OpenCL cryptographic computations for FAB coin

Todor Milev[*]
todor@fa.biz

June 4, 2018

1 Introduction

Public/private key cryptography is arguably the most important aspect of modern crypto-currency systems. The somewhat slow execution of private/public key cryptography algorithms appears to be one of the main bottlenecks of FAB's Kanban system.

In the present text we describe our port of the cryptographic functions that make up the computational core of fabcoin to the GPU programming language OpenCL. Whether this port is to be used in the final version of Kanban - as opposed to using directly the original fabcoin/bitcoin cryptopgrahic library from which the port was derived - remains to be decided.

2 Port of secp256k1 to openCL

The primary result of our investigation is the porting of the fabcoin core/bitcoin core cryptographic library **secp256k1** from C/C++ to openCL. We took special care to keep our code as valid C/C++. I do not believe there has been any performance penalty introduced by the port, however that can be tested in the future.

1. All calls to malloc/free were refactored to a custom memory pool.

2. Inputs and outputs were pipelined into single memory buffers.

3. In this way, our ported code allows lock-less parallelization in both OpenCL (where that is enforced by design) and in C++, where that is the simplest and most reliable way to achieve parallelization.

   To the reader familiar with C++ threads, the parallelization model of OpenCL consist in providing the equivalent of thread creation achieved by a so called kernel queue and the equivalent of thread join.

4. We provided both an openCL and a std::thread benchmark for our code.

5. Should we decide to focus on C++ and abandon openCL, the std::thread C++ parallelization can remain in use.

6. We implemented a TCP server to handle parallelization of a one-channel pipeline of cryptographic computation requests to openCL. The server requires further testing for reliability.

   It remains to be decided whether that design will be retained.

   The design has the advantage that it should allow for easy implementation of asynchronous calls. Should the computational engine hang due to a programming mistake, this design would allow the server's driver to restart it without any noticeable downtime.

7. For the purposes of our tests, we connected our TCP server to a node.js server for handling of networking.

   It remains to be decided whether the node.js code will remain in use.

   The advantages of the node.js server are that it provides a secure well-tested out-of-the box platform for asynchronous message based networking. Node.js's main limitations are that it has restrictions on RAM memory use, however those can be solved by moving all large RAM allocations to the computational engine.

[*]FA Enterprise System

8. Unlike our benchmarks where all code runs using both openCL std::thread, we have yet to implement std::thread parallelization in our TCP server. In view of the other properties of our setup, this should be achieved quite easily.

   We would like to note that running both C++ and openCL in the same executable should incur neither run-time overhead nor boot time slow down.

We based our port off of Pieter Wuille's C project libsecp256k1 [7], a development branch of the bitcoin core crypto library. According to his github profile, Pieter Wuille appears to be the *de facto* lead developer of bitcoin's ECDSA implementation. We chose the development branch as it is also the base of the top-google-search (partial) OpenCL port [6]. We did not choose to use [6] directly as the port appeared to not be self-contained and contained only an implementation of the signature verification function 4) from Section 2.3.

2.1 OpenCL findings

Our investigation of the feasibility of running cryptography using OpenCL, and using the GPU in general, remain inconclusive. Our initial findings are as follows.

- The performance of openCL vs C++ on CPUs varies enormously depending on the specifics of the software task. For example, C++ appears to outperform openCL 6 to 9 times in signature verification, whereas, in the opposite direction, OpenCL outperforms C++ by a factor of 6 to 8 on double-hash SHA256 computations (mining).

  All aforementioned benchmarks were carried on the same devices, running the same code except for different parallelizations (openCL vs C++) and different respective compilers.

- In our implementation, CPUs outperform GPUs on openCL on all tasks, including mining. Since this information does not match the experience of other users, this must be due to the specifics of our openCL port. Clearly there is a lot of room for improvement in this direction.

- We saw very little difficulty porting the SHA256/mining code (the initial port happened in less than a day). However, our crypto library took more than 3 weeks to port. Even so, our current cryptographic library port fails to build on 1 out of 4 GPUs, and runs incorrectly on 2 out of 4, with only 1 NVidia GPU running all code as expected.

  Our (non-mutually exclusive) hypotheses for the causes of this failure are the following.

  – GPUs have very small stack sizes (between 32kb and 64kb). A large program such as the cryptographic library may trigger those limits. This is not something that would easily be diagnozed in a small program such as a SHA256 miner.
  – Yet-to-be-diagnosed race conditions, possibly related to overflowing stacks (see the previous).
  – Issues with the OpenCL drivers.

  Except for drivers and hardware support, it is my belief that all of these issues can be investigated and reliably fixed, however that will require additional development time.

- Performance is extremely sensitive to the software task. Functions that access a lot of RAM memory, such as signature verification, run much faster under C++. Functions that access a very limited range of memory addresses, such as SHA256, run much faster under openCL, even on the same device.

  Clearly the location of memory addresses and the amount of RAM memory used depends on the algorithms used, so performance appears to be much more a function of software than of hardware. Our implementation of signature verification, `secp256k1_ecdsa_sig_verify` (same as in fabcoin core/bitcoin) is optimized to use a lot of RAM memory but not as much computational power. It does so by storing points on the elliptic curve secp256k1 using their WNAF representation (windowed non-adjacent form). This method is quite inefficient memory-wise and can be traded off for a much smaller RAM memory use, but much larger processor use. Whether that would result in performance gains on specific devices remains to be investigated. Further investigation of the matter will require additional development time.

- On the positive side, online sources such as Nvidia's guide [5] suggest that custom-tailored software optimizations may (or may not) yield between 4- and 10- fold speed optimizations on Nvidia hardware, depending on the particular device and program type, see [5, page 22]. As our code is already heavily optimized for CPUs, it remains feasible that a GPU may outperform a CPU of a similar class, should a considerable effort be put into software optimizations.

  Even without such optimizations, OpenCL remains a reasonable option as it allows the same code to run on both the CPU and on multiple GPUs (without the need of much engineering), effectively adding the speeds of all involved devices.

- Since our openCL code runs also C++, all common errors are caught promptly. However, hunting for hardware specific errors - for example, stack overflow due to GPU limitations of otherwise correct code - is greatly hampered by very long build times.

2.2 OpenCL vs CUDA

We also considered CUDA (an Nvidia proprietary language), but - for the time being - chose to use OpenCL instead as that allows the use of AMD GPUs, Intel system-on-chip GPUs and, last but not least, the Intel/AMD CPUs running the host system. In addition, the use of OpenCL should allow for future ports to the upcoming Intel GPUs [2].

While our primary reason for choosing OpenCL over CUDA was hardware portability and the ability to run code on the CPU, our decision was partially supported by the ambiguous studies of the performance of OpenCL vs CUDA. For example, [3] reports between 13% and 63% slower performance of OpenCL vs CUDA, however the more recent [4] reports a slightly better performance of OpenCL vs CUDA. These reports do appear to depend heavily on the particular benchmark used. Even assuming the most pessimistic estimate of 63% slower performance of OpenCL, the ability to run our software on the CPU and non-Nvidia hardware appears to be a reasonable trade-off.

We must mention that, in support of CUDA over OpenCL, Jason Hong and Sam Gong have informally reported much better performance on SHA256 computations when using CUDA - up to 7-fold increase of performance. The huge differences in speed reported internally may be a function of CUDA-specific code optimizations or a function of the specifics of SHA256 computations. From our experience with OpenCL, we know that performance is greatly affected by the RAM memory use and access patterns. For example, our performance metrics of openCL vs C++ swung up to 8 fold in both directions, for a total of up to 64 fold performance difference (2 orders of magnitude) on the same device. I would not be surprised if similar phenomena manifest themselves while programming using CUDA. Should we choose to use CUDA, whether the performance of the other cryptographic functions will be closer to Jason's experience or to the benchmarks reported in [3], [4] remains to be investigated.

Our choice of OpenCL has not been finalized yet; whether we switch to CUDA or to another parallel computation framework (such as openMP, openACC, or simply using std::thread) remains to be decided.

2.3 The four main cryptographic functions

The four main cryptographic functions used to run FAB coin (and all other major crypto-currencies, powered by various algorithms) are as follows.

1. Secure hashing function.
2. Private/public key pair generation.
3. Cryptographic signature.
4. Cryptographic signature verification.

1) Among many secure hashing functions, FAB uses SHA256 (secure hash algorithm 2, 256 bit). The same algorithm is also used in bitcoin. The secure hash is used to digest messages into 256 bit = 32 byte lengths. SHA256 has the property that, given a message and its digest, it is practically impossible to forge another message that has the same digest. In this way, SHA256 can be reliably used to identify data.
2) Private keys are kept secret and are used to sign data - most importantly, transactions/transfers of funds. The public key is a function of the private key that is published openly alongside a message that may or may not be signed with the private key.

Private/public key pairs have the property that, given a message, a signature and a public key, everyone can verify that the signature was correctly generated with an unknown private key that matches the known public key, but no one can guess the private key from which the signature and the public key were generated. Furthermore, given a message, no one can generate a signature matching a given public key without knowing the secret private key that corresponds to it.

The main use case of private/public key pairs is as follows. A message - say, a statement of transfer of funds - is published in the open. To consent with the transfer, the original owner of the funds produces the cryptographic signature of the transfer message. Since the original owner's public key is published in the open and is therefore known, everyone can verify the correctness of this transfer, but only the original owner can generate it.
3) Cryptographic signatures are used to sign messages using private keys as described in 2). Among many cryptographic schemes for signing messages, FAB coin uses the industry standard ECDSA (Elliptic Curve Digital Signature Algorithm) over the elliptic curve **secp256k1**, see Section 2.4. This is the same algorithm stack as the one used in bitcoin.

The cryptographic signature is a function of the message being signed, the private key, and, in the particular case of (EC)DSA, a one-time use random number.
4) The signature verification algorithm is used to verify signatures generated as described in 2). The signature verification is a function of the signed message, the signature, and the public key of the signature owner.

2.4 A brief description of ECDSA

Following Bitcoin, FAB coin uses the standard public/private key cryptography ECDSA over **secp256k1**. Here, ECDSA stands for Elliptic Curve Digital Signature Algorithm and **secp256k1** stands for the elliptic curve:

$$y^2 = x^3 + 7$$

(we do not specify the base point here), over the finite field:

$$\mathbb{Z}/p\mathbb{Z},$$

where

$$p = 2^{256} - 2^{32} - 977. \quad (1)$$

2.5 Benchmarks

Speed. All tabulated devices ran the same code. With the exception of a small header file (27 lines for OpenCL and 67 lines for C++), the code is both valid OpenCL and valid C/C++. The total code size is, at the time of writing, 7153 lines of code (= 5049 actual lines + 1294 comments + 810 blank) in a total of 22 files.

Name	Description	Function	non-stack RAM/comp. unit
mine SHA256$^{\circ 2}$	vary input of SHA256(SHA256(x)), fetch best	`sha256_twice_GPU_fetch_best`	read: 32 bytes write: 32 bytes
SHA256	hash function used in fabcoin	`sha256GPU`	read: Message size + 4 bytes write: 32 bytes
init	initializations needed for signatures and verifications	`secp256k1_ecmult_gen_context_build`, `secp256k1_ecmult_context_build`	read: none write: (3.7 + 6) MB
Pub. key	generate public key	`secp256k1_ecmult_gen`	read: 3.7 MB [1] write: 76 bytes
Sign	Sign a message securely, hardened	`secp256k1_opencl_sign`	read: 6MB[2] write: 76 bytes
Verify	Verify a signature of a message against a public key	`secp256k1_ecdsa_sig_verify`	read: 6 MB write: 230 KB

Label	Description
NI	not implemented yet.
ND	did not run yet due to colliding drivers; older versions may have ran correctly
NC	Compiles builds and runs without errors but produces incorrect results
NB	Does not build

Device	run-time	mine SHA256$^{\circ 2}$	SHA256	Pub. key	Sign	Verify
Nvidia Quadro K2000 (2 computational units)	OpenCL 1.1	21,260/sec	19,007/sec	171/sec	121/sec	66/sec
Intel® Xeon® CPU E5-2630 @ 2.60GHz (12 cores)	OpenCL 1.2	5.98 million/sec	96,028/sec	2,063/sec	1,044/sec	639/sec
	C++ std::thread	0.99 million/sec	36,040/sec	13,937/sec	6,471/sec	5,505/sec
Intel® HD Graphics (SOC, see next) @ 1.1Ghz Intel® (23 computational units)	OpenCL 2.0	76,365/sec	55,769/sec	**NC**	**NC**	**NC**
Intel® Core™ CPU i5-8250U @ 1.6Ghz (8 cores)	OpenCL 2.0	5.55 million/sec	130,844/sec	1,765/sec	1,018/sec	533/sec
	C++ std::thread	0.73 million/sec	48,865/sec	13,643/sec	5,961/sec	4,136/sec
Nvidia GeForce GTX 1080 @1.86Ghz (20 computational units)	OpenCL 1.2	46,759/sec	34,967/sec	**NB**	**NB**	**NB**
Radeon™ RX 480 Graphics @1266 Mhz	OpenCL 1.2	**ND**	31,621/sec	**NC**	**NC**	**NC**
Intel® Core™ CPU i5-7600 @ 3.50GHz (4 cores)	OpenCL 2.0	7.89 million/sec	237,161/sec	1,497/sec	2,592/sec	856/sec
	C++ std::thread	1.01 million/sec	84,259/sec	21,719/sec	6,930/sec	5,093/sec

Compilation times. Measures the difficulty of testing code changes.

[1]Can be reduced to 230 KB

[2]Can be reduced

Device	run-time	SHA256	init	Pub. key	Sign	Verify
Nvidia Quadro K2000	OpenCL 1.1	0.5 sec	23.3 sec	3.5 sec	6.1 sec	11.5 sec
Intel® Xeon® CPU E5-2630 @ 2.60GHz (12 cores)	OpenCL 1.2	0.1 sec	379.2 sec	147.7 sec	360.3 sec	362.6 sec
Intel® HD Graphics (SOC, see next) @ 1.1Ghz Intel® (23 computational units)	OpenCL 2.0	0.3 sec	69.4 sec	2.8 sec	19.3 sec	39.9 sec
Intel® Core™ CPU i5-8250U @ 1.6Ghz (8 cores)	OpenCL 2.0	0.1 sec	417.7 sec	154.5 sec	382.6 sec	379.8 sec
Radeon™ RX 480 Graphics @1266 Mhz	OpenCL 1.2	0.2 sec	45.1 sec	1.9 sec	11.3 sec	23.9 sec
Intel® Core™ CPU i5-7600 @ 3.50GHz (4 cores)	OpenCL 2.0	0.1 sec	234.7 sec	90.8 sec	235.0 sec	232.4 sec

3 Conclusions

In view of the fact that our cryptographic library port parallelizes well on the CPU and runs on the GPU, I see no reason to rush towards potentially difficult software optimizations - for now.

As proposed by Jason Hong, I think that we should delay further development of our cryptographic functions, and instead focus on building a first version of Kanban. I think we should focus on profile our entire system before we start pursuing difficult software optimizations.

Jason has proposed to me that I clone fabcoin, deploy a network of clones on AWS (Amazom Web Services) and build a performance profile on this live system. From there, we should proceed to gradually modify the system to perform the functions of Kanban as proposed in the white paper.

I think Jason's proposal is the right way to proceed for the time being; we should revisit GPU optimizations when the main functionality of Kanban works and we have a proper benchmark to measure the main bottlenecks in the system.

3.1 Data handling in Kanban

I do not see an in-GPU database as a high priority. Instead, I propose we use C++'s RAM memory in the first version of Kanban, and then focus on upgrading to a GPU database when time permits and as dictated by our (yet to be constructed) benchmarks.

For the first version of Kanban, I would propose we store the main tables using the C++ data structure `std::unordered_map`. Constant-time lookup of a table whose rows are labeled by addresses can be achieved at the cost of about 20 bytes per stored row of information. Assuming we store account balance (say, 8 bytes) and an extra 4 bytes of information (say, number of confirmations) for every address, we can achieve constant-time address lookup in

$$\underbrace{34}_{\text{address size}} + \underbrace{8}_{\text{account ballance}} + \underbrace{4}_{\text{meta info}} + \underbrace{20}_{\text{to achieve constant time lookup}} = 56 \text{ bytes}$$

per address. According to `https://blockchain.info`, at present there are about 440,000 unique addresses. Therefore we can store the account information of all bitcoin addresses using only

$$56 \cdot 440,000 = 24640000 \approx 25 \text{ MB}.$$

Should we want to hold 100 addresses for every alive human right now, we need only

$$100 \cdot 7.3 \cdot 10^9 \cdot 56 \approx 40 \text{ TB}$$

of information. The use of $1,000$ shards brings down the memory use of each system to the completely feasible amount of 40 GB. It therefore seems reasonable to postpone the engineering of more specialized data structures to the time when FAB coin starts getting near the estimates above, as at that time I expect we will have a lot greater engineering resources available.

We should note that it is possible to store all UTXOs (unspent transaction outputs) in RAM memory for a system as large as is bitcoin at present (some 56.6 million UTXOs at the time of writing, according to `https://blockchain.info`). According to [1], 52.5 million UTXOs take about 3 GB of RAM. To achieve constant time lookup, one needs about 20 bytes per transaction, so another

$$52.5 \cdot 10^6 \cdot 20 = 1.05 \text{ GB}$$

of RAM memory. In this way, the entire bitcoin UTXO data set can be stored and looked up using a little more than 4 GB. With sharding, it should be possible to scale the storage of all UTXOs thousands at a scale thousands of times larger than bitcoin at present.

References

[1] Sergi Delgado-Segura, Cristina Prez-Sol, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomart. Analysis of the bitcoin utxo set. Cryptology ePrint Archive, Report 2017/1095, 2017. https://eprint.iacr.org/2017/1095.

[2] Jason Evangelho. Intel is developing a desktop gaming gpu to fight Nvidia, AMD, https://www.forbes.com/sites/jasonevangelho/2018/04/11/intel-is-developing-a-desktop-gaming-gpu-to-fight-nvidia-amd/#54a74d33578f. 2018.

[3] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and opencl. *CoRR*, abs/1005.2581, 2010.

[4] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kolodziej, and Christoph W. Kessler. Benchmarking opencl, openacc, openmp, and CUDA: programming productivity, performance, and energy consumption. *CoRR*, abs/1704.05316, 2017.

[5] www.nvidia.com NVIDIA Corporation. NVIDIA openCL best practices guide, version 1.0, https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[6] Author: https://github.com/hhanh00. Project secp256k1-cl, https://github.com/hhanh00/secp256k1-cl. 2014.

[7] Pieter Wuille and contributors. libsecp256k1 https://github.com/sipa/secp256k1. 2015.