On Leveraging GPUs for Security: discussing k-anonymity and pattern matching

Candidato: Leonardo Jero
matricola: 279319

Relatore: Roberto Di Pietro, PhD

Correlatore: Flavio Lombardi, PhD

Anno Accademico 2011/2012
Maggio 2013
# Contents

1 Introduction .................................................. 3
   1.1 Background .............................................. 4
   1.2 Motivation and goals .................................. 4
   1.3 Organization of the work .............................. 5

2 Security and GPUs ........................................ 6
   2.1 Security issues and GPUs ............................. 6
      2.1.1 Privacy and GPUs .................................. 6
      2.1.2 Privacy and k-anonymity ......................... 7
      2.1.3 Integrity, Malware and Pattern Matching .... 8
      2.1.4 Pattern Matching and GPUs ..................... 11
      2.1.5 Why use GPUs? ................................... 12
   2.2 GPU Computing .......................................... 12
      2.2.1 CUDA and OpenCL ................................. 12
      2.2.2 JavaCL and Aparapi ............................... 14

3 k-anonymity through micro-aggregation ............... 18
   3.1 Definitions .............................................. 18
   3.2 Microdata disclosure protection techniques ...... 20
      3.2.1 k-anonymity ....................................... 21
      3.2.2 Disclosure risk and information loss .......... 23
      3.2.3 Micro-aggregation ................................. 25
      3.2.4 Present approaches ................................ 27

4 k-anonymity on GPUs ..................................... 30
   4.1 MDAV ..................................................... 30
      4.1.1 Analysis of the algorithm ....................... 31
      4.1.2 How we parallelized MDAV on GPUs ........... 33
   4.2 HyperCubes .............................................. 35
      4.2.1 Analysis of the algorithm ....................... 37
      4.2.2 How we parallelized HyperCubes on GPUs .... 39
4.3 GPU issues we encountered and solved .......................... 40

5 An in-depth evaluation of synthesized code .......................... 43
5.1 Experimental test-bed ................................................. 43
5.2 MDAV ........................................................................ 43
5.3 HyperCubes ................................................................. 54
5.4 Information loss .......................................................... 56
5.5 Other issues ................................................................. 57

6 Discussion and conclusion ..................................................... 59

A Relevant excerpt from Parallel MDAV and HyperCubes codes ............................. 61
Bibliography ....................................................................... 71
Chapter 1

Introduction

In recent years the need to solve complex problems that require large computing resources in shorter time has especially arisen. Some of these in the scientific field are: weather forecast, seismic simulations, chemical reactions simulation and studies on the human genoma [1]. All of them belong to the “Grand Challenge Problems” set. As can be noted, solving these problems within strict time limits is very interesting for many areas of scientific research.

Other relevant problems exist in the computer science and business fields: database management and data-mining [2], search engine development on the web [3], medical diagnostic, advanced graphic [4], virtual reality [5], network broadcasting, highly complex information security [6] and data confidentiality [7]. In the past, the solution of these problems used to take large amounts of time, due to the need of leveraging very large computing capacity to solve them. Now, thanks to the development of parallel computing, larger and cheaper resources are available, that can be used to address such problems. Parallel computing potentially allows:

- to solve larger problems than before;
- to improve computing time;
- to reduce costs (by splitting work among multiple cheaper processors).

Despite the hopes placed in parallel computing, some problems, requiring super-polynomial algorithms, are still far from being solved in acceptable time. However parallelization can help reduce computing time in order to speed up results availability.
1.1 Background

The evaluation of parallel computing went through various steps. First approaches to this programming paradigm were threads. These, introduced via libraries in the programming languages, including Fortran, C, Python and Java, allowed to start the development of some sort of parallel code. However this result could be improved, because on single-core CPUs threads were only virtually and not physically parallelized. To solve this issue, hardware industry started developing increasingly complex CPUs, that evolved from the dual-core to the current quad-core.

Leveraging threads and hardware development of CPUs, other languages, in addition to the above, introduced the possibility of multi-threading, such as Mathematica [8] and Matlab [9], two purely mathematical languages. Despite such evolutions, more relevant advances in parallel computing are due to the adoption of modern GPUs (Graphics Processing Units), as general purpose computing resources. It suffices to say that thanks to their usage the execution speed of parallel algorithms can get a thousand-fold increase.

1.2 Motivation and goals

The motivations of our work are multifold. Given that there are many embarrassingly parallel problems [10], the development of libraries that allows the usage of GPUs opens new scenarios, in many different fields, thanks to the computational speed of graphic cards. Given the large set of interesting problems that can potentially benefit from parallel computing we are interested in studying and devising a feasible solution using GPU technologies, in particular for mathematical problems that can help ensuring a better security to computing resources. The great and novel GPU computing potential gives the opportunity to quickly ensure security and privacy properties in an effective way. Starting from the results and knowledge gathered in the course “Tecniche Informatiche Avanzate” and in the seminary on GPUs held at the Department of Mathematics and Physics of the University of Roma Tre, we began to be curious on the possibilities of actually solving such problems. Our main goal is to investigate the effectiveness and efficiency of leveraging GPUs for computer security. As part of the effort we will study in particular methods for ensuring integrity and confidentiality using GPUs. In addition, we will evaluate the effectiveness and efficiency of such approaches in order to discover bottlenecks and opportunities for further development and results.
1.3 Organization of the work

The thesis is organized as follows. In Chapter 2 we introduce the security problems, in particular privacy and integrity. We also discuss several approaches to guarantee data integrity through pattern matching. In Chapter 3 we introduce the concept of microdata and k-anonymity. We also describe the techniques proposed in literature to solve the k-anonymity problem. In particular we describe the main micro-aggregation approaches. In Chapter 4 we analyze two algorithms based on the micro-aggregation technique, MDAV and HyperCubes, and we show which are the parallelizable steps of these algorithms and how we parallelized them. In Chapter 5 we discuss tested setup and results obtained with our parallel code. In Addition, we introduce the main issues in using GPUs and how we solved them. Finally, in Chapter 6 we present the contribution, final considerations and future works.
Chapter 2

Security and GPUs

2.1 Security issues and GPUs

In this section we will discuss about security problems, in particular confidentiality and integrity, and about some approaches to solve them with GPUs. Confidentiality refers to limiting information access and disclosure only to authorized users and preventing access by unauthorized ones. In order to preserve confidentiality, data encryption is largely used. Actually, in our work we will focus on privacy in statistical databases. The difference between confidentiality and privacy is that, whilst the first concerns data, the second concerns people. However, encryption is used also to ensure privacy in large statistical databases, even if excessive usage of cryptography is not possible, because of the risk of information loss.

Integrity refers to the guarantee that information has not been altered; it includes the concept of “data integrity”. It also includes “authenticity”, that concerns the linkability of the author of a document. In our work we are interested particularly in how to avoid attacks to data integrity. In fact, viruses can be created to damage it. One of the most common technique used by anti-virus software is based on “pattern matching”, the act of finding a word or a particular sequence of typefaces (i.e. patterns) in a larger given text.

2.1.1 Privacy and GPUs

In recent years the old paper-based files are being replaced by electronic files that can be easily accessed and copied. This large amount of data contains lots of information about a large number of people: due to the great increase of this stored personal data, there is a real need for efficient methods that protect privacy.

“Privacy is the ability of an individual or group to seclude themselves or
information about themselves and thereby reveal themselves selectively. The right not to be subjected to unsanctioned invasion of privacy by the government, corporations or individuals is part of many countries’ privacy laws, and in some cases, constitutions” [11].

To ensure privacy several techniques have been developed recently [12]. The most used techniques have a computational cost not less than quadratic: this is a big issue when the dataset to be anonymized has a large cardinality. For this reason privacy ensuring techniques can benefit from a GPU approach: thanks to their hardware features, GPUs can potentially improve performance by a constant factor that depends on the number of cores, allowing to reduce the overall computational time.

In this thesis, we will deal in particular about clustering techniques, specifically micro-aggregation, that is based on the k-anonymity concept. GPUs can be very helpful also in the k-anonymity field, as shown in [13].

![Figure 2.1: Execution flow of the k-Means algorithm presented in [7]](image)

2.1.2 Privacy and k-anonymity

Many data protection techniques, which are aimed to protect the privacy of the users whose data is stored, are based on the k-anonymity concept. In simple terms, k-anonymity divides the data in clusters of k or more elements and substitutes the elements in each k-cluster with a unique synthetic one, commonly the mean of the elements.

The majority of the techniques that ensure k-anonymity have a computational time equal to $O(n^2)$, so they are not really high-performance when
the number of elements increases. For this reason GPUs usage could be very helpful. Two sorts of fusion of these techniques with the use of parallel computing are proposed in [7] and in [13]. In [7] the authors modified the k-Means algorithm, parallelizing it using CUDA (Fig. 2.1 shows the execution flow of this parallel k-Means algorithm). In [13] an algorithm, that analyzes the k-anonymity approach and is suitable for extracting tuples with the same values, was proposed. We will discuss the k-anonymity problem largely in chapter [3].

2.1.3 Integrity, Malware and Pattern Matching

During the last decades computer science has increasingly shared large amounts of data. With respect to such requirement is the concept of data integrity: at present the probability that a shared file is a malware is rather high. According to the necessity of preserving data integrity, many malware detection techniques (MDTs) were defined. The MDTs can be divided in two main categories: signature-based and anomaly-based.

Signature-based detections work by comparing data signatures with the signatures of the known malware. Signature refers to the hash string of a particular file, but recently also to other statistical properties or characteristics of malicious behavior, such as portions of strings, i.e. patterns, contained frequently in particular malware families.

Anomaly-based detections work by monitoring system activity and classifying it as either normal or anomalous. They attempts to detect any type of misuse that falls out of normal system operation. This approach can find out malware also without having a signatures database, but has the disadvantage of an high percentage of false positives. For a brief survey of MDTs we refer to [14].

Many signature-based anti-viruses use pattern matching tools. Pattern matching algorithms analyze data and compare it against a database of signatures to detect known viruses. So pattern matching allows anti-viruses to ensure data integrity.

Pattern matching approaches

There are many approaches to the pattern matching problem. The naive approach is the brute-force algorithm: it consists in checking for patterns starting the comparison at each text position. This algorithm can be slow if text and pattern are repetitive. In the worst case, if pattern length is $M$ and text length is $N$, this algorithm needs $M \times N$ comparisons; its computational
complexity is $O(M \times N)$. This complexity is not very good, especially when $N >> M$.

A simple improvement to the brute-force algorithm was given in 1975 by Knuth and Pratt and independently by Morris [15] (KMP algorithm). They noticed that a pattern matching problem could be represented by a deterministic finite automaton (DFA) [16], obtained by the specific pattern. The advantage of defining a DFA is a linear time for computation: each step of a DFA is just a state change. Before starting the search of the pattern, the algorithm needs a pre-processing step, that creates an array “next”, obtained from the pattern. This array allows to know for each position which is the next position where it is possible to find an occurrence on the pattern. The running time to build the “next” array is $O(M)$ and the time to compare the pattern to the text is $O(N)$. Total running time is $O(N + M)$.

In 1977, Boyer and Moore developed a new pattern matching algorithm (BM algorithm). The main idea behind the algorithm is that more information is gained by matching the pattern from the right than from the left [17]. As KMP, BM algorithm consists of a pre-processing step and a research step. It is based on two heuristic rules:

- **Bad character rule.** It can be applied if the bad character, i.e. the text symbol that causes a mismatch, occurs somewhere else in the pattern. Then the pattern can be shifted so that it is aligned to this text symbol. If the text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by $M$ positions behind this text symbol.

- **Good suffix rule.** The basic idea of the Boyer-Moore Good Suffix Rule is to study the internal structure of the pattern and use that information to look for duplicated sub-strings in the pattern. In particular, duplicates are searched at the end of the string. In fact, if we successfully matched a suffix of the pattern, but eventually failed at some point, we can start up the next comparison with the pattern shifted over to the duplicate of that suffix.

In general the BM algorithm running time is $O(M \times N)$, but, when the alphabet is large compared to the length of the pattern, i.e. when the text is larger than the pattern ($N >> M$), the algorithm performs the evaluation with a computational time $O(N/M)$.

As the good-suffix heuristics is rather complex to implement, sometimes there is a need for a simpler algorithm that is based only on the bad character rule. This led to a variant of the BM algorithm, proposed by Horspool in 1980.
(BMH). It improves the pre-processing of a particular array that allows to remove completely the good suffix rule.

Multiple pattern matching approaches

Sometimes we need to find more than one pattern in a text. In this case we will talk about multiple pattern matching. The simplest solution to this problem is to try a match for each pattern we are looking for. Clearly this solution has not a good computational complexity, that is $O(K \times (\text{Time for each search}))$, where $K$ is the number of patterns.

A better and larger used solution was given by Aho and Corasick [15] in 1975. As for KPM algorithm, the development of this new algorithm (AC) is based on the notion of DFA.

The main idea of this algorithm is to take advantage of the similarities between patterns to improve computational speed. This algorithm needs a pre-processing step in which two functions are defined: the $\delta$ and the $\varphi$ functions. The first is called transition function: it returns the next automaton state in case of match; it returns the blank value in case of mismatch. The second is called failure function: it returns the next automaton state in case of mismatch, i.e. when $\delta$ returns the blank value. When we find a mismatch during the search for a particular pattern $P_i$, it can be that some already computed characters are in the prefix of another pattern $P_j$. The $\varphi$ function allows to continue the search for $P_j$, avoiding to start again from the beginning. With the AC approach the computational time decreases to $O(|P_1| + ... + |P_k| + |N|)$.

In 1979, Commentz-Walter (CW) proposed a new algorithm [19] that merged the principal features of BM and AC. Like AC, CW is based on a DFA that can recognize all the patterns in a set $P$. The main difference between these two algorithms is in the automaton definition: CW process the DFA with the reverse of every single pattern in $P$. So the algorithm can take advantage of the BM right to left scan technique, that yields a sub-linear behavior on the average. However, it must be said that CW does not maintain linear search time for the worst case.

In 1994, Wu and Manber proposed a new algorithm for multi pattern matching (WM) [20]. Their idea was to concentrate the design of the algorithm on typical searches rather than the worst case behavior. The WM algorithm uses the same search technique of BM. However, takes advantages of three tables instead of only one. The total complexity of WM is given by $O(M + (BN/m))$, where $M$ is the total size of the pattern set $P$, $N$ is the length of the text, $m$ is the minimum length of a pattern in $P$ and $B$ is a parameter that depends by the algorithm.
For a short and more complete survey of all the well known pattern matching algorithms we refer to [21].

2.1.4 Pattern Matching and GPUs

As mentioned above, many malware detectors use pattern matching tools to preserve against malware or in general threats to integrity. Pattern matching algorithms analyze the incoming data and compare it against a database of signatures to detect known malware. For instance two of the most used open source intrusion detectors, ClamAV [22] and Snort [23], adopt AC as matching algorithm.

Unfortunately, the number of malware increases constantly, and with it the amount of signatures increases proportionally, resulting in a problem of scalability. For this reason, GPUs have an important role in improving anti-virus performance. In fact, it has been shown that graphic cards are very efficient and highly effective at accelerating the pattern matching operations of network intrusion detection systems: in [24] and [25] authors developed a GPU adaptation of the AC algorithm; in [26] the main pattern matching algorithms, such as BM and KMP, are tested on GPU.

Figure 2.2: GrAVity architecture [27]

In recent years an interesting proposal was given, regarding pattern matching improvements, with GrAVity [27], a rebuilding of ClamAV anti-virus using GPUs. The ClamAV algorithm is divided in two steps. The first step scans data, searching a match with a prefix of each virus signature. If a
suspicious file was found, it is sent to the second step that checks it more in-depth. The first step is slower, because clean data is larger than viruses. The idea in GrAVity is to use GPUs in the first step to improve scans speed. GrAVity architecture is depicted in Figure 2.2.

2.1.5 Why use GPUs?

Using GPUs for security problems can potentially have the following advantages:

- GPUs are potentially highly effective at accelerating the pattern matching operations of network intrusion detection systems and the clustering operations of k-anonymity based techniques;
- there is a potential speed-up when we use GPUs for virus scans and for data anonymization;
- without the support of parallel computation, many storage system optimizations cannot be deployed because of their associated overheads;
- with GPUs many security tools can be parallelized, improving their performances [24] [25] [26] [27] (see also chapter 4).

We want to actually prove that GPUs would be useful for solving specific problems. In order to do so, in this thesis work we will devise, implement and test real algorithms.

2.2 GPU Computing

A GPU is an hardware component that can be considered a CPU co-processor. It allows to increase the computing speed of images, 3D especially. In the last ten years it has played a fundamental role in videogames industry. Recently came out the idea of take advantages of these components for other tasks, not exclusively graphical: this has resulted in the birth of GPGPU (General-Purpose computing on Graphics Processing Units) [28]. Despite that, until some time ago, GPUs were not easily programmable.

2.2.1 CUDA and OpenCL

The programmability problem of the GPUs has been partially solved, thanks to the efforts of NVIDIA and AMD, that implemented some libraries/drivers
to allow the usage of these graphic cards on a large scale.

The first of these libraries was CUDA, developed by NVIDIA, that fits in the major programming languages: C, Fortran, Python and Java. Despite CUDA allowed to make a considerable progress in GPU programming, this library is closely linked to the NVIDIA hardware components. So code written with CUDA is not portable on graphic cards of other producers. For overcoming this issue it has been developed a new library, OpenCL. Based on the programming language C and originally proposed by Apple and ratified by AMD and NVIDIA, it was accomplished by the no-profit Khronos Group (moreover OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors and GPUs).

OpenCL has the advantage of a larger portability: it can be executed both on NVIDIA and AMD GPUs and recently also on Intel ones.

As regards the syntax, OpenCL and CUDA are relatively similar [29]: they have similar memory hierarchies and processing elements (Fig. 2.4).
Also OpenCL, like CUDA, is integrated as a library in several languages. For example, JavaCL [30] is a binding that allows to join the Java language features with the OpenCL ability to communicate with the GPUs. This approach can lead to considerable benefits in terms of ease of programming and code portability. In 2011, in order to improve JavaCL, AMD released another Java API that has similar features: Aparapi (“A PARallel API”) [31], that increasingly simplifies the GPU programmability. We will discuss about JavaCL and Aparapi in the following.

### 2.2.2 JavaCL and Aparapi

**JavaCL**: is an API that wraps the OpenCL library to make it available to the Java platform. With JavaCL, Java programs can execute tasks directly on graphic cards and benefit from their massive parallel power. JavaCL code looks as follows:

```java
CLContext context = JavaCL.createBestContext();
CLProgram program = context.createProgram(myKernelSource).build();
CLKernel kernel = program.createKernel(
    "myKernel",
    new float[]{u, v},
    context.createIntBuffer(Usage.Input, inputBuffer, true),
    context.createFloatBuffer(Usage.Output, resultsBuffer, false)
);
...
```

---

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What is it</strong></td>
<td>HW architecture, ISA, programming language, API, SDK and tools</td>
<td>Open API and language specification</td>
</tr>
<tr>
<td><strong>Proprietary or open technology</strong></td>
<td>Proprietary</td>
<td>Open and royalty-free</td>
</tr>
<tr>
<td><strong>When introduced</strong></td>
<td>2006</td>
<td>2008</td>
</tr>
<tr>
<td><strong>SDK Vendor</strong></td>
<td>NVIDIA</td>
<td>Implementation vendors</td>
</tr>
<tr>
<td><strong>Free SDK</strong></td>
<td>Yes</td>
<td>Depends on vendors</td>
</tr>
<tr>
<td><strong>Multiple implementation vendors</strong></td>
<td>Yes: Windows, Linux, Mac OS X; 32 and 64 bit</td>
<td>Depends on vendors</td>
</tr>
<tr>
<td><strong>Heterogeneous device support</strong></td>
<td>No, just NVIDIA GPUs</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Embedded profile available</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 2.5: CUDA vs OpenCL properties
The rationales behind choosing JavaCL are many:

- it allows to write and compile once, run everywhere: one-jar for all platforms, and JavaCL takes care of some well-known platform discrepancies / bugs;

- it is a programmer-friendly API with simple memory management;

- it allows to have very competitive performance;

- it guarantees advanced kernel code management: kernels are stored as simple Java resources and can be included directly from their JARs.

JavaCL has also some issues [32]:

- sometimes there are problems with the management of the execution errors. For example, when a problem occurs with the launch of a kernel on the GPU, it is possible that the program encounters difficulties and reverts back the execution on the CPU without warnings. It is not good for developers, which cannot spot the problem without an accurate debugging;

- sometimes there are problems in launching a large number of kernels (encountering an invalidKernelArgs Exception);

- there are also difficulties with huge files, like images;

- sometimes there are portability issues.

Aparapi (“A PARallel API”): like JavaCL, allows Java developers to take advantage of the computing power of GPU devices by executing data parallel code fragments on the GPU rather than being confined to the local CPU. In contrast to JavaCL, Aparapi does this by converting Java bytecode to OpenCL at runtime and executing it on the GPU. When Aparapi fails to convert bytecode to OpenCL and run it on the GPU, it will revert back in a Java thread pool (JTP), that is the Aparapi fall-back mode.

The Aparapi development is aimed to programming simplification. In fact, JavaCL requires the developer to learn OpenCL and to explicitly coordinate the movement of data buffers to and from the GPU devices. If OpenCL is not available at runtime, the developer would be required to provide an alternate Java version of the algorithm and explicitly arrange for execution
Aparapi allows developers to write their data parallel code once in Java (by extending a Kernel base class). The Aparapi runtime will either convert the resulting bytecode to OpenCL for execution on a GPU, or if OpenCL is unavailable or the code cannot be represented in OpenCL, Aparapi will run using a thread pool. So Aparapi extends Java’s paradigm “Write Once Run Anywhere” to include GPU devices. Sample usage looks like:

```java
Kernel kernel = new Kernel(){
  @Override public void run(){
    int i= getGlobalId();
    result[i]=int A[i] + int B[i];
  }
};
Range range = Range.create(result.length);
kernel.execute(range);
...
```

For an example of OpenCL code elaborated by Aparapi see Appendix A.

Before starting our dealing with Aparapi, we report some issues:

- not all Java bytecode can be converted to OpenCL;
- only primitive scalars and simple arrays of primitives are supported:
multi-dimensional Java arrays of primitives are not stored in contiguous memory;
support for Object arrays exists, but requires reflective copying of fields into parallel primitive arrays behind the scenes.

- Kernel.run() reachable methods:
  - cannot be recursive or use switch, break or continue;
  - can only call other members of the Kernel instance, either base class methods or user supplied (for ex.: use Kernel.sqrt(float) instead of Math.sqrt(float)).

Despite these problems, Aparapi is a great resource for parallel computing, according to the large amount of experimental results, that can be found in [35], in which it can be noted that Aparapi performance is much better than any Java thread.

Aparapi is still in its infancy, but offers promising perspective. It works like the others existing Java bindings, but it provides:

1. ease of portability of millions of existing legacy Java code;
2. ease of programming for not-coder and not-hackers;
3. however, generated code can still be examined.

In our work we are going to study some applications of GPGPU to security problems and search for advantages and issues in using graphic cards for programming.
Chapter 3

k-anonymity through
micro-aggregation

3.1 Definitions

Data sharing is increasingly popular and required by various types of organizations: governmental, public and private. Every day a large amount of data is published on the net. The protection of these data against an improper information disclosure is an increasingly fundamental issue.

In the past, released data were in tabular or statistical form. This type of data is called macrodata. Macrodata is aggregated data that concern users or organizations, usually in bi-dimensional tabular form. Macrodata protection techniques are based on the obfuscation of some attribute values that can lead to an information disclosure.

More recently information began to be released in the form of microdata, specific portions of data that allow the user to make any type of statistical analysis. In contrast to pre-computed statistics, microdata ensures more information flexibility and availability.

As microdata refers to well determined individuals, called respondents, there is the need to ensure data anonymity before its publication.

To guarantee anonymity data owners frequently remove or encrypt the explicit identifiers, such as name, address and phone number. However, de-identified data contain also other informations, as sex, race, birth date, etc... Such values can be used to re-identify respondents, resulting in an information disclosure. From now on we will focus on microdata. For a more complete discussion on macrodata we refer to [12].

Microdata contains a set of attributes related to single individuals of a population. Microdata can be represented with a table composed by tuples (i.e.
records) with values in a set of attributes. In Fig. 3.1 an example of micro-data table is depicted.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Race</th>
<th>DoB</th>
<th>Sex</th>
<th>ZIP</th>
<th>Job</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asian</td>
<td>31/10/74 F</td>
<td>00142</td>
<td>Employee</td>
<td>Hypertension</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>01/05/73 F</td>
<td>00142</td>
<td>Employee</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>04/02/88 M</td>
<td>00142</td>
<td>Employee</td>
<td>Chest pain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>24/02/90 M</td>
<td>00143</td>
<td>Employee</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>11/09/87 M</td>
<td>00143</td>
<td>Businessman</td>
<td>Hypertension</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>16/01/88 M</td>
<td>00143</td>
<td>Businessman</td>
<td>Short breath</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>13/10/53 F</td>
<td>00143</td>
<td>Businessman</td>
<td>Short breath</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>01/01/56 F</td>
<td>00143</td>
<td>Teacher</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White</td>
<td>22/12/85 M</td>
<td>00133</td>
<td>Teacher</td>
<td>Chest pain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White</td>
<td>26/07/86 M</td>
<td>00133</td>
<td>Teacher</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White</td>
<td>12/12/77 F</td>
<td>00138</td>
<td>Driver</td>
<td>Short breath</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Example of de-identified medical microdata table

Called $V$ a set of microdata, the attributes in $V$ are usually classified as follows [36]:

- **Identifiers**: These are attributes that unambiguously identify the respondent. These attributes are removed/encrypted in the pre-processing step of the anonymization of the data (for instance SSN and Name in Fig. 3.1).

- **Quasi-identifiers**: a quasi-identifier is a set of attributes in $V$ that, in combination, can be linked with external information to re-identify the respondents to whom the records in $V$ refers. Unlike identifiers, quasi-identifiers cannot be removed from $V$ (for instance Date of birth, Sex and Race).

- **Confidential outcome attributes**: These are attributes which contain sensitive information on the respondents. Example: Disease.

- **Non-confidential outcome attributes**: These are attributes which contain non-sensitive information on the respondents. Example: Job. Note that these attributes can be part of a quasi-identifier. So they must not be neglected when protecting a data set.
3.2 Microdata disclosure protection techniques

Given an original microdata set $V$ the main goal is to release a protected microdata set $V'$, such that the risk of information disclosure is low and the users analyses on $V$ and $V'$ gives similar results.

Microdata protection techniques (MPTs) can be classified into two main categories: masking techniques and synthetic data generation techniques (See Fig. 3.2) [12].

Figure 3.2: Classification of microdata protection techniques (MPT’s) [12]

Masking techniques take microdata and modify it producing new data with the same statistical features. Furthermore after elaboration respondents privacy is protected. These techniques are classified as:

- **non-perturbative**: original data are not modified but only some attributed are deleted;

- **perturbative**: data are modified.

Synthetic techniques take the original set of tuples $V$ and substitute it with a new set $V'$ generated in such a way to preserve statistical properties of the original set. The generation process is based on statistical models. Released microdata $V'$ can be fully synthetic or mixed with the original data (i.e. partially synthetic).

Note that MPTs can work with different data types: continuous data and categorical data. The former are numerical type attributes, on which arithmetic operations can be done. Example: age. The latter are attributes that assume a limited number of values and arithmetic operations cannot be defined on them. Example: job.

In literature there are many masking and synthetic techniques. Some of the most popular are: suppression, generalization and top/bottom coding (among the non-perturbative masking techniques); re-sampling, rounding, swapping and micro-aggregation (among the perturbative masking techniques);
bootstrap, latin hypercube sampling, hybrid masking, random response (among
the synthetic techniques). For a more complete list of the MPT’s see [12].
In our work we will focus on the micro-aggregation technique.

3.2.1 k-anonymity

Before dealing about micro-aggregation, we need to introduce the concept of
k-anonymity [36]:

**Definition 3.2.1.** A dataset is said to satisfy k-anonymity for \( k > 1 \) if, for
each combination of values of quasi-identifiers, at least \( k \) records exist in the
dataset sharing that combination.

Note that, if a protected dataset \( V' \) satisfies k-anonymity, the re-identification,
i.e. mapping a record in \( V' \) to a non-anonymous record in the external data
source (see §3.2.2), is not possible: the best a possible attacker can do is to
map groups of at least \( k \) records.

When in 1998 Samarati and Sweeney defined k-anonymity [37], their ap-
proach suggested to reach k-anonymity through a combination of generaliza-
tion and local suppression techniques.

- **Local suppression:** it suppresses the value of an attribute and replace
it with a missing value, so that the possibilities of analysis are limited.
This technique obscures some attribute values (sensitive cells), that
could contribute significantly to the disclosure risk. For instance, in
the table in Fig. 3.1, we can suppress the attributes ZIP and Job of
the last tuple, that are unique quasi-identifiers.

- **Generalization:** it substitutes the values of a particular attribute using
more generic values. The technique is based on the usage of a hierar-
chy of generalizations, where the most general value is the root of the
hierarchy and the leafs are the most specific values. In Fig. 3.3 the
hierarchy of the ZIP attribute is depicted. So a generalization process
replaces sensitive values, represented by the leafs, with the values of
the parent nodes. Different generalizations are possible according to
the particular hierarchy used.

In Fig. 3.4 an example of a k-anonymized table is depicted.

In 2004 Meyerson and Williams [38] showed that ensure k-anonymity
through generalization and suppression is an NP-hard problem. In fact, find
the optimal combination of these two techniques is an open issue: a careless
Figure 3.3: Generalization hierarchy for ZIP attribute of the table of Fig. 3.1

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Race</th>
<th>DoB</th>
<th>Sex</th>
<th>ZIP</th>
<th>Job</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>00133</td>
<td>00138</td>
<td>00142</td>
<td>00143</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>31/10/74</td>
<td>F</td>
<td>0014*</td>
<td>Employee</td>
<td>Hypertension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>01/05/73</td>
<td>F</td>
<td>0014*</td>
<td>Employee</td>
<td>Obesity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>04/02/88</td>
<td>M</td>
<td>0014*</td>
<td>Employee</td>
<td>Chest pain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian</td>
<td>24/02/90</td>
<td>M</td>
<td>0014*</td>
<td>Employee</td>
<td>Obesity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>11/09/87</td>
<td>M</td>
<td>0014*</td>
<td>Businessman</td>
<td>Hypertension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>16/01/88</td>
<td>M</td>
<td>0014*</td>
<td>Businessman</td>
<td>Short breath</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>13/10/53</td>
<td>F</td>
<td>0014*</td>
<td>Businessman</td>
<td>Short breath</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>01/01/56</td>
<td>F</td>
<td>0014*</td>
<td>Teacher</td>
<td>Obesity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>White</td>
<td>22/12/85</td>
<td>M</td>
<td>0013*</td>
<td>Teacher</td>
<td>Chest pain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>White</td>
<td>26/07/86</td>
<td>M</td>
<td>0013*</td>
<td>Teacher</td>
<td>Obesity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>White</td>
<td>12/12/77</td>
<td>F</td>
<td></td>
<td></td>
<td>Short breath</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4: Table of Fig. 3.1 k-anonymized by applying the generalization and suppression techniques to the attributes ZIP and Job

Combination may cause a loss of data utility \[36\]. Furthermore, the use of generalization to ensure k-anonymity causes several problems, first of all, an high computational cost, as shown in the following theorem \[36\].

**Theorem 3.2.1.** For an attribute with \(c\) categories, there are \(2^c - c - 1\) possible generalizations.

**Proof.** Generalization replaces a subset of categories with new general categories. So the number of generalizations is equal to the number of subsets of categories containing more than one category. There are \(2^c\) subsets of categories, of which \(c\) consist of a single category and one is the empty subset. So there are \(2^c - c - 1\) subsets containing more than one category. \(\square\)

Furthermore, there is another problem: not all the possible generalizations make sense at the semantic level. Moreover, in literature authors disagree on how to apply generalization: some propose a global usage, others
local. On the one hand, global usage implies a greater loss of information, because computed also records that do not need to be changed. On the other hand, local usage is more difficult and makes data analysis more complex. Also suppression has some issues: it is not used homogeneously in literature; in fact some authors propose to suppress the whole tuple, while other propose to suppress only sensitive attributes. In addition when a suppression is applied, the user cannot analyze data without highly specific software. In conclusion, for ordinal attributes, using generalization and local suppression to achieve k-anonymity is far from perfect, but could still be considered. However, for continuous attributes in a quasi-identifier, generalization and local suppression are definitely inadequate. For all the reasons above other MPTs were proposed.

3.2.2 Disclosure risk and information loss

The central question in microdata protection is avoiding information disclosure that can be related to specific individuals. It is necessary to keep a good balance between the right of the individual to the privacy and the right of the society to the knowledge. The solution to this issue is in the middle between [39]:

1. no modification, i.e. maximal data utility but no privacy, and
2. total encryption, i.e. absolute privacy but no data utility.

To reach this balance, the performance of any MPTs is measured in terms of information loss and disclosure risk.

Disclosure risk

There are two types of information disclosure: identity disclosure and attributes disclosure. Identity disclosure means that a specific identity can be linked to a tuple in the microdata table. Attributes disclosure means that a portion of information related to a limited subset of respondents can be revealed. Generally, the factors that have the major impact on the identity disclosure are two:

- population uniqueness: it means that the probability that a respondent, the unique with a specific combination of attributes, was identified is high if these attributes are findable in microdata;
- re-identification: it means that the released microdata is linked to another published table, where the identifiers have not been removed.
The main method for measuring the risk of identity disclosure is record linkage. For other methods we refer to [12].

Record linkage consists in finding a match between a tuple in microdata and a tuple in a public external and non-anonymous information source (for instance a voter list). Since it is not possible to determine how many are the external sources that can be used by a malicious user, statistical analyses are computed on protected microdata. Record linkage methods can be classified in three categories: deterministic, probabilistic and distance-based.

**Deterministic methods** try to find an exact match between one or more attributes of tuples in different datasets. The main issue of these methods is in not considering that attributes have a certain importance in finding a link.

**Probabilistic methods** are defined in this way: given two datasets $D_1$ and $D_2$, the set of all the possible pair of tuples $(d_{1i}, d_{2j})$ is computed, where $d_{1i} \in D_1$ and $d_{2j} \in D_2$. Each pair is associated with a probability that depends by the frequency of each real match. If the probability is lower than a fixed value $T_1$, the tuples are not considered linked; if the probability is more than a fixed value $T_2$ the pair is considered a match; if the probability is between $T_1$ and $T_2$ a human evaluation is needed to decide if the pair is a match or not.

**Distance-based methods** are defined in this way: given two datasets $D_1$ and $D_2$, each tuple $d_{1i} \in D_1$ is matched to the nearest tuple $d_{2j} \in D_2$. These methods require the definition of a distance function and are not suitable for categorical attributes.

**Information loss**

The information loss measure is strictly related to the cases for which information is used. So it is not possible to establish a general measure of information loss. However, used measures are based on the concept of analytically valid and analytically interesting: the former means that microdata preserve approximately its statistical features, such as mean and co-variance; the latter means that microdata contains an adequate number of attributes that can be analyzed in the correct way.

In our work we will use as measures of information loss the **Sum of Square Errors** (SSE), the **Sum of Absolute Square Errors** (SSA) and the **Total Sum of Squares** (SST), that are defined as follows:

\[
SSE = \sum_{i=1}^{g} \sum_{j=1}^{n_i} (x_{ij} - \bar{x}_i)(x_{ij} - \bar{x}_i) \quad (3.1)
\]

\[
SSA = \sum_{i=1}^{g} n_i(\bar{x}_i - \bar{x})(\bar{x}_i - \bar{x}) \quad (3.2)
\]
\[ SST = SSE + SSA = \sum_{i=1}^{g} \sum_{j=1}^{n_i} (x_{ij} - \bar{x}_i') (x_{ij} - \bar{x}_i) \] (3.3)

where \( g \) is the number of clusters created by partitioning the dataset, \( n_i \) is the cardinality of the \( i \)-th cluster, \( x_{ij} \) denotes the \( j \)-th data vector in the \( i \)-th cluster, \( \bar{x}_i \) denotes the average data vector over the \( i \)-th cluster and \( \bar{x} \) denotes the average data vector over the whole set of \( n \) respondents.

A measure \( IL \) of the percentage of information loss can be obtained from:

\[ IL = \frac{SSE}{SST} * 100 \] (3.4)

### 3.2.3 Micro-aggregation

Micro-aggregation results a natural approach for k-anonymity. It is a family of perturbative MPTs (see §3.2), originally defined for continuous data and recently extended to categorical data.

Micro-aggregation methods represent records with \( d \) attributes in a dataset \( D \) as points in the \( \mathbb{R}^d \) Euclidean space. From this point of view, we can give the following definition [39]:

**Definition 3.2.2** (Micro-aggregation problem). *Given a data set \( D \) with \( n \) records in a characteristic space \( \mathbb{R}^d \), the problem consists in obtaining a \( k \)-partition \( P \) of \( D \), so that the SSE of \( P \) is minimized. Once \( P \) is obtained, each record of every part of \( P \) is replaced by the average record of the part.*

Then, given the original data file, micro-aggregation consists of building small clusters partitioning data and then substituting each original data with the centroid of the correspondent cluster. “\( k \)” is a parameter of the method. The larger the \( k \), the larger the information loss and the lesser the disclosure risk.

A \( k \)-partition is said to be optimal if minimizes the information loss in each cluster \( g \).

For this reason each cluster \( g \), obtained through micro-aggregation, has a cardinality such that \( k \leq |g| < 2k \), according to the following propositions [40]:

**Proposition 3.2.1.** An optimal solution to the \( k \)-partition problem of a set of data exists such that its clusters have cardinality greater than or equal to \( k \) and less than \( 2k \).

**Proof.** The dimension of a cluster in a \( k \)-partition is always greater than \( k \). If \( P \) is an optimal partition and the \( j \)-th cluster is supposed to have a cardinality greater than \( 2k \), we can consider the partition \( P' \) resulting from
the division of the j-th cluster in smaller clusters with cardinality greater than k and less than 2k, leaving the rest of the clusters unchanged. It can be noted that $SSE_{P'} \leq SSE_P$, because the sum of squares of each cluster in $P'$ is less than or equal to the sum of squares of each cluster in $P$. Therefore, $P'$ is also optimal. If $P'$ still contains clusters with more than 2k records, we can repeat the process to obtain $P''$, where each cluster has less than 2k elements. At this point $P''$ is optimal and has each cluster with less than 2k elements.

Definition 3.2.3. For a given dataset, a k-partition $P$ is said to be finer than a k-partition $P'$ if every cluster in $P$ is contained by a cluster in $P'$.

Definition 3.2.4. For a given dataset, a k-partition $P$ is said to be minimal with respect to the relationship “finer than” if there is no k-partition $P' \neq P$ such that $P'$ is finer than $P$.

Proposition 3.2.2. For a given dataset, a k-partition $P$ is minimal with respect to the relationship “finer than” if and only if it consists of groups with sizes $\geq k$ and $< 2k$.

Proof. Let $P$ be a minimal k-partition. By definition, the size of all its clusters is $\geq k$. If there is a cluster with size $\geq 2k$, it can be possible to split it in clusters of sizes $\geq k$ and $< 2k$, obtaining the k-partition $P'$. Then, $P'$ is finer than $P$. It follows that $P$ is not minimal unless its clusters have sizes $\leq 2k$. On the other hand, let $P$ be a k-partition having clusters of size $\geq k$ and $< 2k$. If a cluster in $P$ is split, then at least one of the obtained clusters will have size $< k$. Such as, any partition obtained from splitting one or more clusters in $P$ is not a k-partition. It follows that $P$ is minimal.

By the propositions above we obtain the following corollary:

Corollary 3.2.1. An optimal solution to the k-partition problem of a set of data exists that is minimal with respect to the relationship “finer than”.

Thus the search space for a k-partition is reduced to partitions constituted by clusters of size between $k$ and $2k$ elements.

Micro-aggregation has the following advantages:

- it is a unified approach, unlike the methods that combine generalization and suppression;
- optimal micro-aggregation is an NP-hard problem, but nearly optimal heuristics method exist;

26
it does not complicate data analysis;

- it protect continuous data without removing their numerical semantic.

In order to preserve information as much as possible, it is fundamental to try to find clusters whose centroid is close enough to all the records in the cluster. Because of this, micro-aggregation tries to minimize the SSE (see formula 3.1) and then also the SST (see formula 3.3). Micro-aggregation methods are divided in two different categories depending on the way the clusters are built [41]:

- data oriented, that allows to create clusters with a variable number of records (always between k and 2k, according to Prop. 3.2.1), making more homogeneous clusters and reducing the SSE value;

- fixed size, in which all clusters have the same cardinality k, except one if the number of records is not divisible by k. This family of methods are faster than data oriented ones but they increase the SSE value.

In the following section we will discuss the most relevant micro-aggregation methods.

### 3.2.4 Present approaches

One of the most commonly used fixed-size heuristic micro-aggregation method is the Maximum Distance to Average Vector (MDAV). This method works as follow: it takes as input the dataset D and it computes the average record \( \bar{x} \) from all the records. Then, it looks for \( x_r \), the furthest record to \( \bar{x} \) and for \( x_s \), the furthest record to \( x_r \). Afterwards, MDAV forms two clusters, one around \( x_r \) and one around \( x_s \), removing from the dataset D the records inserted in these two clusters. Then it computes again all these operations, until D is empty. We will describe in detail the MDAV algorithm in §4.1. It must to be said that, although the complexity of the fixed size methods is lower than that of the data oriented ones, MDAV has a complexity \( O(n^2) \), so for very large values of n, this method does not scale very well.

An improved algorithm is the Centroid Based Fixed Size (CBFS) proposed in [42]. In this method the global centroid \( \bar{x} \) is fixed from the beginning, so CBFS does not recompute it in each step of the algorithm, allowing to improve performance. However complexity remains of the order of \( O(n^2) \).

MDAV has also some data-oriented variants. For instance, in [43] an algorithm similar to MDAV is proposed: V-MDAV (Variable MDAV). This algorithm adds a step in the cluster creation process for extending the cluster size if some conditions are verified. With this approach it is possible
to improve clusters homogeneity, but computational time increases significantly.

In [44], a new algorithm based on bipartite graphs is presented. This approach has an interesting property: the obtained SSE is upper bounded, that is the obtained SSE is not greater than twice the SSE obtained through the optimal micro-aggregation. However, the SSE values obtained with this method are equal or worse to the ones obtained with MDAV. Finally, in [45], an alternative method for data oriented micro-aggregation based on non-homogeneous generalizations (i.e. not all the records grouped in the same cluster are generalized in the same way) was presented. Although this proposal is relatively fast, it still presents a complexity of $O(n^2)$ like MDAV.

In literature there are also many dat-oriented methods. For instance, in [46], the usage of a partition based on a $2^d$–tree structure is proposed, for partitioning the whole dataset in several subsets, limiting the maximal and minimal number of records for each partition. Instead, in [49] a different approach is proposed: the Euclidean Space is divided in small uniform clusters (i.e. Hypercubes of the Space) and then each record is positioned in the Hypercube to which it belongs. This method, that we will describe in detail in §4.2, fits very well for uniform distributions and allows to reduce computational costs to $O(n)$. However this method cannot be applied to other distributions that differs from the uniform one.

As we said, multivariate micro-aggregation is an NP-hard problem, but in literature there are several polynomial approaches to the univariate case. Therefore, many authors use projection techniques to reduce the multivariate micro-aggregation problem to the univariate case. A couple of approaches that follow this proposal are defined in [47] and [48]. In [47], data points are projected into a one-dimensional space using the Hilbert space filling curves. In [48], the set of records is split in two halves, according to the value of the attribute with greatest variance. Then the method is applied recursively, until all partitions have between k and 2k elements.

In [49] authors define a new method to create clusters, called importance partitioning. Each cluster is created using a sequential optimization algorithm, instead of finding at the same time the k-1 records closest to a certain record. Given an initial record, authors first find the nearest record in the dataset. Then, records are added one by one, since a k-cluster is created, choosing each time the record nearest to the center of the cluster. This technique allows to obtain more homogeneous clusters, but has a high computational cost. However, this method can be applied to several micro-aggregation algorithms (based on geometrical strategies, such as MDAV and CBFS) and it can be a good tool that allows to reduce information loss.

Finally, when datasets are very large, micro-aggregation techniques are inop-
erable in practice. For this reason it is very common the use of partitioning methods. The idea is to partition data in several parts and then to apply micro-aggregation separately to each part. Although partitioning methods allow to protect very large datasets, it is important to note that with this approach the SSE values increase, reducing the utility of the protected data.

Discussion

The majority of the approaches depicted above are computationally costly. Sometimes a solution is needed before a given deadline or as soon as possible. This is the reason why novel approaches to speed up clustering are needed as we will see in the following.
Chapter 4

k-anonymity on GPUs

As mentioned above, it is interesting to leverage GPU resources to improve the performance of any algorithm that is partially or totally parallelizable. In literature there are several proposed methods to improve a particular k-clustering technique, the k-Means algorithm [50]. Actually this algorithm is commonly used for data mining and not for micro-aggregation. In fact, it does not guarantee k-anonymity but only a partition of the dataset in k clusters. So the meaning of the “k” is different from that of the k-anonymity. However, results due to GPU usage are very interesting [50] [51] [52]. In our work we will apply techniques, similar to those cited above, to the micro-aggregation problem. We will study in particular two micro-aggregation algorithms: MDAV and HyperCubes.

4.1 MDAV

The Maximum Distance to Average Vector (MDAV) algorithm, or rather its generic version, was proposed in 2003 by Hundepool et al. as an evolution of the multivariate fixed-size micro-aggregation method described in [40] by Domingo-Ferrer and Mateo-Sanz.

The steps of the algorithm are the following:

1. The average record $\bar{x}$ of all records in the data set is computed. The most distant record $x_r$ to the average record $\bar{x}$ is considered (using the squared Euclidean distance).

2. The most distant record $x_s$ from the record $x_r$ considered in the previous step is found.

3. Two clusters are formed around $x_r$ and $x_s$, respectively. One cluster
contains \( x_r \) and the \( k - 1 \) records closest to \( x_r \). The other cluster contains \( x_s \) and the \( k - 1 \) records closest to \( x_s \).

4. If there are at least \( 3k \) records which do not belong to the two clusters formed in Step 3, go to Step 1 taking as new data set the previous data set minus the clusters formed in the previous instance of Step 3.

5. If there are between \( 2k - 1 \) and \( k \) records which do not belong to the two clusters formed in Step 3, compute: a) the average record \( \bar{x} \) of the remaining records; b) the most distant record \( x_r \) from \( \bar{x} \); c) the cluster containing the \( k - 1 \) records closest to \( x_r \); d) the other cluster containing the rest of records. Exit the Algorithm.

6. If there are less than \( k \) records which do not belong to the clusters formed in Step 3, form a new cluster with those records and exit the Algorithm.

All these steps are formalized in Alg. 1.

4.1.1 Analysis of the algorithm

Even if MDAV is one of the most used micro-aggregation algorithm, it must to be said that it does not always reach an optimal result. In Fig. 4.1 it can be noted that the cluster 4, which contains four records inside, has a record (the red one) that does not belong to its optimal cluster.

![Figure 4.1: Output of the MDAV algorithm on a simple example](image)
Algorithm 1 MDAV: Maximum Distance to Average Vector

Data: Data set $D$, Integer $k$, Integer $n$.
Result: $k$-partition.

while $n < 3k$ do
  $\bar{x} \leftarrow \text{ComputeCentroid}(D)$;
  $x_r \leftarrow \text{ComputeMostDistantRecord}(D, \bar{x})$;
  $x_s \leftarrow \text{ComputeMostDistantRecord}(D, x_r)$;
  for $x \in \{x_r, x_s\}$ do
    $\text{ClusteredSet}_i \leftarrow \text{ElaborateCluster}(D, x)$;
    $n \leftarrow (n - k)$;
    $D \leftarrow D \setminus \text{ClusteredSet}_i$;
    $i \leftarrow i + 1$;
  end for
end while
if $n \geq 2k$ then
  $\bar{x} \leftarrow \text{ComputeCentroid}(D)$;
  $x_r \leftarrow \text{ComputeMostDistantRecord}(D, \bar{x})$;
  $\text{ClusteredSet}_i \leftarrow \text{ElaborateCluster}(D, x_r)$;
  $n \leftarrow (n - k)$;
  $D \leftarrow D \setminus \text{ClusteredSet}_i$;
  $i \leftarrow i + 1$;
  $\text{ClusteredSet}_i \leftarrow \text{ElaborateLastCluster}(D)$;
else
  $\text{ClusteredSet}_i \leftarrow \text{ElaborateLastCluster}(D)$;
end if
return $k$-partition: ClusteredSet.
This behavior, due to the use by MDAV of fixed size clusters, can cause an increase of information loss, i.e. the obtained clusters does not represent accurately the dataset. In fact, as can be noted, the red record in the figure should be placed in cluster number 3, instead of cluster number 4. Improvements to this MDAV behavior are proposed in [43] with V-MDAV. In the article authors add a step that dynamically extends a computed cluster if some distance conditions are verified.

Now we will present an analysis of the computational complexity of the algorithm. Before MDAV algorithm stops, it has to create $\left\lfloor \frac{n}{k} \right\rfloor$ clusters of $k$ records (i.e. $k$-clusters). Assuming that $\frac{n}{2}$ ungrouped records remain on average, the complexity of creating one cluster is given by:

- **CenC** (Centroid computation): the computation of the centroid of the remaining ungrouped records: $O\left(\frac{n}{2}\right)$;
- **MDRC** (Most distant record computation): the search of the most distant record $x_r$ from the centroid, which requires $\frac{n}{2}$ distance computations: $O\left(\frac{n}{2}\right)$;
- **OMDRC** (Other most distant record computation): the search of the most distant record $x_s$ from $x_r$, which requires $\frac{n}{2} - 1$ distance computations: $O\left(\frac{n}{2}\right)$;
- **CluC** (Clusters computations): the creation of two clusters, one around $x_r$ and one around $x_s$. This step takes $O\left(\frac{n}{2}\right)$ operations for each record inserted in one of the two clusters. So for this step the total number of operations is $O\left(2(k - 1)\frac{n}{2}\right)$.

Therefore, since there are $\left\lfloor \frac{n}{2k} \right\rfloor$ iterations, the overall computational time is given by: $[O\left(\frac{n}{2}\right) + 2 * O\left(\frac{n}{2}\right) + O((k - 1)n)] * O\left(\frac{n}{2k}\right) = O\left(\frac{n^2}{4k} + \frac{n^2}{2} \right) \approx O\left(n^2\right)$. Established that the MDAV algorithm has a quadratic complexity, our main goal is to find out the steps of the algorithm that can be parallelized, in such a way to improve MDAV performance reducing its computational cost.

Unfortunately, the use of the GPUs cannot change the asymptotic behavior of an algorithm, so what we can do is to reduce the computational cost to $O\left(\frac{n^2}{M}\right)$, where $M(> 1)$ is a variable that depends on the features of the GPU at our disposal (for ex. number of cores).

### 4.1.2 How we parallelized MDAV on GPUs

As we can see, MDAV use really intuitive geometric steps to create the final $k$-partition. In our work we tried to parallelize the followings:
- the computation of the centroid $\bar{x}$;

- the search for $x_r$ and $x_s$;

- the creation of each k-cluster.

It is simple to note that all these operations are computations of maximum and minimum distances: maximum distances for the elaboration of $\bar{x}$, $x_r$ and $x_s$; minimum distances for the elaboration of the clusters around $x_r$ and $x_s$. Maximums and minimums are not highly parallelizable, because their final results depends by each single record. Then, during computation a sort of comparison between records is needed. For this reason, GPU’s threads cannot work on single records independently. To solve this issue we proposed the following method, based on a map-reduce approach and shown in Fig. 4.2:

1. we introduce a new parameter, the $\text{chunkSize}$, that allows to establish how many records each GPU thread will process;

2. we define a vector, $\text{indexes}$, that contains the indexes of the records which have to be still inserted in a cluster;

3. each thread processes adjacent records $x_i, \ldots, x_{i+\text{chunkSize}}$ and store the relative max/min in the i-th position of $\text{indexes}$;

Figure 4.2: Example of computation of the Max on GPU with records with number of attributes=1 and $\text{chunkSize}=3$. To make the image more readable, the vector of the indexes, that actually has an important role, is omitted.
4. to complete the procedure another GPU call is needed: this execution compares the values stored in *indexes* in the previous step and gives as output the absolute max/min, that is subsequently passed to CPU;

5. the *indexes* vector, modified during the previous steps, is restored, with the exception of the index found in step 4.

Finally the computational flow of our hybrid version of MDAV (see §5.2) is depicted in Figure 4.3.

![Figure 4.3: Frame of GPU based MDAV, with compaction each 10000 records](image)

In §4.3 we will discuss about the main parallelization issues that we encountered.

### 4.2 HyperCubes

In literature there are several micro-aggregation algorithms that allow to solve the k-anonymity problem with a complexity equal to $O(n^2)$. Trying to improve this computational time Solanas and Di Pietro proposed in [39] a
new algorithm.
Assuming a uniform distribution of the records the main idea consists in partitioning the hyperspace to which records belongs, instead of compute records similarity through the squared Euclidean distance. By partitioning the hyperspace the records are implicitly grouped according to their distribution. The records that are in the same hypercube are very likely to be similar. Note that, as mentioned in §3.2.4 this algorithm can be applied only to datasets with uniform distribution.
The algorithm is shown in Alg. 2.

Algorithm 2 Efficient Hyper-Cubes-based Micro-aggregation

**Data:** Data set $D$, Integer $k$, Integer $n$.

**Result:** $k$-partition.

```
D' ← Normalize(D);
l ← ComputeHypercubeEdgeLength(n, k);
HC ← GenerateSetOfHypercubes(l);
for each record $r_i$ in $D'$ do
    <HC$_j$, $r_i$> ← AssignRecordToHypercube(HC, $r_i$);
    Card(HC$_j$):= Card(HC$_j$)+1;
end for
for each Hypercube in HC do
    if Card(HC$_j$) < k then
        Neigh=DetermineHypercubeNeighbour(HC$_j$);
        FuseHypercubes(HC$_j$, Neigh);
        HC ← RemoveHypercubeFromSetOfHypercubes(HC$_j$);
        Card(Neigh):= Card(Neigh)+Card(HC$_j$);
    end if
end for
return set of tuples $<HC_j, r_i>$
```

In details, the algorithm receives as input the parameter $k$ and the dataset $D$ of the records, which are treated as points of the $\mathbb{R}^d$ hyperspace. So the records are stored as tuples of length $d$. In the first step the records contained in $D$ are normalized to assure that the hyperspace has a unitary edge. The next step computes the optimal length $l$ of each hypercube edge, that depends by the number of records $n$ and the parameter $k$. $l$ can be computed as:

$$l = \sqrt{\frac{k}{n}}$$
Once $l$ is determined, the set of hypercubes $HC$ is created. An hypercube is determined by its bounds in each dimension. Each hypercube is identified by a unique index. The number of hypercubes is given by the formula:

$$NofHC = (1/l)^d$$

After the generation of the $HC$ set, each record is assigned to an hypercube $HC_i$. After these steps all records are assigned to an hypercube. However, due to the fact that $l$ is an approximation, a little fraction of hypercubes can contain less then $k$ records. As $k$-anonymity is ensured, all hypercubes must contain at least $k$ records. So low-cardinality hypercubes are fused with their closest neighbours. Note that, by construction, every low-cardinality hypercube has a complete hypercube adjacent to it. The fusion of these hypercubes is computationally cheap.

Once all low-cardinality hypercubes have been fused, the algorithm finishes and returns the set of tuples $< HC_j, r_i >$, that represents the obtained $k$-partition.

### 4.2.1 Analysis of the algorithm

During the computation the algorithm creates exactly $(1/l)^d$ hypercubes. The steps that contribute to the computational costs are the following:

- **Normalization**: it is not relevant. It can be assumed that the data set is just normalized. However the normalization can be computed and its complexity can vary depending on the way it is done. So normalization can be guaranteed with a computational time of $O(n)$.

- **Hypercubes generation**: this step consists simply in the creation of the $HC$ set. If the hypercubes are considered in a logical way (i.e. created at runtime during the records assignment phase), this step does not affect the complexity of the algorithm.

- **Records assignment**: this is the focal point of the algorithm. Each dimension of each record is compared with the dimensional bounds of the hypercubes and it is added to the correct hypercube. As each record has to be verified, the computational cost of this step is $O(n)$. Note that, due to the hypercubes usage, the distance computation between the records is not needed, unlike MDAV.
• **Hypercubes fusion**: as mentioned above, this step is computationally cheap and can be not considered in the computational costs.

Then, the overall computational time is given by $O(n) + O(n) = O(2n)$. Moreover, if normalization is not considered, complexity becomes $O(n)$.

**Issues**

Although the HyperCubes algorithm is the first in literature that is able to micro-aggregate very large datasets in linear time, it has some issues anyway.

- The algorithm does not work well with small values of the parameter $k$. In fact, the goodness of the algorithm is related to a probability measure: the probability that at least an hypercube has a number of records that is less than $k/2$ and more than $2k$. This measure is governed by:

$$Pr \leq \frac{1}{6 \ln n}$$

To comply with this rule we have to impose $k = 12 \ln n$. So, if the number of records is very large, we have to choose a large value for the $k$ parameter. For a more complete discussion, see [39].

Thus, if it is necessary to obtain clusters of very small cardinality, it is possible to use this algorithm to partition the data in clusters of smaller cardinality and then apply a classical micro-aggregation algorithm to refine the $k$-partition.

- The algorithm can stop with some hypercubes of cardinality more than $2k$. So the partition obtained is not always an optimal partition. To solve this issue it can be added a final step that splits hypercubes with a too large cardinality.

- The algorithm works well only with records with uniform distribution. With different ones, the SSE increases, due to the fact that records are not uniformly positioned in the hyperspace and it can be that many of these were placed in a single hypercube, whilst other hypercubes remain empty.

This issue can be solved relaxing the assumption that $l$ is a constant. Particularly it is possible to tune $l$ to leverage the different data density in the hyperspace. Another approach consists in the use of *fuzzy boundaries*: the edges of each hypercube are not fixed, i.e. $l$ is not constant and can vary during computation if an hypercube cardinality is too high or too small.
4.2.2 How we parallelized HyperCubes on GPUs

The first thing to note is that the HyperCubes algorithm is easier to parallelize than the MDAV one. In fact, while MDAV requires max/min computations, that are not totally parallelizable, HyperCubes uses simple comparisons which can be computed independently. Then the part of algorithm that we parallelized is the assignment of the records to the hypercube they belong to.

![Diagram of GPU based HyperCubes]

To minimize memory usage we enumerated the hypercubes and used a vector, “HCcard”, whose length is the number of the hypercubes and that contains in the i-th cell the cardinality of the i-th hypercube. To store in which hypercube each record is positioned, we used a vector, “position”. The GPU kernel analyzes each record and stores its position in the relative cell of “position”, incrementing by one the cardinality of the hypercube to which the record belong.

Let \( \overline{x} = (x_1, \ldots, x_n) \) be a record: to compute the position of \( \overline{x} \) we used the formula:

\[
    \text{position}(\overline{x}) = \left[x_1 \ast t\right] \ast t^0 + \left[x_2 \ast t\right] \ast t^1 + \ldots + \left[x_n \ast t\right] \ast t^{n-1}
\]
where \([\bullet]\) is the floor function and \(t\) is given by
\[
t = \sqrt[\sqrt{N}]{\text{OfHC}} = \frac{1}{t}
\]

Finally, the part of the algorithm that merge the hypercubes with a cardinality lower than \(k\) is implemented in the serial way. If a hypercube has a low cardinality, it is merged with a randomly chosen neighbour.

Figure 4.4 shows the computational flow of our parallel version of HyperCubes.

4.3 GPU issues we encountered and solved

The first issue in the use of GPUs and Aparapi was due to the fact that GPUs memory is not shared with CPUs memory. As such we had to take care about the transfer cost and try to minimize the buffered transfers; Aparapi supports only primitive scalars and simple arrays of primitives. So we were forced to work only with vectors.

MDAV

Instead of the method outlined in Fig. 4.2 we could use another one easier to implement: we could launch a single thread for each record and compare each distance with a primitive value \(\text{Max/Min}\) readable by all threads. Despite the simplicity of implementation, we would have to synchronize threads to avoid concurrent writings on \(\text{Max/Min}\); this would result in a loss of performance. So we opted for the approach described in §4.1.2

At the beginning of our work we used the Aparapi implicit buffer handling, that allows to keep the code simpler, avoiding to manage explicitly transfers between CPU and GPU. Therefore, using the Aparapi implicit buffer we can simply launch the GPU kernel (i.e. the code that we want to run on the GPU) through a couple of code lines, leaving to Aparapi the task to copy the data back and forth from the GPU. This approach has an important issue: since the data buffer is managed directly by Aparapi, for each launch of the kernel all data are transferred on GPU and then, when the GPU finishes its tasks, data are transferred back on CPU. This makes performance widely worse and represents the main parallelization bottleneck.
To solve this issue, we converted the code, changing the buffer from implicit to explicit. The explicit buffer handling allowed us to copy data on the GPU memory only during the first kernel execution and leave data on it until the end of the last execution. So if on the one hand the code became less readable, on the other hand we gained a good performance improvement.

The explicit buffer handling is setted via calling the method `setExplicit(true)` of the `Kernel` class: this setting allows to enable the methods `put(.)`, with which data can be transferred from CPU to GPU, and `get(.)`, with which it can be done the converse.

Note that to compute $\pi$, $x_r$, $x_s$ and each k-cluster, different portions of code are needed. Unfortunately, a kernel cannot have inner methods. Furthermore, kernels does not share memory. As such, the definition of more than one kernel results in the need of copy data multiple times on the GPU memory; clearly this approach should be avoided, because it wastes memory resources. Waiting for the possibility of defining multiple methods into a single kernel, we solved this issue according to: we divided the kernel code in several portions, one for each method needed, in a form similar to a “switch” (i.e. the methods are numbered). When we needed to launch a particular method, we simply had to specify the number of the method and run the kernel. The usage of an emulated multiple entrypoints kernel needed the manage of the GPU via a scheduling: thus the CPU had to handle the various launches of the different portions of the kernel code.

HyperCubes

As we have parallelized HC in a simpler way with respect to MDAV, we did not encounter many issues. However, when the GPU kernel analyzes each record and stores its position in the relative cell of “position”, it needs to increase by one the cardinality of the correspondent hypercube: we synchronized this step with the “kernel.atomicAdd” method provided by Aparapi to avoid concurrent writings by different GPU threads.

Another issue that had to be considered, is due to the “bank conflict”. When a vector is stored in the CPU or in the GPU memories, its elements are organized in banks of memory: each bank contains some elements of the vector. Therefore, if there is the need to read two adjacent elements, it cannot be done at the same time. In fact, the elements of each bank can be accessed one at a time. As such, when we work on GPU, we have to pay attention to this feature that can potentially decrease performance. However, we overcame this limitation adopting a smarter `chunkSize` (similar to the one we used in MDAV, see §5.2). As such, each GPU thread analyzes a fixed number of
adjacent elements, one at a time. In this way we avoided bank conflict issues.
Chapter 5

An in-depth evaluation of synthesized code

5.1 Experimental test-bed

In this chapter we will discuss the experimental results that we obtained using some PCs provided by Roma Tre University:

- a PC featuring AMD Athlon™ II X4 640 Processor and the GPU AMD Radeon HD 6800 Series (lab 19);
- a PC featuring AMD Phenom™ II X4 945 Processor and the GPU NVIDIA GeForce GT 640 (lab 20);
- a PC featuring Intel(R) Core(TM) i5 CPU and the GPU NVIDIA Tesla C2050 / C2070 (lab 21).

All the PCs support OpenCL 1.2, Java 7 and the last Aparapi version (2013-01-23).
Particularly we will show the performance improvements that we gained using parallel versions of MDAV and HyperCubes algorithms.

5.2 MDAV

Our work mainly focused on the implementation of a parallel version of the MDAV algorithm.
A first simple result is depicted in the chart of Fig. 5.1. It can be noted that, despite the usage of the GPU is not very advantageous for a small size of the
problem, it can improve relevantly performance when the dimension of the problem grows up. Particularly, in Fig. 5.1 we can note that on lab20 the usage of the parallel version of MDAV becomes convenient when the number of records to be processed are about 40000, whilst on lab21 we obtain the same benefits at about 70000 records. It is important to emphasize that the performance improvement are strictly related to the used hardware (i.e. the particular CPU and GPU). In fact, lab21 features our most performant GPU, but also our most performant CPU. Furthermore, on lab19, that features a slower GPU, the advantages are not as good as on the others architectures. Note that in the figure we measured the time for the computation of only two k-clusters at a time. For this reason the computational costs seem to be linear instead of quadratic.

As mentioned, in the MDAV algorithm results and performance depends
by two input parameters, the number of records, \( n \), and the size of each cluster, \( k \). We said also that in order to parallelize MDAV, we introduced a new parameter, the \( \text{chunkSize} \). Theoretically the computational time of the algorithm is related to these three parameters: clearly the higher the number of records, the higher the computational time. Actually, the \( k \) parameter, that is relevant in terms of data utility and information loss, it does not influence performance, because the MDAV complexity is \( O(n^2) \): so it does not depend by \( k \). Finally, about the \( \text{chunkSize} \): this parameter allows to determine how many records each GPU thread processes at the same time, in order to find the max/min (see §4.1).

As our results show, the \( \text{chunkSize} \) affects performance. If we choose a chunk size too large, we force the GPU to use few processors: this increases the execution time for each max/min; if we choose a chunk size too small, we force the GPU to use more cores than those available. For instance, if we have a GPU with 512 cores and we require the launch of 520 different threads, the GPU will execute the first 512 threads and then, when each launched thread will be finished, it will execute the remaining 8 ones. This causes two rounds of execution on the GPU. Too many GPU launches waste time. Therefore, there is a chunk size value that allows to optimize the execution and that is related to the particular used hardware.

To verify all these observations we have represented in Fig. 5.2 and 5.3 the results of the performance of lab20, and lab21 with \( \text{chunkSize} \in [50 : 400] \), \( k \in [3, 9] \) and \( n \) (i.e. the number of records) fixed to 60000, 100000 and 200000. Each record has dimension equal to 3 (i.e. each record has 3 attributes). Note that for this experiments we used a synthetic dataset of records with uniform distribution. It can be noted that the lowest ratio \( \frac{\text{GPU execution time}}{\text{CPU execution time}} \) is obtained on lab 20, even if the fastest GPU is featured by lab 21. Thanks to these graphs the best values of the \( \text{chunkSize} \) that allows to improve performance are findable empirically.

The experimental results suggest that the optimal value for the \( \text{chunkSize} \) is similar to \( c\sqrt{n} \), where \( c > 0 \) is a value that depends by the hardware. However, this has not been verified and has to be investigated yet. In fact, while in the graphs for lab19 and lab21 the trend of the \( \text{chunkSize} \) is quite regular, in the graphs for lab20 the trend is discontinuous and does not allow to determine with precision the optimal value for the \( \text{chunkSize} \).

In Fig. 5.4, 5.5 and 5.6 we have plotted at the same time the results obtained with different \( \text{chunkSize} \) values on lab 19, 20 and 21, to better show the influence of this parameter on performance. Moreover \( n \in [0 : 200000] \), \( k \in [3 : 9] \) and \( \text{chunkSize} \in [50 : 400] \). It must to be said that the computational time remains \( O(n^2) \), even if it seems that the trend of the graphs is linear, due to the GPU parallelization factor.
Figure 5.2: Performance as execution time in seconds of parallel MDAV on lab 20 with $k=3,5,7,9$ and $\text{chunkSize} \in [50 : 400]$. Records follows uniform distribution. We fixed $n$ to 60000, 100000 and 200000, to show when the use of the GPU starts to be beneficial. Note that the $\text{chunkSize}$ value is relevant in terms of performance. On this architecture the trend is discontinuous. Furthermore, when $n$ is small the GPU usage is not beneficial.
Figure 5.3: Performance as execution time in seconds of parallel MDAV on lab 21 with \( k=3,5,7,9 \) and \( \text{chunkSize} \in [50 : 400] \). Records follow uniform distribution. We fixed \( n \) to 60000, 100000 and 200000, to show when the use of the GPU starts to be beneficial. Note that the \( \text{chunkSize} \) value is relevant in terms of performance. On this architecture the trend is quite regular. Furthermore, when \( n \) is small the GPU usage is not beneficial.
Figure 5.4: Performance as execution time in seconds of parallel MDAV on lab 19 with $k=3,5,7,9$, $\text{chunkSize} \in [50 : 400]$ and $n \in [0 : 60000], [0 : 100000], [0 : 200000]$. Records follow uniform distribution and have dimension equal to 3. We have plotted at the same time the results obtained with different $\text{chunkSize}$ values, to better show the influence of this parameter on performance. On lab 19, that features the slowest GPU at our disposal, parallelization in not beneficial for small values of $n$. 
Figure 5.5: Performance as execution time in seconds of parallel MDAV on lab 20 with $k = 3, 5, 7, 9$, chunkSize $\in [50 : \infty]$, and $n \in [0 : 100000]$, [0 : 200000]. Records follow uniform distribution and have dimension equal to 3. We have plotted at the same time the results obtained with different chunkSize values, to better show the influence of this parameter on performance. On lab 20, we obtained the best speed up.
Figure 5.6: Performance as execution time in seconds of parallel MDAV on lab 21 with $k=3,5,7,9$, $chunkSize \in [50 : 400]$ and $n \in [0 : 60000]$, $[0 : 100000]$, $[0 : 200000]$. Records follow uniform distribution and have dimension equal to 3. We have plotted at the same time the results obtained with different $chunkSize$ values, to better show the influence of this parameter on performance. On lab 21, we obtained the best parallel results in terms of execution time.
Moreover, Fig. 5.7 shows that records distribution does not affect the performance. This is in agreement with the MDAV algorithm: in fact, MDAV generates the k-clusters through max/min computations, that are not affected by the particular distribution.

Also the problem dimension (i.e. the dimension of the space in which each record lives) does not affect the trend of the graphs, as shown in Fig. 5.8.

Figure 5.7: Performance as execution time in seconds of parallel MDAV on lab 21 with $k=3$, $chunkSize \in [50:400]$ and $n \in [0:60000], [0:100000], [0:200000]$. We have plotted the results obtained with different two distributions: normal and Zipf. Trends remains the same.

Figure 5.8: Performance as execution time in seconds of parallel MDAV on lab 21 with $k=3$, $chunkSize \in [50:400]$ and $n \in [0:60000], [0:100000], [0:200000]$. We have plotted the results obtained with different problem dimension: records has 6 attributes instead of 3. Trend remains the same.
Our first parallel code had a bottleneck that affected performance: the vector of the records was not modified in the GPU memory even if a particular record was inserted in a k-cluster. It means that the GPU had to manage a large number of indexes even when the number of records to be elaborated became smaller. To solve this issue we modified the code: we introduced the possibility of compacting the vector of the records each time that a certain number of these have been clustered. In Figure 5.9 the benefits of this approach are depicted. The spikes are due to the need to copy back on the CPU the vector of the records each time we want to compact it. Despite each compaction causes a loss of time, it is still advantageous because it allows to reduce the time for the subsequent k-clusters creation, improving both partial and global performance. Compaction allowed to further improve performance, even if this choice became an input variable that has to be used carefully. In fact, too many compaction during the computation cause too many copies back and forth from the GPU memory and as mentioned this affects performance.

Figure 5.9: Compaction vs no compaction approaches of MDAV on lab 19 with $n \in [0 : 100000]$. Spikes are due to the compaction efforts.
Gathering all the observations above, we concluded the improvements of our code, implementing an hybrid version of the MDAV algorithm, that we called Hybrid MDAV. This implementation allows to choose whether and how to use compaction and re-adapts automatically the \textit{chunkSize} value after each compaction, following the trend of $c\sqrt{n}$. The best benefit of Hybrid MDAV is in allowing the choice of using GPU only when it is more convenient than CPU. For instance, on lab20, when the number of records to be still clustered is less than 60000-70000 we do not have benefits in using parallelization, so computation is moved to the CPU. Note that the moment in which computation is shifted depends on the particular hardware that we are using.

Thanks to Hybrid MDAV we could finally reach the main goal of our work: we shown that GPUs could be very useful in improving security techniques performance through parallelization. The improvements are shown in Fig. 5.10, where it is depicted how, thanks to parallelization, it is possible to reduce the costs of MDAV in terms of computational time. Particularly we tested Hybrid MDAV:

- on lab 20 with $n \in [0 : 10^6]$. When $n = 10^6$ our hybrid version has a speed up equal to 2.3;
- on lab 21 with $n \in [0 : 2 \times 10^6]$. When $n = 2 \times 10^6$ our hybrid version has a speed up equal to 1.8.

![Figure 5.10: Performance of Serial MDAV and of our Hybrid MDAV on lab 20 with $n \in [0 : 10^6]$ and on lab 21 with $n \in [0 : 2 \times 10^6]$. It is very interesting to note the good speed up obtained thanks to our hybrid version of the algorithm.](image-url)
To conclude the discussion of the experimental results of MDAV, we made a comparison chart about the performance of the architectures we have used. Figure 5.11 shows the importance of the GPU hardware with respect to the performance. The same algorithm works much better if we have at our disposal a more powerful graphic card. It is clear the difference in performance between AMD (lab 19) and NVIDIA (lab 20 and 21).

![Figure 5.11: Performance comparison as execution time in seconds of parallel MDAV between the GPUs featured by lab19, lab20 and lab21. The parameter k is set to 3, the number of records is equal to 100000 and the chunk size is fixed to the optimal for each architecture.](image)

5.3 HyperCubes

After implementing MDAV, we parallelized the HyperCubes algorithm. The performance of our parallel code are depicted in Figures 5.12, 5.13 and 5.14. We compared the parallel version with the serial one, varying the number of records in the range \([10^6 : 9 \times 10^6]\). On lab19 the parallel version did not led
many improvements (note that lab19 features the slowest GPU); conversely on lab20 and on lab21 parallelization increased performance. It has to be noted that HyperCubes is faster than MDAV and computation ends after a few seconds. For this reason, the data transfer from CPU to GPU is not irrelevant, even if in our code is done only once. Then, if we had at our disposal an APU (Accelerated Processing Unit), in which CPU and GPU share memory, we would have been able to increase performance further.

Figure 5.12: Comparison between serial and parallel HyperCubes algorithms on lab19

Figure 5.13: Comparison between serial and parallel HyperCubes algorithms on lab20
The main issue in the parallelization of HyperCubes was the scarce GPU memory. In the experimental charts we could not study datasets with more than 9 millions records. The graphs in Figures 5.12, 5.13 and 5.14 were extended with pointed lines, according to performance trend, to show how the performance would have been if we had at our disposal GPUs with more memory capacity.

5.4 Information loss

In our work we did not focus on the analysis of the loss of information caused by data aggregation. In fact, our parallel algorithms produce the same results of the serial ones (i.e. parallelization do not affect SSE and SST). As such, the information loss is the same as for the serial case. However, we computed the information loss $IL$ for different values of $n$ and $k$. More evaluations can be found in [39] and [56].

<table>
<thead>
<tr>
<th>n</th>
<th>k</th>
<th>MDAV Inf. Loss</th>
<th>HC Inf. Loss</th>
<th>Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>3</td>
<td>1.32</td>
<td>2.63</td>
<td>1.31</td>
</tr>
<tr>
<td>1000</td>
<td>3</td>
<td>0.27</td>
<td>0.55</td>
<td>0.28</td>
</tr>
<tr>
<td>100000</td>
<td>3</td>
<td>0.06</td>
<td>0.13</td>
<td>0.07</td>
</tr>
<tr>
<td>1000</td>
<td>5</td>
<td>2.81</td>
<td>3.82</td>
<td>1.01</td>
</tr>
<tr>
<td>1000</td>
<td>5</td>
<td>0.53</td>
<td>0.85</td>
<td>0.32</td>
</tr>
<tr>
<td>100000</td>
<td>5</td>
<td>0.11</td>
<td>0.19</td>
<td>0.08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>k</th>
<th>MDAV Inf. Loss</th>
<th>HC Inf. Loss</th>
<th>Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>7</td>
<td>3.58</td>
<td>5.00</td>
<td>1.42</td>
</tr>
<tr>
<td>10000</td>
<td>7</td>
<td>0.75</td>
<td>1.15</td>
<td>0.40</td>
</tr>
<tr>
<td>100000</td>
<td>7</td>
<td>0.16</td>
<td>0.24</td>
<td>0.08</td>
</tr>
<tr>
<td>1000</td>
<td>9</td>
<td>4.31</td>
<td>5.82</td>
<td>1.51</td>
</tr>
<tr>
<td>10000</td>
<td>9</td>
<td>0.93</td>
<td>1.35</td>
<td>0.42</td>
</tr>
<tr>
<td>100000</td>
<td>9</td>
<td>0.20</td>
<td>0.29</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Figure 5.15: Comparison between the information loss of MDAV and HyperCubes
As we expected HC causes a higher loss of information, although it allows to reduce the overall computational cost. On the other hand, MDAV is better in terms of information loss, but it has a quadratic complexity instead of linear. As proposed by its authors, HC could be a very helpful pre-process step, whose results were exploited by a classical micro-aggregation algorithm, such as MDAV, to obtain a better k-partition in a reasonable time.

5.5 Other issues

Aparapi helped us in our work, allowing us to implement our parallel code in a natural way, given its object-oriented nature. Nevertheless there are some main issues, that we have already mentioned, that could be studied and improved.

- The implicit buffer handling is very useful for simple codes and very manageable for beginners who want to join the GPGPU computing. In fact it allows to hide the difficulties of managing the buffers between CPU and GPU. However, this Aparapi feature is not so useful in implementing more complex projects: there is a lack of performance due to too many data transfers between CPU and GPU. So for more complex projects users are forced to use the explicit buffer handling, that results in a worsening of code readability. However, even using the explicit buffer handling, the code is still more readable and simpler than the corresponding one in the OpenCL version.

- Unfortunately Aparapi does not support multiple entrypoints for kernels. So the management of multiple methods is not very easy. We could use multiple kernels but it would be useless because different kernels does not share memory. This issue, that probably will be solved soon by the Aparapi developers, can be circumvented with a multiple entrypoints simulation, as mentioned in §4.1.2.

- Another important issue is the limited GPU memory. GPU memories have a capacity of not more than some hundreds of Megabytes. So the amount of data that can be stored on them is not very large. For instance, in our case, we cannot store on the GPU vectors with cardinality greater than $10^7$. This could be a big issue with micro-aggregation algorithms, that require the whole dataset to be stored in the GPU memory. However, at the moment real microdata datasets does not exceed the cardinality of $10^6$ records. Furthermore, GPUs are evolving, increasing their memory and their performance. As such, the
trend towards successful GPU usage in micro-aggregation is set.
Chapter 6

Discussion and conclusion

Recent GPU improvements led to an increased use of these hardware devices for tasks not related to graphical issues. With the advent of GPGPU (see §2.2), programmers have taken advantage of GPUs increasingly, trying to improve a wide variety of algorithms via parallelization. According to the need to make GPUs easily programmable, the manufacturers of the graphic cards developed some parallel computing libraries such as CUDA, by NVIDIA, and OpenCL, by Khronos Group. Privacy and security problems can benefit from parallel computing resources.

Contributions

In our work we contributed to improve GPU usefulness for security, showing that it is possible to improve performance thanks to this hardware component. Furthermore, Aparapi allowed us to extend Java’s paradigm “Write Once Run Anywhere” to parallel computing.

To reach this goal we have studied the k-anonymity problem, emphasizing its mathematical aspects but also the need for fast results calculation, in order to keep the privacy of the respondents of a large amount of datasets all over the world.

We have performed an in-depth thorough study of the actual parallelization issues of algorithms on GPUs. As a consequence, we have learnt a big deal of information over the advantages, benefits and caveats of such approach. Particularly, we have focused on the micro-aggregation technique, investigating two algorithms, MDAV and HyperCubes. We have improved their performance and we have also proposed an hybrid version for the MDAV algorithm.

We seen that parallelization is very helpful in improving performance of algorithms which contain parallelizable portions of code. Furthermore, GPUs
allow the parallelization to be very performant, even if there are still several issues related to the usage of graphic cards.
In conclusion we have collected a large amount of high-quality experimental data that can be used further to better model actual GPU usage. These GPU collected results are particularly interesting, as the trend for future GPUs and CPUs will see:

1. a widening of the gap for performance and
2. a unified address space, resulting in a scaling issues improvement.

Lessons learnt

Our work allowed us to make some observations about Aparapi and parallelization.
Despite several issues, such as the lack of support for multiple entrypoints kernel, Aparapi has shown to be a very powerful tool, that allows to fully exploit the power of the GPUs.
We have learnt how to improve performance avoiding the main bottleneck due to GPU usage: the data transfer between CPU and GPU. We have also learnt how to achieve the best performance from different hardware architectures, managing the chunkSize value (see §5.2).
In conclusion it is not trivial to parallelize on GPUs: the development of parallel code implementing tools is not finished yet.

Future works

After having devised a hybrid version of the MDAV algorithm, we are going to improve our approach allowing the micro-aggregation of larger datasets.
In practice we are going to overcome the GPU memory limits by partitioning the dataset to be micro-aggregated and by computing micro-aggregation of different subsets on different GPUs. In this approach it is important to pay attention to how data are distributed on the GPUs, in order to avoid the increase of the SSE. A solution to this issue consists in totally ordering the records in the dataset, by one or two components, before partitioning it [40].
Appendix A

Relevant excerpt from Parallel MDAV and HyperCubes codes

Hybrid MDAV Kernel class

//Kernel inputs:
//final int[] indexes
//final int[] nOfRec

import com.amd.aparapi.Kernel;

public class HybridKernel extends Kernel {

    int[] result=new int[1];
    int[] spdim=new int[1];
    int[] border=new int[2];
    int[] nOfRec=new int[1];
    int[] flag=new int[1];
    int[] blockDim=new int[1];
    int[] removable=new int[1];
    int[] count=new int[1];
    int[] k=new int[1];
    int[] indexes;
    int[] auxIndexes;
    int[] toBeRemoved;
    float[] records;
    float[] z;
    float[] r;
    float[] s;
    float[] sum;

    final static int kMax=1;
    final static int kMin=2;
    final static int kRemoveIndex=3;
final static int kSetIndexes=4;
final static int kCentroid=5;
final static int kSetAuxIndexes=6;
final static int kSetFlag=7;

int function;

@Override public void run(){

    //The following portion of code find
    //the most distant record from Centroid

        if(function==kMax){
            //t is the number of records analyzed by each thread
            int i=0, j=0, w, t=0, x, init, stop;
            int gid = getGlobalId();
            int size=getGlobalSize();
            float dist=0, max=0;
            if(size!=1){
                if(gid!=size-1){
                    t=blockdim[0];
                }else{
                    t=border[1];
                }
            }

            init=gid*blockdim[0];
            stop=(gid*blockdim[0])+t;

            for (i=init;i<stop;i++){
                dist=0;
                x=indexes[i];
                for(j=0;j<spdim[0];j++){
                    dist=dist+(records[j+x]-z[j])*(records[j+x]-z[j]);
                }
                dist=(float)sqrt(dist);
                if(dist>max){
                    max=dist;
                    indexes[gid*blockdim[0]]=indexes[i];
                }
            }
        }

        if(size==1){
            if(nOfRec[0]%blockdim[0]==0){border[0]=n0fRec[0]/blockdim[0];}
            else{border[0]=(nOfRec[0]/blockdim[0])+1;}

}
max=0;
for(i=0;i<border[0];i++){
    dist=0;
    x=indexes[i*blockdim[0]];
    for(j=0;j<spdim[0];j++){
        dist=dist+(records[j+x]-z[j])*(records[j+x]-z[j]);
    }
    dist=(float)sqrt(dist);
    if(dist>max){
        max=dist;
        indexes[0]=x;
    }
}
result[0]=indexes[0];

if (flag[0]==0){
    for(i=0;i<spdim[0];i++){
        r[i]=records[w+i];
        z[i]=r[i];
    }
    flag[0]=1;
} else {
    for(i=0;i<spdim[0];i++){
        s[i]=records[w+i];
    }
    flag[0]=0;
}

//The following portion of code find
//the closest record to r or s

} else if(function==kMin){
    int i=0, j=0, x, y, init, stop, t=0;
    int gid = getGlobalId();
    int size=getGlobalSize();
    float dist=0, min=1000000000;
    if (flag[0]==0){
        for(i=0;i<spdim[0];i++){
            z[i]=r[i];
        }
    } else {
        for(i=0;i<spdim[0];i++){
            z[i]=s[i];
        }
    }
}

if (size != 1) {
    if (gid != size - 1) {
        t = blockDim[0];
    } else {
        t = border[1];
    }

    init = gid * blockDim[0];
    stop = (gid * blockDim[0]) + t;

    for (i = init; i < stop; i++) {
        dist = 0;
        x = indexes[i];
        for (j = 0; j < spdim[0]; j++) {
            dist = dist + (records[j + x] - z[j]) * (records[j + x] - z[j]);
        }
        dist = (float)sqrt(dist);
        if (dist < min || i == init) {
            min = dist;
            indexes[gid * blockDim[0]] = x;
            toBeRemoved[gid] = i;
        }
    }
}

if (getGlobalSize() == 1) {
    if (nOfRec[0] % blockDim[0] == 0) { border[0] = nOfRec[0] / blockDim[0]; }
    else { border[0] = (nOfRec[0] / blockDim[0]) + 1; }

    min = 1000000000;
    for (i = 0; i < border[0]; i++) {
        dist = 0;
        y = i * blockDim[0];
        x = indexes[y];
        for (j = 0; j < spdim[0]; j++) {
            dist = dist + (records[j + x] - z[j]) * (records[j + x] - z[j]);
        }
        dist = (float)sqrt(dist);
        if (dist < min || i == 0) {
            min = dist;
            indexes[0] = x;
            removable[0] = toBeRemoved[i];
        }
    }
}
result[0]=indexes[0];
nOfRec[0]=nOfRec[0]-1;

if(blockdim[0]>nOfRec[0]) {blockdim[0]=blockdim[0]/2;}
}

// The following portion removes the index of
// the last record inserted in a k-cluster
}

} else if(function==kRemoveIndex){
    int t=getGlobalId()+removable[0];

    indexes[t]=auxIndexes[t+1];

// The following portion reset 'indexes' vector
// modified during the last max/min computation
}

} else if(function==kSetIndexes){
    int gid=getGlobalId();

    indexes[gid]=auxIndexes[gid];

// The following portion compute Centroid
}

} else if(function==kCentroid){
    int i=0, j=0, w, t=0, x, init, stop;
    int gid = getGlobalId();
    int size=getGlobalSize();
    float dist=0, max=0;

    if(size!=1){
        if(gid!=size-1){
            t=blockdim[0];
        } else{
            t=border[1];
        }

        init=gid*blockdim[0];
        stop=(gid*blockdim[0])+t;

        for (i=init;i<stop;i++){
            x=indexes[i];
            w=gid*spdim[0];
            for(j=0;j<spdim[0];j++){
                sum[w+j]=sum[w+j]+records[x+j];
            }
        }

        // Additional code for Centroid calculation...
    } else{
        // Special handling for singleton case...
    }
}
if (size==1) {
    if (nOfRec[0]%blockdim[0]==0) {border[0]=nOfRec[0]/blockdim[0];}
    else {border[0]=(nOfRec[0]/blockdim[0])+1;}
    for (i=1;i<border[0];i++) {
        w=i*spdim[0];
        for (j=0;j<spdim[0];j++) {
            sum[j]=sum[j]+sum[w+j];
            sum[w+j]=0;
        }
    }
    for (j=0;j<spdim[0];j++) {
        z[j]=sum[j]/nOfRec[0];
        sum[j]=0;
    }
    border[1]=nOfRec[0]-(nOfRec[0]/blockdim[0])*blockdim[0];
}

// The following portion reset 'auxIndexes' vector:
// an auxiliary vector
else if (function==kSetAuxIndexes) {
    int gid=getGlobalId();
    auxIndexes[gid]=indexes[gid];
    if (gid==0) {
        border[1]=nOfRec[0]-(nOfRec[0]/blockdim[0])*blockdim[0];
        if (count[0]%k[0]==k[0]-1) {
            if (flag[0]==0) {
                flag[0]=1;
            } else {
                flag[0]=0;
            }
        }
        count[0]++;
    }
}

// The following portion manage 'flag' variable
// needed for the GPU scheduling
else if (function==kSetFlag) {


if(flag[0]==0){
    flag[0]=1;
} else{
    flag[0]=0;
}

Hybrid MDAV Kernel
(OpenCL auto-generated code)

typedef struct This_s{
    int function;
    __global int *blockdim;
    __global int *border;
    __global int *indexes;
    __global int *spdim;
    __global float *records;
    __global float *z;
    __global int *nOfRec;
    __global int *result;
    __global int *flag;
    __global float *r;
    __global float *s;
    __global int *toBeRemoved;
    __global int *removable;
    __global int *auxIndexes;
    __global float *sum;
    __global int *count;
    __global int *k;
    int passid;
}This;
int get_pass_id(This *this){
    return this->passid;
}

__kernel void run(
    int function,
    __global int *blockdim,
    __global int *border,
    __global int *indexes,
    __global int *spdim,
    __global float *records,
    __global float *z,
    __global int *nOfRec,
    __global int *result,
    __global int *flag,
__global float *r,
__global float *s,
__global int *toBeRemoved,
__global int *removable,
__global int *auxIndexes,
__global float *sum,
__global int *count,
__global int *k,
int passid
)
{
    This thisStruct;
    This* this=&thisStruct;
    this->function = function;
    this->blockdim = blockdim;
    this->border = border;
    this->indexes = indexes;
    this->spdim = spdim;
    this->records = records;
    this->z = z;
    this->nOfRec = nOfRec;
    this->result = result;
    this->flag = flag;
    this->r = r;
    this->s = s;
    this->toBeRemoved = toBeRemoved;
    this->removable = removable;
    this->auxIndexes = auxIndexes;
    this->sum = sum;
    this->count = count;
    this->k = k;
    this->passid = passid;
    
    if (this->function==1){
        int i = 0;
        int j = 0;
        int t = 0;
        int gid = get_global_id(0);
        int size = get_global_size(0);
        float dist = 0.0f;
        float max = 0.0f;
        if (size!=1){
            if (gid!=(size - 1)){
                t = this->blockdim[0];
            } else {
                t = this->border[1];
            }
        }
        int init = gid * this->blockdim[0];
        int stop = (gid * this->blockdim[0]) + t;
        i = init;
for (; i<stop; i++){
    dist = 0.0f;
    int x = this->indexes[i];
    j = 0;
    for (; j<this->spdim[0]; j++){
        dist = dist + ((this->records[(j + x)] - this->z[j]) * 
                       (this->records[(j + x)] - this->z[j]));
    }
    dist = sqrt(dist);
    if (dist>max){
        max = dist;
        this->indexes[gid * this->blockdim[0]] = this->indexes[i];
    }
}
}
if (size==1){
    if (((this->nOfRec[0] % this->blockdim[0])==0){
        this->border[0] = this->nOfRec[0] / this->blockdim[0];
    } else {
        this->border[0] = (this->nOfRec[0] / this->blockdim[0]) + 1;
    }
    max = 0.0f;
    i = 0;
    for (; i<this->border[0]; i++){
        dist = 0.0f;
        int x = this->indexes[(i * this->blockdim[0])];
        j = 0;
        for (; j<this->spdim[0]; j++){
            dist = dist + ((this->records[(j + x)] - this->z[j]) * 
                           (this->records[(j + x)] - this->z[j]));
        }
        dist = sqrt(dist);
        if (dist>max){
            max = dist;
            this->indexes[0] = x;
        }
    }
    int w = this->indexes[0];
    this->result[0] = w;
    if (this->flag[0]==0){
        i = 0;
        for (; i<this->spdim[0]; i++){
            this->r[i] = this->records[(w + i)];
            this->z[i] = this->r[i];
        }
        this->flag[0] = 1;
    } else {
        i = 0;
        for (; i<this->spdim[0]; i++){
this->s[i] = this->records[(w + i)];
}
this->flag[0] = 0;
}
} else {
...
...
...
} else {
    if (this->function==7){
        if (this->flag[0]==0){
            this->flag[0] = 1;
        } else {
            this->flag[0] = 0;
        }
    }
} return;
}

Parallel HyperCubes Kernel

//Kernel inputs:
//final float[] records, final int[] HCcard,
//final int[] position, final int[] support

//support[0]= number of records
//support[1]= problem dimension
//support[2]= 1/(hypercubes edge)
//support[3]= chunkSize

Kernel kernel=new Kernel(){
    @Override public void run(){
        int gid = getGlobalId();
        int i, h, w, y;
        int which;

        for(h=0;h<support[3];h++){
            which=0;
            y=-1;
            for(i=gid*support[1]*support[3]+h*support[1];i<gid*support[1]*support[3]+
                h*support[1]+support[1];i++){
                y++;
                which = (int)(which+(int)(records[i]*support[2])*pow(support[2],y));
            }
        }
    }
}
w=gid*support[3]+h;
position[w]=which;
atomicAdd(HCcard,which,1);
}
Bibliography


[32] https://groups.google.com/forum/?fromgroups=#relaxtopic/aparapi-discuss/ryA-8IvjI0AU/overview


