rendering, remeshing, curvature tracing, mesh simplification, and surface smoothing. In this section we describe some classic algorithms categorized by their access patterns.

2.1.1 Mesh compression and rendering

Mesh traversal is most widely used as a substep of mesh compression that makes input mesh more compact and efficient for the subsequent processing. Typical mesh format is composed of a list of vertex coordinates and a list of polygonal faces referencing to the vertices. If a vertex participates in n triangles in the mesh, it will be processed n times when rendering the mesh, for example. By reordering the vertices and compressing the connectivity information during the traversal, the redundancy can be largely reduced.
We classify mesh compression/encoding approaches into four classes: triangle strip, spanning tree, triangle conquest, and layered decomposition.

Triangle strip methods attempt to divide a 3D mesh into long strips of triangles, and then encode these strips. Deering[12] first introduced the concept of the generalized triangular mesh: efficient linear triangle strips. A generalized triangle strip begins with an arbitrary face, and continuously chooses a direction, i.e. an edge, to extend the strip. A strip usually stops when reaching a triangle with on forward connections. An example of a generalized triangle strip is shown in Figure 2.1. Based on Deerings work, Chow [13] proposed a mesh compression scheme optimized for real-time rendering. The triangle strip representation can be applied to a triangular mesh of arbitrary topology.

Turan[14] proposed that the connectivity of a planar graph can be encoded using two spanning trees: a vertex spanning tree and a triangle spanning tree. See Figure 2.2. Taubin and Rossignac[15] presented a topological surgery approach to encode mesh connectivity based on[14]. The vertex spanning tree is a least branch spanning tree in which edges recursively propagate(depth-first search) from an arbitrary root vertex to its connecting vertex(child) and splits into branches where necessary in order to achieve minimum branching. The formation of vertex spanning tree cuts the mesh into a planar polygon of which the dual graph forms triangle spanning tree. Triangle spanning trees can be seen as a set of triangle strips connecting at a branching node while each strip serves as a branch of the triangle tree.

Triangle conquest approaches[16, 17, 18, 19] start with an arbitrary seed face and push
Figure 2.2: (b) shows the vertex spanning tree of (a). Cutting through the vertex tree edges produces topological simply connected polygons (c) (d). Bounding loop (e) is the boundary of the polygon. The dual graph of the polygon is the triangle spanning tree (f). This figure is from[15].
its three vertices to an active vertex list; the seed face and its vertices are flagged conquered. In each iteration the front vertex in the active list works as pivot which tries to conquer its one-ring free vertices in clockwise or counterclockwise order and add them to the active list (breath-first search). If v has not been conquered yet, it is now flagged conquered, and its valence is output to the code sequence; if v has already been conquered and belongs to the active list, which implies the conquest splits the unconquered region into two: a closed region and an open region, the conquest is subsequently split into two. See Figure 2.3.

Bajaj et al. [20] presented a connectivity encoding method using a layered structure of vertices where a triangular mesh is decomposed into several vertex layers and triangle layers. See Figure 2.4. The 0th vertex layer is a randomly chosen vertex of the mesh. The k th vertex layer (k > 0) includes a vertex v if v is not included in any previous vertex layer and there exists an edge e = (v, v’) where v’ is included in the (k-1)th layer (breath-first search). The k th triangle layer includes a triangle t if t has at least one vertex in k th vertex layer and t is not included in any previous triangle layer.

It’s not hard to see that the 1D access pattern of all the applications in this section is triangle strips even though how a strip proceeds may differ. If we look at the access patterns in 2D, the traversals all grow ring by ring (strip by strip) which are spatially close (connecting) to each other. We call this type of access pattern region growing. Region growing reveals good spatial locality as vertices or triangles sharing edges are likely to be accessed consecutively.

2.1.2 Curvature tracing, symmetry detection and remeshing

Curvature tracing is another important problem that can be generalized as a linear mesh traversal problem in the context of shape analysis, symmetry detection and remeshing.

In [21], Alliez et al. approximated curvature tensor field and computed integrated principal curvature lines along the principal directions as illustrated in Figure 2.5. A sequence of adjacent triangles and their vertices is consecutively processed in one curvature line tracing. Mitra et al. [22] employed Alliezs algorithm in their work of symmetry detection and improved it by adding noise avoiding scheme. In [23] and [24], streamlines or wrinkles are traced as piecewise linear curves sampled from the vector field over the underlying surface by integrating the 2D ordinary differential equation.

1D patterns from curvature tracing are also triangle strips. However, instead of growing a region ring by ring, the 2D pattern is more like straight lines instead of circles. In addition,
Figure 2.3: A run of triangle conquest. The thick lines show the active lists. Conquered area is shown in dashed lines. This figure is from [16].
CHAPTER 2. RELATED WORK

Figure 2.4: An illustration for layered traversal. The first three layers are generated from left to right in turn. This figure is from [20].

Figure 2.5: Lines of curvatures (following the principal directions) are traced given the direction field. This figure is from [21].
the lines can scatter across the mesh, and the end vertex of a line is not necessarily connecting to the beginning vertex of the next line by an edge.

2.1.3 Mesh streaming and out-of-core problems

Mesh streaming is another use case of mesh traversal. Mesh streaming is designed for out-of-core problems when large meshes cannot be entirely loaded into memory. Isenburg and Lindstrom[1] proposed the first streaming format where triangles and vertices are reordered in four ways to compare performance: spectral sequencing of the dual graph, spatial sorting along the longest bounding box extent, breadth-first topological sorting, and depth-first sorting. The latter two use the same BFS and DFS methodologies described in Section 2.1.1.

There are more out-of-core algorithms designed for surface reconstruction[25], Delaunay triangulation[26], surface simplification[27, 32], and mesh visualization and rendering[28, 29, 30, 31] in particular. The mesh traversal embedded in them can be categorized into single-resolution approaches, such as the four from[1], and multi-resolution approaches which add layered index structure that coarsen the mesh level by level to the original data, such as the bounding volume hierarchies (BVH) trees from[33] and octrees[32].

A traversal approach determines a particular access pattern and subsequently implies how data locality should be managed, in line with ideal data movement, to optimize performance. We describe the existing optimizations in the following two sections.

2.2 Cache-efficient algorithms and mesh layouts

Since mesh traversal applications exhibit certain access patterns, many algorithms have developed their cache-efficient(cache-aware or cache-oblivious) versions where computations and data are reordered for an efficient cache use. A cache-oblivious algorithm is an algorithm in which data layout is independent from cache size, while a cache-aware algorithm tunes data layout based on cache size. Figure 2.6 illustrates the definitions of cache-aware and cache-oblivious. A survey by Tchiboukdjian et. al. [34] discussed different cache-efficient mesh layouts in graphics and geometry processing.

A widely used technique for cache-efficient algorithms is blocking or tiling: elements are mapped in memory and accessed by blocks of size B to fit in a cache line. As mentioned in
CHAPTER 2. RELATED WORK

Figure 2.6: Cache-oblivious memory model. This figure is from[4].

Figure 2.7: (a)irregular space-filling curves (b)regular space-filling curves
Chapter 1, grid[2] works fairly well for independent individual computations or any other algorithms that base their memory access patterns on regular ordering space filling curves[35]. Regular ordering space filling curves are illustrated in Figure 2.7. When it comes to mesh traversal problems, however, grid may no longer be valid as a solution because evidently the access patterns become diverse and more complex than regular space filling curves. Access patterns like irregular strips and spirals, as described in Section 2.1.1, are more generally being seen in mesh traversal algorithms. Taking triangle strips[12] for example, neighbour triangles or vertices sharing an edge are visited contiguously without necessarily having a width and the path can be random. With cache, enforcing a grid layout may lead to a worst case if the sequential accesses happen to be orthogonal to the data layout since cache linearly fetches data to local memory.

Methods based on minimum linear arrangement[3] are also biased towards a particular access direction. S.-E. Yoon et. al. [4] proposed a method where the layout is locally optimized. OpenCCL[36] presented in [4] casts the mesh layout problem as a graph optimization problem. To describe the access pattern of the application, the user must provide a graph where vertices represent data and edges link data that are likely to be accessed in sequence at runtime. They developed a local metric to decide if a swap of some vertices improves the layout.

Past works[37, 39, 40, 41] reorder triangles to form triangle strips. They assume the cache holds a FIFO queue for vertex reuse and the cache size is known to the algorithm. Bogomjakov and Gotsman[38] introduced a universal rendering sequence algorithm based on space-filling curves that work for any cache sizes. In[42], the authors proposed a cache-aware mesh compression scheme that decomposes the mesh into parallel vertex chains using BFS and then cuts each chain into cache-sized subsets.

Another approach[33] optimizes mesh layout with the help of bounding volume hierarchies (BVH) trees. To generate an efficient layout, they use the OpenCCL algorithm and provide two types of links in the access graph: links representing spatial locality in the mesh and links representing parent-child locality in the BVH tree. We employ the tree idea but with EMM in this work.

By reordering mesh data in compliance with spatial locality, cache reuse is improved. However, the performance is still limited since cache linearly prefetches the next line in memory (1D) which is not necessarily the soon-to-be-accessed data in 2D. Thus, it is ideal if data can be prefetched in a 2D manner - this is fulfilled by EMM systems which are
described in the next section.

2.3 Explicit memory management

Managing the memory hierarchy introduces trade-offs in terms of performance, code complexity, and optimization effort. While multi-core processors based on coherent hardware-managed caches provide the abstraction of a single shared address space which is convenient for parallel programming, they only allow programmers to improve locality implicitly through reordering of instructions or realignment of data in main memory. In contrast to hardware managed caches, software-managed local memories introduce per-core, disjoint address spaces where it is the software's responsibility to manage data, i.e. explicit management. Programmers can explicitly manage locality and decide what and when data is placed in local memories, what data gets replaced and what the data layout is in local memories, which can differ from the layout of data in off-chip DRAM.

Multi-core processors with explicitly managed memory, such as Cell and GPUs[5, 6], originated in the computing domains of games and graphics and have recently emerged as general-purpose high-performance computing platforms. These processors have data-parallel components as accelerators which have high on-chip bandwidth, many scalar or SIMD cores, and explicit data transfers between fast local memories and external DRAM. More and more application benchmarks are under active research for their EMM version of optimizations.

Schneider et al. [43] compared two EMM models: Cellgen and Sequoia[44], both of which expose the data locality problem of applications to a programmer. By annotating the data referenced in the parallel section, programmers implicitly tell Cellgen what data they want transferred to and from the local stores. The Cellgen compiler then takes care of managing locality, by triggering and dynamically scheduling the associated data transfers. In Sequoia, locality is strictly enforced as tasks which can only reference local data. Sequoia allows programmers to explicitly define data and computation subdivision through a specialized notation. Using these definitions, the Sequoia compiler generates code which divides and transfers the data between tasks and performs the computations on the data as described by programmers.
Kamil et al. [45] evaluated the runtime performance of both implicit and explicit optimizations on a stencil computation application, which computes solutions of partial differential equations using finite difference methods on adaptively-refined meshes. In their explicit blocking algorithm (on Cell), a time skewing [46] scheme is used to explicitly cut space and requires the user to specify a cache block size. The grid is divided into cache blocks by several skewed cuts, as illustrated in Figure 2.8, in order to preserve the data dependencies of the stencil. Results show an explicitly managed memory allows for the elimination of cache misses, and the potential impact of inefficient blocking was completely hidden by the significantly improved memory efficiency and memory bandwidth.

While these EMM programming models expose the data locality problem to a programmer, the programmer must learn and understand the new constructs of these approaches.
Chapter 3

Background

Before we can present our EMM approach, it is important to first understand the background knowledge in this work. We will describe the mesh representation used in this thesis, followed by the introduction of the platform STHORM which carries all of our experiments. We discuss its features and performance in this chapter.

3.1 Mesh representation

In this thesis we are working on triangle meshes. Since triangles are the simplest form of a 2D structure and can always create a planar structure despite quadrangles, they are ubiquitous in computer graphics and geometry processing. The most widely used triangle mesh representation, being the input typically accepted by modern graphics hardware, usually consists of a vertex list containing 3D coordinates of the vertices and a face list where each face/triangle has three indices pointing to its three vertices as illustrated in Figure 3.1.

The choice of the data structure is determined by the operations from the algorithm that need to be supported efficiently. Triangle-based data structure is the most widely used data structure that facilitates the efficiency of adjacency queries and hence is chosen for this application.

The Abstract Data Types(ADTs) of mesh, vertex and triangle is as follows:

```c
struct Vertex{
```
unsigned int name; //vertex index
double coor[3]; //vertex coordinates
int vfConnec[12]; // all triangles incident to this vertex
};
struct Triangle {
    unsigned int name; //triangle index
    int vertexIndices[3]; //3 vertices indices
    int neighbors[3]; //3 edges
};
struct Mesh{
    struct Triangle* triangles; //triangle array
    struct Vertex* vertices; //vertex array
    int verNr, triNr;
};

All vertices of a triangle are indicated in the triangle structure by their indices. Neighboring triangles of a triangle are also included in the triangle structure. All triangles sharing
a vertex are recorded in the vertex structure.

It is important to mention that the vertex and triangle lists do not necessarily reflect any ordering. The lists are obtained after parsing mesh data files in which the ordering of vertices and triangles is typically arbitrary, or from the result of mesh generation. This mesh representation is used for subsequent optimizations (mesh data reordering and mesh partitioning) in this work.

### 3.2 Platform STHORM

The platform we perform all the experiments on is an upcoming processor called STHORM designed by STMicroelectronics. The microprocessor is designed for high core-count and low energy costs. The designers of STHORM have removed cache-coherency in lieu of software management. Each STHORM node has 16 in-order processors, and includes two dedicated data transfer units, the Direct Memory Access (DMA) controllers, which is a special hardware device that can asynchronously move data around between memory hierarchies.

<table>
<thead>
<tr>
<th>Memory Region</th>
<th>Access Type</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Memory</td>
<td>Read</td>
<td>200</td>
</tr>
<tr>
<td>Local Memory</td>
<td>Read/Write</td>
<td>1</td>
</tr>
<tr>
<td>DMA Transfer (for x bytes)</td>
<td>Read/Write</td>
<td>266 + 2\left[(x - 8)/8\right] + 208\left</td>
</tr>
</tbody>
</table>

Table 3.1: Memory access latency for STHORM Simulator. This table is from [53].

Data can be transferred around the chip using either word-by-word transfers (direct read) or by using a block transfer. Block transfers can be a few bytes to hundreds of kilobytes, performed asynchronously by DMA controllers; they are the most efficient memory transfers that can be performed on EMM systems [47, 48].

Table 3.1 shows the timing values for each type of local and remote access and DMA transfer time. The simulator model does not model memory contention. The number of processor cycles it takes for DMA to transfer x bytes. The equation is based on real hardware measurements. The DMA controller incurs 266 cycles for initial programming, 2 cycles of pipeline delay per 8 bytes transferred, and a hardware reprogramming cost of 208 cycles.
every 1024 bytes. In addition, the software functions used to initiate a DMA transfer are equivalent to what would be done on real hardware. The software functions read and write DMA controller registers through the unified address system to initiate a transfer.

Table 3.1 omits write for main memory because the simulator assumes that writes will be buffered and returns 1 cycle of access latency. However, the simulator does not model a hardware write buffer, and so the simulator is very optimistic. We focus on read performance of the system.

![Figure 3.2](image.png)

Figure 3.2: The left figure shows the performance of DMA, direct reads, and cache. DMA overheads for different transfer sizes are shown on the right. This figure is from [53].

In order to compare EMM performance with hardware cache, we augmented STHORM with a Dinero IV cache simulator[49]. Dinero is a functional hardware cache simulator that emulates an associative cache. It is configured to be 256K, 4-ways, 64 byte lines, with prefetching enabled. Dinero’s prefetching model linearly prefetches the next nearest 64 byte cache line.

The timing model for the hardware cache utilizes the DMA transfer mechanism. If the access is a miss, then the processor stalls for a DMA transfer of 64 bytes. If the access is a miss that also causes a prefetch, the processor stalls for a DMA transfer of 128 bytes. If the access is a hit, then the processor stalls for 1 cycle as if the data was in local memory. If the processor accesses a prefetched line before it is ready, the processor is stalled for the
remaining time. All misses also incur an access to local memory of 1 cycle.

While block transfer is the most efficient, it does not always account for the best transfer method for a particular application. Given the fact that small block transfers incur high overhead, it is important to understand when to use DMA transfer or direct read in order to design an effective program. We discuss how transfer size and software overhead will affect performance in the following subsections.

3.2.1 DMA overhead

Software overhead is a main factor to determining the DMA transfer size. The timing measurements of DMA transfer in Figure 3.2 include software overhead, such as for-loop overhead and the overhead of initiating a DMA request. In absolute terms, overhead grows linearly as the data size scales up. In real case, however, as shown in Figure 3.2(right), by doubling the size of the buffer, the slope is decreased by more than a factor of 2. Consequently, large transfers are preferred by DMA.

3.2.2 Minimal DMA transfer size

The minimal efficient transfer size is the smallest unit of data worth transferring using DMA, instead of direct read or hardware cache. Figure 3.2(left) shows the total cycles (y-axis) of DMA, direct read, and the Dinero cache to read a varying number of bytes (x-axis) from main memory. Values for direct reads beyond 256 bytes are not shown because they are very large.

These results show that using DMA to transfer less than 64 bytes is inefficient, and that DMA can outperform cache when transferring 1024 bytes or more. If one wishes to transfer 32 bytes or less, it is best to just directly read these values from main memory. Overall, an efficient program would try to work with data blocks of at least 64 bytes to be faster than directly reading memory, and 1024 bytes to rival the performance of cache.

These tests, however, are being somewhat unfair to the cache because they didn’t perform any computation to the data after they have been loaded. No latency is hidden despite of the fact that prefetches can complete while the processor is working; the processor is just stalled waiting for a prefetched line’s asynchronous transfer to finish.

Armed with this minimal efficient transfer information, we are able to wisely choose data movement approach depending on application’s requirements.
Chapter 4

EMM optimizations

In this chapter we present several EMM strategies for three representative mesh traversal applications which have three distinct access patterns. An EMM strategy is composed of a coherent mesh layout, an effective data fetching scheme, and a suitable transfer size, all of which are mainly determined by access pattern. We detail the designs of the components in Section 4.1-4.2, and in Section 4.3, we show how to choose EMM strategy and how the components coordinate with each other in order to optimize a particular application.

4.1 EMM layouts and data structures

Since block transfers are the most efficient memory transfers that can be performed on EMM systems, our EMM layouts are built based on memory blocking(tiling). We present three mesh layouts: Grid, Interleaving Tiles and Adaptive Quadtree, all of which partition the mesh, grouping data spatially close to each other. Given the fact that the access patterns of most geometry processing algorithms typically show spatial locality, we believe that such layouts will facilitate retrieving contiguous data and therefore provide a better trade-off between memory access latency and the extra instruction workload caused by querying to the layouts, initiating DMA transfers and stalling the runtime (wait for DMA transfers to finish).

Considering this extra instruction workload we choose these three layouts that are efficient for queries and require relatively simple additional data structures to implement. The three layouts have different advantages respectively in this context. Grid is the most intuitive model that requires minimal additional data structures upon the basic mesh representation.
CHAPTER 4. EMM OPTIMIZATIONS

Figure 4.1: (a) shows a cell in Interleaving Tiles is surrounded by six other cells (b) shows a cell in Grid is surrounded by eight.

It regularly partitions mesh data into cells of same size while cell data are linearly arranged in main memory either by rows or by columns. Nevertheless, it may impose large overhead when prefetching nearest neighbor cells: there are eight nearest neighbors for a grid cell. In attempting to mitigate the overhead, we propose the Interleaving Tiles (IT) layout. As illustrated in Figure 4.1, each cell in IT is surrounded by six neighbors. As a result, IT can save $(8-6)/8=25\%$ overhead than Grid when prefetching all one-ring neighbors is preferred. Both Grid and Interleaving Tiles, however, do not deal with the problem when mesh elements are far from being uniformly distributed across the mesh, and the application is sensitive to the distribution. Cells containing too much data cause high transfer latency, whereas small cells are not worth transferring considering the significant DMA software overhead. In order to solve this problem, we examine a quadtree layout. Adaptive Quadtree subdivides a mesh until mesh elements are evenly distributed. While Adaptive Quadtree has the advantage of balancing data load for every transfer, it incurs significant more instructions than Grid or Interleaving Tiles as finding nearest spatial neighbors using a tree structure is much more complicated. Consequently, we need to carefully trade-off between data balancing and the extra instruction workload according to the requirements of an application.

The pipeline of the layout generation can be divided into three steps. The first step is to
split mesh triangles and vertices spatially into the layout; the number of cells is determined by application’s access pattern. Then we give every vertex and triangle a layout index so as to facilitate efficient data mapping between mesh element’s (vertex or triangle) linear ID and the layout. In the last step, we reorganize the mesh data in main memory according to the layout.

In this work the applications we examine are all triangle-based; there are no updates or simple visit-or-not updates on vertices whereas both read and write operations occur on triangles. Consequently, we allow duplicate vertices but not duplicate triangles when building the layouts. By partitioning input mesh, mesh triangles are distributed into cells where the triangles completely contained in the cell of k are stored in k; for each triangle t partially contained in the cell of k, t is assigned to the cell its centroid lies in. All the three vertices of a triangle will be assigned to the same cell as the triangle. In other words, a cell contains all the triangles whose centroids are lying in this cell, and contains all the vertices attached to these triangles. Therefore, a vertex in the mesh can be in multiple cells. After the partition, every cell works as a sub-mesh that is self-contained with complete data.

The three layouts employ the same idea for layout-indexing the triangles and vertices. The details will be described for each of them respectively in Section 4.1.1-4.1.3.

In order to reorganize the data according to the layouts, zero-length arrays are used as they allow variable-length object:

```c
struct CellData{
    int verNr, triNr;
    int length;
    char data[0]; //allocate a chunk of memory for vertices and triangles in a cell
};
```

By using such a structure, DMA can transfer vertex data and triangle data in a cell as a whole and cells can have variable size.
4.1.1 Grid

The first layout we examine is the grid layout. We divide 2d planar mesh into k grid cells of same spatial size. See Figure 4.2. Now that each cell performs as a sub-mesh and has its own vertices and triangles, we give a layout index to vertex and triangle such that the layout index can reflect the cell a triangle/vertex belongs to as well as the position it is in the cell. The indexing scheme is shown in Figure 4.3. Likewise, a global to local mapping of vertex index is needed such that a triangle in a cell can reference to its vertices efficiently within the cell. However, since duplicate vertices are allowed, there is no one-to-one mapping from global to local. We solve this issue by making triangle reference its local vertices via pointers. An illustration is given in Figure 4.4. Since the layout does not change during the runtime, using pointers won’t cause any errors in this case. While the vertex data structure remains the same after the layout indexing phase, the triangle ADT is altered as follows.

```
struct Triangle{
  unsigned int name; //triangles original index(ID)
  int vertexIndices[3]; //3 vertices original indices(IDs)
  int neighbors[3]; //3 sharing-edge neighbor triangles
  struct Vertex* vertices[3]; //points to the 3 vertices in local cell
```
CHAPTER 4. EMM OPTIMIZATIONS

Figure 4.3: An illustration of the triangle reindexing methodology. A triangle with ID 800 has a layout index 14812, where the first three digits 148 refers to the cell ID and 12 implies its position in the cell.

Figure 4.4: Once mesh is partitioned into sub-meshes (from left to right), referencing vertex via global index becomes inefficient; we address this by making references as pointers.
Note that the original global index (ID) for vertex and triangle is maintained. We keep a hash table which stores the global-to-local mapping (original ID to layout index) for vertex and triangle in local memory. If the mesh dataset is too large that the hash table cannot fit in local memory, we use DMA transfers to shift parts of the hash table between local and remote memory.

4.1.2 Interleaving Tiles

![Interleaving Tiles layout](image)

Figure 4.5: Interleaving Tiles layout

We propose an improved grid-analogous layout: Interleaving Tiles. As illustrated in Figure 4.5, adjacent rows in the Interleaving Tiles layout are shifted half a tile distance to each other so that a tile now is surrounded by six other tiles instead of eight in Grid. Same as grid cells, tiles are of same spatial size and use the same layout indexing strategy as Grid does. With such a layout, the DMA software overhead can be reduced by \((8-6)/8 = 25\%\) compared to the grid layout whenever the EMM optimization requires prefetching all surrounding cells at a time.
4.1.3 Adaptive Quadtree

Figure 4.6: Octree layout. This figure is from [32].

While Grid and Interleaving Tiles divide up mesh spatially, they don’t guarantee that mesh elements are distributed uniformly to each cell. The density is an important factor to runtime performance in the matter of DMA transfer overhead and minimal efficient DMA transfer size described in Section 3.2. On one hand, transferring a cell with too few nodes causes the same amount of overhead as of a big cell. On the other hand, a heavy cell containing too many nodes causes the processor to stall longer on the transfer. In addition, large cells may have lower data usage ratio depending on program’s data access pattern. To this end, we examine a quadtree representation that is able to keep the cells balanced in terms of element distribution.

We employ the Octree idea from [32] and accommodate it to our 2D case. A Quadtree node works as an index, analogous to a B-tree for database systems, referencing to the underlying data. The data structure of each Quadtree node is defined as follows:
struct QuadtreeNode{
    int name;
    struct QuadtreeNode *parent;
    struct QuadtreeNode *children[4];
    int isLeaf;
    int cellName;
    struct CellData* leafCell; // Pointer to the underlying cells of data in main memory
};

The Quadtree is constructed in two steps:
1. Given the cellsize, which is determined by the application’s access pattern, we know the roughly number of levels of the quadtree being built. A full quadtree having all the leaf nodes the same depth (equal to a grid) is then formed by processing all input triangles. We call it a raw quadtree. The cellsize in italic refers to the number of triangles in each cell through the paper.
2. We merge up or subdivide the leaf nodes in the raw quadtree till each cell referenced by a leaf node has maximum cellsize triangles. We restrict the difference of leaf nodes depth to no more than three levels. This condition guarantees that loading a leaf and all of its adjacent leaf nodes has a bounded space complexity.

Figure 4.6 illustrates the Quadtree structure and the coherent mesh layout. Again we add layout index to mesh triangles to assist fast searches in the tree. The Quadtree layout index of a triangle should be able to direct the search down the tree to the leaf node and find the block of data containing this triangle. To achieve this, we set the index to be the combination of leaf node path and triangles local position. An example of triangle layout-indexing is illustrated in Figure 4.7. The same methodology of adding triangle-to-local-vertex references is applied to all the three layouts.

4.2 EMM data fetching schemes

For good memory access latency, we want to make sure that vertices and triangles assumed to be accessed next are located in local memory. When an application accesses a mesh element that is contained in a particular cell, we transfer the cell (for all the three layouts) into local memory. Since vertices and triangles sharing edges are likely to be requested
Figure 4.7: Triangle reindexing for Quadtree. A triangle with ID 800 has a layout index 22412, where the first three digits 224 shows the path from root to the leaf containing that triangle and 12 implies its position in the leaf.
contiguously, after the application has finished processing the requested mesh object, we are likely to have already loaded the next object the application will request.

Whether prefetching surrounding cells into local memory is depending on application. The naive case is that we do not prefetch any other cells apart from the in-process one. Naive fetching transfers a cell from main to local memory when an element in that cell is requested by the program yet not currently in local memory. If an application has a random access pattern where prefetching may not work, we use this naive fetch. Fortunately, many applications expose relatively regular access patterns, hence we propose the following two prefetching schemes that can handle most of predictable accesses.

Figure 4.8: Convolution prefetching. The red-color-shaded cells are the active, in-process cells. We prefetch the yellow-color-shaded cells, (a) for Grid (b) for Interleaving Tiles and (c) for Quadtree.
4.2.1 Convolution prefetching

We define convolution prefetching as a prefetch of all one-ring surrounding cells. Figure 4.8 illustrates convolution prefetching for each of the three layouts in Section 4.2. To mitigate waiting for these unnecessary transfers to complete, we initiate asynchronous transfers to bring them into local memory, but only wait on transfers when an application requests an object contained in the cell. In this way memory latency can mostly be hidden albeit at added cost of extra overhead for initiating the asynchronous transfers.

This prefetching scheme is designed for applications like region growing, where computation propagates to surrounding cells in the near future.

4.2.2 Extrapolated prefetching

If the accesses are not propagating in all directions, for example, the pattern is a line or a curve with very few sharp turns, it is still possible to develop an effective prefetching scheme. In this case, the next cell requested by the application can be extrapolated based on history path. We employ a simple predicing scheme: we remain a FIFO queue in local memory to store the relative positions of the consecutive cells that have been recently brought in; the relative position with the highest probability is chosen to determine the prefetched cell.

We define four types of relative position: left(1), top(2), right(3), bottom(4). As illustrated in Figure 4.8(a), cell 36 holds 1(left) to cell 37; cell 16 holds 2(right) to cell 37; cell 38 is 3 to cell 37 and cell 58 is 4; cell 15 could be either 1 or 2, and the other 3 corner cells work the same way. We prefetch three cells each time according to the relative position for the Grid implementation.

The cells fallen in the four relative positions vary a little for the Interleaving Tiles: In figure (b), cell 15, 36, 58 are to the top of cell 37; cell 16, 38, 59 are to the bottom of cell 37; cell 15, 16 are on the left while cell 58, 59 are on the right. Thus, each time either two or three cells are prefetched. Quadtree uses the same idea but will have a maximum of four cells prefetched at a time.

We list the aforementioned EMM strategies in Table 4.1. Note that the layout and the fetching scheme in a row do not suggest a binding.
Table 4.1: EMM strategies. The layout and the fetching scheme in a row do not suggest a binding.

### 4.3 Mesh traversal applications

In this section, we show how the EMM strategies in Table 4.1 can be applied on mesh traversal programs and how the components coordinate with each other. We examine three applications that have distinct and representative data access patterns.

#### 4.3.1 Curve tracing

![Figure 4.9: The left figure shows the tensor field on the mesh and the right figure shows the curves traced from the tensor field. This figure is from [37].](image)

As mentioned in Section 2.1, curvature/curve tracing is a common and important geometry processing application that can be generalized as a linear mesh traversal problem in the context of shape analysis, symmetry detection and remeshing. In this work we use the curve tracing application presented in [24] as our workload. It simulates the generation of
winkle curves of deformed cloth. As illustrated in Figure 4.9, the curves are traced from
the vector field, which infers the magnitude and the direction of tension, over the 2D planar
surface by explicitly integrating the 2D ordinary differential equation:

\[
\begin{align*}
\gamma'(u) &= \pm v(\gamma(u)) \\
\gamma'(0) &= s
\end{align*}
\]

In the equation, \( v(u) \) is the vector at point \( u \) calculated using barycentric coordinates.
Lambda is the non-zero step size, assigned to be half of the average edge length of triangles
of the mesh. Each curve is starting from a vertex in the mesh and stops when reaching the
mesh boundary or intersecting with previous curves or meeting given thresholds.

A sequence of adjacent triangles and their vertices is consecutively processed as a curve
proceeds. In each step of the discrete Euler integration, the curve walks a distance of
lambda on the planar along the vector direction at that point, while detecting if there is
an intersection with any edge of the underlying triangle within the step. The occurring of
an intersection leads to the query of the \((x+1)\) th triangle along the curve trace that is
returned by the \( x \) th triangles neighbor of that intersecting edge. The pseudo program of
the application is shown in Pseudo code 4.1.

In pseudo code 4.1, the program starts with a random triangle and an arbitrary vertex
on that triangle as the starting point of the curve. The getCurvelineSegment phase takes
the current triangle and one end of the curve segment, processes it and obtains the other
end of the segment. Function triangleIntersection specifies the next triangle that is about
to be processed. The function is passed with the object of the current triangle plus the line
segment instance, and returns the index of the intersecting edge which indicates the next
triangle. Next triangle continues to become the current triangle in the next iteration until
the next triangle becomes invalid.

The 2D access pattern is curves. A curve does not have width and it is possible to
predict its "next step". However, curves from this application are short as they stop tracing
when intersecting with another one. Hence, we expect that small cells(transfers) paired with
naive fetching, i.e. no prefetching of any surrounding cells, would perform better than the
other EMM strategies. The choice of mesh layout, as discussed at the beginning of Chapter
4, depends on the prefetching scheme, the distribution of the input mesh, and the EMM
system as well. If the EMM system is instruction-efficient, the extra instruction load caused
by querying to Quadtree is then negligible. Otherwise using Grid would provide a better
Algorithm 4.3.1: CurveTracing(\textit{void*})

\begin{verbatim}
vertex_t p1, p2; // curve vertex
triangle_t *current_triangle = getTriangle(mesh, 0);
p1 = current_triangle \rightarrow getRandPoint();
while(current_triangle)
{
    p2 = getCurvelineSegment(current_triangle, p1);
    while(current_triangle && triangleIntersection(current_triangle, p1, p2))
    {
        int edge_index = triangleIntersection(current_triangle, p1, p2);
        int current_triangle_index = current_triangle \rightarrow getNeighbor(edge_index);
        current_triangle = getTriangle(mesh, current_triangle_index);
    }
    p1 = p2;
}
\end{verbatim}

Program 4.1: Pseudo program for curve tracing

In absence of the intersection constraint, if curves can be traced until they reach mesh boundary, Interleaving Tiles paired with extrapolated prefetching should have better performance. We demonstrate it in Chapter 6.

\subsection{4.3.2 Region growing}

The second application is a region growing application. As detailed in Section 2.1.1, region growing describes the access pattern as a pattern grown ring by ring(circle by circle), which is the most common pattern for mesh traversal applications. We examine the classic connectivity encoding algorithm for 3D meshes presented by\cite{16}. An illustration of the traversal process is given in Figure 2.3. The encoding program takes a triangle mesh and outputs a closed and compressed topology of it.

The encoding starts with an arbitrary seed face fseed, its three vertices are added to an active vertex list and their corresponding valences are output to the code sequence, which will eventually be compressed using entropy encoders. The face fseed and its vertices are
flagged conquered. The vertices in the active list are in turn chosen as the pivot vertex which tries to conquer its connecting vertices in the counter-clockwise order. If a connecting vertex \( v \) has not been conquered yet, it is now flagged conquered, and is pushed into the active list; if \( v \) has already been conquered and belongs to the active list, it means the conquered area has separated the unconquered area in two parts. The active list is then split in two: the (internal) sub-list is pushed to the stack for future treatment, while the conquest continues using the (external) list. When the current pivot is full (i.e. with no free edges), it is removed from the list. The pseudo code is given in Pseudo Code 4.2.

The access pattern is completely different from that of the curve tracing program: the program accesses the mesh data in a ring by ring propagation manner in 2D space. We have already prepared the convolution prefetching for such a pattern. For mesh layout, Interleaving Tiles is more preferable than Grid, given that Grid transfers 2 more cells than Interleaving transfers in the case of convolution prefetching. Note that data distribution in region growing is not as important as in curve tracing since all the data items in a cell are likely to be requested in the near future for region growing applications, whereas curve tracing only requires a linear portion of the data. As a result, we don’t consider Quadtree in this case.

### 4.3.3 Random access

If mesh data are randomly accessed, such as spatial sorting and vector sorting, EMM strategies are likely to have worse performance than hardware cache. We use this example to contrast the applicability of EMM strategies.

By random access, a program accesses mesh elements in an arbitrary order. It usually indicates that the elements are processed independently or with minimal queries to the connectivity information. We take the sorting step from the curve tracing application in Section 4.3.1 for example. Before tracing wrinkle curves, the tension vectors are sorted by their magnitude. Every vertex will be read but the order is not concerned in this case. We could just read the vertices sequentially from the beginning of the vertex list which is organized in an arbitrary order.

We demonstrate the effectiveness of our EMM strategies for each of the three applications with performance statistics in Chapter 6.
Algorithm 4.3.2: EncodeConnectivity(mesh_t * m)

```c
stack list_stack; // stack of active lists
list active1, active2;
vertex_t * focus;
while not all triangles of m conquered{
    triangle_t * fseed = m -> getTriangle(0);
    vertex_t * v1 = fseed -> getInternalVertex(0);
    vertex_t * v2 = fseed -> getInternalVertex(1);
    vertex_t * v3 = fseed -> getInternalVertex(2);
    active1.add(v1, v2, v3);
    output(v1, v2, v3);
    list_stack.push(active1);
    while(!list_stack.isEmpty())
    {
        active1 = list_stack.pop();
        while(!active1.isEmpty())
        {
            focus = active1.front();
            active1.pop_front();
            foreach incident vertex i of focus{
                vertex_t * v = focus -> getIncidentVertex(i);
                if(v -> isUnconquered){
                    active1.add(v);
                    v -> flag(conquered);
                    output(v);
                }
                else if(v -> isInActiveList() && !v -> isFull())
                {
                    active2.add(v);
                    list_stack.push(active2);
                }
            }
        }
    }
}
```

Program 4.2: Pseudo program for region growing
Chapter 5

Distance profiler and interface

In Section 4.3 we analyzed the access patterns of three applications and designed EMM strategies accordingly. While the EMM strategies effectively improve performance, programming them places significant burden to application programmers: it requires programmers to manually move the data, and more importantly, it requires adequate understanding of an application’s data accesses such that data can be effectively arranged. In this chapter,
we try to transfer the burden to software so that data movement can be managed automatically (by the software) according to the application’s access pattern. We introduce an access-descriptor called *distance profiler*, which effectively communicates an applications data accesses to a runtime system and subsequently triggers appropriate EMM optimizations.

Distance profiler describes and recognizes data accesses in a distance representation which is then used to map the application to an EMM strategy. With distance profiler, the underlying runtime library orchestrates appropriate data realignment and data movement for the application. An illustration of the relationship between access pattern, distance profile and EMM strategy is given in Figure 5.1.

![Figure 5.1](image)

**Figure 5.1**: Distance profile and EMM strategy

In need of such a distance representation, it is important to realize that same application can have distinctly different access patterns and therefore should employ different EMM strategies. For instance, Figure 5.2 shows two access patterns, both of which are called Z curves. They are obtained from the same space-filling curve algorithm but on two different meshes. However, the spatial order of the accesses for (a) is more regular and consequently should have different prefetching scheme with (b).

![Figure 5.2](image)

**Figure 5.2**: Z curves on different meshes have distinctly different access patterns

On the other hand, same EMM strategy may be applied to different patterns. Taking space-filling curve again for example, pattern (b) in Figure 5.3 is an illustration of Gray-code [50] space-filling curve which looks evidently different from (a). However, if we see every four accesses (first order) as a group, the patterns become exact same hence should applied
Figure 5.3: Z curve and Gray code have different access pattern but if every four accesses are grouped as one, the patterns become exact same hence should applied with the same set of EMM optimizations.

Our distance profiling approach resolves these two difficult situations. When running a program, in the first few iterations, the distance profiler traces the accesses to mesh data and then computes the accumulative spatial distance for each access \( i \) using Equation 5.1. We keep the number of iterations small due to the added overhead incurred by the distance profiling.

\[
\sum_{k=0}^{i-1} \| p_i - p_k \| \tag{5.1}
\]

This accumulative method not only infers the spatial locality of sequential accesses effectively, but also distinguishes global patterns. The key to the effectiveness of our method lies in keeping track of the distance between current access and the first access. Considering this example, in Figure 5.4, for both (a) and (b), any access \( i \) have good locality within its \( k \) predecessors and successors \([i-k, i+k]\) (\( k \) is a small number), while the distance from the \( i \)th access to the first access for (b) is much bigger than that for (a). This phenomenon is reflected in Equation 5.1 since it calculates the accumulative distance between every history access with the current one. Figure 5.5 shows a distance profile. The x-axis is the sequence of accesses, and the y-axis shows their accumulative distance.

Notice that many methods are able to reflect locality. Intuitively, locality is computed within \( k \) consecutive accesses where \( k \) is a small positive number, i.e. for each access \( i \) we
compute the distance between \( i \) and \( (i-k) \) to show locality. However, this type of methods fails to serve our interest since they are incapable of describing access patterns in 2D.

Based on the three applications that have been examined, we obtain a primitive "space" of distance profile (DP) samples, and a mapping to the space of EMM strategies. When a new DP comes, we use nearest neighbor methods to classify it to one of the samples and subsequently find the appropriate EMM strategy. If the new distance profile is far away from the existing samples, we make it a new sample and a new EMM strategy should be applied.

In Table 5.1, we list the current distance profile examples and their EMM strategy mappings as well as the implemented applications that use them to achieve optimal performance. A distance profile with sudden jumps or drops indicates arbitrary accesses which have no spatial locality; a distance profile showing slow and linear increase or decrease implies good locality between accesses; a distance profile showing exponential increase suggests a scanline or curve shape access pattern. A mix of any of these three gives us a hint about the changing of the programs access pattern. It is important to mention that the table is expandable to accommodate more applications; new distance profiles and EMM strategies can be added.
Figure 5.5: A distance profile. The x-axis is the i th accesses, the y-axis shows its accumulative distance.
into existing pool in response to new application requirements.

Finally, we provide a set of interface functions for application programmers to trigger distance profiling and data fetching schemes yet hide the implementations completely from application programmers. The role of the interface is illustrated in Figure 5.6. Once the mesh data is read into main memory, we freeze the data and a distance profile is generated. After the distance profile is classified, the underlying system accordingly chooses the right EMM strategies and orchestrates effective data movement in coordination with the architecture of the underlying hardware.

```
// Mesh interface
Mesh.new(filename);  // create a new mesh structure based on a file
Vertex Mesh.getVertex(id);  // returns the vertex object with the specified ID
Triangle Mesh.getTriangle(id);  // return a triangle object with the id

// Triangle and Vertex interfaces
Vertex.getTriangles();  // returns a list of triangle IDs incident to this triangle
Vertex.setData(VertexData d);  // set application data on this vertex
Vertex.getData();  // return application data on this vertex
Triangle.getVertices();  // return a list of vertex IDs in this triangle
Triangle.setData(TriData d);  // set application data on this triangle
Triangle.getData();  // return application data for this triangle
Triangle.getNeighborTriangles();  // returns a list of triangle IDs sharing edges of this triangle
```

The Abstract Data Types (ADTs) for mesh, vertex and triangle are also part of the
Table 5.1: Distance profile (DP) table. The DP with sudden jumps or drops indicates arbitrary accesses which have no spatial locality; the DP showing slow and linear increase or decrease implies good locality between accesses; An exponential increase in DP suggests a scanline or curve shape access pattern. A mix of any of these three gives us a hint about the changing of the programs access pattern.
interface. We use the mesh representation described in Section 4.1 to implement the ADTs. Mesh data is loaded from a file, and then vertex and triangle objects can be looked up by the application using a user-defined ID. Vertices can be queried for the ID of each triangle they are part of, and Triangles can be queried for each vertex they contain. Both vertex and triangle ADTs support storing and retrieving application specific properties via a get/set data interface. While the application programmer uses three separate ADT interfaces to interact with the Mesh, it is important to realize that this is a stylistic choice with a single underlying implementation across these interfaces. In addition, new architectures can be supported by replacing underlying implementations.
Chapter 6

Experiments and results

We evaluate the EMM strategies in table on the three applications (curve tracing, region growing, and random access) and compare them to a standard implementation using STL-style containers [51] on hardware cache, and compare them to direct read. Direct read is word-by-word transfer in which each transfer accesses to remote memory. The direct read version uses the same STL implementation.

The STL implementation uses vectors to hold the mesh representation described in Section 3.1, i.e. the vertex and triangle lists are implemented as STL vectors. While the STL vector provides a convenient way for referring to mesh objects, it is not responsible for the spatial organization of the mesh itself.

6.1 Experiment setup

All the approaches are implemented and tested on STHORM system using C. STHORM is an upcoming processor with explicit managed memory designed by STMicroelectronics. Each STHORM node has 16 processor cores running at 400 Mhz that share a 256 kilobyte local store. It includes two dedicated data transfer units, the Direct Memory Access (DMA) controllers. The experiments are conducted on one core.

The simulated hardware cache is 256K, 4-ways, 64 byte lines, with prefetching enabled. It prefetches the next nearest 64 byte cache line. The full description of this cache simulator is in Section 3.2. Both the EMM methods and the cache methods of memory access are measured by the cycles it takes to sequentially execute the curve tracing algorithm.

The input mesh is a 2D triangle mesh with 20000 faces and mesh elements are not
uniformly distributed. The data structures built for triangles and vertices take 1.4 megabytes in main memory.

Note that the size of the input mesh is small compared to real problems since the local store of STHORM is limited (256KB) and its remote store is 6MB. As a result, the conclusion drawn from these experiments are not necessarily general.

6.2 Performance metrics

The runtime performance is expressed as processor cycles. Processor cycles for EMM implementations are composed of L3(main memory) access cycles, L1(local memory) access cycles, instruction cycles, instruction loading cycles and DMA cycles. For the cache version, memory accesses will be counted as cache hits and misses. Thus, processor cycles for cache include cache hit cycles, cache miss cycles, instruction cycles and instruction loading cycles.

6.3 Results

Figure 6.1-Figure 6.2 show how the EMM strategy components: mesh layout, prefetching scheme, and cell size (transfer size) affect the performance of the curve tracing application. Each bar shows a breakdown of where the cycles were spent for that implementation. Note that STHORM is not instruction-efficient. As a result, Quadtree has bad performance in all the experiments. However, as can be seen in Figure, the DMA time for Quadtree is almost half less than the other two, which shows great potential to be employed on instruction-efficient EMM systems.

Based on the results from Figure 6.1 and Figure 6.2, we pick the EMM strategy with best performance - Grid & naive fetching & small cell size, and compare it to the STL implementation with hardware cache, and to direct read. Results are shown in Figure 3. EMM(Grid) performs better than Cache(Grid) but slightly worse than Cache(Original). Cache(Original) runs the program on the original, unorganized (or unknown) mesh layout, while Cache(Grid) runs on the grid layout same as in EMM(Grid). EMM(Grid) performing close to cache is a good result since EMM(Grid) does not employ any prefetching scheme. Note that we were being somewhat unfair to the cache since the cache implementation does not involve any optimizations.
For the curve tracing application, we also measured its performance if there is no constraint on the length of the curves, i.e. the curves are traced as long as they can be until they reach the boundary of the mesh. Figure 6.4 and Figure 6.5 show that, in this case, Interleaving Tiles paired with extrapolated prefetching performs the best among the EMM strategies, and it outperforms the cache implementation by 12x.

Figure 6.6-Figure 6.7 show the same set of experiments for the region growing application. Tiles with convolution prefetching performs 4.5x faster than Direct Read, and 1.2x faster than Cache. The Cache(Tiles) and the EMM(Tiles) bar demonstrate that, even if data are organized by locality, EMM fetching schemes can still outperform cache as cache can only do next-line(1D) prefetching and the transfer size is fixed.

The result from Figure 6.9 is used to show that, for applications with random spatial access, we should use cache instead of EMM.

The experiments demonstrate the effectiveness of our EMM strategies and how access pattern determines the design of the EMM strategies.

Figure 6.1: Short curves: performance of different EMM mesh layouts and fetching schemes
CHAPTER 6. EXPERIMENTS AND RESULTS

Figure 6.2: Short curves: performance of different transfer size

Figure 6.3: Short curves: performance comparison between direct read, cache and EMM
CHAPTER 6. EXPERIMENTS AND RESULTS

Figure 6.4: Long curves: performance of different EMM mesh layouts and fetching schemes

Figure 6.5: Long curves: performance comparison between direct read, cache and EMM
Figure 6.6: Region growing: performance of different EMM mesh layouts and fetching schemes

Figure 6.7: Region growing: performance of different transfer size
Figure 6.8: Region growing: performance comparison between direct read, cache and EMM

Figure 6.9: Random: performance between direct read, cache and EMM
Chapter 7

Conclusion and future work

In this work we investigate EMM optimizations for geometry processing applications. We present three mesh layouts and two prefetching schemes tailored for three representative applications and demonstrate their effectiveness. We then introduce the distance profiler which can describe access patterns and enable the system to automatically map an application to a proper EMM strategy. Finally, we provide a set of interface functions which trigger the distance profiler and data movement, and ultimately make the EMM optimizations transparent to application programmers.

This work remains in its preliminary stages. Currently the distance profile table only contains four distance profile samples and five EMM strategies: the table should be expanded to cover more applications; a more advanced distance profiling method is pursued such that it can provide more accurate characterization of data access pattern. More importantly, the experiments with the distance profiler do not include a performance evaluation. Computing the distance profiles and performing the mapping to EMM strategies would incur an overhead which degrades performance. It remains to be investigated what kind of performance trade-off would be obtained with such an overhead.

Future work on EMM optimizations for geometry processing will attempt to address these issues. Additionally, we wish to investigate the portability of the EMM strategies, distance profiler and interface to other EMM systems other than STHORM. We also want to try different mesh partitioning methods and different mesh representations to further the optimizations according to an application’s requirements.
Bibliography


[38] A. Bogomjakov and C. Gotsman, Universal rendering sequences for transparent vertex caching of progressive meshes, in Proc. of GRIN 01, 2001, p. 81.


[53] Craig Mustard. Application Design for Explicitly Managed Memory. (unpublished)