Exhaustive Key Search on Clusters of GPUs

Davide Barbieri, Valeria Cardellini, Salvatore Filippone
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università di Roma "Tor Vergata", Roma, Italy
davide.barbieri@ghostshark.it, cardellini@ing.uniroma2.it, salvatore.filippone@uniroma2.it

***Abstract*—Exhaustive search is generally a last resort for solving a problem: each possible state of a system is generated and evaluated against a condition to find if the problem solution is attained. In some cases, for example in the reversal of cryptographic hash functions that make use of the salting technique, there are very few valid alternatives. However, the set of candidate solutions can be extremely large and therefore very substantial computing resources are needed to walk through the search space in a reasonable time. On the other hand, exhaustive search is very often embarrassingly parallel and so the task can be easily accelerated by distributing the work on a multitude of devices. In this paper we propose a pattern to parallelize general exhaustive searches on a heterogeneous and hierarchical network of computing nodes. We validate this pattern by applying it to the reversal of MD5 and SHA1 hash functions, both at coarse grain (work dispatching among nodes) and at fine grain (work made by each thread), reaching linear scalability with increasing computing power of the participating nodes. In particular, we show how to implement and optimize the hash key search on a GPGPU, achieving near-maximal throughput on various models of NVIDIA devices programmed with CUDA.**

I. Introduction

A hash function takes an arbitrary string as input and produces an output containing a fixed number of bytes; this output is often called a *digest*. Since the cardinalities of the sets of possible inputs and of possible outputs typically differ, more than one string can be mapped to the same hash product; for this reason, it is called a *one-way* function.

Some hash functions are suitable to securely store passwords in a database, since a malicious attempt to read the password by an attacker will be limited to the retrieval of the hashcode. However, to be a good candidate for cryptographic use, a hash function should produce collisions (that is, producing the same output for different inputs) very rarely, especially for inputs that use the same charset and comparable lengths. It should be very difficult to guess a possible input for that hash.

Studying the amount of time and resources needed by a brute-force attack to retrieve a password is a key step in understanding the actual level of security provided by a cryptographic hash function. In some working environments, it is a standard procedure to make periodic cracking tests, called *auditing* sessions, to assess the reliability of the employees' passwords.

There are various ways to create a hash lookup function:

- brute forcing;
- dictionary attack;
- lookup tables;
- rainbow tables.

Usually, there is no simple way to find a string that produces a certain hash, but it is conceptually possible to enumerate all possible strings, compute their hashes and compare them with the target hash. This exhaustive search is often referred as a *brute force* attack.

For short input strings brute forcing is a feasible method, but the number of possible inputs grows exponentially with the number of characters. For example, the number of strings containing at most 8 alphabetic characters (both lower and upper case) is $\approx 54,508$ billions; with 10 characters it becomes $\approx 147,389,520$ billions.

The number of attempts can be drastically reduced if a *dictionary* of recurring words is involved in the string set production. A hybrid technique that uses a dictionary along with a list of common password patterns provides a good way to guess longer passwords.

Lookup tables can be generated storing the mapping between each string with its digest to speed up subsequent searches, but such method becomes quickly unmanageable for the amount of memory required to store the table.

The *rainbow tables* method is used to achieve a trade-off between hash cracking speed and size of lookup tables. It concentrates in less space the information about solutions, but a certain amount of computation is needed to lookup a key.

The last two methods are completely useless when the key is concatenated with a random string in a technique called *salting*, since the corresponding hashcode for each key changes. Although the salting technique makes strings longer, it does not increment the search space since the random part of the string (the salt) to be concatenated is known by definition.

Another application of exhaustive search that is gaining interest is the production of secure transactions of a virtual currency on a peer-to-peer network. For example, in the Bitcoin network [1] transactions' consistency is based on blocks of data that are generated in a process called *Bitcoin mining*. In this process an exhaustive search is performed to find a 32-bit value (*nonce*) that is used as input to a hashing function based on the SHA256 algorithm, producing a hash with a certain number of leading zero bits (which is provided by the network and increases in time). Generating a block is rewarded by the network with a certain amount of money. Usually, since the mining of a block requires a processing power too large to be pursued by a single *miner*, communities of users (even thousands of people) over the Internet join and collaborate, dividing the search space and sharing rewards on the basis of the computing power contribution.

It is clear that the problems described above require a huge processing power to be solved in an acceptable amount of time. It is also usually impossible to obtain this processing power using a single device; hence, the interest in designing and implementing software that scales across multiple nodes, each containing multiple devices, that are organized a complex network.

Very large computing networks, like a *Bitcoin mining pool*, are constructed with the hardware that is commonly found in desktop computers, like multi-core CPUs and GPUs. These devices have very different specifications and exhibit different processing power.

In this paper, we present a parallelization pattern to implement exhaustive searches on hierarchical and heterogeneous architectures; then, we apply this pattern to the brute forcing of MD5 and SHA1 digests on a cluster of GPUs. Lastly, we show how to optimize the exhaustive search kernel in order to exploit the full throughput of NVIDIA CUDA devices. Our experimental results demonstrate that the proposed parallelization pattern applied to password cracking allows to achieve near-maximal throughput on various models of NVIDIA devices programmed with CUDA and in most cases outperforms well-known brute-force tools on a single GPU.

The remaining of the paper is organized as follows. In Section II we review related work on the bruce force method. In Section III we discuss how to parallelize the exhaustive search and present our parallelization pattern. Then in Section IV we describe how to apply the exhaustive search pattern to the design of a distributed password cracking system based on a cluster of GPUs. In Section V we discuss some optimizations that allow to maximize the performance of the exhaustive search kernel on CUDA-based GPUs and present some experimental results in Section VI. Finally, we draw some conclusions and give hints for future work in Section VII.

II. Related Work

Distributed password cracking aims to distribute the password cracking work among multiple nodes; in [2] Marechal analyzed various techniques (such as Markov chains [3]) that can be applied to distributed password cracking.

Previously published works on the brute force method have validated their model by implementing only the dispatching part over multiple nodes of a password cracking, while using the external tool BarsWF [4] to actually execute the cracking task on CPUs and GPUs [5]. Other published results show less than 10% efficiency when they apply the same model on fine grain parallelization [6]. A homogeneous parallel brute force cracking algorithm that performs all the work on the GPU side (including the generation of all candidates) has been proposed by Vu et al. in [7]. However, depending on the size of the character pool and the the password length, their algorithm may require a large amount of memory (some Gbytes) to store all the possible combinations in the GPU and this is not practical.

In our approach we tackle both the coarse-grain (network nodes) and the fine-grain (grid of GPU threads) parallelization, achieving up to 100% efficiency on a single GPU and up to 90% on a GPU cluster. Differently from [7], our approach requires a minimal amount of memory (less than 1 Kbyte) and does not require any initialization phase and separate generation of passwords. Moreover, our solution pattern can be applied to other exhaustive search strategies, beyond the current password cracking application we consider in this paper as a case study.

III. Parallelizing Exhaustive Search

In an exhaustive search method, also known as *brute-force*, we must enumerate all candidate solutions of a particular problem and execute a test to verify if any of them satisfies a particular condition. Since the evaluation of each candidate solution is generally independent from all the others, an exhaustive search offers very good parallelism potential: the solution space can be partitioned in many ways with very few, if any, constraints.

Only a very small amount of data must be scattered at the beginning of the computation to each computing node, to allow the generation of the solutions' subspace; we must also collect periodically a fairly small amount of data from each device to eventually terminate the search if a stop condition is met (e.g., a satisfactory number of solutions has been found).

Although in many cases such subdivision seems trivial, some design considerations should be kept in mind when optimizing the performance on complex topologies of computing nodes with different computing power.

In this section we present a parallelization pattern that can be applied to implement an exhaustive search on a heterogeneous and hierarchical architecture, like a node with a CPU and multiple GPUs or a cluster of such nodes. We particularly emphasize the control of performance levels, in order to achieve the best efficiency on the largest possible number of architectures.

A. Problem Definition

Given the set S of all possible solutions to a class of problems, we can execute an exhaustive search if there exist:

- a bijective function f from the set of natural numbers $N = \{0, 1, 2, 3, \dots\}$ into S, where S is either finite or countable;
- a test function $C : S \to 0, 1$.

Therefore, an exhaustive search involves the generation through $f(i)$ of all possible solutions, conducting tests with an increasing identifier i, and the test of the condition $C(f(i))$ for each entry.

Note that $f(i)$ can be trivial or it can follow a heuristics to favor testing of the most likely solutions. The test function C can be also arbitrarily complex: indeed, it might possibly require every other possible solution to be generated before evaluating the current candidate.

Furthermore, we define the operator *next* such that $\mathit{next}(i, f(i)) = f(i + 1)$. In many instances the execution of the *next* operator is much faster than the execution of $f(i+1)$, because it can be obtained with few manipulations of the $f(i)$ element's data.

Given the following cost functions:

- $K_f(i)$, the cost to generate a candidate solution from an identifier;
- $K_{next}(i, f(i))$, the cost to generate a candidate solution from another candidate;
- $K_C(f(i))$, the cost to evaluate a candidate;

the cost K_{search} of an exhaustive search over a set of n possible solutions on a single process is:

$$K_{search} = K_f(i_0) + \sum_{i=i_0}^{i_{n-2}} K_{next}(i, f(i)) + \sum_{i=i_0}^{i_{n-1}} K_C(f(i))$$

or if $next(i, f(i)) \equiv f(i+1)$,

$$K_{search} = \sum_{i=i_0}^{i_{n-1}} (K_f(i) + K_C(f(i))).$$

If $K_{next}(i, f(i)) < K_f(i+1)$ then the process' efficiency, defined as the time needed to test a solution over the time needed to generate the solution and then test it, will increase for larger n.

To distribute the process over multiple nodes we may use a master task, which

- scatters to the computing nodes the minimum data needed to generate the candidate solutions;
- waits for the completion of the computation on all nodes;
- gathers the results from the nodes;
- optionally executes a merge condition test on the received results.

The last step may be mandatory for algorithms where the test function C returns 0 when it can confidently exclude a solution but for which 1 is no guarantee that a solution has been actually found. For example, when searching for a vector that minimizes a cost function, each node would find the minimum in the provided subspace; then, the merge function would find the minimum cost among all the results of the participating nodes.

The master task introduces the following cost functions:

- $K^j_{scatter}$, the cost incurred by the master to send the data needed to generate the solutions' subspace to the j-th node;
- K^j_{search}, the cost incurred by the j-th node to search the solution through its subspace;
- K^j_{gather}, the cost incurred by the master task to receive the results from the j-th node;
- K_{C_M}, the additional cost incurred by the master to apply the optional merge function.

The search within each subspace is independent from all others, so that K^j_{search} can scale perfectly with the number of participating nodes; the merge function, however, should wait for all the results to be executed. The total cost K_D related to the dispatching of an amount of work, that spans from dispatching the main data to computing nodes to gathering their results, satisfies in the best case:

$$K_D >= \max_j(K^j_{scatter} + K^j_{search} + K^j_{gather}) + K_{C_M}$$

$$K_D <= \sum_j K^j_{scatter} + \max_j(K^j_{search}) + \sum_j K^j_{gather} + K_{C_M}.$$

In many cases $K^j_{scatter}$ and K^j_{gather} are fixed costs and become negligible for sufficiently large problems. For large intervals, K_D will depend almost exclusively on $\max_j(K^j_{search})$; therefore, we can conclude that K_D will depend on the performance of the slowest node.

Maximizing performance requires both ensuring that each node has enough work to make optimal use of its resources as well as balancing the load, so that no node is left idle while waiting for others.

When $K_C(f(i))$ and $K_{next}(i, f(i))$ are constants, we can easily obtain the minimum amount of work to dispatch such that all nodes will exhaust their computation in the same time and at high efficiency, following these steps:

- perform a tuning step to estimate for each node j the minimum number of candidates n_j needed to achieve a given target efficiency, and get the peak throughput X_j;
- find the node with the maximum throughput $X_{max} = \max_j X_j$;
- balance the work on each node j with respect to $\max$, by setting its workload as $N_j = N_{\max} \cdot (X_j / X_{\max})$; since for each j we have $N_j \geq n_j$, this implies $N_{\max} \geq n_j \cdot (X_{\max}/X_j)$; therefore $N_{\max} = \max_j(n_j \cdot (X_{\max}/X_j))$;
- the number of solutions to be tested by the node j will then be $N_j = N_{\max} \cdot (X_j / X_{\max})$

Therefore, the dispatcher task will provide work to its connected nodes with a granularity given by the size N_j of each interval of candidate solutions.

The tuning step could be skipped when a performance model that correlates efficiency, performances, and size of the search subspace for the considered algorithm is available. An approximated model could be built offline by performing a sequence of tests with increasing search size on each node of the cluster.

In a hierarchical topology, the task will dispatch work to other network's subtrees (that is, to dispatcher tasks on other nodes). In this case, the same assumptions outlined so far hold, since they can be considered as computing nodes with a throughput that is the sum of the throughputs of the child nodes and with a minimum number of candidates needed to provide high efficiency that is equal to $N_{node} = \sum_j N_j$. Optionally, N_{node} could be arbitrarily increased to minimize

the overhead caused by the dispatch and merge steps over the full computation.

However, a major constraint of our pattern is the assumption that the size of the intervals that are periodically assigned to the node could be arbitrarily large, while remaining irrelevant compared to the size of the whole search space for the considered cluster.

The proposed pattern can be extended to a dynamic network that can be configured at runtime, by executing the above mentioned steps each time the number of depending nodes or their actual performance metrics vary.

The same approach could be used to provide a minimum fault tolerance model, since it should be possible to monitor the activity of nodes and recalculate the partitioning of the search space each time a set of nodes becomes temporarily inactive. This model in some conditions may not be enough convenient, since the inactivity of a dispatching node would block the contribution of all the nodes in the dispatching sub tree.

IV. Password Cracking

In this section, we describe the application of the exhaustive search pattern introduced in Section III to the design of a distributed password cracking system for MD5 and SHA1 hashes. The system runs on multiple nodes with multiple GPU devices.

Given a charset of N characters, the $f(i)$ function should generate an enumeration that associates a unique unsigned integer to a string. An effective method is to consider a string as an arbitrarily long number represented in base N (that is, using N symbols); so, $f(i)$ would be implemented as an encoding routine that converts a number in this particular number system, associating each digit to the corresponding ASCII character.

An example of such mapping would be the function that translates natural numbers to strings over the charset a, b, c in the following way:

$$[0, 1, 2, 3, 4, 5, 6, 7, \dots] \rightarrow [\epsilon, a, b, c, aa, ab, ac, ba, bb, \dots], \tag{1}$$

with ϵ denoting the empty string.

The function in (1) corresponds to the algorithm shown in Figure 1.

Require: $id \in \mathbb{N}, charset = [' a', ' b', ' c', \dots])$
Ensure: $str = f(id, charset)$
  $str = []$
  while $id > 0$ **do**
    $id \leftarrow id - 1$
    $currentCharId = id \bmod length(charset)$
    $currentChar = charset[currentCharId]$
    $str = currentChar \oplus str$
    $id = \lfloor \frac{id}{length(charset)} \rfloor$
  end while
  return str

Fig. 1. Pseudocode for the $f(id)$ operator; $\oplus$ is the string concatenation operator

Require: $str = f(id, charset)$
Ensure: $nextStr = f(id + 1, charset)$
  $nextStr = str$
  $currentPosition \leftarrow length(str) - 1$
  repeat
    $temp \leftarrow (id + 1) \bmod length(charset)$
    $nextStr[currentPosition] \leftarrow charset[temp]$
    $id \leftarrow \lfloor \frac{id}{length(charset)} \rfloor$
    $currentPosition \leftarrow currentPosition - 1$
    if $currentPosition = -1$ **then**
      $nextStr[length(str)] \leftarrow charset[0]$
      return $nextStr$
    end if
  until $temp = 0$
  return $nextStr$

Fig. 2. Pseudocode for the *next* operator

The generated string $f(i)$, that is, the candidate solution to the cracking problem, should be checked with the test function $C(f(i))$. This is essentially the application of the hash function to the string and the comparison of the result with the input hash. The conversion function $f(i)$ requires more time for longer inputs, and in practice it can become dominant with respect to the hash function.

On the other hand, the *next*$(f(i))$ function can be obtained with a much smaller effort. A possible implementation is described in Figure 2; in most cases it modifies just a single character.

For relatively small strings, that is less than 57 characters, the execution time of both algorithms is essentially independent of the string length. For longer strings, the intermediate result of the hashing algorithm may be saved and reused for a large number of instances sharing the first bytes of the string; thus, for each key we can process only the last block of 64 bytes.

When we apply the performance model presented above, it is then reasonable to approximate the cost of the test function K_C to a constant. This means that the algorithm that dispatches work to computing units and nodes can select intervals of keys just considering the size of each interval and the computing performance of the target node, disregarding the keys lengths. In particular, the ratio between the number of identifiers to be provided to different nodes should be equal to the ratio of the computing power of the nodes.

Given a charset made by N elements, the number of unique strings of length K is N^K, and the number of unique strings with length from K_0 to K is $N^{K_0} + N^{K_0+1} + \dots + N^{K-1} + N^K$. In closed form:

$$\mathbf{S}^{\mathbf{K}}_{\mathbf{K_0}} = \sum_{i=K_0}^{K} N^i = N^{K_0} + N^{K_0+1} + \dots + N^{K-1} + N^K$$

$$N\mathbf{S}^{\mathbf{K}}_{\mathbf{K_0}} = N^{K_0+1} + \dots + N^{K+1} = \mathbf{S}^{\mathbf{K}}_{\mathbf{K_0}} - N^{K_0} + N^{K+1}$$

Therefore,

$$\mathbf{S}^{\mathbf{K}}_{\mathbf{K_0}} = \frac{N^{K+1} - N^{K_0}}{N - 1} \tag{2}$$

If $N = 1$, then

$$\mathbf{S^K_{K_0}} = \sum_{i=K_0}^{K} 1^i = 1^{K_0} + 1^{K_0+1} + \cdots + 1^{K-1} + 1^K$$

that is,

$$\mathbf{S^K_{K_0}} = K - K_0 + 1 \tag{3}$$

Given a maximum and minimum size for our solution, we can use Equations (2) and (3) to compute the size of our search space. Our $f(id)$ is similar to the concept of a *base-n-like* number found in [6], in which the dispatching task itself needs to generate strings and recursively subdivide the search space, and needs some overhead when working with keys having different lengths.

A. GPU Kernel

The hash functions that we implemented on GPU are the *Message Digest algorithm 5* (MD5) [8] and the *Secure Hash Algorithm 1* (SHA1) [9]. Both of them work in principle on arbitrarily long strings of any length; however, in the context of our application we limited the maximum number of character to 20.

Mapping an interval of solution identifiers to the threads of a CUDA grid is straightforward; the environment guarantees that each thread has a unique identifier that can be used to select a unique subset of the search space.

This requires that each thread should call the conversion routine for each testing key; to reduce the time spent on the conversion routine, it is possible to assign a larger number of strings per thread by applying the $next$ operator. In this case, each thread would generate its start identifier multiplying its unique id by a factor equal to the number of solutions per thread to be tested.

The operating system may put a limit on the maximum time that a driver of a graphic card should wait for the completion of a running kernel; we can easily bypass this problem by adjusting the amount of tests per call and spreading the computation over multiple grids.

In our previous works on the CUDA platform ([10], [11] and [12]) we found it easier and effective to implement an optimized kernel for a class of cases that met a set of favorable conditions, then we adapted the other cases by e.g., padding the input to meet alignment requirements in global memory; similar concepts apply to the current problem.

We know that it is possible to implement a fast MD5/SHA1 function packing characters inside unsigned integer registers [13]. Therefore, we manage strings by aligning them to integer variable boundaries, i.e. multiples of 4 characters, padding with the *EOF* character as necessary.

V. GPU Optimizations

In this section we describe how to maximize the performance for a SIMT-oriented GPU like a CUDA device, by limiting the kernel overhead and maximizing the throughput of a MD5 hash function. The same considerations can be applied to other hash functions, including SHA1.

A GPU kernel grid should have a sufficiently large number of threads to be efficient, since all multiprocessors should be used at the same time and hazards caused by instruction dependencies should be hidden by other active warps scheduled on the same multiprocessor. It is also important that each thread should produce a certain quantity of useful work per kernel call to reduce the impact of the thread overhead on the total execution time.

Since the application at hand is clearly limited by the throughput of arithmetic instructions, in order to optimize its performance it is very important to study the characterization of the load and the way the GPU hardware fetches and executes the various instructions involved.

With this knowledge we can build a model useful to identify the bottleneck of our kernels and to forecast the percentage of the theoretical peak performance we can achieve.

A. CUDA Multiprocessor Architecture

CUDA capable GPUs are divided in several architecture families, which are identified by some sets of specifications called *compute capabilities*. GPUs having the same compute capability share the same multiprocessor design; what varies is essentially the number of multiprocessors, the amount of dedicated RAM, and the clock frequency of compute units and memory.

The CUDA model forces the developer to design a parallel algorithm in order to be automatically scalable on the available multiprocessors. The developer has to maximize the throughput of the single multiprocessor defining how a block of threads does a part of the whole work. If we want to optimize our kernel for the widest possible set of devices, we have to produce an optimized version of code for each compute capability, and not necessarily for each device model.

At the time of this writing, there are 8 different compute capabilities in the NVIDIA GPUs [14]. Most of their differences concern the global memory access, caching hierarchies, and floating-point precision. These properties do not affect the performance of our kernel, since memory accesses are very infrequent, On the other hand, what is interesting for us is the throughput of each class of instructions and how these instructions are issued to the different pipelines. Therefore, we can actually simplify our considerations by identifying a smaller set of multiprocessor architectures.

Table I summarizes the architecture specifications for each compute capability. We exclude from our considerations the devices having compute capability 3.5, since we were unable to get access to such type of device in time for this writing.

TABLE I. MULTIPROCESSOR ARCHITECTURE

Compute capability	1.*	2.0	2.1	3.0
Cores per MP	8	32	48	192
Groups of cores per MP	1	2	3	6
Group size	8	16	16	32
Issue time (clock cycles)	4	2	2	1
Warp schedulers	1 single-issue	2 single-issue	2 dual-issue	4 dual-issue

An arithmetic instruction belonging to a warp (a set of 32 threads) is issued by a warp scheduler to a group of cores.

As shown in Table I, from compute capability 2.1 and above, the warp schedulers are dual-issue; this means that each warp scheduler can dispatch two independent instructions of the same warp in the same clock cycle. Moreover, since the number of warp schedulers in these architectures is less than the number of groups of cores, the kernel should provide enough *instruction level parallelism* (ILP) to exploit the computing power of all the cores.

There are some arithmetic instructions, however, that cannot be executed on all cores. Different throughputs are listed in the CUDA programming guide [14] for different arithmetic operations, specific to each NVIDIA architecture family. Such information is summarized in Table II.

TABLE II. INSTRUCTION THROUGHPUT

Compute capability	1.*	2.0	2.1	3.0
32-bit integer ADD	10	32	48	160
32-bit bitwise AND/OR/XOR	8	32	48	160
32-bit integer shift	8	16	16	32
32-bit integer MAD	8	16	16	32

To understand which units actually carry out the instructions presented in Table II, we had to write some *ad-hoc* kernels repeating many times a certain set of instructions, because such a level of detail is not present in the NVIDIA official documentation. What we figured out is that:

- devices of compute capability 1.* execute each of the instructions in Table II using the same cores; in some cases, integer additions are also executed on the *special functions units* which provide an additional throughput of 2 instruction/cycle per multiprocessor;
- devices of compute capability 2.* execute each of the instructions in Table II using the same cores; among these, instructions with lower throughput are only executed on a single group of 16 cores.
- devices of compute capability 3.0 execute integer ADD and logical operations on 5 of the 6 groups of 32 cores, while they execute integer shifts and integer MAD (multiply and add) on only 1 group of 32 cores.

The thread overhead can be reduced by limiting the amount of code outside of the main hash function's body. To do this, we let each thread generate and test more than a single string, thus iterating the core instructions for a substantial number of times.

The hashcode to be looked up can be passed to the GPU kernel through constant memory since all threads will read it, so that it can be read very quickly. The constant memory can also be used to load the substring common to all the strings generated on the GPU.

B. The Main Bottleneck

In Table III we list the number of arithmetic instructions that a single MD5 hash function requires on a multiprocessor; we are simply counting all the operations that cannot be evaluated at compile time in the CUDA source code.

TABLE III. INSTRUCTIONS COUNT (MD5)

32-bit integer ADD	320
32-bit bitwise AND/OR/XOR	160
32-bit NOT	160
32-bit integer shift	128

We verified how these instructions were actually compiled into machine code for each target architecture we are considering, by using the `cuobjdump -sass` tool included in the CUDA toolkit.

Some differences among the target architectures are related to how the *bit rotate* operator is actually translated by the compiler into machine code. The left rotation of x by n bits is implemented in CUDA as $(x << n) + (x >> (32 - n))$ on unsigned integers.

On target architectures with compute capability 1.* we found that this operation is translated into a pair of `SHL` and `SHR` shift instructions plus an additional `ADD` instruction. When compiling for compute capability 2.* and 3.0, the same code is translated in a `SHL` instruction followed by a `IMAD.HI` (integer multiply-add) instruction. The latter emulates the `SHR` instruction ($a >> N$) by performing a multiplication ($a * 2^{32-N}$) that actually shifts left by $32-N$ in a temporary 64-bit register; then, the `.HI` part means that the most significant 32-bit half of the result should be taken and added to the resulting register.

On different compiler versions the same operation is translated into a `SHR` instruction followed by an `ISCADD` (integer add with scale) instruction. The latter represents a shift left followed by an addition. These two versions are totally interchangeable, since they provide the same function and performance.

Devices with compute capabilities 3.5 provide higher performance opportunities since they can execute a 32-bit rotate operation in a single machine instruction, called *funnel shift* [15]. This new, previously unsupported instruction performs a complete rotation, or the work of two shift instructions and one add, and has a double speed, so that the overall throughput is quadrupled with respect to compute capability 3.0 [14].

In all these cases, the number of `ADD` decreases since `ISCADD`, `IMAD`, and the funnel shift instructions implicitly perform the addition.

Table IV shows the actual count of the above mentioned instructions in the kernel optimized for strings of length 4. We have found that the compiled code that produces a single MD5 hash is only made by these instructions, while the overhead caused at each iteration by the *next* operator is less than the 1% of the time spent by the hash function.

The unary `NOT` operations are omitted since they are merged with other instructions in the final phase of compilation.

Looking at the throughput specification in Table II and the compilation output in Table IV, we can identify 3 classes of arithmetic instructions for both the MD5 and SHA1 kernels:

- addition instructions;

TABLE IV. ACTUAL INSTRUCTION COUNT (MD5)

	1.*	2.* and 3.0
IADD	284	220
AND/OR/XOR	156	155
SHR/SHL	128	64
IMAD/ISCADD	0	64

- logical instructions;
- shift/MAD instructions.

An optimization, which was originally introduced in the BarsWF password cracker [4], achieves a speedup of about 1.25 in almost all architectures, since it reduces each class of instructions more or less by the same percentage. To explain this trick, note that we can test a candidate solution for a MD5 lookup in two ways:

- we can start from a string, then apply the MD5 steps to generate a hash, and compare it with the target hash;
- we can start from the target hash, then apply the inverse MD5 steps to generate a string that should be equal to a reference A.

Both approaches lead to a procedure with 64 steps. The MD5 algorithm has one interesting property: the first block of 4 bytes is used inside the first step of the algorithm but not in the last 15 steps. Therefore, we can mix these two approaches in this way:

> If a thread iterates modifying only the first block of 4 bytes, it can apply once the second approach to revert the hash function by 15 steps, then it can iterate over the different strings (with different prefixes) using only the first 49 steps of the first approach and testing the result with the reverted hash.

Since we can cache results from each iteration, such a kind of optimization produces a speedup for both sequential and parallel architectures.

This technique can be applied if just one thread iterates to modify characters from the first 4 bytes of the string; therefore, we have to modify the $f(i)$ and *next*$(i, f(i))$ functions in such a way that the mapping described in (1) becomes:

$$[0, 1, 2, 3, 4, 5, 6, 7, ...] \rightarrow [\epsilon, a, b, c, aa, ba, ca, ab, bb, ...]. \quad (4)$$

This can be achieved by a simple change to line 6 of the algorithm in Figure 1:

$str = str \oplus currentChar$

and the body of the algorithm in Figure 2 in order to modify the prefix of the string accordingly.

Another optimization can help us save three more steps in most of the cases. Each step modifies just 4 of the 16 bytes result; the last four steps therefore will produce one part of the final result each. So, rather than waiting for the completion of all the four steps, we can anticipate the checks as soon as each part is computed.

In Table V we report the instruction count of the optimized kernel.

TABLE V. REAL INSTRUCTIONS COUNT (MD5)

	1.*	2.* and 3.0
IADD	197	150
AND/OR/XOR	118	120
SHR/SHL	90	46
IMAD/ISCADD	0	46

To identify further tuning opportunities we need to know if the full throughput for each class of instruction can be achieved. With the *NVIDIA CUDA Profiler* [16] we found that the kernel does not achieve any instruction level parallelism, since the number of instructions dispatched in a dual-issue fashion is very low (less than 10%). This means that on devices with compute capability 2.1, we leave a group of cores unused most of the time, while on devices with compute capability 3.0 two groups of cores are left unused.

On compute capability 2.* and 3.0, the ratio between addition/logical operations and shift/MAD operations is $R = \frac{270}{92} = 2.93$. On Fermi cards, ignoring the unused groups of cores, the shift/MAD instructions would be completely overlapped with addition/logical instructions, since only 2 groups of cores can be used and $R > 1$. On Kepler cards, the shift/MAD instructions would become the bottleneck, hiding the addition/logical instructions, since 4 groups of cores could be used but $R < 3$. Therefore, we need to account for different critical paths on different architectures.

By using the CUDA intrinsic *__byte_perm* it is possible to reduce the number of shifts needed by MD5 on compute capability 3.0. The intrinsic function is used to execute a rotation by 16 bits in a single instruction, thus having the same cost of a single shift instead of two.

In Table VI we can find the instruction count of the final, optimized kernel.

TABLE VI. REAL INSTRUCTIONS COUNT FOR THE OPTIMIZED KERNEL (MD5)

	1.*	2.* and 3.0
IADD	197	150
AND/OR/XOR	118	120
SHR/SHL	90	43
IMAD/ISCADD	0	43
PRMT (__byte_perm)	0	3

In this case, on the compute capability 3.0, shifts and additions contribute equally to the bottleneck, since $43 + 43 + 3 = 89 \approx \frac{270}{3}$. At first glance, replacing a left shift by four bits with four separate additions:

$t = a + a$ // $a >> 1$
$t = t + t$ // $a >> 2$
$t = t + t$ // $a >> 3$
$t = t + t$ // $a >> 4$

should provide a better throughput, since addition has 5 times the peak throughput of a shift operation, but in the context of our kernel it would actually decrease the overall performances. Providing a better ILP factor would also be pointless on c.c. 3.0, since the additional cores would be used by a portion of code (addition/logical instructions) that is already completely hidden.

A better ILP factor, that is achievable interleaving the production of the hash of two strings at a time, is nevertheless a good choice on Fermi, since that architecture is limited by addition/logical instructions.

The same kind of analysis and optimizations were applied to the implementation of the SHA1 hash function, which shows an even lower ratio between addition and shifts/MAD operations (≈ 1.53).

VI. Experimental Results

In this section, we present some numerical results of the cracking system described in this paper. First, in Section VI-A we describe the experimental setting; then, in Section VI-B we discuss the experimental results.

A. Reference Hardware

Our tests were made on a small network of PCs, each one equipped with one or two GPUs. The system is heterogeneous and the performance power of the network tree is deliberately unbalanced to demonstrate the system flexibility.

The network is composed by the following nodes:

- Node A, which holds a CUDA device:
  - one NVIDIA Geforce GT 540M;
- Node B, which holds two CUDA devices:
  - one NVIDIA Geforce GTX 660;
  - one NVIDIA Geforce GTX gTi;
- Node C, which holds a CUDA device:
  - one NVIDIA Geforce 8600M GT;
- Node D, which holds a CUDA device:
  - one NVIDIA Geforce 8800 GTS 512.

The hardware specifications of the GPUs are summarized in Table VII.

TABLE VII. GPU specifications table

	8600M	8800	540M	550Ti	660
Multiprocessors	4	16	2	4	5
Cores	32	128	96	192	960
Clock (MHz)	950	1625	1344	1800	1033
Compute capability	1.1	1.1	2.1	2.1	3.0

The network topology is the following:

- Node A dispatches part of the work to nodes B and C;
- Node C dispatches part of the work to node D.

B. Performance Results

In Table VIII we show the average performance of our cracking software when executed on a single GPU. As performance index we consider the throughput in terms of Mkeys/s (million key tests per second). Table IX shows the same metric for the execution on the entire network. The search space is the set of password containing up to 8 alphanumeric characters, both lower and upper cases.

We compare the performance results of our approach with the throughput achieved by Cryptohaze Multiforcer [17] and BarsWF [4] tools on a single GPU. Both are open source high performance brute-force tools and provide support to NVIDIA CUDA devices. Moreover, we compare the throughput achieved by our approach to the theoretical throughput, that is the maximum throughput that we can expect from our implementation of MD5/SHA1 on the target hardware. We calculated the theoretical throughput by considering the number of instructions required by each MD5 hash function listed in Table VI and the properties of the concerned architecture as follows.

On compute capability 1.* devices there is only one warp scheduler, so all types of warp instructions executed on the same multiprocessor will be serialized. Each multiprocessor will receive a batch of hash functions to be executed and, in the best case, they will be dispatched evenly between multiprocessors and executed at the peak throughput. Each function will require a multiprocessor to execute $N = N_{ADD} + N_{LOP} + N_{SHM}$ instructions, comprising additions, logical operations, and shifts/MAD operations. The time needed to execute all these instructions for a large enough set H of hashes will be at least $T_H = \frac{H \cdot N_{ADD}}{X_{ADD}} + \frac{H \cdot N_{LOP}}{X_{LOP}} + \frac{H \cdot N_{SHM}}{X_{SHM}}$; so the average execution time per hash is $T = \frac{T_H}{H} = \frac{N_{ADD}}{X_{ADD}} + \frac{N_{LOP}}{X_{LOP}} + \frac{N_{SHM}}{X_{SHM}}$. The throughput of a single multiprocessor, in terms of hash tests in the time interval, should be $X_{MP} = \frac{1}{T}$; ideally, the whole GPU would provide $X_{1.*} = X_{MP} \cdot MP\_count$, where MP_count is the number of multiprocessors in the GPU.

On compute capability 2.1 devices, a multiprocessor has two dual-issue warp schedulers, while on 3.0 devices each multiprocessor has four dual-issue warp schedulers. Finding a general formula for these architectures is more complicated, since instructions can be executed in parallel on the same multiprocessor; however, we can compute an upper bound considering the properties mentioned in the previous section. The throughput of the different instructions is only limited by the fact that some instructions can be executed on only one group of cores on the same multiprocessor.

On 2.1 devices there are three groups of cores; additions and logical instructions can be executed on any group of cores on the multiprocessor. Therefore, we may expect that an MD5 lookup kernel, having three times additions/logical operations as shifts/MAD operations, should evenly occupy all the groups of cores. Hence, we can estimate the theoretical throughput treating all types of instructions equally, as $X_{2.1} = \frac{X_{ADD/LOP} \cdot MP\_count}{N_{SHM} + N_{ADD} + N_{LOP}}$.

On the other hand, the SHA1 function has only one and a half as many addition/logical instructions than shift/MAD instructions; thus, one group of cores would be always busy on shift/MADD operations, while the other two would have some idle cycles. The theoretical throughput would be $X_{2.1} = \frac{X_{SHM} \cdot MP\_count}{N_{SHM}}$.

However, in actual practice there is not enough instruction level parallelism in either algorithm, so that the bottleneck is given in both cases by addition/logical instructions.

On 3.0 devices shifts and MAD operations are sufficient to overlap completely other operations; therefore, $X_{3.0} = \frac{X_{SHM} * MP\_count}{N_{SHM}}$. We expect a similar result to hold in actual

measurements, since 4 groups should be enough to run in parallel shift/MAD operations and addition/logical operations for both MD5 and SHA1.

TABLE VIII. THROUGHPUT ON SINGLE GPU

	8600M	8800	540M	550ti	660
MD5 (theoretical, MKey/s)	83	568	359.4	962.7	1851
MD5 (our approach, MKey/s)	71	480	214	654	1841
MD5 (BarsWF, MKey/s)	71	490	205	560	1340
MD5 (Cryptohaze, MKey/s)	49.4	316	146	410	1280
SHA1 (theoretical, MKey/s)	25	170	128	345	390
SHA1 (our approach, MKey/s)	22	137	92	310	390
SHA1 (Cryptohaze, MKey/s)	20.8	132	68	185	377

TABLE IX. THROUGHPUT ON WHOLE NETWORK

	theoretical (MKey/s)	our approach (MKey/s)	efficiency
MD5	3824.1	3258.4	0.852
SHA1	1058	950.1	0.898

By observing the results in Table VIII, we can see how the actual performance of the devices with compute capability 1.* (8600M and 8800) are near to the expected throughput with few differences. One reason for these differences is the lack of ILP that prevents the SFU (Special Function Unit) to be used to execute additions (thus the additions throughput per multiprocessor falls from 10 instructions/cycle to 8 instructions/cycle). On Fermi architecture (540M and 550Ti), since we do not exploit the maximum throughput of ADDs and logical operations (which form the bottleneck) for the lack of ILP, the actual performance is quite far from the theoretical one. On the other hand, on the Kepler architecture (660) we achieve roughly the maximum expected efficiency, that is 99.46%.

The same table also shows how our code can provide similar, or even better, performances than well known tools, such as BarsWF and Cryptohaze Multiforce, that aim to maximize the performance. For example, on the Kepler architecture BarsWF and CryptoHaze Multiforce achieve 72.39% and 69.15% of the theoretical throughput, respectively.

The results of the full network reported in Table IX show an actual overall throughput that is roughly equal to the sum of the throughputs of the single devices, thus showing an almost perfect parallelism achieved by our system.

VII. CONCLUSIONS

In this paper we have proposed a parallelizing pattern for exhaustive searches in a hierarchical and heterogeneous environment and we have proved its effectiveness by building a cracking system for MD5 and SHA1 hashcodes that works on a infrastructure composed of multiple nodes having different GPU devices. Furthermore, we have shown how to identify the performance bottleneck on a GPU implementation that is limited by arithmetic and logical instructions and how to optimize it.

In future work, we plan to apply the proposed parallelization pattern for exhaustive searches to other architectures, including multicore CPUs, manycore systems, and GPUs from manufacturers other than NVIDIA. Our major goal would be to obtain a high efficiency even on clusters of greater complexity, size, and heterogeneity. To achieve this, we would need to investigate better failure models that would provide a smart way to reconfigure the cluster topology when a subset of dispatching nodes becomes inactive.

REFERENCES

[1] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2008. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[2] S. Marechal, "Advances in password cracking," *Journal in Computer Virology*, vol. 4, no. 1, pp. 73–81, 2008.

[3] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proc. of 12th ACM Conf. on Computer and Communications Security*, ser. CCS '05. ACM, 2005, pp. 364–372.

[4] S. Mikhail, "World fastest MD5 cracker BarsWF." [Online]. Available: http://3.14.by/en/md5

[5] J. A. Dev, "Cracking MD5 hashes by simultaneous usage of multiple GPUs and CPUs over multiple machines in a network," in *Proc. of 2nd Int'l Conf. on Advances in Electronics, Electrical and Computer Engineering*, ser. EEC 2013, 2013, pp. 383–387.

[6] J. Zou, D.-D. Lin, and G.-C. Mi, "A universal distributed model for password cracking," in *Proc. of 2011 Int'l Conf. on Machine Learning and Cybernetics*, ser. ICMLC 2011, vol. 3, 2011, pp. 955–960.

[7] A.-D. Vu, J.-I. Han, H.-A. Nguyen, Y.-M. Kim, and E.-J. Im, "A homogeneous parallel brute force cracking algorithm on the GPU," in *Proc. of 2011 Int'l Conf. on ICT Convergence*, ser. ICTC 2011, 2011, pp. 561–564.

[8] R. L. Rivest, "The MD5 message-digest algorithm," Internet RFC 1321, United States, Apr. 1992. [Online]. Available: http://tools.ietf.org/html/rfc1321

[9] D. Eastlake, 3rd and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," United States, Sep. 2001, rFC 3174.

[10] D. Barbieri, V. Cardellini, and S. Filippone, "Generalized GEMM kernels on GPGPUs: experiments and applications," in *Parallel Computing: from Multicores and GPU's to Petascale*, ser. Advances in Parallel Computing. IOS Press, Apr. 2010, pp. 307–314.

[11] ——, "Sparse computations on GPGPUs," DISP, Univ. Roma Tor Vergata, Tech. Rep. RR-12.90, Jan. 2012. [Online]. Available: http://art.torvergata.it/handle/2108/76470

[12] ——, "Fast uniform grid construction on GPGPUs using atomic operations," in *Proc. of Int'l Conf. on Parallel Computing*, ser. ParCo 2013, Sep. 2013.

[13] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conf.*, ser. GTC 2010, 2010. [Online]. Available: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

[14] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, July 2013.

[15] ——, *PTX: Parallel Thread Execution ISA Version 3.2*, July 2013.

[16] "CUDA Profiling Tools Interface." [Online]. Available: https://developer.nvidia.com/cuda-profiling-tools-interface

[17] "Cryptohaze Multiforcer." [Online]. Available: http://www.cryptohaze.com/multiforcer.php