Universidade de Aveiro Electrónica, Telecomunicações e Informática,
2011

Rui Figueiredo

Massively parallel identification of RFID tags

Computação massivamente paralela para identificação de marcadores RFID

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2011

Rui Barbosa de Figueiredo

Massively parallel identification of RFID tags

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor André Zúquete, e do Professor Doutor Tomás Oliveira e Silva, Professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**António Manuel de Brito Ferrari Almeida** Professor Catedrático da Universidade de Aveiro (por delegação do Director de Curso)
vogais / examiners committee	**Leonel Augusto Pires de Seabra de Sousa** Professor Catedrático do Instituto Superior Técnico, Universidade Técnica de Lisboa
	André Ventura da Cruz Marnôto Zúquete Professor auxiliar da Universidade de Aveiro (orientador)
	Tomás António Mendes Oliveira e Silva Professor associado da Universidade de Aveiro (co-orientador)

agradecimentos / acknowledgements

Aproveito este espaço para deixar algumas palavras de agradecimento á família e amigos, por me aturarem nos momentos em que estive presente, mas sobretudo por compreenderem os longos períodos de ausência, motivados pelo forte desejo de ver este mestrado concluído.

"Last but certainly not least", aqui vai um grande obrigado ao meus dois orientadores, por me manterem motivado ao reconhecer o trabalho aqui empenhado, sempre com grande simpatia e disponibilidade.

Resumo

Nos dias que correm, tem-se assistido a uma grande evolução dos sistemas de identificação através de marcadores RFID, frequentemente sem se dar a devida importância à componente de privacidade nos mesmos. A presente dissertação pretende explorar um paradigma de identificação de marcadores com o intuito de colmatar esta lacuna, recorrendo à utilização de uma função dificilmente invertível, criptográfica ou de síntese, para a geração no marcador de um identificador pseudo-aleatório a partir do identificador real do mesmo, bem como de um conjunto de números aleatórios gerados pelo marcador e pelo leitor. Contudo, torna-se necessária uma pesquisa ao longo de todos os identificadores atribuídos, que por questões de desempenho é realizado de uma forma massivamente paralela. Desta forma, impede-se o seguimento de objectos ou pessoas associados ao marcador por entidades ilegítimas, que não tenham acesso a uma base de dados de todos os identificadores atribuídos

Abstract

In recent years, there has been a large evolution of identification systems through the use of RFID tags, often with some disregard for privacy concerns. In this dissertation a paradigm will be explored focusing on the use of a well known cryptographic standard or hashing function to generate a pseudo-random identifier from the real identifier as well as a set of random nonces from the tag and reader. However, a search is required along the set of assigned identifiers, which for the sake of performance shall be done resorting to a massively parallel approach. This way, it becomes unfeasible for an illegitimate reader to relate two activation sessions of the same tag without access to the database of all the assigned identifiers.

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

1.1 Motivation: untraceable car identification

RFID tags are being progressively used to identify many types of objects, namely cars traveling on highways. But car identification with RFID creates a security problem, as it enables (1) the tracking of car movements by unauthorized persons and (2) the execution of actions initiated by the proximity of a particular car (e.g. bomb explosion). Therefore, car tags should not use constant identifiers to prevent unauthorized identifier-based tracking activities.

On the other hand, using random identifiers and an higher-level application protocol capable of identifying the tag, as in electronic passports, is not particularly interesting, since the contact time between the legitimate reader and a moving car may be too short to enable it.

1.2 Objectives

With this in mind, the main objective of this work will be to study a paradigm of management for identifiers sent by RFID tags that would prevent its tracking. In order for this to happen, the tags must generate different identifiers every time they are activated and only legitimate entities should be able to correlate those identifiers. As such, the identifier generated by the tag should be the result of a hard-to-invert function (**f**) applied to a base identifier $\mathbf{ID}_{\boldsymbol{base}}$, that would remain constant throughout the life of the tag, and to a couple of random numbers generated by itself, **tagRandom**, and another generated by the reader, **readerRandom**, as follows:

$$\mathbf{ID}_{\boldsymbol{i}} \equiv < \mathbf{tagRandom}_{\boldsymbol{i}}, \mathbf{f}(\mathbf{ID}_{\boldsymbol{base}}, \mathbf{readerRandom}_{\boldsymbol{i}}, \mathbf{tagRandom}_{\boldsymbol{i}}) > \tag{1.1}$$

If the function **f** is not invertible, such as a hashing function, like SHA-1 [1] or MD5 [2], or a cryptographic function, such as AES [3], where $\mathbf{ID}_{\boldsymbol{base}}$ is the constant tag identifier, then a search for the latter performed by an illegitimate reader will require an exhaustive traversal of the entire base ID space, which can be as large as required in order to minimize the probability of discovery.

A legitimate reader, however, should be able to recover $\mathbf{ID}_{\boldsymbol{base}}$ from $\mathbf{ID}_{\boldsymbol{i}}$. This is technically viable if the reader has access to a database of all the assigned $\mathbf{ID}_{\boldsymbol{base}}$ values, requiring

the computation of $\mathbf{ID}_i$ for all the base IDs in the database until one is found that matches $\mathbf{ID}_{base}$ for the given random numbers.

For an external observer, the tag identifier will look as a purely random number, thus not conveying any useful information regarding the object it is attached to, or even its owner. But for someone possessing the base ID of the tag, it is easy to check whether or not the computed, pseudo-random tag identifier is the correct one.

As a side-effect, this cryptographically-based, pseudo-random identifier generation implementation prevents a tag from being cloned, as long as we are able to keep the secrecy of the base ID deployed in each tag. Nevertheless, the compromise of a tag's base ID is not a problem for all other base IDs, since they are all different and (preferably) random. This side effect was already identified in previous works (e.g. [4]).

1.3 Possible privacy-preserving tag identification protocol

The tag identification protocol runs conceptually as follows (see Figures 1.1 and 1.2). First, both the tag and the reader generate random challenges, **tagRandom** and **readerRandom**. The reader communicates its challenge to the tag, which uses both challenges and its base ID to compute an identifier with a given non-invertible function.

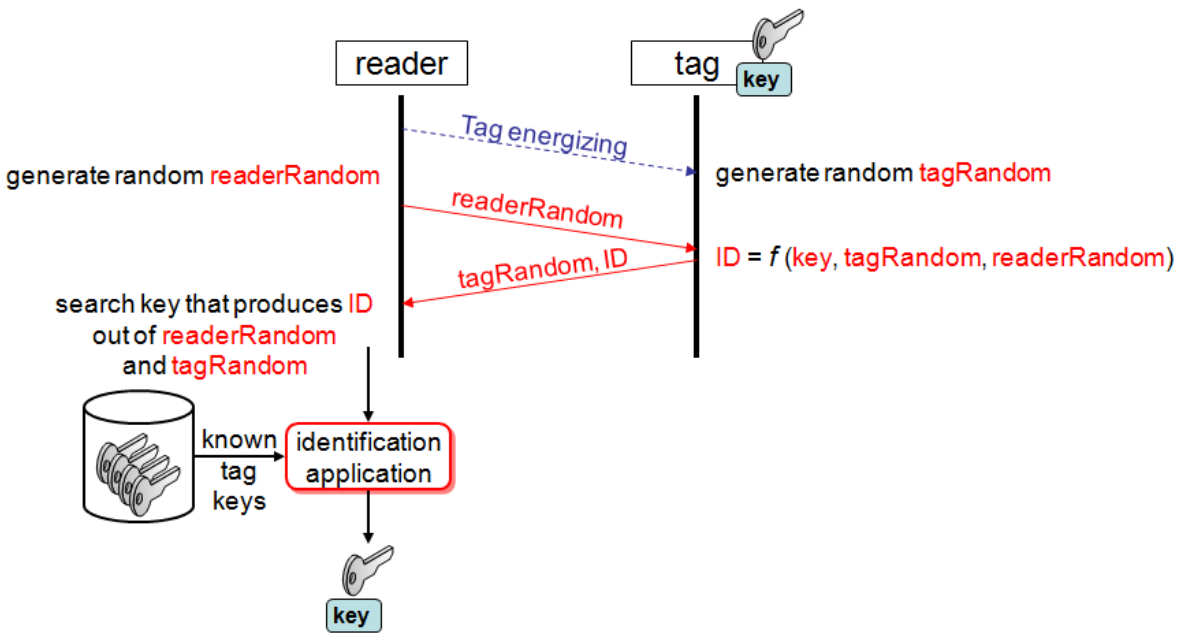

Figure 1.1: Tag identification protocol using the MD5 digest function: conceptual message exchanges

The tag then sends its challenge and the identifier to the reader. The identification application will then search through all the known base IDs, corresponding to valid tags, to find out the one that generated the identifier from the two challenges used. This exhaustive search is similar to a password guessing attack using a dictionary.

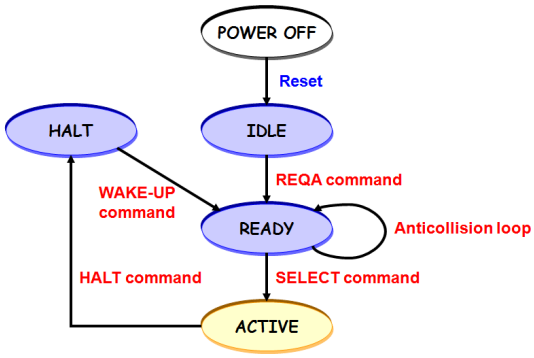

Figure 1.2: Tag identification protocol using the MD5 digest function: ISO 14443 A RFID tag state transition

---

For our tests we used a very demanding set of dimensions for the challenges and the base IDs: 64-bit challenges (**tagRandom** and **readerRandom**) and 128-bit keys and identifiers ($\mathbf{ID}_{base}$ and $\mathbf{ID}_i$). This dimension for base IDs makes it unfeasible to perform an exhaustive search throughout the whole base IDs space.

This protocol can be implemented as an extension to the existing RFID standards, namely the widely used ISO 14443 A standard [5]. Figure 1.2 shows a diagram of the state transitions of an ISO 14443 A compliant RFID tag according to the messages received from a reader. The **tagRandom** value can be generated immediately after a tag Reset, before entering the IDLE state. **readerRandom** can be piggybacked in a REQA command, enabling the tag to compute the ID before entering the READY state; the ID is then used throughout the anticollision loop and in the tag selection. Finally, the tag can piggyback **tagRandom** in its SACK answer to the SELECT command, enabling the identification application to identify a tag immediately after its ID-based selection.

1.4 Problem

In spite of the good level of privacy enabled by this approach, a problem is created due to the costly computation required to infer $\mathbf{ID}_{base}$ from $\mathbf{ID}_i$. Since RFID systems are based on the cheapness of tags, relaying the cost to the readers and allowing the use of millions of the former, a possible scalability issue is raised. For large numbers of deployed tags, the information systems are required to search through possibly all of those base IDs (worst-case scenario) to find a match, which would require the computation of **f** as many times as there are base IDs.

Given the relatively small number of execution units on a traditional CPU, this easily parallelizable search can be, if the number of base IDs is in the hundreds of millions, very slow. In 2008 there were nearly 256 million highway vehicles in the US [6], so for a system of this type to be useful in real world applications, the time required to perform a search within a space of 250 million base IDs has to be relatively small.

1.5 Solution

Given the nature of the problem, this type of highly parallelizable operations will be better suited for a General Purpose Graphical Processing Unit, from now on solely referred to as GPGPU. This relatively recent technology could be used to perform the function **f** for many base IDs at the same time, in order to derive the base ID from a pseudo-random identifier. Due to its high availability and ease of programming, Nvidia GPGPUs will be used, using CUDA. The CUDA programming language was used instead of AMD Stream or OpenCL due to Nvidia's stronger commitment to the GPGPU world, and the greater market share that followed.

1.6 Contribution

On the following pages, we will focus on the optimization of different $\boldsymbol{f}$ unidirectional functions, and compare them with each other and their respective performance on different architectures. Next, a searching solution will be detailed in accordance with the protocol proposed in 1.3. Based on the premise that some base IDs will be used more often than others, and as such should be searched for before, several algorithms will be studied that allow a reduction of the average search time for a given workload. Finally, a distributed system will be proposed and implemented to assess the scalability of this model.

For simplicity's sake, the base IDs will be referred to hereafter as simply keys.

In terms of target architectures for performing massive key searching, our main target was the Nvidia S1070 Tesla GPGPU, with a total of four 240-core devices. In our performance tests we used only one of these devices, thus a quarter of its power. For establishing performance comparisons with other hardware platforms, we used two other GPGPUs, an Nvidia Fermi Geforce GTX480 (480 cores), and an Nvidia GT335M (72 cores). An hexacore CPU, the Intel Core i7 980X, with AES-NI capabilities was also used for the same purpose.

Chapter 2

Context

The following sections intend to familiarize the reader with the most important aspects of the underlying concepts of both GPU programming and the MD5 and AES algorithms, necessary to fully understand what will be discussed afterwards.

2.1 GPU

In today's society, computer-generated images are part of everyday life, be it from TV, advertising, movies, computer gaming, medical imaging and even newspapers. This helps explain the rapid evolution of computer graphics. A graphics processing unit (GPU), is an integrated circuit dedicated to perform the computations required to render graphics, offloading this work from the CPU for performance reasons, since these calculations are more efficiently performed by specialized hardware. These processors perform operations, mostly over matricial representations of data, in order to perform texture mapping, polygon rendering, rotation and translations of vertices of the polygons and oversampling and interpolation transformations to reduce aliasing. Modern GPUs also introduce the capability of decoding the most popular types of video compression standards.

Due to the increasing work that is being offloaded to Graphics cards of modern computers, they have become a small computer of their own, with a dedicated processor (the GPU itself) and RAM memory. This result stems from a growing trend of improving the visual component of Human-computer interactions. The aforementioned computations involve a great deal of parallelism, typically a small program where the machine instructions remain the same over different inputs/outputs. As such, up to hundreds of ALUs can be used in a single GPU and if one were to increase the programmability of said hardware, it stands to reason that the same could conceivably be used to solve computational problems from different areas.

It was this leap in programmability that marked the advent of the GPGPU, General Purpose Graphics Processing Unit, a Graphical processing unit in every way, with the added capability of being able to process data not related to the rendering of graphics. Examples of uses of this relatively new technology are being deployed in such fields as signal processing (both video and audio), ray-tracing, bioinformatics, financial modeling, fluid dynamics, physical simulations, statistical models, computer vision and cryptanalysis, and even the search for extra-terrestrial intelligence. [7, 8, 9, 10, 11, 12, 13]

Figure 2.1 shows a comparison of transistor usage in a CPU *versus* a GPU. It is clear that the CPU devotes a large portion of its silicon space to control flow and caching circuitry. On

the other hand, such mechanisms are not needed for GPUs, as the large volume of data being processed hides the latency of memory operations. In some cases, this enables performance improvements over CPUs of 2 orders of magnitude.

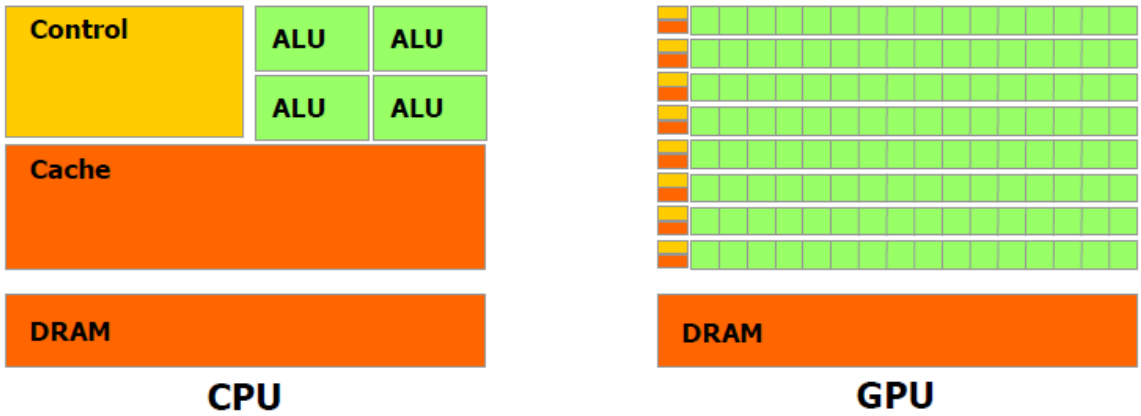

Figure 2.1: CPU vs GPU transistor usage, from [14]

Nvidia GPGPUs can be separated according to their compute capability (more information on section 2.2). Since this work was devised for a Tesla S1070 computing system, consisting of 4 GPGPUs of compute capability 1.3, this is the architecture that we will be focusing on. As such, every GPGPU is a collection of streaming multiprocessors (SM). Each SM has 8 processing cores, 16384 32-bit general purpose registers, and 16 KiB of shared memory [14]. More recent GPGPUs have different characteristics.

2.2 CUDA

CUDA, short for Compute Unified Device Architecture, is a proprietary parallel computing architecture developed by NVIDIA and introduced in November 2006 [14]. It provides a programming interface to NVIDIA GPUs, based on the C programming language [15], albeit with some restrictions and other extensions. These restrictions involve the inability to use recursion or function pointers, and the sub-program is required to run across multiple disjoint memory spaces, i.e., no overlapping of the data memory space is possible. Wrappers also exist for other high-level languages, such as Java, Perl, Python, among others.

Each CUDA parallel sub-program, called a kernel, is composed of a user-specified number of threads, each of which running exactly the same code (not necessarily at the same time). The threads are grouped in blocks. Each block is assigned to one SM. When there are enough hardware resources, up to 8 blocks can be running in a single SM. Threads of the same block can use simultaneously (part of) the shared memory of each SM; it is also possible to synchronize them. Threads of different blocks cannot be synchronized easily, and can only communicate via (slow) global memory.

The threads of each block are grouped in so-called warps, which are groups of 32 threads. All threads of a warp are executed simultaneously (actually, in 4 clock cycles because there are only 8 processing cores in each SM). Each SM queues the warps that can be run. To hide the long latency of memory accesses, many warps should be assigned to each SM, i.e., the

number of threads per block times the number of blocks than can be assigned to a SM should be large.

2.2.1 Thread coordinates

Each thread receives two sets of coordinates: one set to specify its coordinates within a block, and another set to specify the coordinates of the block. Blocks are organized in one-, two-, or three-dimensional arrays of threads; the dimensions of a block, and the coordinates of a thread within a block, are stored in the fields `x`, `y`, and `z` of the variable `blockDim`, respectively `threadIdx`. The blocks are organized in a one- or two-dimensional array of blocks, called a **grid**; the dimensions of a grid, and the coordinates of a block within a grid, are stored in the fields `x` and `y` of the variable `gridDim`, respectively `blockIdx`. Since each thread has access to these variables, it is easy to compute the global coordinates of a thread.

Figure 2.2 presents an example of a two-dimensional Grid, with one-dimensional blocks. For simplicity's sake, the number of blocks per thread has been reduced, but in a typical kernel it's value should be a multiple of 32 (the number of threads in a warp), such as 128 or 256. Note that this number has to be constant for every block in a Grid.

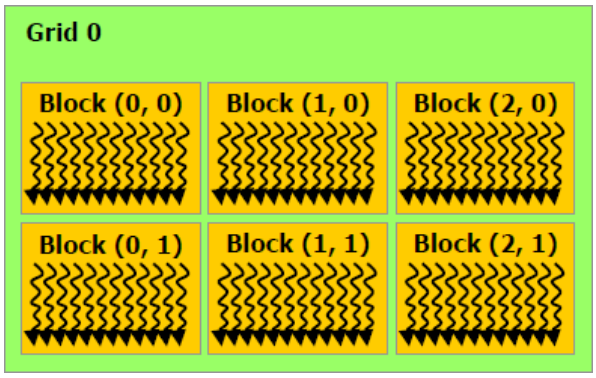

Figure 2.2: Grid Example. The legend Block(i,j) represents a block with coordinates `blockIdx.x=i` and `blockIdx.y=j`. Each thread inside each block is individually numbered with `threadIdx.x`.

For the threads of a given warp, threadIdx.y and threadIdx.z are constant, while threadIdx.x takes consecutive values. This has to be taken into consideration when accessing the memory. Each global memory access reads 32, 64, or 128 bytes. All, or at least most of the bytes read should be consumed by one of the threads of the warp. In our case, since the MD5 and AES algorithms perform a significant amount of computation for each key (which has 16 bytes), it was found to be convenient to store the four 32-bit words of each key in consecutive memory positions.

2.2.2 Threads' termination after a match

Terminating an exhaustive search when a match is found is not easy to do in the CUDA programming model when several devices are used at the same time. Terminating the threads of the same block is easy and fast; one only needs to test a completion flag stored in shared memory. Terminating the threads launched in a given device is also easy, but slower; one only needs to test a completion flag stored in the global memory (of that device). Terminating the

threads launched on the other devices is more complicated, requiring the use of page-locked memory, shared between all devices.

To test the feasibility of its use, a simple program spawning 4 kernels was created, one for each device of the the Tesla S1070 system. The first kernel simply terminated right after launch, setting a boolean variable (the one shared between all devices in page-locked memory) to true. The remaining kernels were locked in a loop, constantly testing that same boolean variable, terminating once it is set to true. Ideally, immediately after the launch of the kernels, one would expect all the kernels to terminate at once. However, due to the latency imposed by the slow inter-device communication mechanism, the time between the termination of the first kernel and the last was measured at up to 300 miliseconds. This overhead makes the exploitation of multiple devices for a single search unworkable, once we factor that the maximum amount of time required to find a match in a set of 100 million keys is, at most, half as long (see Section 6.4). As such, we focus our research on the use of a single device per search.

A much better solution to solve the scalability problem is the use of multiple devices were each one processes a different workload. It is this method that will be presented in chapter 7.1. In case the number of assigned keys becomes too large to fit in the memory of a single device, one can also distribute the array of keys between multiple devices, without the need to resort to page-locked memory.

2.2.3 CUDA memory model

In order to write optimized CUDA code, a deep understanding of the memory model is required.

Figure 2.3 illustrates the memory hierarchy for CUDA devices. Apart from the large number of 32-bit general purpose registers, threads also have access to a local, per-thread memory situated in global memory, and hence comparatively very slow (in the order of 400 to 600 cycles v.s. 4 [14]). A group of threads of the same block also have the ability to use a shared memory space. Two other models exist for constant data, i.e., data that is written by the host and consumed by the device, namely constant memory and texture memory. These two types of memory are well suited for when all threads read the same value, and due to it being cached, access times are comparable in terms of latency to the memory.

To provide the reader with a better understanding of the CUDA programming interface, refer to the Code Snippet 2.1. In this trivial example, an array of characters is copied from a memory position to another, in the GPGPU.

Initially, the host allocates the memory in the device's global memory, using the instruction *cudaMalloc*, the pointers input_d and output_d then become associated to positions of memory in the device, with no possibility of being used directly by the CPU. To copy memory to/from the device/host, *cudaMemcpy* is required. Once this setup is performed, the kernel is launched, and the GPU performs the code in function "kernel". Finally, the output array is copied to the host's RAM and displayed. To compile this program, one can simply execute "nvcc hello.cu" on any machine where the CUDA compiler is installed, and the single output file contains all the CPU and GPU code required. To run the program, a simple "./a.out" is all it takes, assuming a GNU/Linux system.

Note that the size of the grid in this case is just one block, with 12 threads, as specified by the new execution configuration syntax ("<<<...>>>") as an extension to the C language. Another instance of an extension in this example is the __global__ keyword, denoting a function

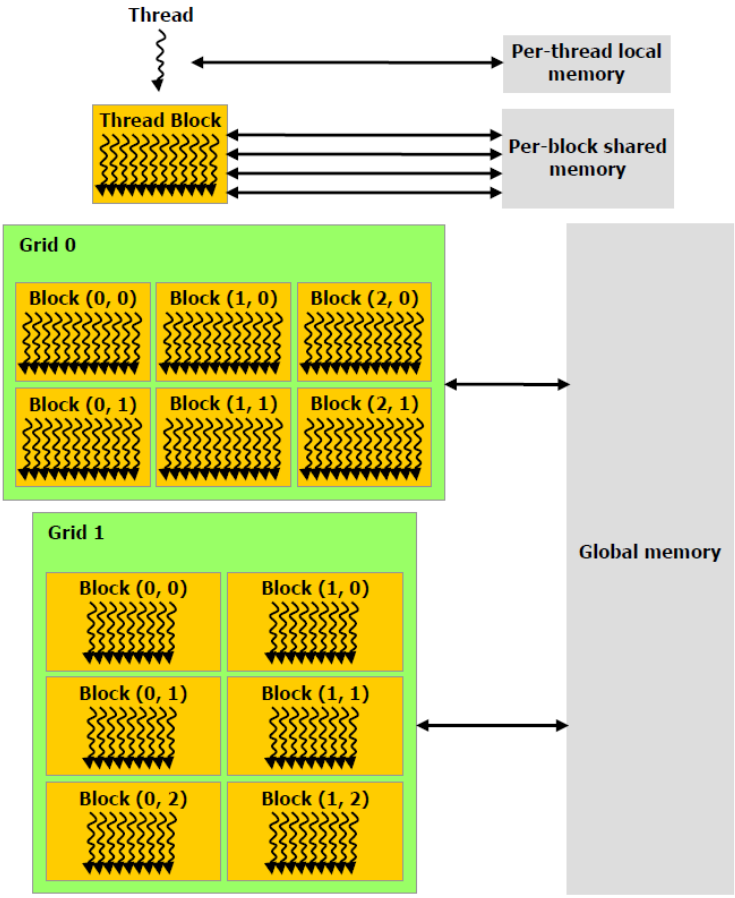

Figure 2.3: Memory hierarchy, from [14]

---

that can only be run in a GPGPU, and launched by a CPU with a given configuration syntax (grid dimension).

The important concept where there is a distinction in the instruction flow as opposed to a CPU-only application, lies in the single GPU instruction. All the 12 launched threads perform their work in parallel, i.e., at the same time, every thread has a sequential Id and that same Id is used to access a single memory position.

Code Snippet 2.1: CUDA Hello World

```
#include <stdio.h>
#include <cuda.h>

// Device (GPU) code
__global__ void kernel(char *input, char *output)
{
   // copy input to output
   output[threadIdx.x] = input[threadIdx.x];
}

// Host (CPU) code
int main()
{
   // Stored in RAM, not visible to the device
   char input[12] = "Hello World";
   char output[12] = "";

   // Device memory space
   char *input_d;
   char *output_d;

   // Allocate device memory (global memory)
   cudaMalloc( (void**)&input_d, 12*sizeof(char) );
   cudaMalloc( (void**)&output_d, 12*sizeof(char) );

   // copy input to device
   cudaMemcpy( input_d, input, 12*sizeof(char), cudaMemcpyHostToDevice );

   // Launch the kernel
   kernel<<<1, 12>>>(input_d, output_d);

   // Retrieve results
   cudaMemcpy( output, output_d, 12*sizeof(char), cudaMemcpyDeviceToHost );
   cudaFree( output_d );
   cudaFree( input_d );

   // Show the output
   printf("%s\n", output);
   return 0;
}
```

2.3 MD5

MD5 [2] is a fast, non-invertible cryptographic hashing algorithm with a 128-bit digest output, designed by Ronald Rivest in 1991.

Initially, a "1" bit is appended to the message, followed by as many "0" bits as required so that its length is congruent to 448, modulo 512. After this first step, the following 64 bits of the last 512-bit block are set to the length of the message in bits, in little endian form. MD5 operates over an internal state of 128-bits, or 16 bytes, divided into a group of four 32 bit words, A, B, C and D. These words are initialized to a constant value at the beginning of the hashing process.

Consider the following auxiliary functions that generate a word out of three 32-bit words:

$$f = \begin{cases} \mathrm{F}(\mathrm{B},\mathrm{C},\mathrm{D}) & = & (\mathrm{B} \wedge \mathrm{C}) \vee \neg(\mathrm{B} \wedge \mathrm{D}) \\ \mathrm{G}(\mathrm{B},\mathrm{C},\mathrm{D}) & = & (\mathrm{B} \wedge \mathrm{D}) \vee (\mathrm{C} \wedge \neg\mathrm{D}) \\ \mathrm{H}(\mathrm{B},\mathrm{C},\mathrm{D}) & = & \mathrm{B} \oplus \mathrm{C} \oplus \mathrm{D} \\ \mathrm{I}(\mathrm{B},\mathrm{C},\mathrm{D}) & = & \mathrm{C} \oplus (\mathrm{B} \vee \neg\mathrm{D}) \end{cases}, where \begin{array}{ccc} \vee & = & or \\ \wedge & = & and \\ \neg & = & not \\ \oplus & = & xor \end{array}$$

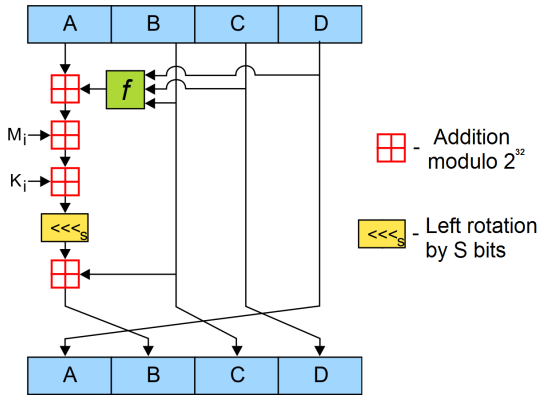

Figure 2.4: MD5 operation, from [16]

For every 16-word block, or 512 bits of input, the state is updated by adding the current state with the result of performing a succession of 16 of the operations depicted in Figure 2.4 for each different auxiliary function. M_i, K_i and S are respectively the 32-bit input, a 32-bit constant and a left rotation of variable amount. The red squares denote an addition modulo 2^{32}.

2.3.1 MD5 collision issues

Digest functions with collision issues, such as MD5 [17, 18, 19, 20, 21, 22], allow people to find pairs of different pre-images that, once hashed, generate the same digest. However, this has no advantage that could ruin this contribution.

Our privacy-enhancing RFID identification system could be misused if an attacker could provide, for a given readerRandom value, a tagRandom and an ID suitable to be matched by a given key. However, without knowing any keys other than itself, a tag has to resort to a random key to get a match with any of the existing keys. The success probability would be $N/2^{-128}$, where N is the total number of assigned keys.

Using collisions, an attacker could try to reuse an ID eavesdropped from another tag to get a match with the same key of that tag. But in this case, on each dialog with a reader he must choose a proper tagRandom that, together with readerRandom (that he does not control) and the key (that he does not know), generates the same ID. We cannot firmly state that this is not at all possible, but without knowing the key (50% of the MD5 input) we find it very unlike to succeed.

In any case, the goal of this work was not to provide a security mechanism to prevent the impersonation of RFIDs (which, by the way, is much easier when they are constant). Our goal was to provide privacy to RFID owners, and this is not endangered by the collision issues of MD5.

Last but not least, we could have used SHA-1 instead of MD5, but we did not for two reasons: first, the 160-bit output of SHA-1 is excessive for our problem, and would represent extra computations on the (slow) RFID tags; second, preliminary performance evaluations of SHA-1 and MD5 in our NVIDIA devices showed that SHA-1 is 2.35 times slower than MD5, because it has nearly twice the number of instructions. Current superscalar CPUs, with their SIMD instructions, are able to blur this performance difference, but GPGUs cannot do so.

2.4 AES: Advanced Encryption Standard

The AES[3] cryptographic cipher, also known as Rijndael, was the winner of a 5 year selection process by the American Government, announced on November 26, 2001. Developed by Belgian cryptographers Joan Daemen and Vincent Rijmen, the original publication of the algorithm specified a base block size of 128-bits and key sizes of 128, but both can be augmented in steps of 32 bits, with a limit of 256 bits for the block size. The AES standard, however, only specifies a block size of 128-bits and key sizes of 128, 192 and 256 bits.

The algorithm works over an internal state matrix of 4×4 bytes, wherein a different number of operations is performed, according to the key size in use. For the purposes of our work, only the encryption process shall be considered, with a key size of 128 bits.

To perform an encryption, a key expansion is required, deriving 176 bytes from the initial 128-bit key. The main operations performed include a word rotation, i.e., a cyclic permutation, and a substitution of bytes over an invertible table, called the S-box. This S-box is obtained via the use of finite field arithmetic, giving the cipher its non-linear properties.

After this expansion, the following computations are performed:

1. AddRoundKey - Where every byte of the state matrix is combined with a round key by means of a bitwise XOR ($\oplus$), illustrated in Figure 2.5.

2. SubBytes - A substitution of every byte, using the aforementioned S-box, illustrated in Figure 2.5.

3. ShiftRows - A transposition where every row of the state matrix is shifted in a different way, illustrated in Figure 2.6

4. MixColumns - A transformation is performed over every column of the state matrix, illustrated in Figure 2.6.

For this last computation, every four bytes of a column are computed as:

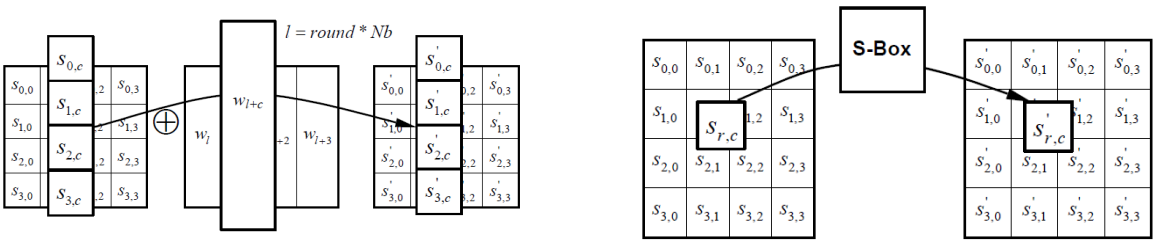

Figure 2.5: AddRoundKey (left) and SubBytes operation (right), in [3]

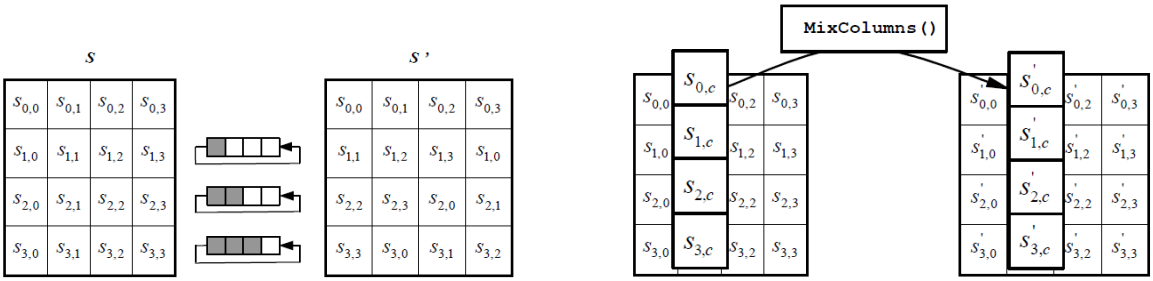

Figure 2.6: ShiftRows (left) and MixColumns operation (right), in [3]

$$
\begin{aligned}
S'_{o,c} &= (\{02\} \bullet S_{o,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\
S'_{1,c} &= S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c} \\
S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c}) \\
S'_{3,c} &= (\{03\} \bullet S_{o,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c})
\end{aligned}
$$

Where $S'_{i,c}$ denotes the state matrix byte in row i and column c after the computation and $S_{i,c}$ the state matrix byte in row i and column c before the computation. $(\{02\}\bullet S_{o,c})$ considers the inputs polynomials over GF(2^8) and multiplies them modulo x^4+1 with $3x^3+x^2+x+2$, in finite field arithmetic. These operations can be performed by resorting to lookup-tables, if the memory of the computational platform is sufficient, in order to speedup the process.

Initially, the input, that is the data that we intend to cypher is loaded to the state matrix, and a first AddRoundKey is performed. Afterwards, 10 rounds are performed comprising a sequence of the 4 previously stated computations, where the final round does not include a MixColumns.

At the time of writing this document, the only published attacks on AES are side-channel attacks on some specific implementations [23, 24, 25, 26], and as such this cipher can be considered cryptographically secure.

In the original Rijndael proposal [27], the authors present an optimization for 32 bit architectures. By defining the operations corresponding to the four steps of a round in a matricial form, one can then incorporate all transformations into a single expression. The simplified result of this combination can then be transformed into a simple XOR of table lookups and of a round key (four table lookups and four XORs per round). This does however require that the target architecture accommodate in memory a set of four tables of 256 words, for a total of 4 kilobytes. This implementation will be referred to hereafter as "T-box".

2.5 AES-NI

AES-NI [28], or AES new instructions is an extension to the popular x86 instruction set, first implemented in some products of the 2010 Intel Core processor family. The purpose of its creation is the acceleration of software using the AES standard, by means of implementing some of the performance intensive computations. The extension comprises the following instructions:

1. AESENC - A single round of encryption (ShiftRows, SubBytes, MixColumns and AddRoundKey)
2. AESENCLAST - The last round of encryption (no MixColumns)
3. AESDEC - A single round of decryption (irrelevant for this application)
4. AESDECLAST - The last round of decryption (irrelevant for this application)
5. AESKEYGENASSIST - Used in the generation of the round keys
6. AESIMC - Used to convert the encryption round keys in a form suitable for decryption (irrelevant for this application)

According to Intel, "AES-NI can be used to accelerate the performance of an implementation of AES by 3 to 10× over a completely software implementation" [29].

Chapter 3

Related Work

S. Weis *et al.* [30] were the first to propose a mechanism for randomizing a tag identifier to avoid its traceability. They proposed a computation of the identifier from a secret value and a random nonce generated by the tag. Both the nonce and the derived identifier are conveyed to the reader, which must then search among all known tag keys to find a match. In our work we introduced another random value, generated by the reader (C_{reader}), and we evaluated the search cost in a massively parallel computing device.

M. Ohkubo *et al.* [31], G. Avoine and P. Oechslin [32], T. Dimitriou [4] and D. Henrici and P. Muller [33] developed alternative approaches where the key of a tag changes each time it is used to produce an identifier; the new key is an hash of the former [31, 32], just a successor of the former, computed in some suitable and secure way [4] or triggered hash chains [33], (further enhanced in [34]). However, these approaches raise critical synchronization issues between the tag and the identification application and often require the authentication of the reader to avoid unwanted key updates in tags.

D. Molnar and D. Wagner [35], D. Molnar *et al.* [36] and T. Dimitriou [37] proposed the use of a tree search structure where each branch has a particular key. A small set of branch keys is uploaded to each tag and they use their branch keys to transform two concatenated random nonces, one generated by the tag and the other by the reader. The identification application uses only the keys on each tree level and on a particular tree branch to identify a tag (with a unique sequence of branch keys). This proposal may in theory speed up the identification of tags but increases the computation within tags (requires many key encryptions) and increases the length of tag replies. Furthermore, it complicates the exploitation of tags with random keys, as they must be carefully initiated, one by one, with a unique set of branching keys. Our approach goes on the opposite direction, as we rely on tags with randomly created keys and on heavy computational power of the identification application in order to keep a reduced computing capability on tags and small message contents.

The interested reader can find many other proposals for protecting the privacy of tags (see, for instance, the review by Lehtonen *et al.* [38]). But we found no evidence of works exploring massively parallel computation for finding the key of a tag among a large set of known keys. Furthermore, many protocols assume that the tag "travels" along many ownership domains (i.e., identifies an object that may have many owners during its lifetime), something that raises security issues related with forward untraceability [39]. However, we do have that problem, since we assume that car identifying tags and the related key owners are constant over time.

Chapter 4

Unidirectional function

In this chapter we will analyze and optimize the performance of the implementations of MD5 and AES in the CUDA programming language.

4.1 MD5

4.1.1 Usage and optimizations

For the implementation of MD5, some modifications to its structure were performed in order to lower the processing time. Since the length of the input bits is lower than the block size (512 bits), only one iteration per key is required, meaning the main loop is unnecessary (the one responsible for going over all the 512 bit blocks of the input message). Additionally, the padding and the appending of the bit size of the message can be hardcoded, since they do not change. Also as a result of only processing one iteration, in the final operation, consisting of the addition of the hash for the current block with the one from the last, since there is no former block, a fixed set of bits is added (the initial state of MD5). This can also be hardcoded.

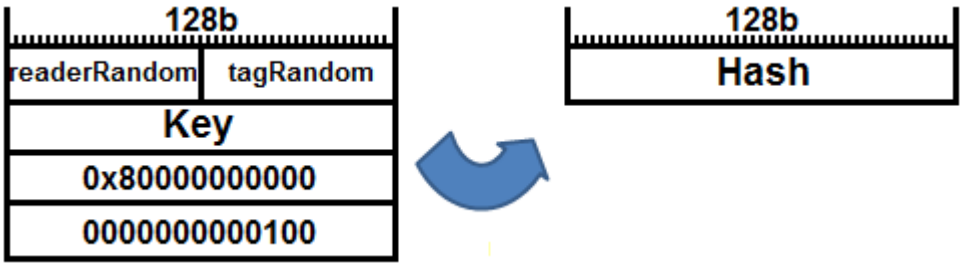

Figure 4.1: MD5 hashing per key

Figure 4.1 shows how the random 64-bit challenges from both the tag, tagRandom, and the reader, readerRandom, plus a 128-bit key, are combined to create the input message to be hashed according to the MD5 algorithm. Also visible is the padding, a 1 after the last bit of the key, followed by zeros until the last 64 bits of the block, where a representation of the length of the input message in bits is appended. Code Snippet 4.1 presents the implementation of MD5 for processing the key at index i of the input variable and storing the result at index i of the output variable.

Code Snippet 4.1: MD5 Code Snippet

```
// Device code
__global__ static void md5_search(uint4 * input
      , uint4 * output)
{
   uint4 buffer[4];

   // Initial MD5 state
   uint4 hash;
   hash.x=0x67452301;
   hash.y=0xefcdab89;
   hash.z=0x98badcfe;
   hash.w=0x10325476;

   // Thread's index for memory access
   int i= blockIdx.y * gridDim.x * blockDim.x + blockDim.x * blockIdx.x +
       threadIdx.x;

   if(!found && i<N)
   {
      buffer[0]=rands; // rands in constant memory
      buffer[1]=input[i]; // keys in global memory

      // pad MD5state and append bit length (256), in little endian
      buffer[2].x=0x00000080;
      buffer[2].y=0x00000000;
      buffer[2].z=0x00000000;
      buffer[2].w=0x00000000;
      buffer[3].x=0x00000000;
      buffer[3].y=0x00000000;
      buffer[3].z=256;
      buffer[3].w=0x00000000;

      // perform a single block-size hashing with MD5
      MD5(buffer,hash);

      // Add the result of the single block hash to the initial state
      hash.x+=0x67452301;
      hash.y+=0xefcdab89;
      hash.z+=0x98badcfe;
      hash.w+=0x10325476;

      out[i]=hash;
   }
}
```

GPGPU		
Tesla	Fermi(H)	
0,163	0,0804	
CPU		
1T	4T(H)	6T(H)
11,2	2,8	1,86

Table 4.1: CPU vs GPGPU performance for MD5, times in seconds, 100 million keys

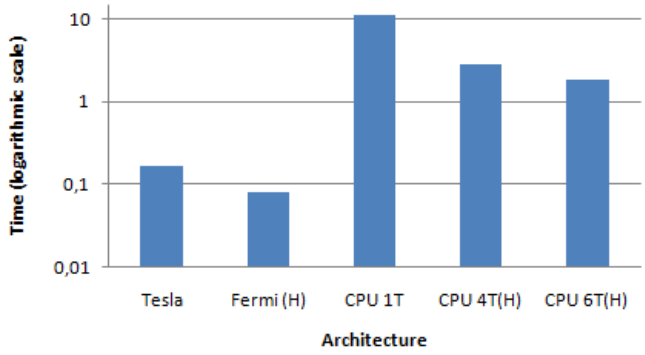

Figure 4.2: CPU vs GPGPU performance, using a base 10 logarithmic scale

4.1.2 Performance evaluation

To evaluate the performance obtained with the previous GPGPU implementation of MD5, a kernel was developed, consisting of an exhaustive computation of MD5 for a large set of input keys, and producing an output of the same length, consisting of the pseudo-random identifiers. The random values are both constant for all the computations. Processing 100 million keys takes about 163 milliseconds.

4.1.3 Comparison with CPUs and other GPUs

Comparing these results with an Intel Core i7 980X CPU clocked at 3.33 Ghz, the GPGPU approach shows its clear superiority for this type of problems as can be seen through Figure 4.2 and Table 4.1, where the two architectures are compared processing 100 million keys. Times were measured using CUDA events for GPGPU code and with GNU/linux's "getrusage" for CPU code (system time + user time).

Initially the CPU code was implemented as a multi-threaded application, but difficulties with timing prompted the use of a single thread version instead, later inferring the maximum performance for the CPU by dividing the result by the number of available cores. Processing 100 million keys takes 11.2 seconds to complete on a single core. Since the CPU in question is a hexacore, the maximum performance, assuming linear scalability, should be in the order of 1.86 seconds. Considering the Tesla GPGPU only takes 163 milliseconds, the improvement is greater than an order of magnitude.

The same code was also tested on a newer GPGPU, namely an Nvidia GeForce GTX480 with 480 cores, also known as Fermi. Since we are dealing with a consumer graphics card, instead of a GPGPU dedicated to running CUDA code, the available memory is lower, but using a smaller set of keys resulted in a halving of the performance set by the devices in the Tesla system (for the reduced set also). This was expected as the core count doubles. No other gains can be extracted from the move to a greater compute capability in this particular application.

4.2 AES

4.2.1 Usage and optimizations

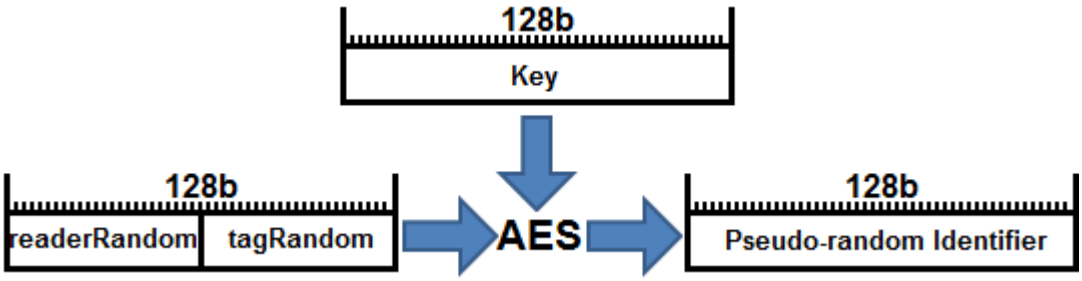

Figure 4.3: Ciphering the random challenges with AES

Since with AES we are dealing with a block cipher, the process of generating the cyphertext is different from that of the generation of the MD5 hash. As depicted in Figure 4.3, the random challenges are encrypted using a key, and the cyphertext is used as the pseudo-random identifier. Note that no decryption is ever performed, due to the fact that the key is unknown. Instead we perform encryptions with all the deployed keys until the cyphertext matches the pseudo-random identifier generated in the RFID tag.

The first step after the successful implementation of AES in C (before porting to CUDA), validated by the test vectors provided in [3] was, in a similar way to that performed for MD5, remove the main loop. As the block size of this cipher is the exact length of the input bits (the two random nonces), only one iteration is required. Since this block size is fixed at 128 bits (remember from section 2.4 that the AES standard comprises 3 possible key sizes, 128, 192 and 256 bits) a small number of irrelevant control instructions can be removed, as well as manual loop unrolling for the 10 rounds of the four inner operations, which did not result in any measurable performance difference, or in a decrease of the number of used registers, suggesting that the compiler already handles that optimization automatically.

Some of the tests performed from this point on will be in a 9 SM device, specifically a GeForce GT335M GPGPU, running at 1.08 Ghz, instead of a 30 SM device from the Tesla S1070 computing system. This move was performed because the 9 SM device is slower in a very consistent way, which makes it easier to find differences between multiple kernels.

As is well established in the literature studying different implementations of AES [27], this cipher's performance can be significantly increased by resorting to tables to perform the transformations over finite fields. These transformations correspond to the byte substitution, from the SubBytes step, and the multiplication of a byte by 2 or 3, modulo $x^4 + 1$, both of them from the MixColumns step. These tables will be referred from now on as, respectively, "Sbox", "Two" and "Three".

Despite the memory bandwidth being at times a major bottleneck for CUDA code, the initial version of the port to CUDA was implemented with the tables, but keeping in mind that it's use could be slower than performing the actual computation, which would be tested at a later date.

Upon compilation of the code, it became immediately clear that several modifications would have to be performed as 176 bytes of local memory per thread were being used. This local memory, despite its name, resides in global memory (it's local only in terms of scope),

global	1 table const	1 table shared from global	3 tables const
505	364.1	384.2	119.74

Table 4.2: Memory type comparison, loading the Sbox table from global memory, constant or shared for a single table and finally loading all tables from constant memory (times in milliseconds)(9 SM device, 1 million keys)

and is used by the compiler to store data that does not fit into the MP's registers. The reference pseudocode that describes the AES algorithm specifies that the key expansion is done entirely before its actual use, hence the need for the 176 bytes of memory to store the key schedule. Careful analysis of the algorithm shows that nothing prevents the key expansion process to be performed in steps, immediately before a given key is needed. The only requirement for this solution is the need to store the last 16 bytes computed, as the new keys are derived from the previous. Initially, both tables were defined as simple byte arrays, in the traditional C manner:

Code Snippet 4.2: lookup tables for CPU code

```
unsigned char Sbox[256]=
{0x63, 0x7c, (...)
};
unsigned char two[256]={...};
unsigned char three[256]={...};
```

For this type of code to compile, it needs to be preceded by the __device__ modifier, indicating that it should be loaded into the device memory prior to the kernel launch. As such, a very large amount of memory reads will be performed, which is far from ideal due to the ever present latency hit.

Several tests were performed to compare the performance when different types of memory are used to store these tables (global, constant or shared). Table 4.2 shows the performance difference between loading the tables from constant memory or shared memory, copied from global memory by the first thread of a warp, only for the Sbox table. The remaining entry corresponds to loading all the tables from constant memory. All of the tests used 1 million keys.

Another way to lower the processing time could be to store the key schedules themselves in global memory, instead of the actual keys. This, however, implies that every key would now require 176 bytes of device memory, instead of just 16 bytes (128 bits). As a result, this would lower the maximum amount of keys that could be searched for using a single device to 9%, or 24 million keys, on the devices of the Tesla S1070 system.

Table 4.3 compares the performance of the AES versions with and without pre-computation of key schedules, over 1 million keys, in a single Tesla device. In spite of the fact that a large number of operations are not performed, the no pre-computation version is only marginally worse, due to fact that a large number of slow memory operations needs to be performed.

In order to check if the computation could be further sped up by not resorting to the tables, but instead perform the necessary computations in the devices ALU's, the table that involves the least amount of computational work, ("two") was replaced by the equivalent operations. As a multiplication in finite field arithmetic is performed modulo a given polynomial, in this

no pre-computation	pre-computation
26.99	25.36

Table 4.3: Pre-computation analysis (times in milliseconds)(30 SM Tesla device, 1 million keys)

table	no-table
112.07	243.24

Table 4.4: Table vs no-table analysis (times in milliseconds)(9 SM device, 1 million keys)

case $x^8+x^4+x^3+x+1$ (0x11B in binary form) this operation is equivalent to a single bitwise shift to the left (equivalent to multiplying by 2), followed by an XOR with 0x1B, if the result of the previous operation is greater than 0XFF (equivalent to the modulo operation). Despite this relative simplicity, the resulting code performed slower than when the "two" table is employed, even when loaded from global memory, proving that the use of lookup tables is faster in CUDA.

Table 4.4 shows the difference between the version where the multiplication by two is performed via a lookup and one where it is computed. Clearly there is a tremendous performance gain enabled by the use of tables.

Returning to a memory analysis, back on a 9 SM GPGPU, Table 4.5, compares the processing time when shared memory is used, but now loaded from constant memory, or another version, also with shared memory but loaded directly from the code, by the first thread of a warp, again only for the Sbox, with a final entry for the version with all the tables loaded in the latter variant.

The final and best approach ended up being the use of shared memory, loaded from the first thread of each warp (note that shared memory is shared between threads of the same warp) as integers, using pointer arithmetic, directly from code, as demonstrated in Code Snippet 4.3: The identifier __align__ is used to make sure that the table is aligned to the nearest 4 byte segment, to enable the use of integers for the loading instead of bytes. __shared__ indicates to the compiler that the following data structure should be treated as shared memory.

The branch guarantees that only the first thread of each warp accesses the memory, and ensures the remaining ones wait before continuing execution.

4.2.2 T-box implementation

The next step was the implementation of the "T-box" version of AES, defined in section 2.4. Since CUDA devices have a shared memory space of at least 16 kilobytes per Multiprocessor, the 4 required 256 entry 4 byte tables would fit comfortably, enabling the evolution of

sharedFromConstant	sharedFromCode	codeAllTables
92.9	92.86	48.17

Table 4.5: Pre-computation analysis (times in milliseconds)(30 SM Tesla device)

Code Snippet 4.3: lookup tables for GPGPU code

```
__align__(4) __shared__ static unsigned char Sbox[256];
__align__(4) __shared__ static unsigned char two[256];
__align__(4) __shared__ static unsigned char three[256];

(...)

// in the Kernel:
if ( !(i%32) ) // i: thread index in the grid
{
   ( (unsigned int *) Sbox)[ 0]=0x7B777C63;
   (...)
   ( (unsigned int *) Sbox)[63]=0x16BB54B0;

   ( (unsigned int *) two)[ 0]=0x06040200;
   (...)
   ( (unsigned int *) two)[63]=0xE5E7E1E3;

   ( (unsigned int *) three)[ 0]=0x05060300;
   (...)
   ( (unsigned int *) three)[63]=0x1A191C1F;
}

(...)
```

Tbox	Previous
1.72	10.82

Table 4.6: T-box analysis (times in milliseconds)(30 SM Tesla device)

the previous first-thread-loads-the-memory-for-the-warp mechanism for bigger tables.

Table 4.6 compares the T-box implementation with the previous fastest kernel, with 1 million keys. The Tbox version is almost an order of magnitude faster.

4.2.3 Comparison with MD5

Comparing the fastest implementation of AES with the previous version of MD5, we can conclude that there is no significant difference in the usage of either a digest function or a cryptografic cypher, as depicted in Table 4.7. As such, both alternatives are feasible for usage in real world scenarios.

MD5	AES
163	172

Table 4.7: MD5 and AES comparison, processing 100 million keys (times in milliseconds)(30 SM Tesla device)

4.2.4 Comparison with similar computations in CPUs and other GPGPUs

Comparing the performance of the exact same code, minus the intricacies of the memory management for the lookup tables, with the same Intel Core i7 980X CPU (and associated timing issues) used for the MD5 tests, as well as a more recent Fermi GPU, the GPGPU is still clearly the winner, even when the AES-NI instruction set is used for the CPU code.

Table 4.8 shows the maximum time required to perform 10 million computations over the input keys, with both the 3 table and 4 table (T-box) versions of AES, for the CPU (with 1 thread or an hypothetical 6-thread interpolated result), and both the Tesla and Fermi GPGPUs. Also included is the performance of an AES-NI implementation on the CPU. A speedup analysis is also presented in Table 4.9, comparing the speedup obtained from the move to a given GPU, for the best case scenario for all the architectures (AES-NI in the CPU and T-box for the GPGPUs).

		3T	4T	NI
CPU	1 Thread	5.5	1.31	0.4
	6 Threads	0.91	0.22	0.06
GPGPU	Tesla	0.108	0.0172	
	Fermi	0.037	0.009	

Table 4.8: CPU vs GPGPU comparison (times in seconds), searching through 10 million keys

CPU 6T NI	Tesla 4T	Fermi 4T
1×	3.5×	6.7×

Table 4.9: Best-case speedup vesus the CPU(times in seconds). For the CPU a 6-threaded version of AES-NI is the reference, whereas the GPGPUs use the 4 table version.

Chapter 5

Key searching Solution

Since the performance results for AES and MD5 were so similar, this chapter will focus on a searching solution for the latter. However, the conclusions drawn from this chapter would also apply to the former.

5.1 Overview of GPGPU search activities

The key search activities are conducted by a GPGPU kernel that has access to all assigned keys. Briefly, searches are as follows:

1. The set of keys is copied from the host to an array in device memory.

2. Both the challenges and the pseudo-random ID computed by the RFID tag are stored into device constant memory, since they are read by all threads, and never overwritten.

3. The search kernel is launched on the device, looking for the key that originated the pseudo-random identifier.

4. The result, i.e., the key of the tag that matches the computed ID with the previous parameters, along with a boolean flag, are written in device global memory in case a match is found.

5. The host program collects the result after the termination of the search kernel.

For consecutive key searches, steps 2 to 5 are performed, always using the keys installed in the device in step 1. The set of keys only needs to be updated when new keys are assigned, or key reordering actions are performed on the host.

Figure 5.1 illustrates the previous memory setup (see Chapter 6). Note that on this particular example, in the end, the first thread found a match, and as such sets the "found" boolean and the "result" variable as required.

The Code Snippet 5.1 shows how the memory is loaded when the program is initiated, as it can and will be reused for multiple kernel launches. When a key is received from an external source (possibly a tollbooth controller) the massively parallel search is issued, and the results are later processed as desired.

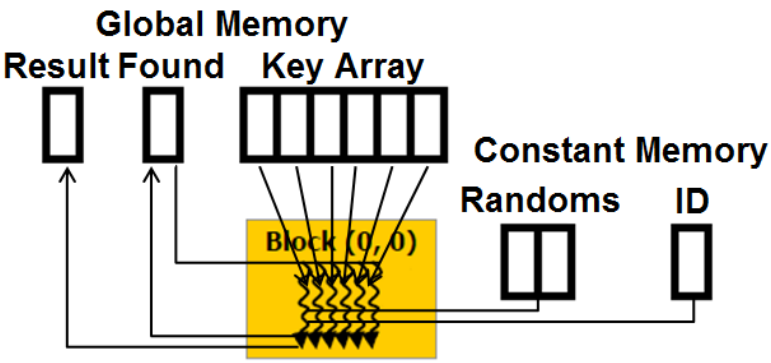

Figure 5.1: Memory setup: global memory and constant memory. The first thread finds a match. Step 1: all the threads read "Found". Step 2: all the threads compute MD5 from the challenges and with their respective base ID. Step 3: all the threads compare the result of MD5 with the pseudo-random ID. Step 4: The first thread found a match, and as such, sets "found" to true and "Result" accordingly

Key load strategy	GPU time (μs)	Global memory read throughput (Gib/s)	Global load efficiency
4 × load `uint`	79794.7	10.8278	1
2 × load `uint2`	78747.0	10.9118	1
1 × load `uint4`	78620.2	10.9895	0.66656

Table 5.1: Profiling values collected with `cudaprofiler`, normalized (per block counts). These values were obtained with three search kernels with the same task: perform a key search among 1 million keys (without actually finding it). The kernels used 3 different strategies for loading keys from global memory: 4 `uint` loads, 2 `uint2` loads or a single `uint4` load.

5.2 Optimization of kernel memory accesses

The large amount of keys that need to be checked presented us with a challenge: how to efficiently access this data in a way that would minimize data transfer from the global device memory. In light of the constraints presented by the CUDA programming model, one needs to ensure the correct alignment of the data structure holding the set of keys to 128 byte segments. Considering the use of `uint4`, this is already handled transparently by the compiler. Furthermore, each thread is required to access either 4, 8 or 16 byte words and for every memory request for a half-warp, the first 8 words need to reside in a 128 byte segment and the last 8 words in the following 128 byte segment. Lastly, the accesses must be done in sequence by each thread.

Given these requirements, and considering the characteristics of the problem at hand, we decided to store the keys sequentially in a memory array of `uint4`, meaning 128 bits per key, times the total number of keys.

Preliminary profiling tests with accesses to keys using four `uint`, two `uint2` and one `uint4` revealed that we achieve the highest overall performance with `uint4`, even with a lower global load efficiency (see Table 5.1).

Testing the boolean flag in all threads, despite requiring a large amount of global memory

Code Snippet 5.1: host program that launches search kernels

```
// Host code (without error checking)
int main()
{
   (...)

   // Load number of threads and hop size to constant memory
   cudaMemcpyToSymbol("N", N, sizeof(unsigned int));
   cudaMemcpyToSymbol("blocks_per_thread", &tc, sizeof(unsigned int));

   // Allocate memory for the device array and load the data (from keys_h)
   cudaMalloc((void **) &keys_d, sizeof(uint4)*N);
   cudaMemcpy(keys_d, keys_h, sizeof(uint4)*N, cudaMemcpyDeviceToHost);

   // Allocate memory for the result and found variables
   cudaMalloc((void **) &result_d, sizeof(uint4));
   cudaMalloc((void **) &found_d, sizeof(bool));

   while (1) {
      // Loop waiting for search requests
      (...)
      // load found to false
      cudaMemcpy(found_d, &notfound, sizeof(bool), cudaMemcpyDeviceToHost);

      // Launch the kernel with BLOCK_N blocks and THREAD_N threads
      search_kernel<<<BLOCK_N, THREAD_N>>>(keys_d, result_d, found_d);

      // Retrieve found and result
      cudaMemcpy(&found, found_d, sizeof(bool), cudaMemcpyDeviceToHost);
      if (found)
         cudaMemcpy(&result, result_d, sizeof(uint4), cudaMemcpyDeviceToHost);

      // Do something with the key (for billing purposes?)
      (...)
   }
   // No more requests to process
   cudaFree(keys_d);
   cudaFree(found_d);
   cudaFree(result_d);
   (...)
}
```

reads, doesn't result in a significant performance loss, and allows the kernel to be stopped immediately after a match is found, thus preventing unnecessary computations over a possibly large number of remaining keys.

Due to the absence of thread cooperation, there is no need to resort to the shared memory explicitly. The only item that needs to be visible to all threads is the found variable, but since it is visible by all threads in the device, as opposed to just the ones in the same thread block, shared memory is not adequate; one has to use (slow) global memory.

5.3 Simplest approach: one thread per key

In the first kernel developed, every thread is responsible for the computation of only one pseudo-random ID, accessing the array of keys in the index for which it is responsible. The challenges and responses are also loaded and the hash of the resulting block is compared against the ID computed by the tag. If there is a match, the key, and a boolean value set to TRUE, will be written in global memory.

This key index is calculated as:

```
i = blockIdx.y * gridDim.x * blockDim.x +
    blockIdx.x * blockDim.x +
    threadIdx.x
```

Due to the 65535 limit for the `x` dimension of a grid (`gridDim.x`), only 8 million keys can be deployed (with a `blockDim.x` of 128) with only `x` coordinates, hence the use of the two-dimensional component of `blockIdx`.

Figure 5.2 illustrates how the memory accesses are performed. Also depicted is how threads are grouped to the same, respectively, `ThreadIdx.x`, `blockIdx.x` and `blockIdx.y`. For this last variable, note that no special meaning is associated with having a different `y` dimension. This is only a way to overcome the 8 million keys limit.

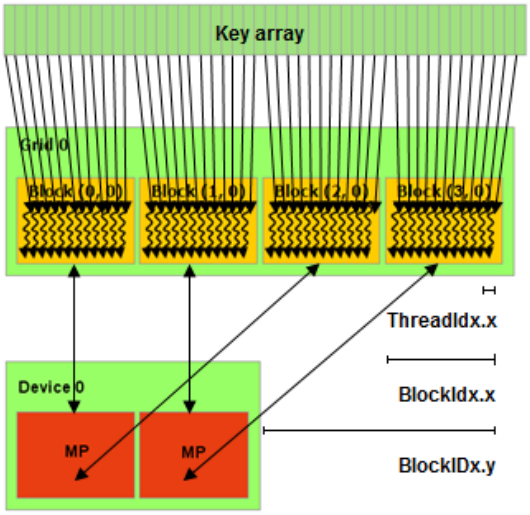

Figure 5.2: One key per thread and its mapping to a device

Code Snippet 5.2: One thread per key

```
// Device code
__global__ static void md5_search(uint4 * global
      , uint4 * result, bool * found)
{
   // Thread's index for memory access
   int i= blockIdx.y * gridDim.x * blockDim.x + blockDim.x * blockIdx.x +
       threadIdx.x;

   if(!found && i<N)
   {
      // Compute MD5

      if (hash==pseudoRndID)
      {
          *found=true;
          *result=keys[i];
      }
   }
}
```

Code Snippet 5.2 shows a simplified version of the proposed search kernel. Note that "found', "keys"' and "result" reside in global memory, "N" and "pseudoRndID" are loaded into constant memory and the remaining data structures are mapped to registers.

As was already mentioned in section 2.2.1, since the number of threads per block is fixed, the size of a grid can not be made arbitrary. As such, this approach is not actually "one thread per key", but the minimum amount of threads that can process the entire set of keys. Hence the need to test if "i<N", stopping the last threads for which the corresponding index falls outside the array of deployed keys.

In order to further increase performance, we implemented a feature that copies an initial portion of the set of keys to its end, ensuring that the number of keys to process is always a multiple of the number of threads available in the device. This means there is no need to perform the "i<N" test.

Performance tests with sets of 50, 100 and 250 million base keys (cf. Figure 5.4) revealed that when there is a match with the key at index 0, the execution time in the device is far from zero. Furthermore, this deviation from zero increases linearly with the total number of keys. This is a clear evidence that the termination of the threads where no key search is effectively performed, only the `found` boolean variable is tested, has a significative overhead, proportional to the total number of keys. Consequently, the approach of having one thread per key, although simple, may not be the most adequate one for achieving the lowest average kernel execution time. In the next section we describe a minor modification that tackles this performance issue.

5.4 Faster approach: several keys for each thread

To solve the aforementioned overhead problem, we devised another search kernel, where the number of threads is always constant and independent of the total number of keys to

process, and is proportional to the number of streaming processors in the device. Every thread processes a subset of the array of keys, using a stride equal to the number of threads; thus, the number of keys searched per thread is a simple division of the total number of keys by the number of threads.

For this new algorithm the key index is calculated as:

```
i = blockIdx.x * blockDim.x + threadIdx.x
```

and for every iteration, we update it as follows:

```
i += blockDim.x * gridDim.x
```

where `blockDim.x`, later denoted by `THC`, is the thread count, and where `gridDim.x`, later denoted by `SP`, is the number of available streaming multiprocessors (CUDA cores) for the device.

Figure 5.3, illustrates how this new kernel compares with the previous, with respect to both the memory accesses and the mapping of the grid in a GPGPU. The previous kernel creates a large grid where only a few blocks are being processed at any one point whereas in this evolution the size of the grid is calculated based on the number of cores available in the device.

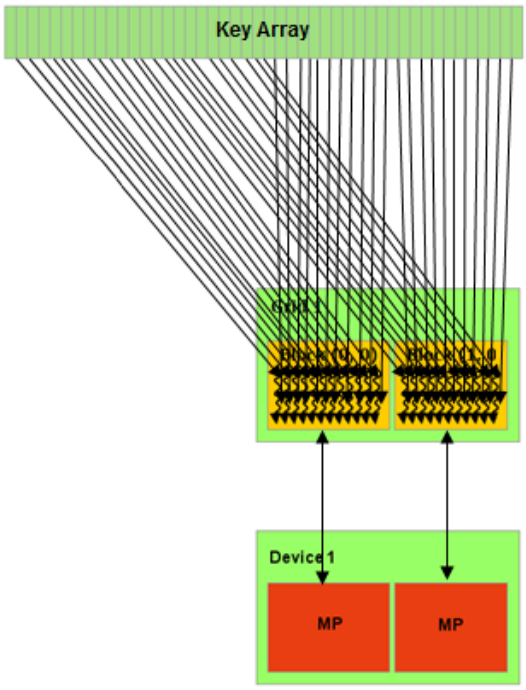

Figure 5.3: Multiple keys per thread and its mapping to a device

Note that once a thread finds a match, setting the "found" boolean to true, in the next iteration all the threads in the device will terminate almost immediately, since there are no more blocks in the grid to be processed.

We also tested a version of this last kernel where each thread processes a sequential array of keys, i.e., the first thread processes the key at index 0, 1, 2... up to Nkeys/THC and so on. Due to the CUDA memory architecture this should be much worse, because in this case

the memory accesses cannot be coalesced, but the measurements revealed that that wasn't the case, at 208 vs. 163 milliseconds for 100 million keys. This can be explained by the fact that these search kernels are limited by their computational intensity, as opposed to memory bandwidth.

The Code Snippet 5.3 highlights the differences between the two kernels proposed so far. HOP is loaded into constant memory as $SP \times THC$. An attempt was made to hardcode this last loop invariant, but performance evaluations did not reveal a subsequent improvement (this would also prevent the use of the code in machines with different GPUs).

Code Snippet 5.3: Multiple keys per thread

```
(...)

// Thread's index for memory access
int i=blockDimx * blockIdx.x + threadIdx.x;

while(!found && i<N)
{
   (...)

   if (...)
   {
       (...)
   }
   i+=hop;
}
```

Note that since there is the need to have some sort of way to track the number of iterations performed per thread, to ensure that the computed index lies within the confines of the array of keys, regardless of its size, the previous optimization where the number of keys is always a multiple of the number of available threads is not performed. If one were to remove the "i<N" test, this kernel could become an infinite loop, even if the number of keys is a multiple of the number of threads, in case the pseudo-random identifier does not match any of the keys in the array for the given random nonces (a scenario that could be caused by errors or denial of service attacks).

Figure 5.4 shows a comparison between the two key search kernels. Not only did the initial overhead, corresponding to the termination of all the remaining threads disappear, but also the revised kernel is consistently faster throughout the search space.

5.5 Performance evaluation

5.5.1 Match time performance

Figure 5.4 shows the time it takes to find a key out of 50, 100 or 250 million keys based on its position in the array. We evaluated this time for both the original kernel (where the number of launched threads matches the number of keys) and the improved kernel, where the number of threads is fixed and each thread handles as many keys as required. The optimized kernel with the optimal value of threads per block, 64, is faster for every position, especially for keys near the beginning of the array.

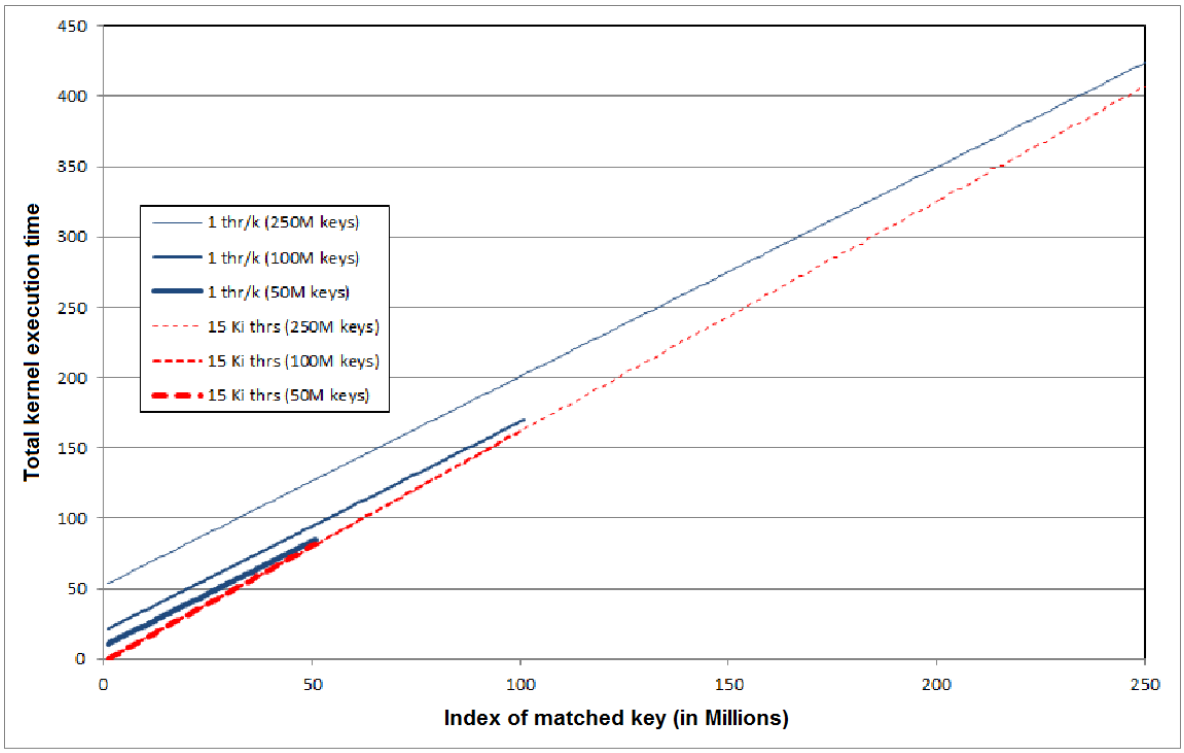

Figure 5.4: Key search time for each key index among sets of 50, 100 and 250 million keys. Solid lines represent values obtained with a kernel that uses one thread per key; dotted lines represent values obtained with a kernel that uses a fixed number of threads (15 Ki).

The results of Figure 5.4 also show that larger key sets increase the performance penalty of kernels using one thread per key at all indexes, while do not affect the kernel with a constant number of threads. The following model explains this fact.

The match time T_i when using N threads for N keys is given by

$$T_i = (i+1) \cdot t_k + (N-i-1) \cdot t_0 + N \cdot t_c$$

while the match time T'_i when using X threads for N keys, with $X < N$, is given by

$$T'_i = (i+1) \cdot t_k + (i+1) \cdot t'_k + X \cdot t_c$$

where $i \in [0, N-1]$ is the matching key index, t_k is the time to perform an MD5 or AES computation, t_0 is the time to perform the test on the (global) boolean variable that signals a match, t_c is the time to create a single thread and t'_k is the extra cost to implement the search cycle when each thread checks more than one key.

The slopes of T_i and T'_i are

$$\frac{\partial T_i}{di} = t_k - t_0$$

$$\frac{\partial T'_i}{di} = t_k + t'_k$$

Clearly, the slope of T_i is always lower than the slope of T'_i, as we can confirm in the graphics of Figure 5.4. But can we get an intersection of the curves with a larger set of keys, i.e., with

a bigger N? Intuitively, from the graphics we see that they will never intersect each other, since each time we increase the number of key, the match time difference for the last key increases. Nevertheless, from our model we can confirm this intuition:

$$T_{N-1} - T'_{N-1} = (N - X) \cdot t_c - N \cdot t'_k$$

Since $N \gg X$, then

$$T_{N-1} - T'_{N-1} \approx N \cdot (t_c - t'_k)$$

Since t_c and t'_k are not influenced by the number of keys N, then if $t_c > t'_k$ for one value of N, that will also happen for all values of N. This is exactly what we observe in Figure 5.4.

5.5.2 Kernel profiling

We evaluated the efficiency of our search kernel with a fixed number of threads using the CUDA profiler. For this evaluation we used 48 million keys, a number that is a multiple of 30 (the number of stream multiprocessors) and is also a multiple of 512 (the number of threads in each stream multiprocessor). Table 5.2 resumes the collected counters when searching for a key that does not exist among 48 million keys using 15360 threads.

Memory read throughout (GB/s)	Global load requests	Global load transactions		Instruction throughput
		128 bytes	32 bytes	
10.9895	4166	25000	12500	1.00344

Table 5.2: Profiling values collected with `cudaprofiler`, normalized (per block counts). These values were obtained with a search kernel that performs a key search among 48 million keys (without actually finding it).

Our search kernel is not memory intensive, it is compute intensive; nevertheless we achieve a memory read throughput that is about 10% of the absolute maximum of 102 GiB/s [40].

The number of 128-byte global load transactions is the minimum possible: the number of blocks was 240, therefore each block had to process 200000 keys, or 3200000 bytes, which implies 25000 128-byte load transactions. On the other hand, the 32-byte global load transactions, due to the testing of the success flag (the same for all threads), should be equal to the number of keys processed by each half-warp, which is exactly 25000. Concluding, the kernel does the minimum possible number of global memory loads to perform the key search.

Finally, the instructions throughput is close to the optimum value of 1, being slightly higher because of the presence of some dual-issue assembly instructions (namely, multiplications for index calculation).

Chapter 6

Searching and ordering

In order to further improve the performance, based on the kernel described in section 5.4, a modification was devised to bring the most frequently matched keys closer to the beginning of the array, and as a result reduce the average time necessary to process a given workload. This makes sense, as generally there is a subset of tags that are activated often, whereas others are seldom used. In spite of the fact that the following modifications were all performed exclusively on the MD5 kernel and respective supporting host code, they could have been used in the AES version, with similar results.

6.1 Real-time bubble reordering

Every time there is a hit, the matched key is swapped to the previous position in memory that also falls in the search space of the same thread. This way, the more frequently matched keys are pushed to the beginning of the array. This reordering paradigm does not need to keep counters for recording the number of matches per key, allowing the global memory of each device to be filled entirely with keys (nearly 268 million 128-bit keys per device of the Tesla S1070 computing system).

Code Snippet 6.1 illustrates how this algorithm works; "hop" refers to the number of threads per block. Note that we need to ensure that if the matched key is already at the beginning of the array, we must not move it.

6.2 Offline reordering

In spite of the good results obtained with the previous kernel (see Section 6.4), the bubble reordering algorithm could in theory prevent an optimal ordering in the event that some threads end up being responsible for a greater number of highly used keys than others. This stems from the fact that a given key, no matter how many times it is reordered, will always be handled by a certain thread, meaning that if the distribution of highly probable keys is not even among all threads, after the convergence of this process there would be some highly probable keys that would take longer than others (the ones taking longer corresponding to the threads with more highly probable keys).

To tackle this minor issue, an idea was envisioned of an offline reordering of keys at certain intervals by the CPU, based on the number of hits for every key, ensuring that the distribution of keys to the threads is as even and optimal as possible. As we will see, this method provides

Code Snippet 6.1: Real-time bubble reordering algorithm's kernel

```
(...)

// Thread's index for memory access
int i=blockDimx * blockIdx.x + threadIdx.x;

while(!found && i<N)
{
   // Compute MD5

   if (/*Computed MD5 == Pseudo-random identifier*/)
   {
       *found=true;
       *result=global[i];

       if(i>=blocks_per_thread)
       {
           // a -> the new index
           // temp, b -> temporary variables
           a=i-hop;
           tmp=global[a];
           global[a]=global[i];
           global[i]=tmp;
       }
   }
   i+=hop;
}
```

a better reordering and spreading of keys among all threads but has the extra cost of keeping a separate hit counter per key. In the kernels tested, 32 bit counters were used, which means a 20% reduction in the amount of keys that can be processed by each device (nearly 215 million 128-bit keys in each of our 4 Tesla 4 GiB devices). Code Snippet 6.2 highlights the modifications that need to be performed to the real-time bubble reordering kernel in order to keep track of counters for all the keys being deployed.

Code Snippet 6.2: Bubble reordering algorithm's kernel with counters

```
(...)
       if(i>=blocks_per_thread)
       {
           a=i-hop;
           tmp=global[a];
           global[a]=global[i];
           global[i]=tmp;
           b=counter[a];
           counter[a]=counter[i]+1;
           counter[i]=b;
       }
       else counter[i]++;
(...)
```

6.3 Real-time random bubble reordering

The final kernel optimization, almost exactly the same as the solution described in Section 6.1, is another way to solve the potential for uneven distributions, but without the need for potentially costly device-to-host transfers of the set of keys and respective hit counter, along with the associated downtime of the reordering method presented in Section 6.2.

Since the reordering of keys after a match will be to memory positions that will no longer be in use for the current search activity, one can assume that there is no need to ensure that a thread only reorders keys that are within its search space. As such, a random number between 0 and $SP \times THC - 1$ is computed by the CPU every time a new search is issued, sending it to the device via constant memory, and this random number will be used to calculate the index of the new position of the matched key. This enables a matched key to move towards the beginning of the array of keys processed by one randomly chosen thread. This way, threads that initially have many frequent keys have the opportunity to spread them among all the other threads, which may contribute to a more even distribution of the most frequent keys among all threads.

Code Snippet 6.3 highlights the differences between the real-time and real-time random bubble reordering algorithms. The only additional step, aside from the use of the new random number, "randomLeap", is the different calculation performed to compute the new address of the matched key. One has subtract the original index of the thread in the Grid (and not in the array of keys) in order to prevent accessing a position of memory before the last $SP \times THC$ segment.

6.4 Performance Evaluation

In order to compare the time evolution of the ordering kernels, as well as the offline ordering method, we created a simulation that generates a large amount of indexes between 0 and one million (the number of keys used in this analysis) according to a Gaussian distribution, to simulate the event that some keys are activated more often then others. We then evaluated the time needed to search 1000 keys corresponding to these indexes in the original array and repeat the process many times. These random numbers are the same for all the simulations.

Figure 6.1 shows the results of the impact of key ordering strategies. Both the reordering kernel and the random reordering kernel show a significant improvement over the basic search, with the former being slightly faster. When an offline reordering is issued, only once after 4 thousand sets of a thousand keys, the algorithm converges much sooner, but one has to consider the time needed to copy data to/from the device, and also the time necessary to actually sort the keys in the CPU.

Number of keys (millions)	Elapsed time (s)	
	Average	standard deviation
50	1.8831	0.012359881
100	3.2590	0.005754226
210	6.4474	0.058963642

Table 6.1: Elapsed time of offline reordering actions on a 3.07 GHz Core i7 950 Intel CPU.

Table 6.1 shows the cost of the offline reordering actions, as a function of the number of

Code Snippet 6.3: Real-time random bubble reordering algorithm's kernel

```
(...)

// Thread's index in the Grid for memory access
int i=blockDimx * blockIdx.x + threadIdx.x;

while(!found && i<N)
{
   // Compute MD5

   if (/*Computed MD5 == Pseudo-random identifier*/)
   {
       *found=true;
       *result=global[i];

       if(i>=blocks_per_thread)
       {
           // a -> the new index
           // buffer, b -> temporary variables
           a=i-blocks_per_thread-(blockDim.x * blockIdx.x + threadIdx.x)+
               randomLeap;
           buffer[0]=global[a];
           global[a]=global[i];
           global[i]=buffer[0];
           b=counter[a];
           counter[a]=counter[i]+1;
           counter[i]=b;
       }
       else counter[i]++;
   }
   i+=hop;
}
```

keys. For this evaluation we used the Tesla host, with a 3.07 GHz Core i7 950 Intel CPU and 12 GB RAM, and sets of 50, 100 and 210 million keys. For each set of keys we ran 10 experiments and computed their average execution time. The time figures, measured with elapsed time between two CUDA events, include the copy of keys and hit counters from the GPGPU to the CPU memory, the reordering of keys according to their counters with quick sort, and the CPU to GPGPU memory transfer of all keys and counters. The counters were initially set up with a Gaussian distribution of hits from 2×10^9 activations.

The values presented in Table 6.1 show that even for very large sets of keys, the pause time imposed by offline reorderings is not critical. Furthermore, as reorderings do not need to be frequent, and can be combined with regular updates due to newly assigned keys, they do not represent a significant increase in the pause time imposed by updates of key sets.

Figure 6.2 illustrates the results obtained when we perform an offline reordering at every thousand sets of keys with and without a reset of their hit counters. With the distribution we used, the former is considerably slower to complete the simulation. From this we can not only conclude that frequent reordering actions are unnecessary but also that the reset after it worsens the performance.

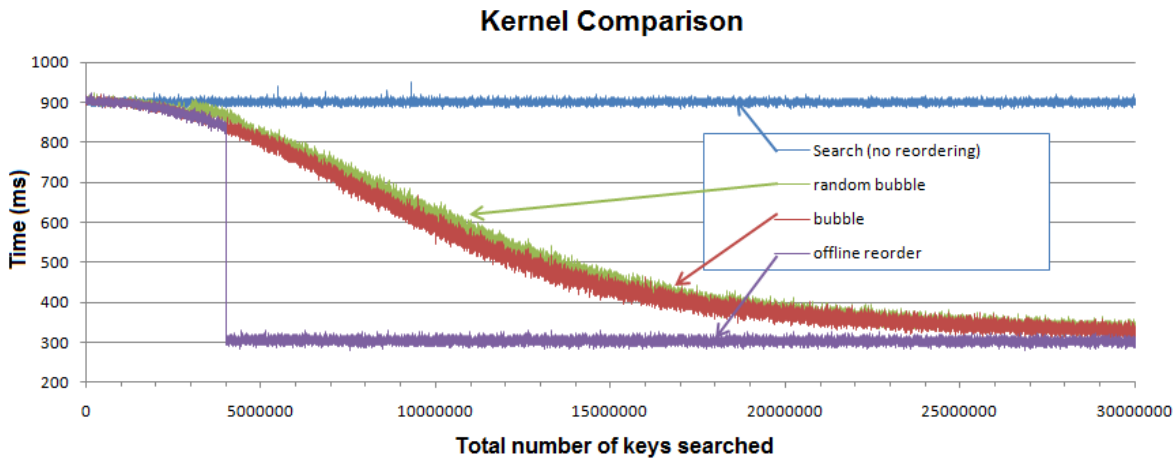

Figure 6.1: Search time for sets of 1000 keys among 1 million keys. The keys are activated according to a Gaussian distribution. Search times were computed for 4 kernels, the original one (without ordering), two using a real-time bubble reordering, either within the keys of the same thread or randomly among threads, and another one using an offline, hit-based ordering. The sharp transition of the offline reorder curve is caused by an offline reordering that occurred after the first 4 million keys.

6.5 Comparison with similar computations in other GPGPUs

We also tested the reordering kernel in a newer Fermi GPGPU. No other modifications to the code were needed, other than the calculation of the number of CUDA cores, from the available stream processors and compute capability of the device (which defines the number of SPs per SM, 8 vs. 32 with compute capabilities respectively lower than or equal to 2.0). This step is needed to infer the ideal number of threads per block. We also recompiled the code to include the option `-code=sm_20`, to account for Fermi's computing capability. Due to problems resulting from Fermi's cache coherency model, it was also necessary to disable the Level 1 cache memory, shared per multiprocessor. This ensures that there are no multiple copies of the global found variable, which would otherwise prevent the kernel from being stopped when a certain thread finds a match.

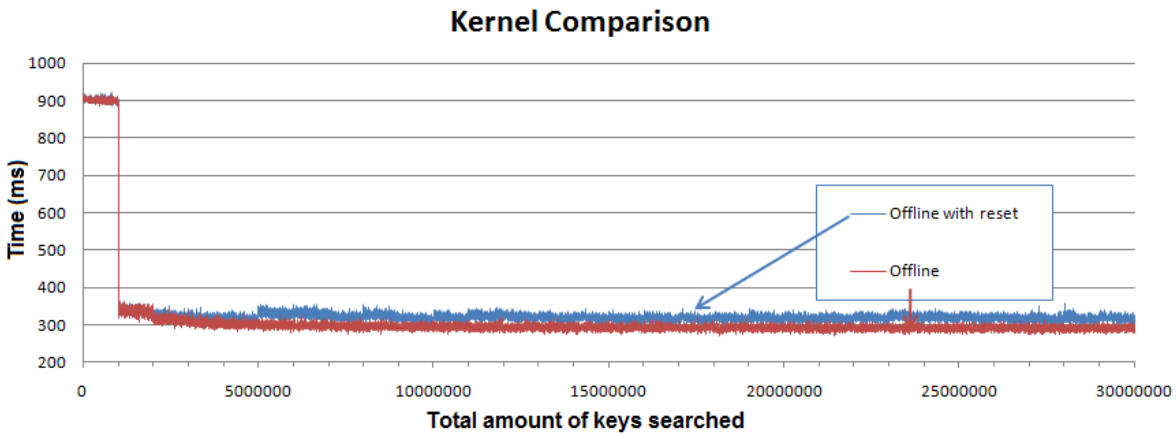

Figure 6.2: Search time for sets of 1000 keys among 1 million keys. The keys are activated according to a Gaussian distribution. Search times were computed with the kernel with real-time key reordering. Offline ordering is performed after searching one thousand sets of a thousand keys, either reseting hit counters or keeping them. The sharp transition of the offline reorder curve is caused by the first offline reordering.

Chapter 7

Scalability analysis

In order to study how an information system based on a parallel approach to the searching problem could be deployed, consider initially a one device solution. This single device performs searches over the deployed set of keys residing in its global memory, and can use one of, or a combination of the proposed ordering solutions (e.g. the use of the bubble sorting kernel without a random jump, combined with an offline reordering of keys, at times when the database is updated for the purposes of being updated with the newly deployed keys).

Suppose that the number of tags is still sufficiently low that they can all be stored in a single device's memory, but the number of necessary searches to find the proper keys corresponding to the generated pseudo-random identifiers over time is too high to be performed by a single device, and as such, N devices are used in parallel to perform the necessary operations. It stands to reason that a highly dependable load balancing solution should be used in conjunction with the computing nodes, assigning to each one a certain identifier to be processed. This solution has a problem, which is the degradation of the time required for the databases of the various different devices to reach convergence, considering, as was already explored in Chapter 6, that some keys are activated more often than others. In fact, the search for M keys, evenly distributed by N devices, would take N times the convergence time of a single device, albeit the total search time should be relatively close to N divided by the total search time required by the single device solution (as the faster convergence of the algorithm should result in a progressive lowering of the average search time).

Note that since the pseudo-random IDs have no correlation to their originating keys, there is no way to purposefully send some keys to a specific device, to tackle the convergence issue.

One solution to this problem would be the grouping of devices into geographical activation regions. Considering the motivation for this work, untraceable car identification, it is safe to assume that in a certain region, the set of highly used keys should be drastically different from another, as most cars typically move mostly within a certain somewhat fixed routine. This way, the convergence would be more effective if one were to separate the databases were the reordering will occur into different regions, a simple assignment of devices to regions would ensue. Furthermore, if the number of deployed threads so permits, it would even be possible to fit multiple regions, or regional key databases into a single device, where the keys are replicated in different orders, and a kernel launch pertains to a certain specific region, and therefore, a specific databases. This could be useful in cases where the number of deployed keys, as well as the number of searches over time are reduced, but a very high index of geographical separation is present. If the geographic locality principle applies in a

strict form, meaning that a key from a certain region will never appear in another, it is even conceivable the existence of disjoint key arrays. Naturally, this last point could also apply for databases in different devices.

However, what if this principle does not apply at all, due to the use of this identification paradigm for a different application, for example. In that case the information system could be set up so that when a certain computing node finds a key, it communicates this to the remaining nodes, that can periodically run a very simple kernel that simply moves this key, or multiple keys in parallel, closer to the beginning of the array. This way, one could use as many devices as needed without sacrificing the convergence time in a significant way.

The final consideration pertains to the unlikely event that the number of keys does not fit into a single device. In this case, we could number the nodes according to their respective database. The device with portion 1 of the array should be named 1, the device with portion 2 should be named 2, and so on. When a new search is needed, two options are possible, either the request is dispatched to a certain group of devices, for which the combined database equals the original, thus optimizing the minimum amount of time per search, or, alternatively, the load balancer could send the request to device 2 if and only if the key was not found in device 1, and so on. Note that when a match is found, the key would only need to be shared between the nodes of the same number, in order to guarantee a minimum convergence time.

7.1 Scalability test

To test the performance of the scenarios proposed in Chapter 6, a scalability test was devised, consisting of the following: consider a multiple CUDA node system networked to a load balancer, as depicted in Figure 7.1.

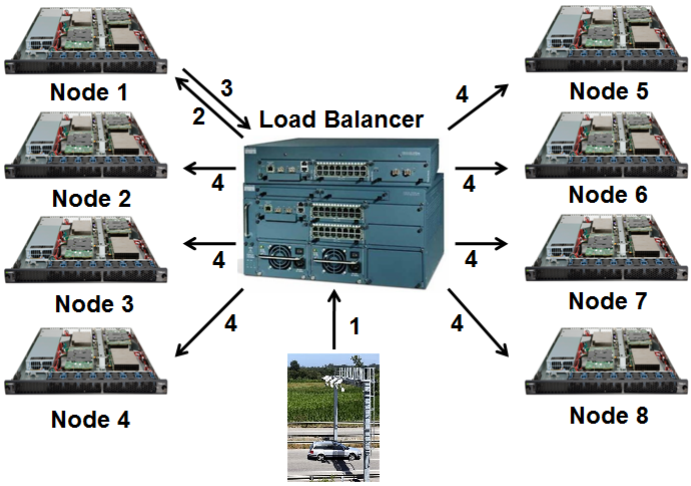

Figure 7.1: Proposed system architecture: Multiple nodes managed by a single load balencer, connected to the tollbooth systems.

When a car crosses the RFID reader, its tag generates a pseudo-random identifier, that is then sent to the information system so that the key can be extracted (1). A load balancer selects the least used node and sends it the pseudo-random identifier (2). Some time later, the node answers the request with the key of the device (3). The next step, (4) is desirable,

although not strictly necessary, and consists of a flood of the key (not the pseudo-random identifier) to all the remaining devices, so that they can proceed to a reorder of the key in their own databases.

Essentially, we introduce a new kernel that only goes through the database of keys, without computing any hashing or cipher function. This operation will be referred to from now on as a "reorder", although strictly speaking an exhaustive search is also performed, since the keys could be in different positions on different devices (it's position, or index in the array can not be used).

This step, as previously stated in chapter 6 allows an optimal convergence of the algorithms. However, the added message traffic, coupled with the time of actually reordering the keys that were not found locally, can possibly render the reordering algorithms useless, in the event that it prevents the nodes from processing as many keys as they could, on a global scale.

7.2 Tested scenario

Due to not being able to procure a suitable amount of CUDA enabled systems, a simulation in traditional x86 hardware was performed, exploring the trade off between the use of the reordering kernels with or without the flooding of keys to all the devices. Note also that we intend in to simulate not CUDA devices, but Tesla S1070 computing systems, or 4 devices grouped together, hence the term "nodes" when appropriate, as opposed to "devices".

Note that the time required to simply reorder a key is lower than that of searching what key originated a given pseudo-random identifier and then perform a reorder. This time difference was measured at roughly 20 times lower.

The main requirements for the simulation are as follows:

1. The failure of one or more devices shall not compromise the system.
2. A new search request shall be sent to the least used device
3. When a new device is connected, a database shall be requested from an already established device. If the local database (stored in a file) is relatively similar to those in the remaining devices, i.e., there haven't been many reorders/searches yet, the local database shall be used.

The node simulators were implemented in C, due to its inherent speed and added realism vis-a-vis CUDA. Due to the added synchronization and mutual exclusion requirements on the Load Balancer, this last program was written exclusively in Java.

7.3 Protocols

The diagram in Figure 7.2 shows the messages exchanged between 2 (not three) nodes and a load balancer. When the device X is started, it connects to the load balancer, querying for a database. As we are still at the beginning of the simulation, no reorder or search has been performed, and as such, the device's X local database should be used, hence the "Go" message from the server. Next, after the 4 threads responsible for handling each of the 4 devices in a node are launched, they connect to the load balancer with an "Hello" message,

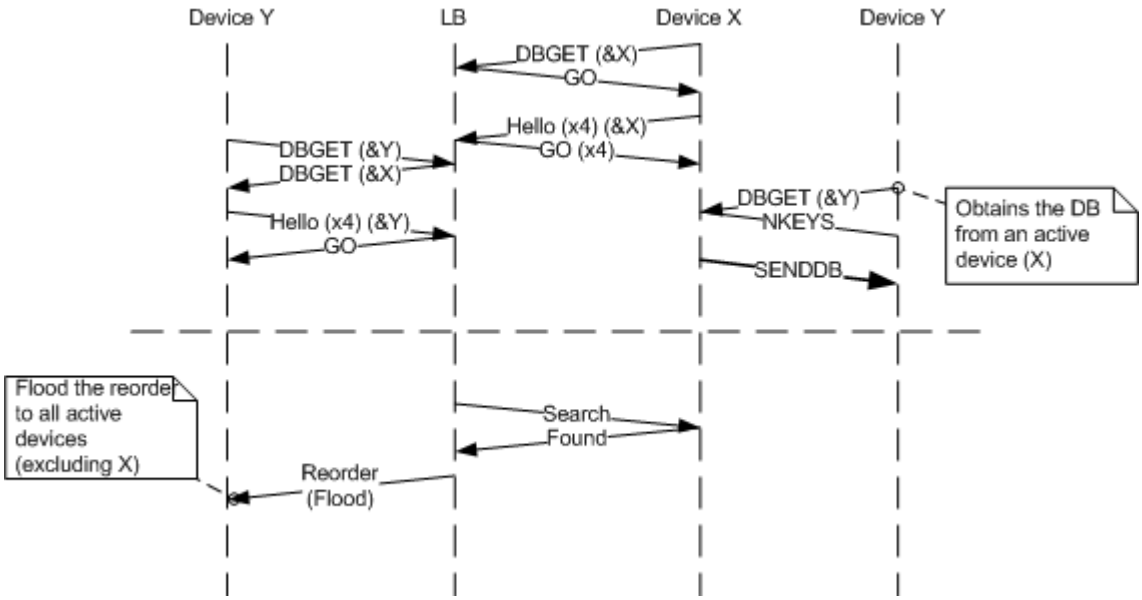

Figure 7.2: Sequence diagram. Initially the mechanism of sharing databases is explained, with the flooding of keys happening later.

to which a "Go" response is issued. Each device is handled independently from the remaining ones in the same node. From this point on, the device X is ready to receive search requests, and when these requests are served, the responses with the original key of the RFID tag will be flooded to all the active devices, if the simulation is configured in that way.

Later, a device Y is connected to the system, but this time, as the databases are already somewhat different from the original, the load balancer tells it to fetch a database from a random active server, in this case X. After a successful transfer (SENDDB), the device Y closes the connection to device X, and its 4 device threads connect to the load balancer, announcing their readiness to receive search and reorder requests. The number of devices available on a node should be configurable.

7.4 Software models

The following 4 figures depict the model of thread cooperation and the state transitions within them, for both the nodes and the load balancer. Squares correspond to passive entities while circles denote active threads. Figure 7.3, referring to a node, shows that a main thread launches the "sendDB" entity, that in spite of its name is responsible for receiving incoming connections and launching the "Worker" threads to handle the database requests themselves, but only after connecting to the load balancer and obtaining a database either locally or via another server. Only after this last step can the main thread launch the "Comn" threads that block waiting for messages from the load balancer, and use the shared memory structures in "smem" to set up a buffer of incoming requests that need to be serviced in the "Device" threads, that handle the actual CUDA simulations. "Smem" groups the various requests for each device (separately), and helps to minimize the latency of the network. The state transitions between these entities can be seen in Figure 7.5.

In order to speed up the simulation, the searches do not perform any computation, such

as MD5 or AES, and are more akin to a simple reorder. When the load balancer sends a search message to a node, with a pseudo-random ID of, say 0x12, this ID is searched for in the database as is, and the output of the search is 0x12. In other words, for the purposes of this simulation, the pseudo-random IDs are the actual keys.

As for the load balancer, Figures 7.4 and 7.6, its Main thread spawns the "Sim" thread, that loads a set of Keys activated in a distribution already discussed in chapter 6.4 from a file, so that they remain consistent across various simulations, and also to avoid unnecessary computation. When a new ID is generated, the "Sim" thread resorts to the "deviceOps" passive entity, responsible for serializing operations on device threads, hence preventing errors. The "device" threads block on input from the devices in the nodes, and also flood them to all other active devices via the "deviceOps" class, once again. The mechanism of load balancing, as demonstrated in Code Snippet 7.1, consists in sending a new search request to the device that has the least amount of pending searches, up to a maximum of 20, in which case the thread sleeps, being awaken when a new device is added to the system. The *send2leastUsed* method is also used to re-distribute the pending requests of a device that has crashed. "devices" is a vector containing all the devices active at present, and devicesLock is the condition variable responsible for guaranteeing mutual exclusion on accesses to the former.

Code Snippet 7.1: Load Balancing mechanism

```
/**
 * Send an entry to the least used device in the
 * Vector 'devices'
 *
 * @param e the entry
 */
public void send2leastUsed(entry e)
{
   synchronized(devicesLock)
   {
      // sleep while no devices are active
      while (devices.isEmpty())
         try {
            devicesLock.wait();
         } catch (InterruptedException ex) {}

      // sleep while every buffer is full
      int lower,index;
      boolean first=true;
      do {
         if (!first)
            try {
               devicesLock.wait();
            } catch (InterruptedException ex) {}

         lower=20;
         index=-1;
         first=false;
         for (int i=0; i<devices.size(); i++)
            if (devices.elementAt(i).getPending()<lower)
            {
               lower=devices.elementAt(i).getPending();
               index=i;
               if (lower==0)
                  break;
            }
      }while (lower==20);

      // send to the device
      devices.elementAt(index).process(e);
   }
}
```

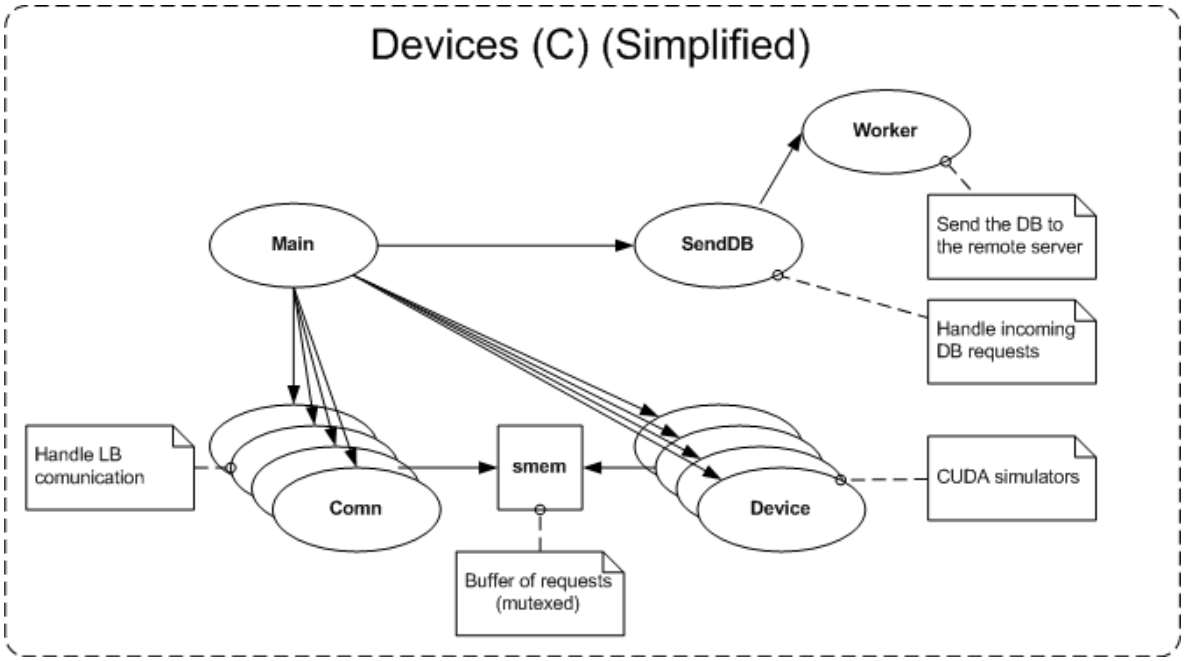

Figure 7.3: Model of thread cooperation in a node, active entities (threads) represented with circles and passive entities with squares

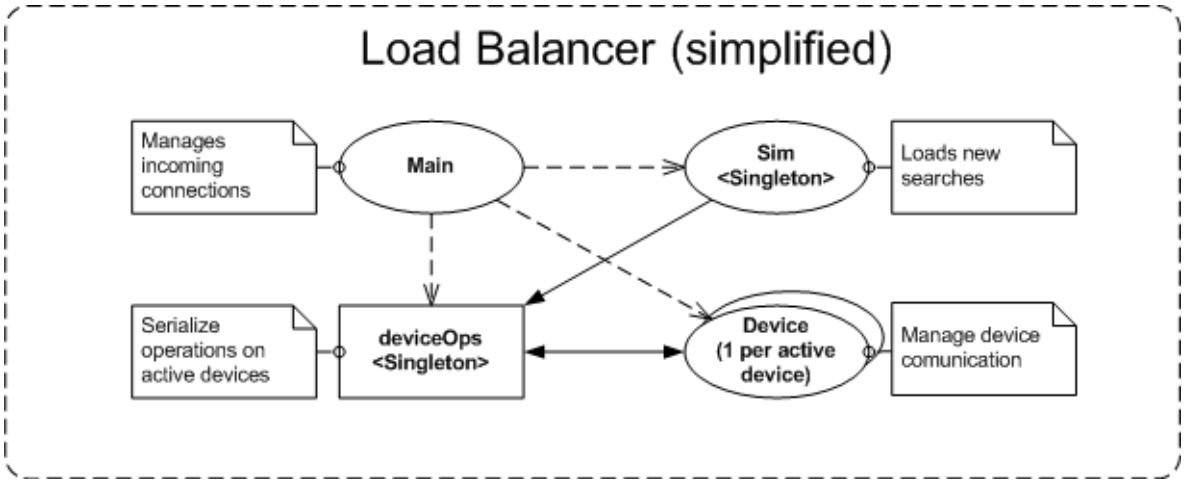

Figure 7.4: Model of thread cooperation in a Load balancer, active entities (threads) represented with circles and passive entities with squares

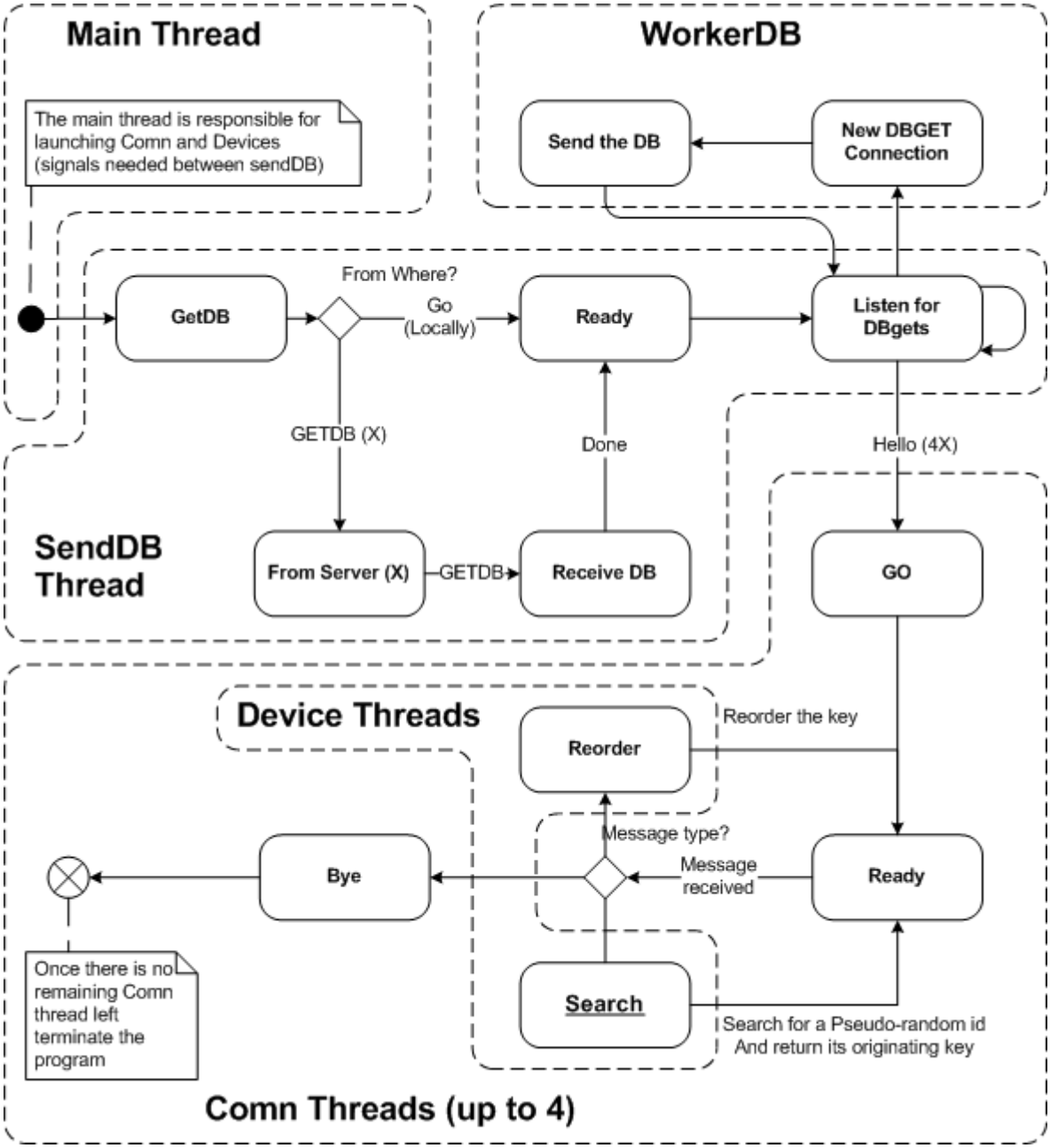

Figure 7.5: State diagram for the nodes, making the distinction between the various entities an their respective functions

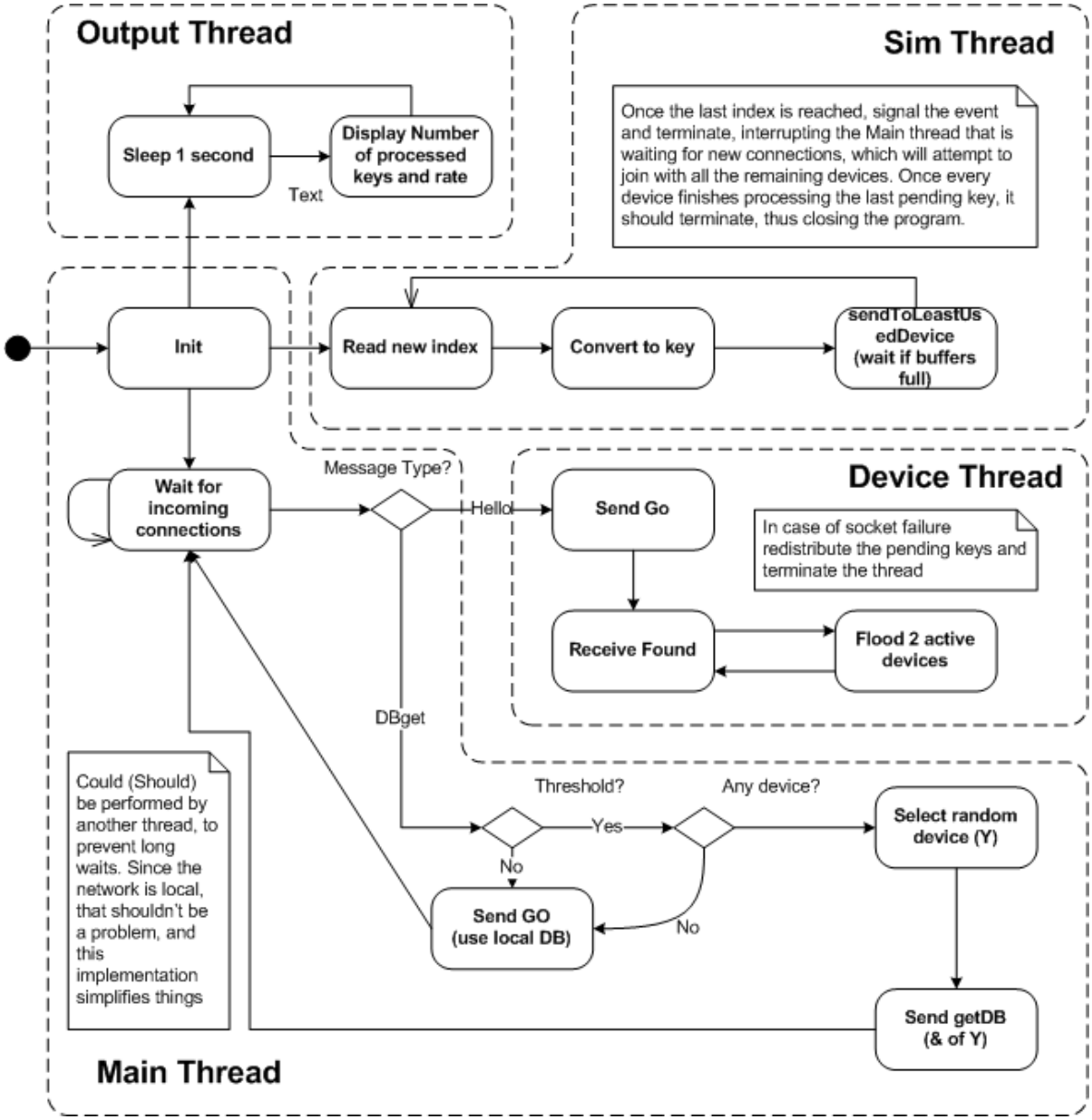

Figure 7.6: State diagram for the Load Balancer, making the distinction between the various entities an their respective functions

7.5 Performance evaluation

To evaluate the performance of the proposed system, several simulations where performed, with a varying number of nodes and alternating the flood of key reorders to on or off. As with the on-device simulations discussed in Section 6.4, sets of 1000 searches are timed, having a Gaussian distribution. In other words, at every thousand keys processed, the current CPU time is recorded an plotted. This method was chosen because it would yield more accurate results than waiting a given amount of time (say a second), and checking how many identifiers have been processed since the last count.

Due to time constraints, the number of keys was reduced by an order of magnitude, so that the simulations could complete in a manageable time interval. Also due to this problem, not all the simulations were performed with equal numbers of search requests, having been stopped once the desired effect (in this case the convergence time) has been clearly identified, except if otherwise stated. As an attempt to keep the results consistent with the GPUs, the size of the jump of a matched key towards the beginning of the array was also reduced by a factor of 10.

Figure 7.7 shows a comparison between the evolution of searches on a one node simulation (4 devices) with or without the reordering of keys on devices where the search did not take place. Clearly when floods are not used the convergence of the algorithm takes a lot longer, as one would expect. There is also a clear indication that as the number of searches performed increases, the system becomes faster when flooding is de-activated, meaning that the reorders are efectively taking up system time that could be used for processing more searches.

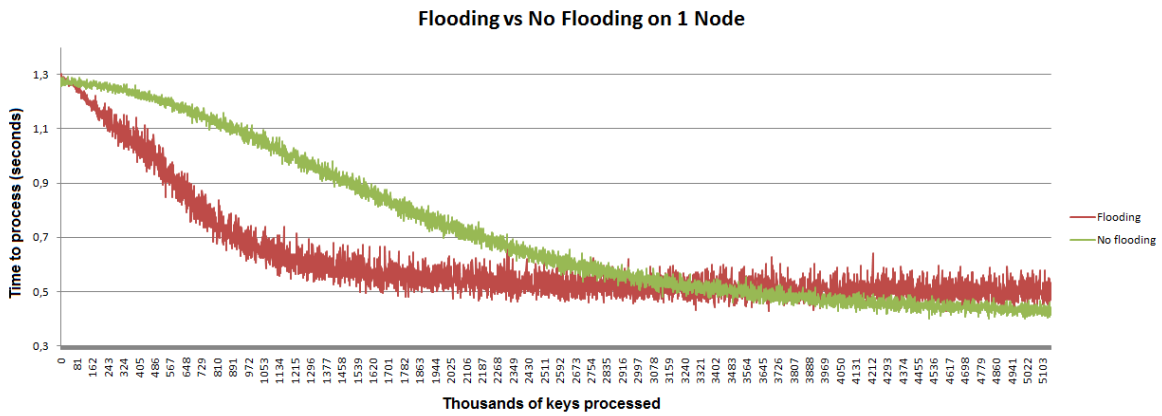

Figure 7.7: Flooding vs no flooding of searched keys, on a single node.

Figure 7.8 shows that an increase in the number of nodes has no effect on the convergence time, however, it becomes clear even with only 3 nodes that the scalability of this solution is being compromised. As we move progressively to a higher number of nodes, only minor differences in the time required to process a thousand keys is noted.

Figure 7.9 demonstrates a clear proportional relationship between the number of devices and the time required for the algorithm to converge, when a flooding of reorder requests is not used. What is not clear is how the time required to search a number of keys scales. However, by recording the time needed to process sets of keys at the very beginning of the simulations, compiled in Table 7.1, and inverting the results, hence obtaining not the number of seconds required to process a thousand, but the number of keys processed per millisecond,

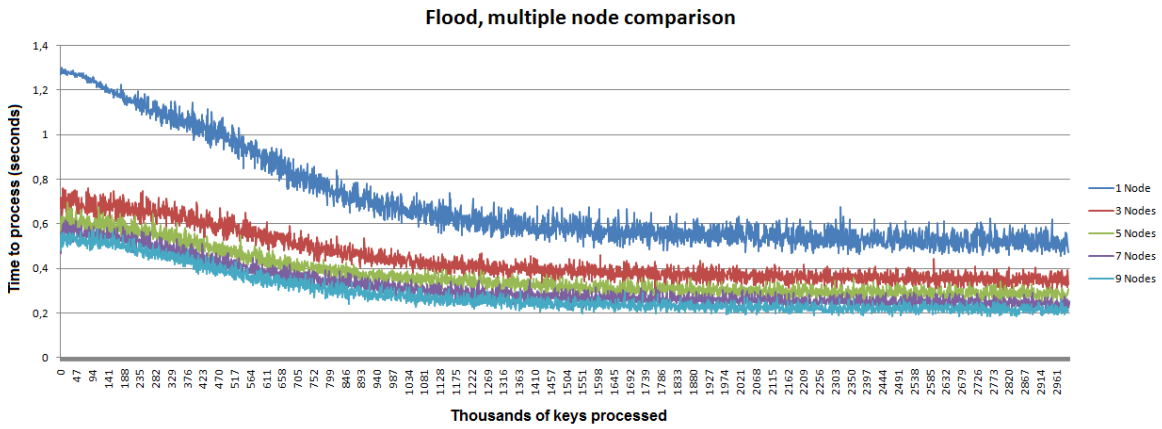

Figure 7.8: Flood across multiple nodes

all doubts are lifted regarding that issue [Figure 7.10]. Clearly there is a huge performance hit associated with the flooding of keys.

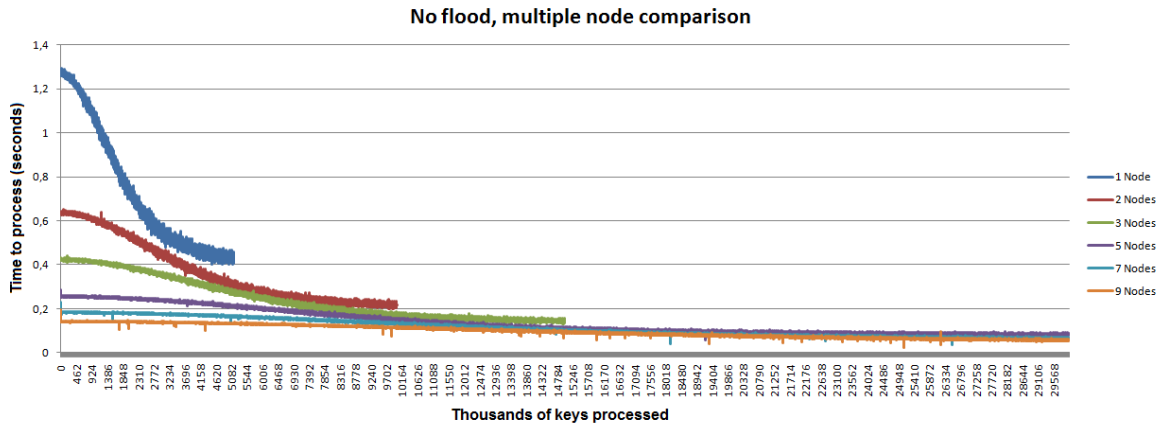

Figure 7.9: Convergence without flooding

Note that it can be advantageous to use the flooding method, provided that the frequency of database updates is sufficiently high (making the current order of the databases suboptimal) and the number of nodes in use is sufficiently low. Every time the database needs to be updated, if the recently added identifiers correspond to RFID tags that will be used frequently from that point on, a significant reorder of the databases would need to be performed, and as was proved previously, that would happen much faster with flooding.

1N	2N	3N	4N	5N	6N	7N	8N	9N
Flooding								
1,29	0,64	0,43	0,32	0,26	0,21	0,18	0,16	0,14
No flooding								
1,29	0,71	0,69	0,64	0,62	0,59	0,57	0,55	0,54

Table 7.1: Initial performance across all possible combinations. Units are the number of keys processed per millisecond.

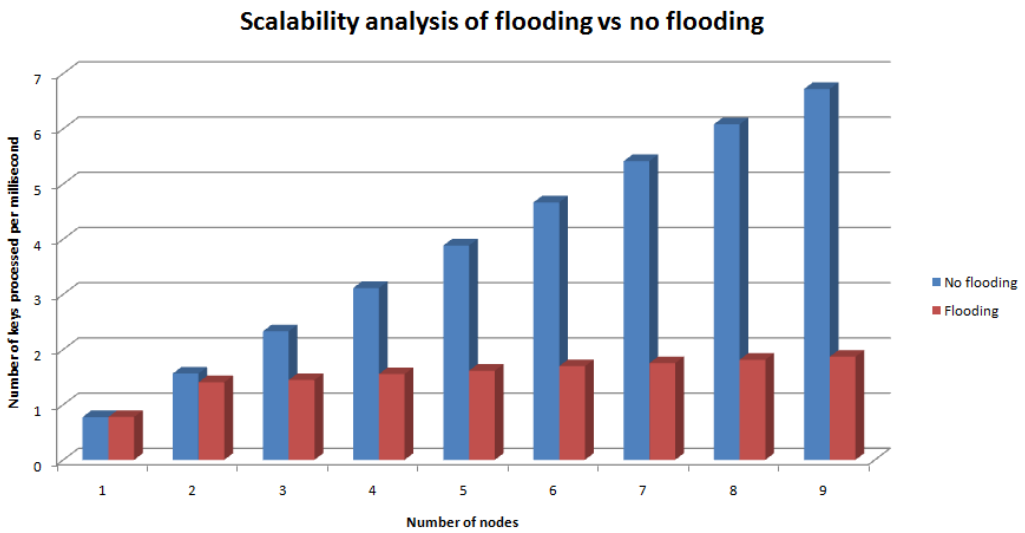

Figure 7.10: Scalability analysis, based on Table 7.1. Units are the number of keys processed per millisecond.

Chapter 8

Conclusions

In this document a study was presented on the identification of car RFID tags with pseudo-random identifiers, resorting to the use of massively parallel computation. A very simple challenge-response protocol was conceived, capable of producing 128-bit pseudo-random identifiers from 128-bit secret keys, compatible with RFID standards such as ISO 14443 A.

Several strategies were also tested, with the purpose of further reducing exhaustive key searches along all known keys. Performance evaluations with a single GPGPU device of an Nvidia S1070 computing system show that it takes less than 408 or 430 milliseconds to find a match with 250 million keys for, respectively, MD5 and AES.

Several ordering algorithms were evaluated with a Gaussian distribution for key activation, that resulted in a significant reduction of the match time either using real-time methods (kind of bubble sorting within the GPGPU kernel) or offline methods (quick sort by CPU application). Both methods can be combined, and the offline one is specially interesting when updating the set of known keys with the keys of all newly deployed tags.

An actual distributed information system was also modeled and implemented, proving that the scalability of such a paradigm is linear up to at least 9 nodes or 36 devices, and that for few devices and frequent database updates a flooding of key reorders is beneficial. When it comes to how the GPGPU model compares to a traditional CPU, even when dedicated instruction sets are used, for the AES computations, the 2 year old GPGPU is, in the worst case scenario, 3.5 times faster than the latest hexacore CPU.

Another important benefit of the GPGPU is the excellent scalability prospects with newer hardware. Even assuming that no benefit can be obtained by architectural improvements, for as long as Moore's law continues to dictate the pace of the industry, the newer GPGPUs will feature additional execution units, which can be used seamlessly with the already existing code by simply increasing the size of the grid. These assumptions were confirmed in our results with the newer Fermi GPGPU. However, one problem pertaining to the reordering of keys persists, due to the fact that the linearity of time to process versus the position of the key in the index could prevent a proper usage of the bubble reordering kernels.

To sum up, this privacy-preserving RFID identification strategy is a viable solution for untraceable car identification by authorized entities, as referred in Section 1.1. In 2008 there were nearly 256 million highway vehicles in the US [6], therefore the key searches with 250 million keys were not very far from real scenarios. Nevertheless, the system can scale up easily by using more GPGPU devices or by splitting keys among them.

Bibliography

[1] National Institute of Standards and Technology (NIST): Secure hash standard (SHA-1). (FIPS PUB 180-1) online at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[2] Rivest, R.: The MD5 Message-Digest Algorithm. RFC 1321 (1992) online at http://tools.ietf.org/html/rfc1321.

[3] National Institute of Standards and Technology (NIST): Advanced encryption standard (AES). (FIPS PUB 197) online at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[4] Dimitriou, T.: A Lightweight RFID Protocol to protect against Traceability and Cloning attacks. In: 1st IEEE/CreateNet Int. Conf. on Security and Privacy for Emerging Areas in Communication Networks (SecureComm 2005), Athens, Greece (2005)

[5] International Organization for Standardization / International Electrotechnical Commision: ISO/IEC 14443-3: Identification cards. Contactless integrated circuit(s) cards. Proximity cards. Part 3: Initialization and anticollision (2001)

[6] Bureau of Transportation Statistics (BTS): National Transportation Statistics, Table 1-11: Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances. online at http://www.bts.gov/publications/national_transportation_statistics/html/table_01_11.html (Checked in Feb 2011)

[7] Lengyel, T.K., Gedarovich, J., Cusano, A., Peters, T.J.: GPU Vision: Accelerating computer vision algorithms. online at http://www.c13software.com (2011)

[8] Kerr, A., Campbell, D., Richards, M.: GPU VSIPL: High-performance VSIPL implementation for GPUs. online at http://www.nvidia.com (2011)

[9] Tolke, J.: Implementation of a lattice Boltzmann kernel using CUDA. Springer Verlag (2008)

[10] Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: IEEE International Conference on Signal Processing and Communications (ICSPC 2007), Dubai, United Arab Emirates (2007)

[11] Trapnell, C., Schatz, M.C.: Optimizing data intensive GPGPU computations for DNA sequence alignment. In: Parallel Computing Vol 35. (2009) 429–440

[12] Ino, F., Gomita, J., Kawasaki, Y., Hagihara, K.: A GPGPU approach for accelerating 2-d/3-d rigid registration of medical images. In: International Symposium on Parallel and Distributed Processing and Applications. (2006) 939–950

[13] Agosta, G., Barenghi, A., Santis, F.D., Biagio, A.D., Pelosi, G.: Fast disk encryption through GPGPU acceleration. In: Parallel and Distributed Computing: Applications and Technologies. (2009) 102–109

[14] NVIDIA: NVIDIA CUDA C programming guide. online at http://www.nvidia.com (Checked in March 2011)

[15] Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edition. Prentice Hall (1988)

[16] Wikipedia: MD5. online at http://en.wikipedia.org/wiki/MD5 (July 2011)

[17] den Boer, B., Bosselaers, A.: Collisions for the compression function of MD5. In: Advances in Cryptology – EUROCRYPT '93 Proc., Lofthus, Norway (1994)

[18] Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199 (2004)

[19] Kaminsky, D.: MD5 To Be Considered Harmful Someday. Cryptology ePrint Archive, Report 2004/357 (2004) http://eprint.iacr.org/2004/357.

[20] Klima, V.: Finding MD5 Collisions – a Toy For a Notebook. Cryptology ePrint Archive, Report 2005/075 (2005) http://eprint.iacr.org/2005/075.

[21] Klima, V.: Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications. Cryptology ePrint Archive, Report 2005/102 (2005) http://eprint.iacr.org/2005/102.

[22] Klima, V.: Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105 (2006) http://eprint.iacr.org/2006/105.

[23] Bernstein, D.J.: Cache-timing attacks on AES. online at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2005)

[24] Tillich, S., Herbst, C.: Attacking state-of-the-art software countermeasures-a case study for AES. In: CHES. (2008) 228–243

[25] Tillich, S., Herbst, C., Mangard, S.: Power analysis resistant AES implementation with instruction set extensions. In: Cryptographic hardware and embedded systems – CHES 2007, Springer Verlag (2007) 303–319

[26] Tillich, S., Herbst, C., Mangard, S.: Protecting AES software implementations on 32-bit processors against power analysis. In: Applied Cryptography and Network Security – ACNS. (2007) 141

[27] Daemen, J., Rijmen, V.: AES proposal: Rijndael. online at http://www.cryptosoft.de/docs/Rijndael.pdf (1998)

[28] Gueron, S.: Intel advanced encryption standard (AES) instructions set. online at http://www.intel.com (2010)

[29] Rott, J.: Intel advanced encryption standard instructions (AES-NI). online at http://software.intel.com (2011)

[30] Weis, S.A., Sarma, S.E., Rivest, R.L., Engels, D.W.: Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems. In: 1st Int. Conf. on Security in Pervasive Computing (SPC 2003), Boppard, Germany (2003)

[31] Ohkubo, M., Suzuki, K., Kinoshita, S.: Cryptographic Approach to Privacy-friendly Tags. In: RFID Privacy Workshop, MIT (2003)

[32] Avoine, G., Oechslin, P.: A Scalable and Provably Secure Hash Based RFID Protocol. In: 2nd IEEE Int. Works. on Pervasive Computing and Communication Security (PerSec 2005), Kauai Island, Hawaii, USA (2005)

[33] Henrici, D., Muller, P.: Providing Security and Privacy in RFID Systems Using Triggered Hash Chains. In: Proc. of the 6th Ann. IEEE Int. Conf. on Pervasive Computing and Communications (PerCom'08), Hong Kong (2008)

[34] Lim, T.L., Li, T., Gu, T.: Secure RFID Identification and Authentication with Triggered Hash Chain Variants. In: 14th IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'08), Melbourne, Victoria, Australia (2008)

[35] Molnar, D., Wagner, D.: Privacy and Security in Library RFID: Issues, Practices, and Architectures. In: Proc. of the 11th ACM Conf. on Computer and Communications Security (CCS 2004), Washington, DC, USA (2004)

[36] Molnar, D., Soppera, A., Wagner, D.: A Scalable, Delegatable Pseudonym Protocol Enabling Ownership Transfer of RFID Tags. In: 12th Int. Works. in Selected Areas in Cryptography (SAC 2005), Kingston, ON, Canada (2005)

[37] Dimitriou, T.: A Secure and Efficient RFID Protocol that could make Big Brother (partially) Obsolete. In: Proc. of the 4th Ann. IEEE Int. Conf. on Pervasive Computing and Communications (PerCom'06), Pisa, Italy (2006)

[38] Lehtonen, M., Staake, T., Michahelles, F., Fleisch, E.: From identification to authentication – a review of RFID product authentication techniques. In: Workshop on RFID Security (RFIDSec 06), Graz, Austria (2006)

[39] Lim, C.H., Kwons, T.: Strong and robust RFID authentication enabling perfect ownership transfer. In Ning, P., Qing, S., Li, N., eds.: Int. Conf. on Information and Communications Security (ICICS '06), Raleigh, North Carolina, USA (2006)

[40] NVIDIA: Tesla S1070 GPU computing system. online at http://www.nvidia.com (April 2010)