Parallel implementations of MD6, a modern cryptographic hash function

Arnaud Bienner, Benoit Dequidt
arnaud.bienner@gmail.com, benoit.dequidt@gmail.com

June 18, 2009

Contents

Abstract

This paper present differents ways to implement parallel versions of the MD6 cryptographic hash function. MD6 is one of the several candidates for the SHA-3 competition, which aim is to find a new hash function to replace the old SHA-2, vulnerable to modern attacks. We tried to see how a cryptographic hash function like MD6 could be efficiently parrelizable, to take the most of the new parallel architectures that are in our computers nowadays (multicores/processors, GPGPU).

1 Introduction and context

Cryptographic hash functions: Information security is essential on cumputers. This buissiness grows a lot due to digital signature and mostly because of the Internet with its huge data flow exchange on it. Cryptographic hash functions can be used to check message integrity. Indeed, you have to be able to detect data corruption . Moreover, they are also used for password verification without saving the cleartext password [1]

SHA-3: For years, MD5 was used as the standard cryptographic hash function. Because of the discovery of new attacks and of the increase of computing power (which made old unefficient attacks now possible to perform), SHA-1 has been developped.
SHA-2 family, a SHA-1 variant, that contains SHA-224, SHA-256, SHA-384 and SHA-512, has been developped too.

Recently, new attacks have been performed against SHA-1. As these attacks are difficult to perform in reality (because they needs lot of computing), as they still need less operations that an exhaustive research, it is enough to say that SHA-1 is not yet cryptographic reliable. Because SHA-2 is based on SHA-1, maybe it could be concern by theses attacks.

Consequently, a competition has been launched by the NIST to suggest a new cryptographic hash function, that will be called **SHA-3**.

One of the most interisting candidate is **MD6**, that we'll study in this article.

Parallelism: For years, there was a contest to CPU frequency with a huge growth of computation abilities. But now, we reached physical limits that bound this evolution.

So now, parallels architectures are under development because they increase the computing power, by adding several cores or processors, instead of increasing the performance of an unique processor.

Because of the video games industry growth, graphic cards have been improved, and particulary their graphic processors unit (GPU). They are made for highly parallel computing applications, like graphic processing, so they get a lot ALU.
Today, it's possible to use those GPU to make computation that are not necessarily closed to graphic computing, with tools like CUDA (see 4).

[1] Password are usually not stored in cleartext for obvious reason but instead in digest form (hash code).

2 MD6

2.1 Presentation

MD6 is one the twenty favorites hash funcion for the SHA-3 competition.

It has been developped by Ronald L. Rivest, a american cryptographer who developed MD5 and participate to the development of RSA, with Shamir and Adleman.

In this section, we'll try to understand how MD6 works, to find some source of parallelism.

We will not smeak about security proofs, that are not the subject of this article, and that have been demonstrated in [1] and [2].

2.2 Principle

MD6 take as parameter a message of length less than 2^{64} bits. It computes a digest of d bits where $0 < d \leqslant 512$ bits.

d defaimt value is 256 bits, but it could change.

Moreover, lot of MD6's parameters, that have default value, can be changed:

Key K : Default is null (with length 0). Could be use for salting.

Level L : As describe in 2.4, MD6 use an quaternary Merkle tree. His height could be specified with L, which is 64 default (that means fully tree based). If L is equal to 0, then compression is made sequentially, with a Merkle-Damgård construction, that give the same result. If L is less than 64, MD6

uses a hybrid mode: first tree-based, from 0 to L, then sequential.

Number of rounds r : default is $r = 40 + \lfloor d/4 \rfloor$, so when default d value (512), $r = 104$.

Other parameters : constantes used in compression function (like Q or the ti) could also be changed.

2.3 Mode of operation

The mode of operation is based on a quaternary Merkle tree.

This mode is interisting for us, because, as a tree, it could be easily parallelizable. In fact, each tree node can be computed in parallel.

The unit measure is the **word**, that is 8 bytes, or 64 bits.

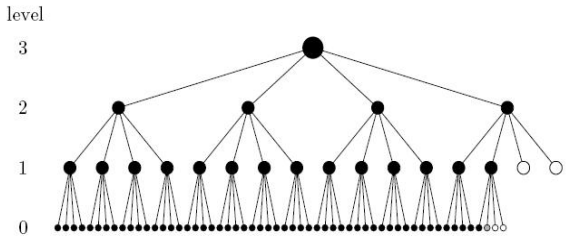

Figure 1: MD6 Merkle tree

Tree's leaves are data from the file we want to hash. They could be padded with 0, to get leaves with a 16 words size (equivalent to 128 bytes or 1024 bits). Moreover, each node should have 4 children, so fictive nodes padded with 0 have to be created if necessary. Each block, composed by 4 nodes, is compressed (using the compression function describe in 2.4) to obtain a 16 words node.

In this way, we go up into the hash tree and we stop when we have only one node: the root.

Digest is the truncated value of the tree's root (last d bits, so last 256)

2.4 Compression function

The compression function has a vector of 89 words in entry, represented by the figure **??** and composed by :

Q a vector of 15 words, equals to the fractional part of $\sqrt{6}$

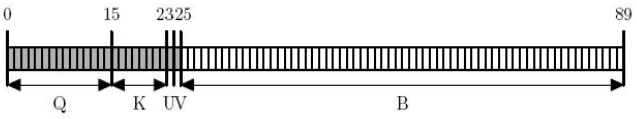

Figure 2: Entry data for compression function

K 8 words of the key (or 0 if there isn't a key)

U 1 word, which point out the block position, we give details in figure3

V 1 word, we give details in figure 4

B 1 bloc of 64 words of data, which match for 4 nodes of 16 words.

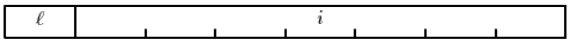

Figure 3: Unique word of id U, auxiliary entry of compression function composed of a byte for the level in the tree and 7 bytes for the position in this level

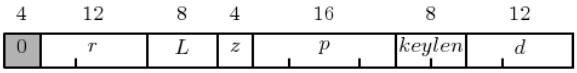

Figure 4: Word of control V, auxiliary entry for the compression function. r is the number of rounds, L the maximun height of the tree, z equals to 1 for the last compression (which one leads to the root) 0 otherwise, p is the number of 0 added (*padding*), $keylen$ the key's size and d the digest's size

The compression function does afterwards r rounds (104 by default), each composed of **16 independents loops which may be parallelizable**. Each of those 16 loops does around 15 logical operations or shift bits[2].

Each round compute 16 words, from 89 words computed before (the 89 first values are the vector of the figure 2 gave in entry for the compression function).

The last 64 words computed are the outputs of the compression function.

[2]logical operations are prefer to arithmetic operations and branch dependent operation to avoid side channel attacks

2.5 Reference implementation

The reference implementation, aviable on the competition website (`http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`) is based on radix's lists, which insure a memory cost in $\log_4 n$.
Each tree level is filled with 4 blocks, when a level is full, we clear it by compressiong the 4 blocks and filled the next level up.

This version is not straight parallelizable, so we decided to do our own MD6's implementation, which compress data level by level. We have a cost in memory in $O(n)$ because at the begining we to store the whole message. We used 2 arrays : one for the working level, and another 4 times smaller for the next level up filled by the blocks' compression results. When the next level up is full, it become the working level and we allocate a new empty array 4 times smaller which become the next level up. We only have reimplemented the operation mode, in order to be able to parallelized is easily. The code, spend most of is time in the compression function (not moddified), our code is as efficient as R.L Rivest's one. On our test machine (which feature are given in **??**) we got a rate average of **27,5 MB/s** (by hashing files of different size).

3 Parallel implementation on CPU using the Intel TBB

3.1 Presentation of the Intel TBB

Threading Building Blocks (TBB) is a librairy in C++ made by intel in order to write programs which benefit of multiprocessors/multicores architectures [3]. It's a librairy packaging a set of data strucutes and algorithm to enable programmers to write their code without thinking of the threads managing like in POSIX threads, Windows threads or Boost Threads, where you have to manage the creation, the sinchronization and the ending of each thread. In the librairy computation are considered as tasks which are automatically distributed on the whole avialable ressources, using efficiently by the same time caches

3.2 Implementation

We choosed to look down our tree by width, from left to right. Thus for each level, we compute in parallel all the blocks of words, the TBB managing the distribution of computations on the differents processor/cores.
We measure on a 850 MB file, the impact of the numbers of processor/cores on the hashing rate. Here are the results on the machine Idkoiff, results are likely the same on ensibm (see the part **??** for details on the machines). On a dualcore, the algorithm is twice faster

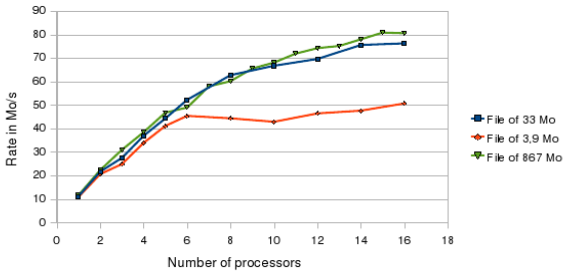

Figure 5: TBB's rate, depending on the numbers of processor/cores for three files of different size

than on one core. However, we noticed that beyond 8 processors/cores, the increase of performance is no longer linear with the number of processors/cores. It might be linked to the memory bus' saturation of the machine. The algorithm is all the same 8 times faster using 16 processors/cores. Note : To use the TBB, we used g++ 4.3.3, whereas for all the others implementations we use gcc 4.3.3. Versions compiled with g++ are 3 times slower than those compiled with gcc for some reasons that we don't understand while the compilation's options and the code are the same.

4 Implementation on GPU

Note : the rates presented in the following parts are calculated from the average hashing rate on several big files of different size. We also included all memories' copy in order to have a result near the reality and the real time felt by the user.

4.1 Cuda

4.1.1 Presentation

Cuda (*Compute Unified Device Architecture*) is a GPGPU technology (*General-Purpose Computing on Graphics Processing Units*). It enable programmers to write program using their graphic processor (GPU) to execute computation usually executed by the central processor (CPU). Cuda is an extension of C language, so it is easy to use cuda's code in others C or C++ programs. Other extension for Python, Fortran and Java are also available. All the NVIDIA's recent graphics card (GeForce series 8,9, Teslas and some Quadro and coming cards) match with cuda.

4.1.2 Special feature of programming

To program in a efficient way on cuda, you have to understand how a NVIDIA graphic card works. A very good and detailed documentation [4] on this subject is provided by NVIDIA.

Graphic cards are composed of several multiprocessors, which shared a same memory and can operate in parallel. Those multiprocessors are composed of dozen of cores which can as well operate in parallel.

Moreover, the graphic card have a global memory for all the processors, but very slow and a constant memory also very slow but this one be layed in the cache. De plus, la carte possède une mémoire globale à tout les processeurs, mais très lente d'accès, et une mémoire constante, lente mais qui peut être mise en cache.

For the API, le main program is called the kernel. This kernel must be called by a function executed by the central processor (CPU). When you call the kernel you need to specify on how many blocks this kernel should be executed and how many threads you have in each block. The execution of blocks on the graphic card are completely independent and are computed on differents multiprocessors. Threads in the same block are executed on the same multiprocessor et can share data (stored in the shared memory). Furthermore, they can be synchronised using a border of synchronization. Each thread have a data strucute, where you can find its number and the number of the block where it is executed. This structure enable the programmer to have a scalable execution.

The kernel can call auxiliary functions which will be replaced (*inline*) in the final code. Complex operations such as memory allocation, memory copies and so on need to be compute by the central processor. Chaque thread possède une structure de données lui permettant de savoir son

In conclusion, we notice that the way to program with CUDA is very special, it's thought for parallel computation and scalable algorithm. And the result depends a lot on the problem and how it is implemented.

4.2 Each block in parallel

In our first CUDA version, we compress each block in parallel, each multiprocessor have to compute 4 nodes into 1, it's likely the same version as the TBB one (see 3.2). Each block in composed by one thread who compress a block of word (MD6 signification). The average rate is about **2,7 MB/s**, so it's 10 times slower than the CPU version...

It's linked with the fact that we don't take advantage of the available parallelism.

4.3 Parallelization of the compression function

The compression of a block of words is composed by independent steps, especially the 16 rounds of loop executed during a round of compression, which is the part of the cost where the program spend the most of its time.

We parallelized those 16 rounds of loop, expecting increase meaningful performances of the program. In order to have the most possible parallel version, we also parallelized all the steps of the algorithm which can be done in parallel, including memory copies in arrays at the begining and at the end of the compression.

We now have a rate of **28 MB/s** (10 times more than the previous version). But, this version isn't faster than the CPU one. Moreover, for small files, time spent for memory copies on the graphic card isn't soften. It seems to be linked to the fact that the compression function doesn't do a lot of computation compared to the memory access. In order to check it, we moddified the compression function to do 160 rounds of loop instead of 16. Thus, the GPU execution time for this version is 10 times faster than the identical version for CPU. Thus in order to increase again performances, we need to restrict access to memory and also increase the number of computation in each block.

4.4 Increase of the number of threads per block

Of course we can't increase the number of computation done by the compression function. To increase the number of threads per block, we decided to compress in each graphic block several block of words in parallel. The problem is that the compression function use an array and use each case only once. We only need an array of 89 words to do the compression function. We never need more than 89 words for a block so now we have an array of $89 \times N$ where N is the number of blocks of words compressed by a graphic block instead of a huge array of $89 + 16 * r$. In facts with many threads compressing many blocks of words, we need to have several working arrays but the shared memory is limited and we can't stored more than one array. We can't either stored those arrays in the global memory for obvious performances reasons.Furthermore a too big array reduce performances because each case is only reached once the it don't use cache system. Thus, we decided to use N circular arrays as working arrays for N blocks of words. Now we can increase the number of threads per blocks.

4.5 Using circular array

The circular array, has a size of 89. In order to use this array, we moddified our implementation to compute each index of the array modulo 89. First we though it would be better to use a circular array of 128 words, in order to use logical computation (with a mask of bits) to compute indexs, instead of the modulo arithmetics computation which cost a lot. But arrays of 128 words where to big to be stored in cache so performances where quite bad.

So we decided to use circular array of 89 words. We use 20×16 threads per blocks instead of 16 with the old version. It's the limit beyond we can't have another new circular array in shared memory. The rate is yet **80 MB/s**. However, the time compuation could be reduce tabulating the values of modulo operation, by storing them in arrays and avoid those arithmetics compuations which cost a lot.

4.6 Tabulation des valeurs pour l'accès au tableau circulaire

We tabulate all the values modulo 89 needed. On tabule toutes les valeurs modulo 89 nécessaires, c'est à dire celles qui pourront être utilisées (de 0 à $r \times 89$). This array is created and copied into the constant memory of the graphic cards. Thus the cost of the computation of an index is replaced by a memory access. Data into constant memory can be cached and a large numbers of threads reached those data, the benefis is important. We now have a rate of **105 MB/s** with this version.

5 Experimentations and results

5.1 Tests machines

We used three tests machines :

- A personal machine with an Intel Core2Duo E6420 @3.1Ghz, a 8800Gts, Linux 2.6.28, gcc 4.3.3
- Idkoiff, a compuation machine of INRIA with eight Dual core AMD Opteron @2.2Ghz, two gtx280, Linux 2.6.26 gcc 4.3.3
- Ensibm a server of Ensimag composed of 16 IBM Power5+ @ 1.5Ghz Linux 2.6.9, g++ 3.4

On the two first machine, we tested all our implementations : basic CPU, CPU with TBB, and the differents CUDA versions. On ensibm we only runed our CPU versions.

5.2 Tests

For each implementation, we measured execution time with and without memory allocation and memory copies, because those costs depends a lot on the machine. Indeed on our personal machine the copy rate is 2200 MB/s versus 700 MB/s on Idkoiff. We measured the hashing rate for CPU and GPU on a set of file with differents length, between 2 bytes and the memory saturation of the graphic card : 60 MB for the 8800Gts, and more than 350MB for the gtx280. On our personal machine we can consired memory cpies which are quite light. Our results are presented on figure 6. GPU versions are much faster than the CPU version for big files, we can notice that costs of memory allocations and memory copies are soften with the length of the file. There is a factor about 5 between the maximal GPU hashing rate and the CPU one : 150 MB/s for the GPU versus 30MB/s for the CPU. Extra memory costs due to copies between central memory and the card memory are soften with the file length.But we couldn't test it on

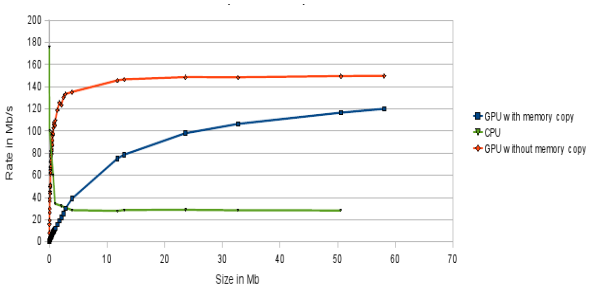

Figure 6: Comparison between CPU version (green),GPU version with memory copies (blue) and without (red)

bigger files because we were limited by the card memory of our 8800Gts. For bigger files we should divided it in a multiple of 4 smaller files and use our algorithm on those 4 smaller files.

We also done the same tests on Idkoiff which got more powerfull CPU and more powerfull graphics cards (in our tests we only use one card).

But on this machine memory costs are very high due to a thin rate (it might be linked to the machine's architecture). Results of those tests are on figure 7 We

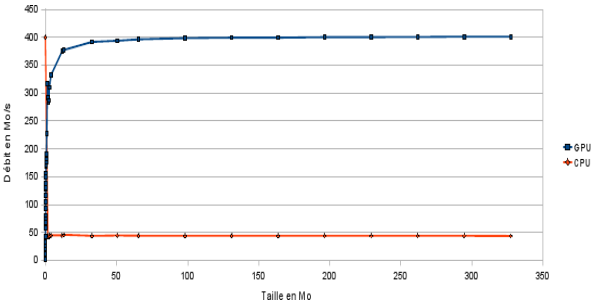

Figure 7: Comparison between GPU version (blue) and CPU version (rouge) on Idkoiff

reached a hashing rate of **400 MB/s** on big files with a gtx280 versus **42 MB/s** on an opteron. The rate reached for the gtx280 is almost 10 times bigger than the opeteron rate, more than 13 times bigger than the Core2Duo one and more than 2.5 times bigger than our 8800Gts rate.

6 Conclusion

MD6 is suitable for parallel implementations on GPU and on multicores/processors.

On CPU, with Intel TBB library, performances increase lineary, while memory access doesn't overcharge computer bus.

Moreover, no much modifications are needed to implement TBB version from the sequential version.

CUDA implementation is more difficult to make, because it's a new technology, and because its suppose to know very well NVIDIA graphic cards architecture, which are different from a CPU architecture.

Also, CUDA has number of limitations (including no recursive call) that make the CUDA implementation more difficult to make.

When programming CUDA application, we must keep in mind some special features. Particulary, because memory copy costs are high, CUDA musn't be use for small data flow.

So it should be interisting to have many versions, and to execute the one which is well suited for the data quantity to hash. The best solution should be to use an adaptative algorithm which will use at best available computing ressources.

Despite these inconvenients, GPU parallel implementation is more efficient than CPU implementation (as soon as file are enough big), despite of a longer development time.

It's interisting to see that GPU are less expensive that dozens of processors, but can be more efficient as soon as computation requirement is important.

Moreover, a GPU consumes less energy than a CPU.

About MD6, we see that even if GPU implementation doesn't use all the powerness of the graphic card, the performances obtained are very good.

In conclusion, we can say that MD6 is very well suited to use parallelism of present and future architecture. This make MD6 a modern cryptographic hash function. Also, we see that GPU powerness, when correctly exploited, can achieve to get very performant implementations of parallelizable algorithms.

References

[1] R. L. Rivest, "The md6 hash function, a proposal to nist for sha-3," October 2008. http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf

[2] C. Y. Crutchfield, "Security proofs for the md6 hash function mode of operation," Master Thesis, Massachusetts Institute of Technology, June 2008. http://groups.csail.mit.edu/cis/md6/docs/2008-06-crutchfield_ms_thesis.pdf

[3] *Intel Threading Building Blocks Reference Manual*, Intel Corporation, 2009. http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference%20(Open%20Source).pdf

[4] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, NVIDIA Corporation, November 2007. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf