Master Thesis

Parallel geometry processing

Author(s):
Poggiali, Dario

Publication Date:
2008

Permanent Link:
https://doi.org/10.3929/ethz-a-005666870

Rights / License:
In Copyright - Non-Commercial Use Permitted
Parallel Geometry Processing

Dario Poggiali

Master Thesis
September 2008

Prof. Dr. Mario Botsch
Prof. Dr. Markus Gross

Analysis and Optimization of Spatial and Appearance Encodings of Words and Sentences
Christian Vögeli
Master Thesis
SS 2005
Prof. Dr. Markus Gross
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction
Ridges are local maxima in a relaxed sense. According to Eberly, Height Ridges are the loci with first derivative equal to zero and second derivative negative, in at least one direction (which is per definition perpendicular to the ridge). Widely used in image analysis, e.g. in medical imaging, there needs to be only one such direction with the mentioned conditions, resulting in ridge curves. In the 3D domain, one usually chooses in advance if it is looked for ridge surfaces (mentioned conditions in one direction) or ridge curves (mentioned conditions in two orthogonal directions). This decision is easily made if one knows that the data contains only structures with circular cross-section (looking for ridge curves) or longish cross-section (looking for ridge surfaces). However, data is often unknown, exhibits structures that are intermediate, or exhibits both types. The goal of this project is to develop a method that is capable of deciding automatically (with possible parametrization by the user) which type has to be looked for at a given region.

Task / Work Packages
• Automatic decision based on eigenvalues of the Hessian, generation of the geometry
• Application of the method to synthetic test data and practical data
• Hysteresis-like mechanisms for avoiding ridges to oscillate between 1D and 2D
• Other (advanced) mechanisms for extraction control and filtering
• Possibly comparison of the results to those from Skeletonization/Medial Axis

Requirements
• C/C++ programming skills
• Sufficient mathematical skills

Remarks
A written report and an oral presentation conclude the work. The thesis is overseen by Prof. Markus Gross and supervised by Filip Sadlo, Institute of Computational Science. For further information or application to this project, please contact Filip Sadlo, IFW C27.1, Tel. 632 71 44, sadlo@inf.ethz.ch.
Abstract

In this thesis we investigate the benefits achieved by exploiting parallel algorithms on typical Computer Graphics problems. Given three different parallel architectures, the impact in term of performance on well known geometry tasks is examined.

We start by considering simple SSE instructions for SIMD level parallelism using Intel® SSE intrinsics. Then, shared-memory parallelism on multiprocessing machines and the dedicated OpenMP™ API are investigated. Finally, we consider General-Purpose computing on Graphics Processing Units together with NVIDIA® CUDA technology. By exploiting these three types of parallelism, a Conjugate Gradient method for sparse linear systems derived by the harmonic parameterization is implemented. Further, a parallel Multigrid solver for finite element simulations based on hexahedral cells is developed.

During the implementations of such methods, timings and speedups are measured, in order to find which parallel paradigm is best situated for each one of these problems.
Acknowledgment

First, I want to thank my supervisor Mario Botsch for his great efforts in constantly motivating me during this thesis, and Peter Kaufmann for his precious and constant availability.

I would also thank Samuele Gantner and Guy Müller for reminding me during my whole school career the correct balance between working and relaxing.

Thanks to Gioacchino Noris and Diego Rossinelli for their precious and inspiring discussions.

Least but not last, special thanks to Denise Spicher and Hans Dubach for acting like a compass in this insidious world called ETHZ.
Parallel Geometry Processing

Introduction
Since the complexity of 3D models is steadily growing their efficient processing requires to exploit all available hardware resources for numerical computing. The goal of this master thesis is to investigate into parallel geometry processing on multi-core CPUs and GPUs, and to compare the speed-ups possible with those architectures. The comparisons will be performed on numerical techniques, such as the conjugate gradient (CG) method, as well as on some simple geometry processing algorithms.

Task Description
The main tasks of this master thesis are:

• Investigation of SSE instructions for instruction level SIMD parallelism.
• Investigation of OpenMP for shared memory parallelism on multi-core CPUs.
• Investigation of CUDA for GPU-based parallel programming.
• Implementation of CG solver using SSE, OpenMP, and CUDA parallelization.
• Exploiting parallelism for some example geometry processing methods.
• Analysis of the respective speed-ups.

Further optional tasks could be:

• Comparison of sparse direct solvers to iterative CG solvers.
• Implementation of a multi-grid solver.

Remarks
A written report and an oral presentation conclude the work. The master thesis is overseen by Prof. Markus Gross and is supervised by Dr. Mario Botsch.

• Hand-out: March 3, 2008
• End-date: September 2, 2008
# Contents

1 **Introduction**  
1.1 Related Work ................................................. 1  
1.2 Overview ...................................................... 2  

2 **Parallel Architectures**  
2.1 Single Instruction Multiple Data (SIMD) ................... 3  
2.1.1 Intel® Streaming SIMD Extension ....................... 4  
2.1.2 Memory Alignment .......................................... 7  
2.2 Shared-Memory Parallel Computers ......................... 9  
2.2.1 Open Multi-Processing (OpenMP™) ....................... 10  
2.3 General-Purpose computing on Graphics Processing Units (GPGPU) ............. 14  
2.3.1 NVIDIA CUDA™ ........................................... 14  

3 **Sparse Linear Systems** ........................................ 21  
3.1 Sparse Matrix Representation ............................... 22  
3.1.1 The Coordinate Storage (COO) .......................... 22  
3.1.2 Compressed Row Storage (CRS) ......................... 23  
3.1.3 Block Compressed Row Storage (BCRS) ................. 24  
3.2 Sparse Linear System for Surface Parameterization .......... 25  
3.3 Sparse Linear System for FEM Simulation ................ 28  
3.3.1 Linear Static Deformation ............................... 28  
3.3.2 Linear Dynamic Deformation ............................ 29  
3.3.3 Stiffness Warping ......................................... 29
## Contents

4 Sparse Linear Solver 31
   4.1 Conjugate Gradient (CG) ............................................. 32
      4.1.1 The Algorithm ................................................. 34
      4.1.2 Parallelizing the Conjugate Gradient ....................... 35
   4.2 Multigrid (MG) ....................................................... 50
      4.2.1 The Algorithm .................................................. 52
      4.2.2 Parallelizing the Multigrid ................................. 52

5 Conclusion and Future Work 57
   5.1 Parallel Architectures ............................................. 57
   5.2 Sparse Linear Solvers ............................................. 58

Bibliography 59
Introduction

Performance has always been a key requirement of real-time applications, especially when dealing with large amount of data. This information needs to be processed by exploiting all the available hardware resources. During the last years, leading semiconductor industries moved toward parallel architectures, and nowadays almost every desktop computer is equipped with multi-core processors and graphics processing units specialized for highly parallel computations. With the advent of this new hardware capabilities, new dedicated application programming interfaces have been developed in order to improve the accessibility of these resources ([NVI], [AA], [Ope]).

3D geometry processing fits perfectly into this new programming model. Usually, 3D surfaces are discretized using a huge number of elements and the algorithms dealing with these are not efficient enough to be used in any sort of real-time application. The spectrum of applications where the quality is proportional to the resolution of the data is wide: from computer games, where often low-poly characters are used in order to guarantee a plausible interaction with the user, to medical applications, where high detailed models of the patients bodies should be used to facilitate the diagnostic or to train new surgeons. By exploiting parallel architectures, these applications can be speeded up significantly, offering for the same updating rate more plausible and realistic looks.

1.1 Related Work

As shown in [MB05], high performance linear solvers are of major importance in geometry processing algorithms. Many efforts have been dedicated on mapping such solvers on parallel architectures. For example, the PARDISO solver [OS] implements a parallel $LU$, $LDL$ or $LL^T$ factorization on shared-memory multiprocessing machines, and it is licensed to thousands of
1 Introduction

Researchers. Even leading semiconductor industries provide more and more highly optimized, extensively threaded libraries for their hardware, as in the case of the Intel® Math Kernel Library [Int07]. Multigrid methods recently started to move towards parallel implementations as well, as suggested by [LSar] for the mesh deformation problem and as shown in [GW06] for simulation of deformable bodies.

General-Purpose computing on Graphic Processing Units is also a technique that during the last years gained more importance. [KW03] shows how to map linear algebra operators to solve set of algebraic equations on the GPU and [BFGS05] implements a Conjugate Gradient and a Multigrid method on NVIDIA® graphics cards. Moreover, leading graphics hardware industries NVIDIA® and AMD-ATI™ have developed dedicated APIs for their GPUs: Compute Unified Device Architecture (CUDA [NVI]) and Close-to-Metal (CTM [AA], now Stream SDK) softwares allow to use the GPU as a computing device. With these new strategies [BCLar] implements a general sparse linear solver and [SHZO07] shows novel GPU implementations of numerical kernels using parallel scan primitives.

1.2 Overview

In this thesis we investigate the benefits of exploiting some parallel resources which nowadays are common to every desktop computer. We start by describing three different types of parallel architectures: first, the Intel® SSE instruction set for low level SIMD parallelism is investigated. Then, we discuss shared-memory multiprocessing programming and the dedicated OpenMP™ API and, finally, we analyze General-Purpose computing on Graphics Processing Units with the new NVIDIA CUDA™ technology (Chapter 2). For each of these architectures the programming models and their hardware implementations are explained. In order to being able to evaluate the benefits achieved by exploiting these parallel architectures, Chapter 3 introduces the mathematical background of some important applications in Computer Graphics, for instance the surface parameterization and the simulation of deformable bodies. Next chapter, Chapter 4, describes two important numerical solvers for systems of linear equations (the Conjugate Gradient and the Multigrid methods) and shows how these routines behave when mapped into the parallel architectures described in Chapter 2. Finally, we draw some conclusions and we suggest some future work in Chapter 5.
Parallel Architectures

In contrast with sequential architectures, where a sequence of instructions is executed one after the other, parallel architectures allow to execute many instructions simultaneously. There are many types of parallelism, and each of them needs dedicated hardware resources. In this chapter we describe three different parallel architectures. We start with a low level parallel technique that allows us to exploit special CPU registers in order to perform the same operation on multiple data. We proceed investigating shared-memory multiprocessing machine, nowadays common to almost every desktop computer and, finally, we discuss a new technique for General-Purpose computing on Graphics Processing Units.

2.1 Single Instruction Multiple Data (SIMD)

*Single Instruction Multiple Data (SIMD)* is a technique that allows us to process a large amount of data of same type that need the same instruction performed on it. With a sequential architecture, all the elements of a data set are processed one after the other, and the same operation is performed on the single elements. A typical implementation of this strategy looks like the pseudocode in Figure 2.1

```
for(int i = 0; i < n; i++)
{
    Type el = data[i];
    operation(el);
}
```

*Figure 2.1:* All the elements in *data* are processed individually.
By exploiting a SIMD architecture, we can process packets of data simultaneously, decreasing the amount of time needed for the computation. The previous example can be rewritten as shown in Figure 2.2.

```java
for(int i = 0; i < n / packetSize; i++)
{
    SIMDType packet = loadPacket(data, i);
    SIMDoperation(packet);
}
```

**Figure 2.2:** SIMD architectures allow us to perform the same instruction on packets of data simultaneously.

Instead of iterating over all the $n$ elements in the data set, we iterate over $n/packetSize$ packets of data of fixed dimension. Thanks to special hardware resources, all the element in one packet are processed concurrently. A visual explanation of this process is given in Figure 2.3. With a sequential architecture the elements of the data set are processed separately (Figure 2.3 (Left)). On the other hand, exploiting SIMD level parallelism, we perform the same instruction on multiple elements simultaneously (Figure 2.3 (Right)).

**Figure 2.3:** Left: data elements are processed sequentially. Right: the same instruction is performed on multiple data elements concurrently.

### 2.1.1 Intel® Streaming SIMD Extension

Intel® introduced an instruction set called *Streaming SIMD Extensions (SSE)* that allows the developers to exploit SIMD parallelism. Intel® processors are equipped with eight 128-bit registers named XMM0, ..., XMM7. Each of these registers packs together four 32-bit single precision floating point numbers, and SSE instructions are performed on these registers. XMM registers can be viewed as vectors containing four elements and they are the basic unit of SIMD instructions (for this reason SIMD computing is also known as vector processing). Implementing an algorithm which exploits SSE instructions requires the knowledge of some basic operations: First, we need to know how packets of data are loaded from main memory into XMM registers, then which instructions can be performed on them and, finally, how the data in the XMM registers is stored back to main memory.
2.1 Single Instruction Multiple Data (SIMD)

Moving Data between Memory and XMM Registers

The first thing we need to understand in order to perform any SIMD instruction is how data is loaded from the main memory into one of the XMM registers. Given an address to main memory, SSE instructions allow to load a continuous portion of memory starting from this address into the destination register (Figure 2.4).

![Figure 2.4: A continuous portion of memory is loaded into a XMM register. The instruction takes as inputs an address src in main memory and a destination XMM register.](image)

Once the data has been loaded into the registers it is possible to perform some operations on it. Suppose we have four single precision floating points $a_0, a_1, a_2$ and $a_3$ that we want to add to other four single precision floating points $b_0, b_1, b_2$ and $b_3$, resulting in $c_0 = a_0 + b_0, c_1 = a_1 + b_1, c_2 = a_2 + b_2$ and $c_3 = a_3 + b_3$. Moreover, suppose that the $a_i$ are consecutively stored in main memory starting from address 0x40, the $b_i$ from 0x80 and the $c_i$ from 0x100. Figure 2.5 shows the configuration.

![Figure 2.5: Example of exploiting SSE instruction. Each array is stored consecutively in the memory, and four elements are processed simultaneously each clock cycle.](image)
2 Parallel Architectures

With a sequential architecture, we first need to load $a_0$ and $b_0$ into two CPU registers. Then, we can add the content of this two registers ($c_0 = a_0 + b_0$) and move back the result to main memory. The same process is repeated until no more elements have to be processed. Using SSE instructions however, we can load 128-bit of memory starting from address 0x40 (all the $a_i$) into register XMM0 and 128-bit starting from 0x80 (all the $b_i$) into register XMM1. We then add XMM1 to XMM0 and finally we store the result back to main memory starting from address 0x100.

Moving data from XMM registers to main memory works similarly to its opposite operation: Given a source XMM register src and a destination address dest, the content of src is stored into 128-bit of continuous memory starting from dest (Figure 2.6).

![Diagram](image)

**Figure 2.6:** The content of register XMM0 is stored back into 128-bit of consecutive main memory starting form address dest.

However, we have to to keep in mind that working directly with CPU registers can be really problematic. Before loading data into CPU registers we need to save their current state in order to be able to recover a consistent state after the operation; if our instructions are nested, we have to take care of using temporary registers to store the results needed for the next instructions, and so on.

Fortunately, Intel® provides a set of SSE intrinsics. With SSE intrinsics, we do not have to care about registers, but instead we have a 128-bit data type __m128 and a set of C++ methods for performing arithmetic and logical instructions. An implementation of the example of Figure 2.5 becomes simply:

```c
void add4(float* a, float* b, float* c)
{
    __m128 vA, vB, vC;
    vA = _mm_load_ps(a); // Load 4 floating points numbers from address pointed by a
    vB = _mm_load_ps(b); // Load 4 floating points numbers from address pointed by b
    vC = _mm_add_ps(vA, vB); // vc = vA + vB
    _mm_store_ps(c, vC); // Store 4 floating points numbers to address pointed by c
}
```

Lines 4 and 5 load 128-bit of data from the main memory pointed to $a$ and $b$. Line 6 adds in a SIMD fashion the vectors of data $vA$ and $vB$ storing the result in vector $vC$. Line 7 stores the
elements in $vC$ to the location in main memory pointed to $c$. The temporary variable $vC$ is not mandatory. Instead of storing the result of the addition of $vA$ and $vB$, we can directly write

```c
_mm_store(c, _mm_add_ps(vA, vB)) or _mm_store(c, _mm_add_ps(_mm_load_ps(a), _mm_load_ps(b)))
```

and let the compiler decide how many registers have to be allocated.

### 2.1.2 Memory Alignment

Memory alignment can have significant influence on performance when moving data from main memory to XMM registers and vice-versa. An address `add` in main memory is **16-byte aligned** when it is a multiple of 16, that is, when $add \mod 16 = 0$. Figure 2.7 shows a typical memory structure underlining the addresses that are 16-byte aligned.

![Memory Alignment Diagram](image)

**Figure 2.7:** Memory alignment. Left: only the green addresses are 16-byte aligned. Center: 16-byte alignment is ensured for the arrays $a$, $b$ and $c$. Right: arrays $a$, $b$, and $c$ are not 16-byte aligned.

Some SSE intrinsics are optimized to load or store packets of data starting from an aligned address, but they fail to execute if this address is not aligned. If memory alignment cannot be ensured, other instructions less optimized have to be used.

Dense Matrix Vector multiplication (DeMV) is a good example to investigate the behavior of SSE instructions when memory alignment is ensured or not. Suppose we store a square matrix $M$ with dimension $N \times N$ in row-major order \(^1\). Moreover, suppose that $N$ is a multiple of 4 ($N \mod 4 = 0$). The entries of $M$ are stored continuously in the main memory starting from

\(^1\)In row-major storage, a matrix is stored in linear memory one row after the other.
2 Parallel Architectures

an address $m$. Our goal is to multiply $M$ with a vector $x$ and store the result in vector $r$. A sequential algorithm for DeMV is given in Algorithm 1.

Algorithm 1 DeMV

1: for $i = 0$ to $N - 1$ do
2: $acc = 0.0$
3: for $j = 0$ to $N - 1$ do
4: $acc = acc + M(i, j) \times x(j)$
5: end for
6: $r(i) = acc$
7: end for

SSE intrinsics are used to speed up the dot product between a row of $M$ and the vector $x$ (Lines 3, 4, and 5). The parallelized implementation of DeMV using SSE intrinsics is:

```c
void DeMV_SSE(float* m, float* x, float* r, int n )
{
    __m128 vM;  
    __m128 vX;  
    union u  
    {                
        __m128 v;     
        float f[4];   
    } acc;
    for(int i = 0; i < n; i++)
    {
        acc.v = _mm_set1_ps(0.0f);
        for(int j = 0; j < n; j+= 4)
        {
            vM = _mm_load_ps(m + i*n + j);
            vX = _mm_load_ps(x + j);
            acc.v = _mm_add_ps(acc.v, _mm_mul_ps(vM, vX));
        } 
    }
}
```

After loading the current values of $M$ and the corresponding values of $x$ into two XMM registers (Lines 16 and 17) we multiply them and we add the result to an accumulator initialized to zero (Line 18). We repeat this procedure until one entire row of $M$ is processed. Before passing to the next row, we sum the values of the accumulator together and we store the result in $r$ (Line 20). To access the single values of a XMM registers, a C++ union data type is used (Line 5). Since all of the members of an union occupy the same location in memory, it allows the same portion of memory to be accessed as different data type. Therefore, the modification of one element of the union will affect all other elements. By declaring an union with two members, namely a __m128 variable and an array with four floats, we can process the data as XMM register and read it as single float value.

The parts we are interested in are where the entries of $M$ and the entries of $x$ are loaded from main memory to one of the 128-bit XMM registers (Line 16 and 17). If the address $m$ is 16-byte aligned, all loading operation will load 128-bit of memory starting from an aligned address. On the other side, if $m$ is not 16-byte aligned, we need to load the values from main memory using a non-optimized instruction. As Figure 2.8 shows, this leads to a deterioration of performance.
2.2 Shared-Memory Parallel Computers

Shared-Memory Parallel Computers are machines with multiple processors that communicate through variables stored in a shared address space. These architectures are often called symmetric multiprocessing machine, i.e., shared-memory parallel computers whose processors can individually access the shared memory space with the same speed (they have Uniform Memory Access, UMA) [BC08]. Figure 2.9 shows the simplified diagram for our dual processors, 8-Core machine.

Each processors has four cores with their own L1 cache. The L2 cache is shared between all the cores in a processor and the two processors (respectively the eight cores) communicate through the main memory. Having caches with different hierarchies (L1 and L2) and different scopes (private L1 cache per core and private L2 cache per processor) requires some important mechanisms in order to ensure cache coherence. Take for example a single processor system: new values computed by the processor are written in cache, where they are kept until the space they occupy is needed to store other, new, values. At this point, all the new values in cache are stored back to the main memory. In a shared-memory multiprocessing machine this strategy does not work anymore: If the new values computed by a single core are stored directly in its private cache, coherence is compromised.

The blue plot shows the speedups for the Dense Matrix Vector multiplication when the address is 16-byte aligned and when optimized instructions are used to load the values from memory to the XMM registers. As the reader can see, this strategy leads to the best performances, speeding up the DeMV operation by a factor of 3. The yellow plot shows the timing when non-optimized instructions are used. Even when these instructions are used, ensuring memory alignment anyway leads to better performances (green plot). However, the theoretical speedup of 4 is not achieved because processing four elements simultaneously takes longer than processing them individually.

Figure 2.8: Speedups for the Dense Matrix Vector multiplication parallelized with SSE intrinsics.

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>(SSE) Aligned (Optimized)</th>
<th>(SSE) Aligned (Unoptimized)</th>
<th>(SSE) Unaligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>2.96</td>
<td>2.32</td>
<td>2.62</td>
</tr>
<tr>
<td>2048</td>
<td>2.91</td>
<td>2.26</td>
<td>2.62</td>
</tr>
<tr>
<td>3072</td>
<td>2.69</td>
<td>2.27</td>
<td>2.62</td>
</tr>
<tr>
<td>4096</td>
<td>2.64</td>
<td>2.22</td>
<td>2.62</td>
</tr>
<tr>
<td>5120</td>
<td>2.89</td>
<td>2.23</td>
<td>2.62</td>
</tr>
<tr>
<td>6144</td>
<td>2.86</td>
<td>2.21</td>
<td>2.62</td>
</tr>
<tr>
<td>7168</td>
<td>2.85</td>
<td>2.22</td>
<td>2.62</td>
</tr>
<tr>
<td>8192</td>
<td>2.85</td>
<td>2.21</td>
<td>2.62</td>
</tr>
<tr>
<td>9216</td>
<td>2.85</td>
<td>2.22</td>
<td>2.62</td>
</tr>
<tr>
<td>10240</td>
<td>2.86</td>
<td>2.22</td>
<td>2.62</td>
</tr>
</tbody>
</table>
cache, they are not accessible from the others cores. Fortunately, many strategies have been
developed in order to solve this issue, and nowadays shared-memory machine are ensured to be
cache coherent.

### 2.2.1 Open Multi-Processing (OpenMP™)

OpenMP™ API is the de-facto standard for development on shared-memory multiprocessing
machines. It consists of a set of compiler directives, library routines and environment variables
that are used in order to control the parallelization of a program. OpenMP™ is based on a thread
paradigm: a process consists of many threads, that is, active executions of instructions within
the process. Threads share the same memory space and, in a multiprocessing system, they can
be executed simultaneously. When a process is created, it consists of a master thread that runs
sequentially until it reaches a region of the code that has to be executed in parallel. At this
point, the master thread forks new threads that execute the parallel region and in the end they
join together (Figure 2.10).

The aim of executing many threads simultaneously is to reduce the computational time needed
to solve a problem. The principle is that a big problem can be decomposed into smaller ones
which are then solved concurrently. However, how the problem is divided is responsibility
of the programmer and there are many aspects (like dependencies between the problems, data
exchange between the processors, thread synchronization, ...) that need to be taken in consider-
ation.
2.2 Shared-Memory Parallel Computers

**Figure 2.10:** If a parallel region is reached, the master threads forks new threads. In the end, they join together, and the control is returned to the master thread.

**OpenMP™ Memory Model**

By default data is shared between the parallel threads and the changes are visible to all of them. However, some parallel algorithms may need private variables, i.e., variables that can be accessed only by a single thread. How the data is accessed is also important: multiple parallel threads can read simultaneously from the same shared memory location, but they need to be synchronized when they write to this location. Letting two or more parallel threads write to the same shared memory location at the same time results in an inconsistent state. To better understand the importance of having different memory scopes, let us come back to the Dense Matrix Vector multiplication example introduced in Section 2.1. First, we have to find a good strategy to subdivide the problem along the processors. Since matrix $M$ is stored linearly row-by-row in the main memory, a good approach is to let each processor compute a contiguous portion of the resulting vector $x$. In this way thread $T_0$ will take care of multiplying the first batch of rows of $M$ with $x$, thread $T_1$ the second batch and so on (Figure 2.11).

**Figure 2.11:** Matrix $M$ is subdivided in batches of rows. Each thread computes a different entry of the resulting vector $x$. Since each thread write its result on different memory locations, there is no need to synchronize them.
Each thread accesses the data of $M$ and of $r$ at different locations and therefore we can leave their entries stored in the shared memory. The entries of $x$ have to be accessed by all threads simultaneously but fortunately each concurrent thread only needs to read their values and therefore vector $x$ can be left in the shared memory.

The implementation of the Dense Matrix Vector multiplication on a 8-Core machine is:

```c
void DeMV_OMP(float* m, float* x, float* r, int n )
{
    int i, iStart, iEnd;
    int width = n / 8;
    int thrid;
    float acc;

    #pragma omp parallel private(i, iStart, iEnd, thrId, acc)
    {
        thrid = omp_get_thread_num();
        iStart = thrid * width;
        iEnd = iStart + width;
        
        for(i = iStart; i < iEnd; i++)
        {
            acc = 0.0f;
            for(int j = 0; j < n; j++)
            {
                acc += m[i*n + j] * x[j];
            }
            r[i] = acc;
        }
    }
}
```

At line 8 the parallel region is defined by the OpenMP™ compiler directive `#pragma omp parallel`. When the master thread reaches this definition, it will create as many threads as processors and each one of them will execute the code in the parallel region. The optional directive `private(<list>)` declares variables $i$, $iStart$, $iEnd$, $thrId$ and $acc$ as private. This means that each thread has its own copy of these variables and the others threads cannot modify them. The private scope of variables $i$, $iStart$ and $iEnd$ ensures that each thread is reading and writing in the proper location in the main memory. $thrId$ is a variable where the ID of the thread that is currently executing the region is stored (in our configuration $thrId$ can be $0, 1, ..., 7$) and its values is returned by the routine `omp_get_thread_num()` (Line 10).

Variable $acc$ is an accumulator for the result of the dot product; if $acc$ were shared, each threads could increment it adding values that belong to other rows of $M$. By defining $acc$ in a private scope, we ensure that each thread has its own accumulator that cannot be modified by anyone else. Since this implementation fails if the dimension $n$ of the matrix is not a multiple of 8, we use the OpenMP™ compiler directive `#pragma omp parallel for private(acc)` that automatically splits the `for` loop among the parallel threads.

In Section 2.1 we saw how the Dense Matrix Vector multiplication can be improved by using SSE intrinsics to speed up the dot product between a row of $M$ and vector $x$. In a shared-memory multiprocessing computer we can subdivide the matrix $M$ and let each processor compute a subset of the resulting vector $r$. In order to exploit all the available hardware resources we mix these two strategies: Each processor is responsible for a batch of rows of $M$ and it uses SSE intrinsics to speed up the computation of the respective entries of $r$. Figure 2.12 shows the timing for all the parallelized algorithms we investigated. By increasing the size of the
problem, the benefits of using parallel threads become more visible. With our best implementation we were able to run the Dense Matrix Vector multiplication about ten times faster than the sequential implementation.

In a hypothetical world, using 8 processors together with XMM 128-bit registers should result in an overall speedup of 32. However, this is quite distant from the performances reachable by the actual implementations. In order to verify how the parallel Dense Matrix Vector multiplication behaves, we run the code varying the number of processors used during the parallelization. Figure 2.13 (Left) shows that the speedups achieved are proportional to the number of processors. On the other side, we noticed that the additional speedup obtained by the SSE registers tend to vanish when more than three parallel threads are used (Figure 2.13 (Right)). However, it is interesting to notice that the same performance obtained by using 8 processors can be reached by only 4 processors together with SSE instructions (Figure 2.13).

**Figure 2.12:** Timing for the Dense Matrix Vector multiplication routines with all the parallel strategies investigated until now.

**Figure 2.13:** Speedups with respect to the parallelized version of Dense Matrix Vector multiplication ($N = 10240$) using one processor. Left: exploiting multiprocessing systems. Right: exploiting multiprocessing systems mixed with SSE intrinsics.
2 Parallel Architectures

2.3 General-Purpose computing on Graphics Processing Units (GPGPU)

General-Purpose computing on Graphics Processing Units (GPGPU) is a technique where the GPU is used instead of the CPU for performing numerical computations. Today’s GPUs can be seen as coprocessors to the main CPU capable of executing a large number of threads in parallel. Moreover, thanks to a dedicated API, programming the GPU has become more accessible than in the past. NVIDIA CUDA™ [NVI] (Compute Unified Device Architecture) software is an extension of the C programming language that allows using the GPU as a computing device.

2.3.1 NVIDIA CUDA™

With the CUDA API, the GPU is viewed as a device able to execute set of parallel threads performing the same instruction on different data, that is, in a SIMD fashion (Section 2.1). CUDA describes both an abstract programming model and its hardware implementation; in the next sections we will explain these concepts.

CUDA Programming Model

The GPU is described as a batch of threads organized in a grid of thread blocks (Figure 2.14).

![Figure 2.14: Thread organization with the CUDA programming model.](image)
Blocks can have two or three dimensions, but grids have to be two-dimensional. All these threads execute the same instructions concurrently on different batches of data. The programmer can choose how many threads need to be created, but it is the responsibility of the device to decide how many threads can run simultaneously. Each thread is indexed by a block coordinate and each block by a grid coordinate. The first indicates the position of the thread in the block, and the second the position of the block in the grid. A thread with block coordinate \((t_x, t_y)\) inside a two dimensional block of dimension \((B_x, B_y)\) is indexed with a thread ID obtained as follows

\[
\text{thread}_{id} = t_x + t_y B_y. \tag{2.1}
\]

Respectively, a block with grid coordinate \((b_x, b_y)\) inside a grid of dimension \((G_x, G_y)\) has a block ID equal to

\[
\text{block}_{id} = b_x + b_y G_y. \tag{2.2}
\]

Threads in the same block can synchronize themselves and they can communicate through a shared memory space. However, blocks in the same grid cannot be synchronized and they are executed independently.

The fastest memory of the device is the shared memory and the programmer should exploit it as much as possible. The following standard strategy should be adopted when programming with CUDA in order to maximize the performance:

- Load data from one of the device memories into the shared memory;
- Synchronize all the threads in the block, so that each thread can safely read the shared memory;
- Process the data in the shared memory;
- Write the results from shared memory back to the device memory.

The big challenge when writing a program on the GPU is to find how to split the job among all the threads. Normally, one thread is created for each result we need to compute. For example, to update a vector with dimension 100, we will create 100 threads, and each of them will compute its corresponding entry. It is the responsibility of the device to schedule all the threads and to decide how many threads can be executed in parallel.

**CUDA Hardware Implementation**

The programming model described in last section is implemented in the device by using a set of multiprocessors. In turn, each multiprocessor contains a set of processors able to execute the same instruction on different data at any given clock cycle. A grid of thread blocks is executed by letting each multiprocessor process batches of blocks, one batch after the other. The blocks that are processed by one multiprocessor in one batch are called active blocks and each of them is split into groups of threads called warps. Each warp has the same number of threads, referred to warp size, and it is executed by the multiprocessor in a SIMD fashion. The warps of the active blocks are called active warps and they are scheduled by a thread.
Figure 2.15: Multiprocessor architecture. A multiprocessor is organized as a set of processors able to execute the same instruction on different data at any given clock cycle.

A scheduler that periodically switches from one warp to another (Figure 2.15). With this hardware implementation, the maximum number of threads that are executed in parallel at a given time is given by the product between the number of multiprocessors and the size of the warps.

CUDA Memory Model

Figure 2.16 shows the different kinds of memory of CUDA. The constant and texture memories are read-only memories and they can be accessed by all the thread blocks. Since they are cached, no special access pattern is needed in order to improve their memory bandwidth. The total amount of constant memory is 64kB, and therefore it is useful only to store small amounts of data. Texture memory can be one- or two-dimensional; a 1D texture has a maximum width of $2^{27}$ 32-bit elements (512 MB), while the maximum size of a 2D texture is $(2^{16}, 2^{15})$ 32-bit elements. Texture memory is particularly attractive because its cache is optimized for 2D spatial locality, and therefore it can be exploited to store two dimensional matrices whose entries are accessed frequently.

The global memory is not cached and therefore is important to know how it must be accessed. Global memory guarantees maximum memory bandwidth when the addresses simultaneously accessed by each thread of a half-warp (i.e. the first or second half of a warp) are arranged so that the memory access are coalesced into a single, contiguous, aligned memory access [NVI07b].
2.3 General-Purpose computing on Graphics Processing Units (GPGPU)

Figure 2.16: The CUDA Memory Model. There are four kinds of memories that can be exploited, namely Global Memory, Texture Memory, Constant Memory and Shared Memory

Moreover, the size of the type of the accessed variables must be 4, 8 or 16 bytes and these must be aligned to 4, 8, or 16 bytes, respectively. Formally, thread number \( i \) should access the address

\[
\text{halfWarpBaseAddress} + i
\]

where \( \text{halfWarpBaseAddress} \) is a pointer to a variable of type \( T \), where \( T \) meets the size and alignment requirements discussed above. Moreover, \( \text{halfWarpBaseAddress} \) should be aligned to \( 16 \times \text{sizeof} \ (T) \) bytes. Figure 2.17 shows some examples of coalesced and non-coalesced accesses to the global memory.

The shared memory is the fastest memory offered by the GPU. Each multiprocessor has its own shared memory: to each block is assigned a different portion of the multiprocessor’s shared memory and therefore the write operations are local to a single block. Since the CPU cannot access the shared memory, a good strategy is to let each thread of a block load data from the device memory to the shared memory. Once the data is loaded, threads can access their local shared space with the best possible performance.

In order to investigate the behavior of the different types of memories discussed until now, we return to the Dense Matrix Vector multiplication example. Each block of threads in the grid is responsible to compute one entry of \( r \). This means that all threads in a block compute the dot product between a row of \( M \) and the vector \( x \). The first step is to load the corresponding entries of \( M \) and \( x \) into the shared memory of the block. Starting from the pointer to the beginning of the current row, each thread loads a consecutive element of the row. If this pointer is 64-byte \( (16 \times \text{sizeof} \ (\text{float})) \) aligned coalesced reading is ensured. At this point, each thread is responsible to multiply one entry of \( M \) with one entry of \( x \), storing the result in a shared array.
At the end, all intermediate results in the shared array are summed together and one of the threads writes the result back to the device memory.

If the dimension of the rows of matrix $M$ is a multiple of 16, the accesses to the device memory are ensured to be coalesced for each row. However, if this is not the case, the right access pattern can be followed for the first row only (Figure 2.18).

We test the Dense Matrix Vector multiplication routine storing the matrix $M$ either in the texture memory or in the global memory. Since vector $x$ is small enough, it is stored in the constant memory. The time needed to move the data between the CPU and the GPU is removed by measuring the timing for 128 calls of the routine. We use the CUBLAS [NVI07a] cublasSgemv routine as reference for the best optimized implementation.

Figure 2.19 shows that by accessing the global memory in a coalesced way we obtain better result than fetching data from the texture memory (blue and green plot respectively). However, we managed to speedup the Dense Matrix Vector multiplication by a factor of almost 8 (in Section 2.2 we reached more or less the same speedup exploiting multiprocessing architecture). This improvement is not really satisfactory. As explained in [NVI07b], GPU programming is...
2.3 General-Purpose computing on Graphics Processing Units (GPGPU)

Figure 2.18: Memory accesses for the Dense Matrix Vector multiplication routine. If the dimension of matrix $M$ is not a multiple of 4 (bottom), the accesses are not coalesced.

well suited for addressing problems with high arithmetic intensity, i.e. the ratio of arithmetic operations to memory operations. The arithmetic intensity of the Dense Matrix Vector multiplication is quite low, and therefore the inefficient memory operations deteriorate the performances of the overall process. Moreover, when the dimension of the problem doesn’t fit anymore the hardware specification of the GPU, we notice a further decrease of speedup (Figure 2.19 when matrix dimension changes from 7168 to 8192 with CUBLAS cublasSgemv routine).

Figure 2.19: Timing for the Dense Matrix Vector multiplication. If coalesced accesses are ensured, the global memory yield to the best performance.

Figure 2.20 shows the behavior of our routines when the data of $M$ is not aligned. By using texture memory, the results are almost identical to the aligned case (green plot). This is explained by the fact that texture memory does not require any specific alignment and in this
case performs even better than the highly optimized cublasSgemv routine. By using global memory, the routines become four times slower. Each thread needs extra time to load data from the device memory, and therefore the arithmetic intensity is even worse than before. We draw the conclusion that each kind of operation needs to exploit the best model it fits. If memory alignment can be ensured, using the global memory is the best solution. However, if memory alignment cannot be ensured, texture memory is a good alternative.

![Figure 2.20: Timing for the Dense Matrix Vector multiplication. When the rows of matrix $M$ are not aligned, texture memory is the best alternative.](image-url)
Sparse Linear Systems

In this chapter we describe the mathematical background of some common applications in Computer Graphics. In the next chapters, these applications are used to evaluate the benefits achieved by exploiting parallel architectures.

The ability to solve linear systems of equations is a key requirement for many of today’s scientific problems. Depending on the nature of the problem, the associated system can have properties that can be exploited in order to save computational time and memory usage. One of the most common property shared by many applications in Computer Graphics is sparsity, i.e. when the majority of the coefficients associated to the linear system is equal to zero. Sparse systems are typically derived from linear piecewise approximations of continuous surfaces. In this document we refer to linear system as a collection of linear equations involving the same set of variables. Moreover, the system is expressed in matrix notation

$$Ax = b; \quad A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}; \quad b = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}; \quad x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

where $A$ is the coefficient matrix, $b$ is the right-hand-side and $x$ the unknown vector. It is not surprising that the coefficient matrix $A$ associated to a sparse linear system is sparse, that is, populated primarily with zeros. In order to efficiently deal with this kind of matrices, many different data structures have been developed ([Sil05], [SUC05]) and for this reason sparse linear systems have become very attractive. The idea common to all these sparse representations is to store only the non-zero values with relative information about their position in the matrix.
Some representations are designed with the aim of optimizing the performance of numerical kernels (like Sparse Matrix Vector multiplication) or to minimize the memory consumption and the access time. A typical sparse matrix has a random sparse occupancy, that is, the non-zero entries are randomly distributed over the whole range of coordinates (Figure 3.1). Usually, this does not affect how the matrix is stored but it is important to keep in mind that dedicated data structures for well known sparsity patterns can be designed in order to guarantee optimality. In the next section we will introduce some of the most used data structures for storing sparse matrices.

3.1 Sparse Matrix Representation

A sparse matrix representation usually satisfies some basic requirements, for instance:

- minimize the memory consumption needed to store the information carried by the matrix.
- avoid all the unnecessary operations involving the zero entries of the matrix,

We will see some storage techniques for sparse matrix that fulfill both requirements.

3.1.1 The Coordinate Storage (COO)

The Coordinate Storage (COO) is the easiest and the more intuitive storage technique. It consists of three arrays: vals, col_ind and row_ind. vals contains the non-zero entries of the matrix in row-major order while col_ind and row_ind contain the columns and row indices of these values. Unfortunately, due to its simplicity, this technique is also inefficient, because the amount of information needed to store the matrix can be further reduced.
3.1 Sparse Matrix Representation

3.1.2 Compressed Row Storage (CRS)

The Compressed Row Storage is the most used storage technique for sparse matrices. It is simple, it reduces the quantity of memory needed to store the information and allows for an easy implementation of parallel algorithms. The non-zero values of the matrix are stored in the $vals$ array. The column indices of these values are stored in the $col_ind$ array. In a third array, $row_start$, pointers to the positions in the array $vals$ corresponding to the beginning of a new row are stored.

Some of the most useful properties of the Compressed Row Storage are:

1. The length of the array $vals$ is equal to the length of the array $col_ind$:

$$vals.size() = col_ind.size();$$  \hspace{1cm} (3.2)

2. The number of non-zero elements in the $i^{th}$ row is equal to

$$row_start[i + 1] − row_start[i];$$  \hspace{1cm} (3.3)
3 Sparse Linear Systems

3. In order to satisfy the second property, the length of the array \texttt{row\_start} is equal to the length of array \texttt{vals} plus one and its last entry is equal to the number of non-zero elements in the matrix:

\[
\text{row\_start\_size()} = \text{vals\_size()} + 1 \quad \text{and} \quad \text{row\_start\_last()} = \text{vals\_size()}.
\] (3.4)

Suppose we want to store a square matrix with dimension \(N \times N\) with \(N_{NZs}\) non-zero elements: the Compressed Row Storage is quite efficient because the memory used to store the entire matrix is \((2 \cdot N_{NZs} + N + 1)\) instead of \(N^2\) (storing all values) or \((3 \cdot N_{NZs})\) (Coordinate Storage). CRS is used as standard representation for the rest of this document. There is a very similar storage technique called Compressed Column Storage (CCS). The idea is almost the same that CRS, with the difference that instead of storing the values in a row-major order and storing pointers to the beginning of rows, the non-zero entries are stored in a column-major order and pointers refer to the beginning of columns. These two techniques are almost equivalent, and is up to the developer to choose which one is best suited for a specific situation.

3.1.3 Block Compressed Row Storage (BCRS)

The Block Compressed Row Storage is quite similar to the CRS. The main difference is that instead of storing the single non-zero entries of the matrix, we store blocks of values. There always are three arrays (\texttt{vals}, \texttt{col\_ind} and \texttt{row\_start}) but in this case they refer to blocks of values in the matrix (generally blocks of \(2 \times 2\) or \(4 \times 4\) dimension). This technique requires more memory when the sparsity pattern of the matrix is completely random, but, on the other hand, it can improve the ratio between memory throughput and numerical computation for some operations.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{bcrs.png}
\caption{Block Compressed Row Storage (BCRS) for Sparse Matrices}
\end{figure}

\[\ldots \quad 0 \quad 1 \quad 9 \quad 0 \quad 0 \quad 8 \quad 0 \quad 0 \quad 3 \quad 1 \quad 5 \quad 0 \quad 0 \quad 0 \quad 7 \quad 9 \quad 0 \quad 0 \quad 4 \quad 0 \quad 0 \quad \ldots\]

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
vals & 7 & 0 & 0 & 3 & 5 & 0 & 0 & 0 & 0 & 2 & 7 & 0 & 0 & 4 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\
\hline
\hline
col\_ind & 0 & 2 & 3 & 1 & 2 & 3 & 1 & 3 & 0 & 1 & 3 \\
\hline
row\_start & 0 & 3 & 6 & 8 & 11 \\
\hline
\end{tabular}
3.2 Sparse Linear System for Surface Parameterization

The parameterization of a surface is a bijective (one-to-one) mapping from the surface to a suitable region. In general, the surface is embedded in $\mathbb{R}^3$ and the target region is a subset of $\mathbb{R}^2$. In this section we explain how a particular mapping from an approximated surface embedded in $\mathbb{R}^3$ to a subset of $\mathbb{R}^2$ can be reduced to solving a sparse system of linear equations.

A Triangle Mesh is a piecewise linear approximation of a surface embedded in $\mathbb{R}^3$. It is characterized by a triplet $M = (V, E, T)$ where $V$ denotes the set of vertices, $E$ denotes the set of edges and $T$ denotes the set of faces (in our case triangles). We call the subset $V_B \subset V$ the set of vertices lying on the boundary of the mesh and $V_I \subset V$ the set of vertices in the interior of the mesh. Moreover, let $N_i$ denote the $I$-ring, the set of edge neighbors, of vertex $v_i \in V$.

A parameterization $\psi$ of $M$ is a continuous one-to-one mapping linear on each triangle $T_i \in T$ from $M$ to a region $\Omega \subset \mathbb{R}^2$. Formally:

$$\psi : M \subset \mathbb{R}^3 \rightarrow \Omega \subset \mathbb{R}^2$$  \hspace{1cm} (3.5)

$\psi$ can be formulated to guarantee some properties of the parameterized surface, like no distortion in angles (conformal mapping) or no distortion in areas (equiareal mapping). One can show that if a mapping is conformal it is also harmonic. Harmonic mappings, however, have the big advantage over conformal mappings of being simpler to compute. Harmonic mapping can be approximated by using a finite element method based on linear elements [FH05] as follows:

1. First, choose a fixed mapping for the boundary of $M$, i.e. $\psi(V_B) = \psi_B$. (usually for disk-like surfaces the boundary is mapped to an unit circle);

2. then, find a piecewise linear parameterization $\psi : M \rightarrow \Omega \subset \mathbb{R}^2$ such that the Dirichlet energy

$$E_D(\psi) = \frac{1}{2} \int_M \| \nabla_M \psi \|^2$$  \hspace{1cm} (3.6)
The Dirichlet energy described by Equation (3.6) can be seen as a measure of the deformation caused by the parameterization, and thus it must be minimized. For a triangle \( T_i \in T \) with vertices \( v_0, v_1 \) and \( v_2 \) and corresponding angles \( \theta_0, \theta_1 \) and \( \theta_2 \) we can show that [FH05]:

\[
2 \int_{T} || \nabla_T \psi ||^2 = \cot \theta_0 || \psi(v_1) - \psi(v_2) ||^2 + \\
\cot \theta_1 || \psi(v_0) - \psi(v_2) ||^2 + \\
\cot \theta_2 || \psi(v_0) - \psi(v_1) ||^2
\]  

(3.7)

and therefore we can find the critical points \( \psi_i = \psi(v_i), v_i \in V_I \) of equation (3.6) by setting

\[
\nabla \psi_i E_D(\psi) = \sum_{j \in N_i} w_{ij} (\psi(v_j) - \psi(v_i)) = 0 \quad \forall v_i \in V_I
\]  

(3.8)

where

\[
w_{ij} = \cot \alpha_{ij} + \cot \beta_{ij}
\]  

(3.9)

and the angles \( \alpha_{ij} \) and \( \beta_{ij} \) are shown in Figure 3.6. Equations (3.8) are the necessary condition for the critical points and, as shown in [AKS03], they yield the following linear system:

\[
\psi(v_i) \sum_{v_j \in N_i} w_{ij} - \sum_{v_j \in N_i \cap V_I} w_{ij} \psi(v_j) = \sum_{v_j \in N_i \cap B} w_{ij} \psi(v_j), \quad \forall v_i \in V_I.
\]  

(3.10)

In matrix notation, system (3.10) becomes \( Ax = b \) with

\[
A = \{ a_{ij} \}_{v_i, v_j \in V_I}, \quad a_{ij} = \begin{cases} 
\sum_{v_k \in N_i} w_{ik} & i = j \\
-w_{ij} & v_j \in N_i \\
0 & \text{otherwise}
\end{cases}
\]  

(3.11)
3.2 Sparse Linear System for Surface Parameterization

Figure 3.7: Triangular meshes with their respective matrices derived from the harmonic parameterization. As we can see, the matrices are sparse and symmetric.

and

\[
x = \begin{bmatrix}
\psi(v_0) \\
\psi(v_1) \\
\vdots \\
\psi(v_{n-1})
\end{bmatrix}, \quad 
b = \begin{bmatrix}
\sum_{v_j \in N_0 \cap V_B} w_{0j} \psi(v_j) \\
\sum_{v_j \in N_1 \cap V_B} w_{1j} \psi(v_j) \\
\vdots \\
\sum_{v_j \in N_{n-1} \cap V_B} w_{n-1,j} \psi(v_j)
\end{bmatrix}.
\] (3.12)

Computing the harmonic map for each vertex of the surface \( M \) is hence reduced to solving this linear system with respect to the unknown vector \( x \). Matrix \( A \) has dimension \( |V_I| \times |V_I| \) and is sparse; the number of non-zero elements in the \( i^{th} \) row is equal to \( |N_i| + 1 \). Matrix \( A \) is symmetric and positive definite and, as we will see in the next section, particular solvers in order to efficiently find the solution of the equations can be used. Normally, we deal with meshes of thousands or hundreds of thousands of vertices and therefore efficient algorithms for the associated linear systems are necessary. Figure 3.7 shows a triangular mesh with different resolutions and the associated sparse matrices derived from the harmonic mapping theory discussed above.
3 Sparse Linear Systems

3.3 Sparse Linear System for FEM Simulation

In this section, the simulation of deformable bodies is discussed. Starting from the continuum elasticity theory, the Finite Element Method (FEM) is used to solve the governing partial differential equations. First, we discuss the static deformation setup, then the dynamic FEM equations and their associated linear systems are investigated.

3.3.1 Linear Static Deformation

The motion of a deformable body is simulated by describing a displacement field \( u(x) \). The deformed body is obtained by displacing each point \( x \) in the undeformed body to \( x = x + u(x) \). Following the finite element method, the body is discretized into a finite number of elements (in our case hexahedra) and \( u(x) \) is defined only on the nodes of such elements. Once the displacement values for each node of a hexahedral element are found, they are linearly interpolated within the element by

\[
\mathbf{u}(x) = \mathbf{H}_e(x)\hat{\mathbf{u}}
\]  

(3.13)

where \( \mathbf{H}_e(x) \) is a matrix containing the shape functions of element \( e \) and \( \hat{\mathbf{u}} \) are the values of the displacement field on the vertices of \( \epsilon_e \). In case of hexahedral elements, the shape functions correspond to the weights of a trilinear interpolation.

The values for \( \hat{\mathbf{u}} \) are computed by applying the elasticity theory. First, by using Cauchy’s linear strain tensor, the strain \( \epsilon \) within an hexahedral element is computed by

\[
\epsilon = \mathbf{B}_e \hat{\mathbf{u}}
\]  

(3.14)

where \( \mathbf{B}_e \) is a precomputed matrix containing the partial derivatives of the shape functions. Following Hooke’s law of elasticity the stress \( \sigma \) within an hexahedral element is

\[
\sigma = E\epsilon = E\mathbf{B}_e \hat{\mathbf{u}}.
\]  

(3.15)

Matrix \( E \) is an elastic modulus and in our case only depends on two scalar values, namely Poisson’s ratio and Young’s modulus. The strain energy depends on the stresses and strains within an element and it is used to find the elastic forces associated to each node of the element. By using Cauchy’s linear strain, one can prove that

\[
(\mathbf{V}_e \mathbf{B}_e^T \mathbf{E} \mathbf{B}_e) \hat{\mathbf{u}} = \mathbf{K}_e \hat{\mathbf{u}} = \mathbf{f}_e
\]  

(3.16)

where \( \mathbf{V}_e \) is the volume of the hexahedral element and \( \mathbf{K}_e \) is the stiffness matrix. By collecting the stiffness matrices of all the elements of the deformable body, the stiffness matrix \( \mathbf{K} \) is built.

The general setup for the static deformation problem is:

- Discretize the body into elements (tetrahedral, hexahedral, polyhedral [MKB+08]);
- Setup the stiffness matrix \( \mathbf{K} \) by combining all the element matrices \( \mathbf{K}_e \);
- Setup the elastic forces \( \mathbf{f} \) (gravity, constraints, interaction forces, ...);
- Solve the linear system \( \mathbf{K} \mathbf{u} = \mathbf{f} \) with respect to the displacement vector \( \mathbf{u} \);
- Displace points \( x_i \) of the body by \( \mathbf{u}(x) \).
3.3 Sparse Linear System for FEM Simulation

3.3.2 Linear Dynamic Deformation

When the dynamics of a body are simulated, the position vector \( x \) changes over the time, i.e. \( x(t) \). In this case, the Lagrangian equation of motion

\[
M \ddot{x} + C \dot{x} + K(x - x_0) = f
\]

(3.17)
governs the behavior of the body. Matrix \( M \) is the mass matrix, matrix \( C \) is the damping matrix and matrix \( K \) is the stiffness matrix derived in the previous section. Given \( M \), the damping matrix \( C \) is often computed by \( C = \alpha M + \beta K \) (Rayleigh damping).

By exploiting the fact that the velocity is the first derivative of the position, that is, \( v = \dot{x} \), the second order ODE (3.17) is split in two first order ODEs

\[
M \dot{v} + C v + K(x - x_0) = f
\]

(3.18)
\[
\dot{x} = v
\]

(3.19)

We use an unconditionally stable implicit Euler integrator to solve the first order ODEs (3.18) and (3.19). Therefore, the iterative schemes are

\[
x_{i+1} = x_i + \Delta t \dot{x}_{i+1} = x_i + \Delta t v_{i+1}
\]

(3.20)
\[
v_{i+1} = v_i + \Delta t \dot{v}_{i+1}
\]

(3.21)

By expanding Equation (3.21) we obtain:

\[
v_{i+1} = v_i + \Delta t M^{-1} (f - C v_{i+1} - K(x_{i+1} - x_0))
\]

\[
v_{i+1} = v_i + \Delta t M^{-1} (f - C v_{i+1} - K(x_i + \Delta t v_{i+1} - x_0))
\]

\[
v_{i+1} + \Delta t M^{-1} C v_{i+1} + \Delta t^2 M^{-1} K v_{i+1} = v_i + \Delta t^2 M^{-1} (f - C v_{i+1} - K(x_i + \Delta t v_{i+1} - x_0))
\]

\[
M v_{i+1} + \Delta t C v_{i+1} + \Delta t^2 K v_{i+1} = M v_i + \Delta t (f - K(x_i + \Delta t v_{i+1} - x_0))
\]

(3.22)

Equation (3.22) describes a system of linear equation \( A v_{i+1} = b \) with

\[
A = (M + \Delta t C + \Delta t^2 K) \quad \text{and} \quad b = M v_i + \Delta t (f - K(x_i + \Delta t v_{i+1} - x_0))
\]

(3.23)

With linearized elastic forces, solving the dynamics described by the Lagrangian equation of motion reduce to solve the linear system (3.23) each time step. Moreover, the stiffness matrix \( K \) is sparse. Since it is also constant, only the right-hand-side \( b \) changes during the simulation. However, under large rotations, linearized forces introduce unrealistic behaviors like growth in volume (Figure 3.8 (Left)). The next section introduces a method called Stiffness Warping to remove these artifacts.

3.3.3 Stiffness Warping

In [MDM+02] Müller et al. propose a method called Stiffness Warping to remove the artifacts caused by the linearized elastic forces. In [MG04] they improve this method by computing the rotational parts of the elements rather than the rotational parts of the vertices. In this approach,
the first step is to extract the element rotational parts $R_e$ of the deformed elements. Then, these elements are rotated back with $R_e^T x$, and a new displacement $R_e^T x - x_0$ with respect to the initial position is computed. The elastic forces associated to these displacements are $K_e (R_e^T x - x_0)$ and they are rotated back to the initial displacement, i.e. $f = R_e K_e (R_e^T x - x_0)$.

In the dynamic setup, the stiffness matrix $K$ is split into two new stiffness matrices $K'$ and $K''$. By denoting $R_e$ the per-element rotational part, these two new matrices are computed by

$$K' = \sum_e R_e K R_e^T$$

$$K'' = \sum_e R_e K$$

and the linear system (3.22) becomes

$$(M + \Delta t C + \Delta t^2 K') v_{i+1} = M v_i + \Delta t (f - K' x_i + K'' x_0)$$  (3.24)

The benefits of using stiffness warping are shown in Figure 3.8 (Right), where the artifacts due to large rotations are no longer present. However, the warped stiffness matrices have to be recomputed every time step. This implies that both the entries of the coefficient matrix and of the right-hand-side of linear system (3.24) change during the simulation.
Sparse Linear Solver

In the previous chapter we saw how the parameterization of a surface and the simulation of deformable bodies reduce to solve a sparse linear systems. In this chapter we investigate two important methods, the Conjugate Gradient method and the Multigrid method, that allow us to efficiently solve these sparse linear systems. Moreover, these methods are then improved by exploiting parallel resources.

There are two big families of linear solvers: Direct Solvers and Iterative Solvers. Direct solvers are solvers that compute an exact solution of the problem with a finite sequence of operations and they are particularly indicated for dense linear system. Iterative solvers are solvers that, starting from an initial guess, find a sequence of approximations to the solution and they fit particularly well to sparse systems. The solvers we investigate in this document belong to the second class.

There are two important measurements for an approximation of the real solution: the error and the residual. The error \( e \) associated with an approximation \( \tilde{x} \) of the exact solution \( x \) of a linear system \( Ax = b \) is defined as

\[
e = x - \tilde{x}.
\]  

(4.1)

Unfortunately, the error is as inaccessible as the exact solution, so more general measurements are considered. The residual \( r \) associated with an approximation \( \tilde{x} \) of the exact solution \( x \) of a linear system \( Ax = b \) is defined as

\[
r = b - A\tilde{x}.
\]  

(4.2)

The relationship between the error and the residual is called the residual equation and forms the base for some important linear solvers (for example the Multigrid method explained in Section 4.2). The residual equation associated with an error \( e \) and a residual \( r \) is defined as

\[
Ae = r.
\]  

(4.3)

\footnote{A \textit{linear solver} is a solver for system of linear equations.}
Understanding the residual equation is quite straightforward: It basically tells us that if an approximation $\tilde{x}$ is equal to the exact solution (the error $e = 0$) the approach to the solution is completed ($r = 0$). In order to define a scalar measurement for the error and for the residual, the most important vector norms used in the literature are the \textit{Euclidean} or \textit{2-norm} and the \textit{maximum norm} defined, respectively, by
\begin{equation}
\|e\|_2 = \left[\sum_{i=0}^{n-1} e_i^2\right]^{1/2} \quad \text{and} \quad \|e\|_\infty = \max_{0 \leq i < n-1} |e_i|.
\end{equation}

### 4.1 Conjugate Gradient (CG)

The \textit{Conjugate Gradient Method} (or CG) is one of the most famous method for solving large sparse system of linear equations where the associated coefficient matrix is symmetric and positive definite [She94]. The foundation of the CG is the \textit{quadratic form}. The \textit{quadratic form} of a vector $x$ is the scalar, quadratic function
\begin{equation}
f(x) = \frac{1}{2}x^TAx - b^Tx + c,
\end{equation}
where $x$ and $b$ are vectors, $c$ is a scalar and $A$ is a matrix. The gradient $\nabla f(x)$ of the quadratic form is computed by
\begin{equation}
\nabla f(x) = \frac{1}{2}A^Tx + \frac{1}{2}Ax - b
\end{equation}
and therefore if $A$ is symmetric, i.e., $A = A^T$, we have
\begin{equation}
\nabla f(x) = Ax - b \quad \text{and} \quad r = b - Ax = -\nabla f(x).
\end{equation}

Finally, the following relationship is derived
\begin{equation}
\nabla f(x) = 0 \quad \text{if and only if} \quad Ax = b.
\end{equation}

As shown in Equation (4.8), the solution of the system $Ax = b$ is a critical point of $f(x)$. Moreover, if the matrix $A$ is positive-definite, this critical point is a minimum (red point in Figure 4.1). In other words, in order to solve $Ax = b$, it is sufficient to find the minimum of $f(x)$. To find this minimum we start from an initial guess $x_0$ and follow a direction $d_0$ that takes us to point $x_1$, closer to the solution. In $x_1$ we follow another direction $d_1$, resulting in $x_2$. This strategy is repeated until the residual is small enough for our purpose, or when a maximum number of iterations is reached. The process is described by the following recursive relationship
\begin{equation}
x_{i+1} = x_i + \alpha_i d_i \quad \text{with} \quad \alpha_i \in \mathbb{R}.
\end{equation}

The crucial part is to define for each iteration the direction $d_i$. Methods such as the \textit{Steepest Descent method} use the opposite direction of the gradient of $f$ (i.e. where $f$ decrease most
4.1 Conjugate Gradient (CG)

**Figure 4.1:** Example of contours of a quadratic form for a $2 \times 2$ matrix (each ellipsoidal curve has constant $f(x)$). Left: Steepest Descent Method; the directions $d_i$ are opposite to the gradient. Right: Conjugate Gradient Method; directions $d_0$ and $d_1$ are A-orthogonal.

quickly) producing a zig-zag path toward the solution (Figure 4.1 (Left)). The main idea of CG is to avoid following the same direction many times; instead it takes the right step the first time a direction is used (Figure 4.1 (Right)).

Instead of choosing the directions $d_i$ orthogonally (as in the case of the Steepest Descent method), the CG computes a set of directions that are A-orthogonal. Two vectors $d_i$ and $d_j$ are A-orthogonal (or conjugate) if

$$d_i^T A d_j = 0. \quad (4.10)$$

Starting from an initial guess, the first direction followed by the CG is $d_0$. The next direction is computed ensuring A-orthogonality with respect to the previous one, that is

$$d_0^T A d_1 = 0. \quad (4.11)$$

Direction $d_2$ however must be A-orthogonal to all previous directions ($d_0$ and $d_1$). This implies that in order to compute the direction $d_i$ for the $i^{th}$ step, we need to store in memory all the previous directions $d_0, d_1, ..., d_{i-1}$. Fortunately, the CG avoids this problem by taking the opposite direction of the gradient, i.e. the residual, as first direction $d_0$ (this is the reason for the name Conjugate Gradient). Therefore, by exploiting the properties of gradient vectors, we need only to store the last direction in order to compute the new one.

The step length $\alpha_i$ associated to the direction $d_i$ is chosen in order to minimize the value of $f$ along $d_i$, and is computed using a line search method by setting the directional derivatives of $f$ to zero. In the next section we show a possible implementation, and the interested reader can refer to [She94] for a complete and detailed explanation.
4 Sparse Linear Solver

4.1.1 The Algorithm

The algorithm for the CG method is listed in [She94]. Since we are more interested in how the algorithm can be improved by exploiting some parallel architectures, we list an implementation (Figure 4.2) and we investigate the routines that need to be parallelized.

In order to implement the CG method only three basic algebra operations are sufficient (listed in Table 4.1).

<table>
<thead>
<tr>
<th>Vector Update</th>
<th>update</th>
<th>( \text{update}(a, b, c, r) \rightarrow r = a + bc \text{ with } b \in \mathbb{R} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot Product</td>
<td>dot</td>
<td>( \text{dot}(v_1, v_2) \rightarrow v_1^T v_2 )</td>
</tr>
<tr>
<td>Sparse Matrix Vector Multiplication</td>
<td>SpMV</td>
<td>( \text{spmv}(A, d, x) \rightarrow x = Ad )</td>
</tr>
</tbody>
</table>

Table 4.1: Three basic algebra operations needed to implement the Conjugate Gradient solver.

The routine takes as inputs the coefficient matrix \( A \), the right-hand-side vector \( b \), a maximum number of iterations \( i_{\text{max}} \) and an error tolerance \( \epsilon \). During the iterations, it updates the solution vector \( x \) initialized with the initial guess and it stops either when the number of iterations is greater than \( i_{\text{max}} \) (i.e. \( i \geq i_{\text{max}} \)) or when the approximation is close enough to the solution (i.e. \( rr_{\text{new}} \leq \epsilon^2 rr_0 \)). \( rr_{\text{new}} \) and \( rr_0 \) are scalar values of the squared 2-norm of the residual \( r \) defined.
Lines 7 and 8 compute the opposite direction of the gradient of $f$ in $x_0$ (which, from Equation (4.7), is equal to the residual). This direction is used to update $x$ during the first iteration. Line 20 computes $\alpha_i$, i.e., how long the current step needs to be in order minimize $f$ along $d_i$ and line 22 updates the solution vector $x$. Line 28 updates $d_i$ computing a new direction $d_{i+1}$, that, from the theory of the conjugate gradient, is $A$-orthogonal to $d_i$.

Figure 4.3 shows the conjugate gradient method applied to the linear system for the harmonic parameterization derived in Section 3.2. At each iteration vector $x$ converges to the solution, and therefore the visual appearance of the textured surface becomes more plausible.

**Figure 4.3:** Iteration steps of the Conjugate Gradient solver for the linear system derived by the harmonic parameterization.

### 4.1.2 Parallelizing the Conjugate Gradient

Parallelizing the update operation does not require any special strategy since each element of the corresponding vectors is independent of the others. The dot operator on the other hand
requires some sort of inter-data or inter-processor communication, since it takes as input two vectors and returns a scalar. In fact, when the job is shared between multiple processors (as in the case of shared-memory computers) or multiple data (as in the case of SIMD instructions) at a certain point we need to collect the information processed by the single processing units. Fortunately, all the parallel architectures we investigate provide some reduction mechanism, i.e. strategy to combine the final results of each batch of jobs. The last operation that has to be taken into consideration is the Sparse Matrix Vector multiplication (SpMV). SpMV is the most expensive routine of the CG; in our tests it takes about 90% of the computational time needed to solve the system.

The next sections present how the update, dot and SpMV operations can be parallelized using the parallel architectures discussed in Chapter 2 and we discuss the benefits, if any, achieved by exploiting these kind of resources.

**Streaming SIMD Extension**

**update** In order to parallelize the update operator using SSE intrinsics we need 4 registers, namely vA, vB, vC and vR. The register vB contains 4 copies of the scalar value b and registers vA and vC are used to load the data from arrays a and c respectively. After the computation, the updated values are stored in register vR where they will be stored back to main memory. In a vector processor with vector length equal to 4 (SSE_VEC_SIZE = 4) four elements are simultaneously processed by:

- Loading the values from the arrays a and c in main memory into registers vA and vC;
- Multiplying registers vC and vB;
- Summing the result of the multiplication with register vA and storing the result in register vR;
- Storing the values in register vR back to the array r in main memory.

By using SSE intrinsics, the example code for the update operator is:

```c
void update(float* a, float b, float* c, float* r, int n)
{
    __m128 vA, vB, vC, vR;
    vB = _mm_set_ps1(b);
    for(int i = 0; i < n / SSE_VEC_SIZE; i++)
    {
        vA = _mm_load_ps(a); // Load 128-bits from memory location pointed by a to vA
        vC = _mm_load_ps(c); // Load 128-bits from memory location pointed by c to vC
        vR = _mm_add_ps(vA, _mm_mul_ps(vB, vC)); // Compute vR = vA + vB * vC
        // Store 128-bits from register vR to memory location pointed by r
        _mm_store_ps(r, vR);
        a += SSE_VEC_SIZE; // Increment pointers a, c and r
        c += SSE_VEC_SIZE; // ensuring 16-byte memory alignment
        r += SSE_VEC_SIZE;
    }
}
```
The pointers $a, c$ and $r$ point to an address in main memory aligned to 16 bytes; in our configuration $\text{SSE\_VEC\_SIZE} = 4$ and $\text{sizeof(float)} = 4$ and therefore the memory reads are aligned at each iteration (Lines 15, 16 and 17). Figure 4.4 shows the speedups for the update operator achieved using SSE intrinsics.

![Figure 4.4: Speedup achieved by using SSE intrinsics for the dot operator.](image)

Since this is a low level parallel paradigm the speedups are independent of the size of the problem. Usually, when algorithms are parallelized extra routines are called (for example forking or joining parallel threads) and the overhead needed for such calls becomes especially visible when dealing with small data set. However, by exploiting XMM registers, we were able to run the update operation two and half times faster than its sequential implementation. At each iteration the result of the updated packet of entries (register $vR$) has to be stored back to main memory and therefore we do not reach the speedup of three achieved in the Dense Matrix Vector multiplication in Section 2.1, where the local results are stored back only when one entire row has been processed.

**dot**  As introduced above, the dot operator is more insidious to parallelize because it involves a reduction. In this case we need three registers: $vV1$, $vV2$ and $vRed$. The first two registers are responsible for loading the values from $v_1$ and $v_2$; the third one acts as an accumulator. After having initialized all the elements of $vRed$ to zero, the following steps are repeated:

1. Load values from arrays $v_1$ and $v_2$ into the registers $vV1$ and $vV2$;
2. Multiply the registers $vV1$ and $vV2$;
3. Update the register $vRed$ by summing the result of the multiplication;
4. Update the pointers to main memory and return to step 1.
4 Sparse Linear Solver

After the iterations, we have to sum, or reduce, all the elements in the $vAcc$ register in order to have the scalar value for the dot product.

```c
float dot(float* v1, float* v2) {
    __m128 vV1, vV2M;
    union u {
        __m128 v;
        float[4] f;
    } vRed;
    vRed.v = _mm_set_ps1(0.0f);
    for(int i = 0; i < n / SSE_VEC_SIZE; i++) {
        vV1 = _mm_load_ps(v1); // Load 128-bits from memory location pointed by v1 to vV1
        vV2 = _mm_load_ps(v2); // Load 128-bits from memory location pointed by v2 to vV2
        // Compute temporary values for the dot product
        vV1 = _mm_add_ps(vV1, _mm_mul_ps(vV1, vV2));
    }
    // Reduction (supposing SSE_VEC_SIZE = 4)
}
```

At Line 3 the union structure for the reduction operation is declared. During the computation at Line 18 the SSE data type member is used and, at the end, we add all the single values of the float array (Line 25). With this strategy, comparing to the sequential implementation, the dot operator runs about three time faster (Figure 4.5).

![DOT Speedup Chart](chart)

**Figure 4.5:** Speedup achieved by using SSE intrinsics for the dot operator.

**SpMV** We now investigate the Sparse Matrix Vector multiplication. The algorithm for SpMV using the CRS matrix representation is shown in Algorithm 2.
Algorithm 2: \texttt{SpMV} with CRS format

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{for} $i = 0$ to $N - 1$ \textbf{do}
\State \hspace{1em} $red = 0$
\State \hspace{1em} \textbf{for} $j = \text{row}_\text{start}[i]$ to $\text{row}_\text{start}[i + 1] - 1$ \textbf{do}
\State \hspace{2em} $c = \text{col}_\text{ind}[j]$
\State \hspace{2em} $red = red + \text{vals}[j] \times x(c)$
\State \hspace{1em} \textbf{end for}
\State \hspace{1em} $r(i) = red$
\State \textbf{end for}
\end{algorithmic}
\end{algorithm}

The outer loop (Line 1) iterates over the rows of the matrix, while the inner loop (Line 3) iterates over the non-zero entries of the current row. Variable $c$ (Line 4) contains the column index of the entry $\text{vals}[j]$ and is used to fetch the suitable value for the multiplication from the vector $x$ (entry $\text{vals}[j]$ as coordinates $(i, c)$). In this case, SSE instructions could be used to multiply four non-zero entries in the array $\text{vals}$ with their respective values from vector $x$. However, as shown in Figure 4.6, the values of $x$ corresponding to consecutive entries in $\text{vals}$ are randomly distributed, and therefore the load operation is extremely expensive.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4_6.png}
\caption{Random memory accesses when multiplying a row of the matrix with the vector.}
\end{figure}

In order to locally linearize the reading operations, the BCRS format with blocks of dimension $4 \times 1$ (Section 3.1) is used. In this way, we can fetch a block of four consecutive values from vector $x$ and multiply it with the corresponding entries of the matrix (Figure 4.7).

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4_7.png}
\caption{With BCRS format, the read operations are locally linearized.}
\end{figure}
The best performance is achieved when the non-zero entries in the matrix are close together, i.e. a small number of zeros is needed to fill the blocks. In real world applications this usually does not appear. If the linear system we are solving derives from the harmonic parameterization setup, the sparseness occupancy of the matrix $A$ depends on the indexing of the vertices in the mesh. Randomly permuting these indices yields the worst case scenario, where on average we need one block for each entry, quadrupling the memory usage (Figure 4.8 (Left)). With a smart permutation (i.e. the Reverse Cuthill McKee (RCMK) reordering [CM69]) we obtain a band matrix, i.e. a sparse matrix whose non zero entries are confined to a diagonal band (Figure 4.8 (Right)).

Let $A_r$ be the matrix obtained by random permutation of the indices and $A_{db}$ the diagonal band matrix. It should be clear that the three linear systems $Ax = b$, $A_r x = b$ and $A_{db} x = b$ are equivalent. Our tests show that when $A_r$ is represented with a $4 \times 1$ BCRS format, we find on average 1.0003 non-zero elements per block, and the $\text{SpMV}$ operation becomes 2 times slower than the sequential implementation. Representing $A_{db}$ with the same format, yields an average of 1.43 non-zero elements per block and the $\text{SpMV}$ operation is 1.25 times slower. These deteriorations of performances are due to the increase of memory movements caused by the auxiliary zeroes needed for fulfill the blocks.

From these results, we conclude that this kind of parallel architecture is not suitable for the Conjugate Gradient. Without any speedup for the $\text{SpMV}$ routine, our solver runs almost equivalently
to a sequential implementation. However, we showed that exploiting SSE instructions can have important benefits for some other problem, like vector updating (Section 4.1.2), Dense Matrix Vector multiplication (Section 2.1) or when mixed with multiprocessors programming (Section 2.2).

**Shared Memory Programming**

We now investigate shared memory multiprocessing programming. With this architecture, each processing unit executes a portion of the job simultaneously.

**update** Referring to Table 4.1, the `update` operator is parallelized by letting each processor compute a subset of the entries of the resulting vector. Since each element is independent from the others, there is no need to synchronize the parallel threads or to define some sort of inter-thread communication. As shown in Section 2.2, OpenMP™ provides a compiler directive to split automatically a `for` loop between all the available processors and therefore the code for the parallel `update` operation becomes:

```c
void update(float* a, float b, float* c, float* r, int n)
{
    #pragma omp parallel for
    for(int i = 0; i < n; i++)
    {
        r[i] = a[i] + b * c[i];
    }
}
```

When the size of vectors `a`, `c` and `r` is large enough, the time needed to fork and join the parallel threads is negligible and the `update` routines scales almost perfectly with the number of processors, as shown in Figure 4.9. The additional speedup achieved in the case of large vectors is due to the L2 cache. Figure 2.9 shows the hardware configuration used for our tests. Using a single core, the amount of L2 cache available is 12 MB. Since the eight cores lie on two different chips, the amount of L2 cache is doubled and therefore it is more probable that the new data are fetched from cache instead of from main memory. Since the `update` operator fits all the requirements for an easy and scalable parallelization, it is straightforward to think about speeding up the computation for each single processor using SSE instructions. The entire job is split between the processors and each of them use the XMM registers for the local computation. With this strategy even on a 8-Cores machine the `update` operator results up to 13 times faster than its sequential implementation.
Figure 4.9: Speedup achieved by using OpenMP together with SSE intrinsics for the update operator.

As explained above, the dot operator involves a reduction. Each thread computes the dot product between two corresponding regions of the vectors $v_1$ and $v_2$. After the computation, each processor has its intermediate result stored in its private memory (Figure 4.10).

To sum these values (or to reduce them), we could use the shared main memory space. A shared variable initialized to zero is updated by each processor by adding their private results. However, in order to guarantee memory consistency, we have to serialize the accesses to the shared variable for each thread. Fortunately, the dedicated APIs for shared-memory programming provide us with optimized directives that implement the reduction operator. These directives are given as argument when a parallel region is defined; they take as inputs a variable $var$ and a binary operator $op$. They create a private copy of variable $var$ for each processor, namely...
4.1 Conjugate Gradient (CG)

\(\text{var}_0, \text{var}_1, ..., \text{var}_{n-1}\) and in the end they compute \(\text{var} = \text{var}_0 \text{ op} \text{ var}_1 \text{ op} ... \text{ op} \text{ var}_{n-1}\). Using this optimized directives, the pseudocode for the \text{dot} operator is:

```c
float dot(float* v1, float* v2, int n)
{
    float red = 0.0;
    #pragma omp parallel for reduction(+; red)
    for(int i = 0; i < n; i++)
    {
        red += v1[i] * v2[i];
    }
    return red;
}
```

Line 4 declares a reduction variable \(\text{red}\) and a reduction operator + with the OpenMP\textsuperscript{TM} directive \text{reduction}. Each processor will have its own copy of \(\text{red}\) initialized to zero and at the end these private variables will be summed together. Figure 4.11 shows the speedup achieved for the \text{dot} operator. With big vectors, the time needed to create and destroy the threads becomes negligible. However, the reduction needs to serialize some accesses to memory and therefore we don’t have a perfect scaling with respect to the number of parallel threads.

![Figure 4.11: Speedup achieved on a 8-Cores machine for the dot operator.](image)

\textbf{SpMV} The last and most important operation is the Sparse Matrix Vector multiplication (SpMV). How the job is partitioned between the processors depends on which representation is used to store the matrix. For example, with a CRS format (Section 3.1) the job is split along the rows of the matrix (similar to the Dense Matrix Vector multiplication as shown in Figure 2.11), while with the CCS format (Section 3.1) it is split along the columns. We use the CRS format, and therefore a different portion of rows of the matrix is assigned to each parallel thread. If the linear system derives from the harmonic parameterization discussed in Section 3.2, each row contains almost the same number of non zero elements, and therefore the job is well balanced between the processors. The parallel version of the SpMV operation becomes:
void SpMV_CRS(float* vals, int* col_ind, int* row_start, float* x, float* r, int n) {
    int c;
    float acc = 0.0f;
    #pragma omp parallel for private(acc, c)
    for(int i = 0; i < n; i++)
    {
        float acc = 0.0f;
        for(int j = row_start[i]; j < row_start[i+1]; j++)
        {
            c = col_ind[j];
            acc += vals[j] * x[c];
        }
        r[i] = acc;
    }
}

The accesses to the vector \( x \) depend on the sparse occupancy of the matrix (Lines 11 and 12). If the entries in a row are close together, there is a good chance that some of their respective elements in the vector will be fetched from cache. In this situation, the matrix reordering algorithms discussed above can bring to an important benefit. The respective speedups for the \( \text{SpMV} \) operation are shown in Figures 4.12. We can see that the performance of the \( \text{SpMV} \) reflects the number of processors used for the parallelization. Moreover, the reordering of the matrix leads to further improvements and in certain cases it doubles the speedup (Figure 4.12 with matrix dimension 352K).

![Figure 4.12: Speedup achieved by exploiting a 8-Core processor for the SpMV operator. Blue: without matrix reordering. Green: with matrix reordering.](image)

### Parallel CG

We are now ready to implement the Conjugate Gradient method. In order to test our CG solver, we uses as reference the GMM \( \text{cg} \) routine [GMM] for our sequential implementation and the Intel MKL \( \text{dcg} \) routine [Int07] for the parallel version. The linear systems are derived form the harmonic parameterization of the surfaces illustrated in Section 3.2. We tested the solver with and without matrix reordering. Figure 4.13 shows the speedups computed by comparing our sequential CG solver to GMM \( \text{cg} \), Intel MKL \( \text{d cg} \) and our parallel CG.
4.1 Conjugate Gradient (CG)

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>GMM</th>
<th>MKL</th>
<th>Parallel CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>21K</td>
<td>0.89</td>
<td>0.90</td>
<td>2.50</td>
</tr>
<tr>
<td>43K</td>
<td>0.88</td>
<td>1.01</td>
<td>3.04</td>
</tr>
<tr>
<td>209K</td>
<td>0.93</td>
<td>1.26</td>
<td>5.57</td>
</tr>
<tr>
<td>352K</td>
<td>0.89</td>
<td>0.89</td>
<td>3.71</td>
</tr>
</tbody>
</table>

**Figure 4.13:** Comparison of GMM cg, Intel MKL dcg and our parallel CG. Intel MKL dcg fails to launch when the size of the matrix is too large.

GMM conjugate gradient performs almost as well as our sequential CG. In contrast, the performances of the Intel MKL routine are surprisingly really poor. Even tough the parallel version of the Intel MKL dcg performs twice as fast compared to its sequential implementation, the speedups are almost negligible. Finally, our parallel CG runs between 3 and 5.5 times faster than its sequential implementation. By using a matrix reordering algorithm, the accesses to memory are more cache friendly and therefore we notice additional speedups up to a factor of 8.5 (Figure 4.14). As we noticed with the CUBLAS cublasSgemv routine in Section 2.3, when the size of the problem does not fit anymore the hardware resources (Level 1 cache, Level 2 cache, ...) we lose some performance, as shown in Figures 4.13 and 4.14 for the 352K example.

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>GMM</th>
<th>MKL</th>
<th>Parallel CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>21K</td>
<td>0.87</td>
<td>1.04</td>
<td>3.54</td>
</tr>
<tr>
<td>43K</td>
<td>0.86</td>
<td>1.17</td>
<td>5.11</td>
</tr>
<tr>
<td>209K</td>
<td>0.87</td>
<td>1.42</td>
<td>8.50</td>
</tr>
<tr>
<td>352K</td>
<td>0.79</td>
<td></td>
<td>4.64</td>
</tr>
</tbody>
</table>

**Figure 4.14:** Comparison of GMM cg, Intel MKL dcg and our parallel CG with matrix reordering.
4 Sparse Linear Solver

GPGPU

The biggest challenge when programming on the GPU is to find a good partitioning of the job between all the threads. Moreover, we have to follow the right access patterns discussed in Section 2.3 in order to maximize the memory bandwidth.

update and dot  Despite our best and persevering efforts, we did not improve the performances of the update and dot operations using CUDA. Our biggest difficulty was to improve the arithmetic intensity of these two operations. Even following the right access patterns to the global memory and using as much as possible the fast shared memory, our implementations run slower than using a single CPU. We use as reference the highly optimized CUBLAS routines cublasSaxpy and cublasSdot. Figure 4.15 shows the speedup with respect to our sequential implementations on the CPU.

![Figure 4.15: Speedup for the update and dot operations using our CUDA implementation (blue) and using CUBLAS (green).](image)

For the update operator, the cublasSaxpy routine performs slightly better than the sequential implementation, running from 1.7 to 3.2 times faster (we achieved more or less the same speedups using XMM registers in Section 4.1.2). In contrast, our implementation runs two times slower (Figure 4.15 (Left)). For the dot operator, even the speedups achieved by the cublasSdot routine are almost visible, running from 5 times slower to 1.5 times faster (Figure 4.15 (Right)). More then being operations with a low arithmetic intensity, their implementation cannot reuse the data fetched from the device memory. Once an element is loaded into the shared memory is only used to compute an entry of the resulting vector. Therefore, the time needed to move data between the shared and global memories is not compensated by the computational efforts of these operations.

SpMV  SpMV is implemented on the GPU by creating one thread for each row of the matrix. If the matrix has dimension $N$, we create $\frac{N}{256} + 1$ blocks of threads that in turn contain 256 threads each. A portion of 256 rows is assigned to each block, and each thread within the block

---

2saxpy (Scalar Alpha X Plus Y) is one of the routine in the Basic Linear Algebra Subprogram (BLAS) library. It consist of a scalar multiplication and a vector addition. saxpy ($n, \alpha, x, y$) computes $y = \alpha x + y$ where $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$. saxpy is almost equivalent to the update operation.
computes the dot product between one of these rows and vector $d$. Block with block index $B_i$ is responsible for the rows whit indices in $[B_i \times 256; B_i \times 256 + 256]$. Each thread $T_i$ within the block loads the entry of vector $d$ with index $B_i \times 256 + T_i$ into the shared memory (this operation accesses the memory in a coalesced way). In this way, when a thread has to fetch a new value for vector $d$, first it checks if this values is already in the shared memory. If this is the case, it loads the value from the shared space without access to the device memory; if is not the case, the thread needs to access to a random position in the memory. With this strategy, the shared memory is used as a cache that try to avoid the threads to access the device memory. The code for the implementation of this strategy is:

```c
__global__ void SpMV_k(float* av, int* aj, int* rp, float* d, float* r, int N)
{
    __shared__ float cache[256];
    unsigned int block_begin = blockIdx.x * blockDim.x;
    unsigned int block_end = block_begin + blockDim.x;
    unsigned int row = block_begin + threadIdx.x;

    if(row < N)
    {
        cache_d[threadIdx.x] = d[row];
        __syncthreads();

        unsigned int row_begin = rp[row];
        unsigned int row_end = rp[row + 1];

        float reduction = 0.0;
        float dj;
        for(int col = row_begin; col < row_end; col++)
        {
            unsigned int j = aj[col];

            if(j >= block_begin && j < block_end)
            {
                dj = cache_d[j - block_begin];
            }
            else
            {
                dj = d[j];
            }
            reduction += av[col] * vj;
        }
        r[row] = reduction;
    }
}
```

At Line 3 the shared cache is declared. At Line 11, each thread loads from the device memory one entry of vector $d$ storing its value on the cache. The synchronization point at Line 12 ensure that all the entries in the cache are loaded. At Line 21 the threads read the columns index of the current non-zero value. If this index is in the cache (Line 23) the thread will not access the memory, but instead it reads the value from the shared memory; otherwise, the threads needs to access the device memory to fetch the entries of $d$ (Line 29).

The matrix can be stored in the global or texture memory. Since we deal with large models, vector $d$ is too large to be stored in the constant memory, and therefore it is stored in the same memory of the matrix. We try different storage strategies: first, we store the matrix with the CRS format in the global and texture memory; then, we use the BCRS with block of dimension $4 \times 1$. Figure 4.16 shows the results.
4 Sparse Linear Solver

As we can see, the best strategy is to store the matrix in the global memory with a CRS format. In this case, our best implementation results two times faster than the sequential one on the CPU. In order to improve the performances for the BCRS format, we tested again the SpMV kernel together with matrix reordering. Figure 4.17 shows that in this case the BCRS format performs slightly better than CRS. With this approach, by loading block of 4 values, we try to increase the computational intensity of the kernel. Again, these results are not really satisfactory. In the best situation, the SpMV operation is speeded up by only a factor of two.

**Figure 4.16:** Speedup for the SpMV operation using CUDA

**Figure 4.17:** Speedup for the SpMV operation using CUDA with matrix reordering

**Parallel CG** By collecting all these parallel operations, we can now implement the Conjugate Gradient method. We use the cublasSaxpy and cublasSdot routines for the update and dot operator and the BCRS format with block dimension $4 \times 1$ to store the matrix. To the best
of our knowledge, the only Conjugate Gradient on the GPU implemented with CUDA is the *Concurrent Number Cruncher (CNC)* proposed in [BCLar]; we use the CNC as reference for our implementation.

As Figure 4.18 shows, the two solvers run almost equivalently. With our best implementation, the solver runs two times faster than its sequential version on the CPU. Moreover, matrix reordering does not lead to any measurable benefit. Since all the basic operations needed to implement the Conjugate Gradient method are dominated by the memory movements between the memories in the GPU, we conclude that this architecture is not yet ready for such kind of operations.

In this section, we examined how the Conjugate Gradient method can be parallelized with different parallel architectures. The best performances are achieved by exploiting multiprocessing machines, thanks to their highly efficient data movements between the processors and main memory. We saw that SSE instructions fail when data is randomly distributed, but they can yield to important benefits with simple operations. In contrast, the efforts on implementing the Conjugate Gradient on the GPU are not comparable to the gains in performance of such routines.
The Multigrid method (MG) exploits the fact that a problem can be represented on different scales of resolution.

Many iterative methods kill the high frequencies of the error very quickly but they fail to converge when this error is smooth. For this reason, they are called smoothers. A well-known smoother for MG is the Gauss-Seidel Method. By restricting the error to a coarser resolution of the problem, its low frequencies are accentuated; we can compute the error in the coarser resolution, prolong it to the fine resolution and finally correct the initial approximation.

Formally, the first step of the MG method is to smooth, or relax, the error associated with an approximation of the solution

\[ \text{RELAX}(A, \tilde{x}, b) \rightarrow \tilde{x} \text{smooth}. \]  

(4.12)

The error \( e \) associated to \( \tilde{x} \text{smooth} \) should be now restricted to a coarser resolution in order to accentuate its remaining low frequencies. Since we do not know yet the exact solution of the system, and therefore we cannot compute the error, we rely on the residual:

\[ r = b - A\tilde{x} \text{smooth} \]  

(4.13)

and we restrict \( r \) instead of \( e \):

\[ \text{RESTRICT}(r) \rightarrow r_c \]  

(4.14)

In order to find the error associated to \( \tilde{x} \text{smooth} \) we solve the residual equation in the coarser resolution. First, we restrict the matrix \( A \):

\[ \text{RESTRICT}(A) \rightarrow A_c, \]  

(4.15)

then we solve the restricted residual equation:

\[ A_c e_c = r_c. \]  

(4.16)

Once we found the solution \( e_c \) we have to prolong it to the initial fine resolution:

\[ \text{PROLONG}(e_c) \rightarrow e \]  

(4.17)

and finally we can correct \( \tilde{x} \) by adding the error \( e \):

\[ x = \tilde{x} \text{smooth} + e \quad \text{since} \quad e = x - \tilde{x}. \]  

(4.18)

By using another MG to solve the residual equation \( A_c e_c = r_c \) in the coarser level, a recursive V-cycle MG is created. When a sufficiently coarse level is reached, the residual equation is solved by using a direct solver and the error is propagated back to the finest resolution.

The two unanswered questions are: first, how can we restrict the residual from a fine to a coarse level and then prolong the error from a coarse to a fine level? And second, how can we restrict the matrix \( A \)? To answer both questions we have to define a restrict operator \( R \) and a prolong operator \( P \).

In our configuration a surface is embedded in a volumetric mesh composed by hexahedral elements. The displacement for each vertex of the mesh is computed by following the elasticity
4.2 Multigrid (MG)

theory explained in Section 3.3. Finally, these displacement are interpolated over the vertices of the embedded surface. The stiffness matrix \( K \) (derived in Section 3.3.3) is built from the hexahedral mesh. By varying the parameters of the voxelisation, we build a hierarchy of hexahedral meshes shown in Figure 4.19.

![Hierarchy of meshes. The surface is embedded in a volumetric hexahedral mesh with different resolutions.](image)

The prolong operator is used to map quantities from a coarse mesh \( M_c \) to a finer one \( M_f \) and it is built as follows: For each vertex \( v^f \) in \( M_f \), the hexahedral element \( e^c \) in \( M_c \) that embeds \( v^f \) is found. Then, by using the local coordinates \((x, y, z)\) of \( v^f \) with respect to the surrounding element \( e^c \), the weights associated to \( v^f \) are computed using trilinear interpolation. Once we found for each of the vertex \( v^f \) in \( M_f \) the corresponding weights associated to some vertices \( v^c \) in \( M_c \) we can build the prolong matrix \( P \). Each row of \( P \) corresponds to a vertex \( v^f \) in \( M_f \) and each column corresponds to a vertex \( v^c \) in \( M_c \). At coordinates \((i, j)\) we find the weight for the vertex \( v^f_i \) in \( M_f \) corresponding to the vertex \( v^c_j \) in \( M_c \). Figure 4.20 should give a better understanding of this process. Since we deal with hexahedral elements, \( P \) has dimension \( |V_f| \times |V_c| \) with a maximum of 8 non-zero entries for each row.

![Construction of the prolong operator. Each rows of \( P \) corresponds to a vertex in the fine mesh and each column corresponds to a vertex in the coarse mesh. The weights \( w_i \) are computed by trilinear interpolation.](image)
4 Sparse Linear Solver

By defining the prolong operator in this way, the restrict operator \( R \) becomes simply the transposed of \( P \), i.e. \( R = P^T \). In the rest of this document we will refer to the restrict operator as \( R \) and the prolong operator as \( P = R^T \). Note that during the construction of the mesh hierarchy the Galerking Property [WL00] is enforced. A coarser representation of the matrix \( K \) can be computed with

\[
K_c = RK_fR^T.
\] (4.19)

As explained in Chapter 3.3, by using an implicit Euler integrator to solve the dynamics of the simulation, a linear system \( Av = b \) has to be solved for each time step. If stiffness warping (Section 3.3) is disabled, matrix \( K \), and respectively matrix \( A \), remain constant during the simulation and therefore the whole matrix hierarchy can be pre-computed. On the other hand, by enabling stiffness warping, matrix \( K \) is not constant anymore, and a new matrix hierarchy has to be computed at each time step. At this point, restricting the residual \( r \) becomes a simple Sparse Matrix Vector multiplication (same for prolonging the error):

\[
r_c = Rr_f \quad \text{and} \quad e_f = R^Te_c.
\] (4.20)

4.2.1 The Algorithm

The pseudocode for the recursive V-Cycle MG is shown in Algorithm 3.

**Algorithm 3** MG(\( A, x, b \))

Require: Matrix \( A \), vectors \( x \) and \( b \)
Ensure: \( Ax = b \)
if coarsest level then
    return \( x = A^{-1}b \)
end if
RELAX(\( A, x, b \))
\( r_f = b - Ax \)
\( r_c = Rr_f \)
\( A^c = RAR^T \)
MG(\( A^c, e^c, r^c \))
\( e_f = R^Te_c \)
\( x = x + e^c \)

In our tests, we use 2, 3 and 4 levels of discretization (Figure 4.21) and a sparse Cholesky Factorization to solve the residual equation in the coarsest level.

4.2.2 Parallelizing the Multigrid

Since the difficulties encountered when dealing with sparse data structures with SSE instructions and GPU based programming, we decided to exploit multiprocessing programming to parallelize the Multigrid method.

The **prolong** and **restrict** operators are easily parallelized because they correspond to a Sparse
4.2 Multigrid (MG)

Relax $A_0x_0 = b_0$
Compute $r_0 = b_0 - A_0x_0$

Restrict $b_1 = R_{01} r_0$
Relax $A_1x_1 = b_1$
Compute $r_1 = b_1 - A_1x_1$

Restrict $b_2 = R_{12} r_1$
Relax $A_2x_2 = b_2$
Compute $r_2 = b_2 - A_2x_2$

Restrict $b_3 = R_{23} r_2$
Solve $A_3x_3 = b_3$

Prolong $e_0 = P_{10} x_1$
Correct $x_0 = x_0 + e_0$

Prolong $e_1 = P_{21} x_2$
Correct $x_1 = x_1 + e_1$

Prolong $e_2 = P_{32} x_3$
Correct $x_2 = x_2 + e_2$

Figure 4.21: Graphical representation of the V-Cycle Multigrid. Starting from the finest level, the residuals are restricted until the coarsest level. There, the error is computed and is used to correct the approximation back to the finest level.

Matrix Vector multiplication. As we saw for the case of the Conjugate Gradient (Section 4.1.2), each thread is responsible for computing a portion of the resulting vector. However, if stiffness warping is disabled, parallelizing these two operators does not lead to any noticeable speedup. Our tests show that the MG is significantly faster than the GMM Conjugate Gradient but most of the computation time is spent during relaxation and solving on the coarsest level (Figures 4.24 and 4.25).

Matrix Restriction

A more interesting scenario appears when stiffness warping is enabled. In this case the crucial part of the solver is the matrix restriction composed of the multiplication of three sparse matrices, namely $R$, $K$ and $P$ (Equation (4.19)). At this point, before parallelization, some optimization techniques have to considered in order to obtain a competitive solver.

Naive Approach The first approach we implemented is to use a third temporary matrix $T$ to store the result of the multiplication $T = KP$ and then we compute $K_T = RT$. However, this strategy is too expensive: most of the computational time is waisted in multiplications by zero and by storing the values of $T$ in the consistent data structure.
Avoiding the temporary matrix  In order avoid storing the values of \(KP\), we expand the summation of the product \(RKPR = RKRT\)

\[
K_c(i, j) = \sum_r R(i, r) \left( \sum_w K_f(r, w) R(j, w) \right)
\]

In this way we can directly compute the entries for \(K_c\). The fact that \(P = R^T\) plays an important role: by storing the matrices in a CRS format, we can access all the data row-wise, avoiding expensive columns-wise accesses that are not suitable with the CRS format.

Precomputing \(K_c\)’s sparsity  Thanks to the fact that the sparseness occupancy of \(K_c\) is known and constant we can avoid computing all its zero entries. Moreover, since \(K_c\) is symmetric, just the lower triangular part of \(K_c\) has to be computed. Our algorithm becomes

\[
K_c(i, j) | j \leq i \land K_c(i, j) \neq 0 = \sum_r R(i, r) \left( \sum_w K_f(r, w) R(j, w) \right)
\]

Precomputing indices intersection  Even with all these improvements, restricting the matrix is still a bottleneck in the implementation.

An entry of the matrix \(K_c\) with coordinates \((i, j)\) is computed by

\[
K_c(i, j) = \sum_c R(c, i) \left( \sum_w K_f(c, w) R(j, w) \right)
\]

It often appears that the dot product between a row \(c\) of \(K_f\) and a row \(j\) of \(R\) leads to zero, because no elements share the same column index. Moreover, the number of non-zero elements that share the same column index is practically always very small. Knowing the coordinates of all non zero elements of our target matrix, i.e. \(K_c\), we can precompute all the useful indices of the elements in \(R\) and \(K\) (Figure 4.22). We proceed in the following way:

- For each non-zero entry of \(K_c\) with coordinates \((i, j)\) we store the index with respect to the array \(vals\) of all the values \(R(i, j)\) that will not be multiplied by zero (i.e. when the intersection of the column indices of the values in row \(c\) of \(K_f\) and the column indices of values in row \(j\) of \(R\) is not empty).

- For each entry \(R(i, j)\) stored in the previous step, we store the indices with respect to the array \(vals\) of the entries in \(K_f\) and \(R\) that are actually useful to compute the dot product between row \(c\) of \(K_f\) and row \(j\) of \(R\).

Since the above optimizations do not require the knowledge about the values of the matrices but only their sparsity structure, they can be precomputed at the beginning of the simulation. In fact, without topology changes of the mesh, matrix \(R\) remains constant over the whole simulation and the sparsity of matrix \(K_c\) can be precomputed after the first step. Thanks to this approach all the useless computation during the restriction of the matrix \(K\) are avoided. However, [Geo07] shows that more efficient algorithm with more sophisticated data structures can be implemented.

At this point the job for the matrix restriction operator have to be divided between the processors. Each processor is responsible for computing a portion of the matrix \(K_c\). Since the matrix is
stored in a CRS format, the partitioning is done between batches of rows. Since only the lower triangular part of $K_c$ is stored in memory and the sparsity of matrices $K_f$ and $R$ are random, this partitioning leads to an unbalanced partition of the jobs between the processors. Counting the total number of operations needed to compute one entry of $K_c$, we can re-partition the job equally over all the processors (Figure 4.23), avoiding situation where the threads with less job have to be inactive waiting for the others threads in the final synchronization point. Even though does not improve drastically the performance in the simulation (we gain about 2 frames per second), the same approach can lead to remarkable improvements in different situations.

Figure 4.22: Matrix restriction operator.

Figure 4.23: Unbalanced (right) and balanced (left) partitioning of the jobs along parallel threads for the matrix restriction operator.
4 Sparse Linear Solver

We compared our Multigrid with the GMM $\text{cg}$ Conjugate Gradient routine with and without Stiffness Warping. As Figure 4.24 shows, the Conjugate Gradient has quadratic complexity while the Multigrid has linear complexity. Without Stiffness Warping the matrix hierarchy can be precomputed and therefore the MG runs between 3 and 30 times faster than the GMM $\text{cg}$ (Figure 4.25). Enabling Stiffness Warping, at each time step the matrix has to be restricted for each coarse level and the MG runs between 1.5 and 5.5 times faster (Figure 4.25).

![Figure 4.24: Timing for the GMM $\text{cg}$ routine and our MG with and without Stiffness Warping.](image)

![Figure 4.25: Speedups of our MG with respect to the GMM $\text{cg}$ routine with and without Stiffness Warping.](image)
Conclusion and Future Work

5.1 Parallel Architectures

Despite their simplicity, Streaming SIMD Extensions instructions can be really powerful and they cannot be ignored. When the data we are processing is uniformly distributed in the memory, they fit particularly well for simple operations. Many numerical routines can be speeded up to a factor of three by exploiting these special registers. However, we have to keep in mind that the actual compilers are getting always smarter: for simple situations auto-vectorization strategies work very well. Sometimes, in our test we did not notice any speedup using SSE instructions because the compiler already exploited XMM registers automatically. However, for more difficult operations, SSE instructions leads to remarkable benefits.

Shared-memory multiprocessing programming revealed to be the most general parallel paradigm we investigated. Its power consists in the optimized memory bandwidth provided by the actual CPUs. Even with random accesses to the memory, the benefits achieved by these architectures are outstanding. Smart caching strategies and optimized inter-processor communications enable this parallel paradigm to guarantee improvement for practically all kind of applications. Also in this context, auto-parallelization technologies starts to reach good results: Intel® C++ compiler provides the possibility of set some parameters to control the auto-parallelization of an application. Usually, the results are not surprising but lot of efforts are devoted into this field. Moreover, thanks to the dedicated APIs, programming on multiprocessing machines is really accessible and easy to learn. From our experience, this is the best parallel technology we investigated.

General-Purpose computing on Graphics Processing Units results to be a really specialized technique. In order to achieve good performances, a lot of requirements have to be fulfilled. The memory access patterns and the arithmetic intensity of an application are the two biggest
issues when programming the GPU. Moreover, since nowadays only single precision floating points are supported by the graphics cards, numerical errors caused by truncations appear. If the problem we want to solve fits these architectures, the speedups can be up to a factor of hundreds. However, especially when dealing with random memory access and operations with poor arithmetic intensity, the benefits are scarcely noticeable.

5.2 Sparse Linear Solvers

We examined the behaviors of two important numerical solvers mapped into parallel architectures. The Conjugate Gradient method is highly parallelizable and performs really well especially on multiprocessing systems. Since it basically consists of just a few algebra operations, its implementation is straightforward. Moreover, it is used in many application where performance is a key requirement and any speedup makes the efforts of the parallelization worth. Future work could be to re-map this solver on the future graphics cards, hoping that the issues described above by that time will no longer be present.

The Multigrid method is a powerful tool. Even without any specialized hardware resources it is one of the best method for solving linear systems. We showed that by using multiprocessing systems the frame rate of a simulation can be doubled. However, Multigrid methods are known to be difficult to implement and especially hard to debug. In the literature, we find entire chapters on how a Multigrid method should be tested in order to ensure a correct execution. Fixed Point Property, Zero Residuals, Compatibility Conditions,... are just a few of the diagnostic tools needed to profile this solver. We tested our implementation only with the basic methods, and therefore a future work must be an intensive and meticulous profiling of all the single phases of our Multigrid. Moreover, further optimization can be achieved; for example, the matrix restriction operator can be improved by exploiting more sophisticated data structures.
Bibliography


Bibliography


Bibliography

