Computer Science • 20(1) 2019 https://doi.org/10.7494/csci.2019.20.1.3114

Piotr Szkotak
Paweł Russek
Kazimierz Wiatr

STUDY OF OPENCL PROCESSING MODELS FOR FPGA DEVICES

Abstract *In our study, we present the results of the implementation of the SHA-512 algorithm in FPGAs. The distinguished element of our work is that we conducted the work using OpenCL for FPGA, which is a relatively new development method for reconfigurable logic. We examine loop unrolling as an OpenCL performance optimization method and compare the efficiency of the different kernel implementation types: NDRange, Single-Work Item, and SIMD kernels. In our conclusions, we compare the metrics of the created FPGA accelerator to the corresponding GPGPU solutions. Also, our paper is accompanied by a source code repository to allow the reader to follow and extend our survey.*

Keywords reconfigurable computing, accelerated computing, high-level hardware synthesis

Citation Computer Science 20(1) 2019: 85–97

1. Motivation

FPGA technology has been around for 35 years now. The idea of semiconductor devices that provide loosely organized logic blocks with connections that could be configured and reconfigured with software has evolved over time. As the technology has advanced and the transistor count of a single chip has increased (from ca. 100,000 in 1984 to 20 billion in 2018), FPGAs have grown from glue logic confined chips (through components suitable for interface and peripheral development) to high-performance custom computing eligible machines [14,18]. The cost and time of hardware development has grown, as the gap between hardware designer productivity and semiconductor capacity has constantly increased. This leads to new design methods and tools that started as schematic-based design entry tools, evolved throughout hardware description languages and platforms for IP core integration, and expanded to high-level hardware synthesis languages (HLS). It is worth noticing that all of the above-mentioned methods co-exist today. The hardware description languages (HDLs) are, in fact, the basic tools for hardware development; however, this method is tedious, time-consuming, and often demands a detailed understanding of the hardware. In contrast, HLSs speed up design time thanks to the relaxed hardware definition rules, imposed processing models, and definite data types. Typically, they share their syntax and structure with commonly used software programming languages (*e.g.,* C/C++) for practical reasons. Additionally, to keep the HLS code compatible with software compilers, hardware synthesis is directed thanks to the extra compiler directives. These directives are necessary, as no software language is rich enough to fully accommodate the designing flexibility that comes with unrestricted hardware architecture. For example, the special directive sets the pipelined execution of the loop; this is a common hardware acceleration strategy that is not necessary for software development.

Today, software development time has shrunk thanks to newer programming abstraction layers, models, and tools. One of the important trends is to produce software that is hardware agnostic; *i.e.,* the code that is portable at the program execution level (*e.g.,* Java). Meanwhile, thanks to the dissemination of multi- and many-core CPUs as well as GPGPU devices, parallel programming languages have drawn the attention of the community. Despite existing traditional models for parallel programming (like Threads, OpenMP, and MPI), new methods have been proposed as a consequence(*e.g.,* Intel Cilk, nVidia, and CUDA). On this scene, the parallel programming language that marks itself is OpenCL (Open Computing Language) [2]. Its underlying programming model is SPMD (Single Program Multiple Data), where the computing device executes many instances of a single program (called a *kernel*). The kernel's threads form the computing grid, and each thread gets a unique identifier to let the programmer distribute the data in the grid. The OpenCL standard allows us to write applications that execute kernels on heterogeneous platforms consisting of different processors or hardware accelerators like CPUs, GPGPUs, and DSPs. The broad list of supported computing platforms makes OpenCL an attractive choice for

many software coders; as a consequence, the OpenCL community continues to grow. Simultaneously, the numerous body of developers presents an opportunity for FPGA vendors to broaden the understanding of reconfigurable computing. As the result, tools like Intel FPGA SDK for OpenCL are offered as methods of FPGA development today.

In our work, Altera OpenCL SDK 16.0 (which is now Intel FPGA SDK because of Intel's acquisition of Altera) was studied. We were particularly interested in comparing the performance of the FPGA platform to other OpenCL platforms. Similar to all HLS tools, Altera OpenCL (AOCL) compilation is controlled by a rich set of program directives; this alters the resulting hardware architecture. Thus, we also wanted to examine and compare the implementation strategies offered by the AOCL compiler. Thanks to our access to the high-performance computing infrastructure of the Academic Computer Centre CYFRONET of the University of Science and Technology in Cracow [1], we conducted our experiments on an OpenCL host with the Nallatech 395 Stratix V D8 FPGA device.

Simultaneously, the work is a piece of the plan to create a password discovery system that allows for the fast elimination of weak user passwords from the ICT infrastructure. System administrators would perform brute force and dictionary password attacks to forewarn users and defeat the real threats. The successful operation of such a system depends on its throughput; therefore, an accelerated platform is favored for the purpose of its execution. Meanwhile, password hashing is an essential part of the practical password authentication procedure; as for safety, a password is not stored as plain text but rather as its corresponding number; *i.e.,* hash. In the common scheme, the password entered by the user is processed by the hash algorithm, and the result is compared to the number stored in the database. It is not possible to convert the hash back to the password for any hash algorithm; thus, the password is safe if the database is stolen. Consequently, we selected the SHA-512 algorithm to carry out our experiments.

2. SHA-512 algorithm

The SHA-512 hash function is a part of the SHA-2 standard that was first published in 2001 (and later updated) by the National Institute of Standards and Technology. The complexity of a good hash function is low enough to let the hash to be calculated without unnecessary overhead; simultaneously, it is high enough to prevent too-easy brute-force attacks. SHA-512 fulfills all of these requirements, taking original messages that are less than 2^{128}-bits long and returning a 512-bit hash. The message, padding, and message length make a message block that is split into N blocks (Fig. 1), and the 1024-bit blocks are processed in consecutive SHA-512 passes. However, in order to limit our calculations to a single algorithm pass, messages of up to 896-bit long are assumed. Thus, the message block can be processed in one SHA-512 pass. For $N = 1$, the message length is up to $1024 - 128 = 896$ bits in this work.

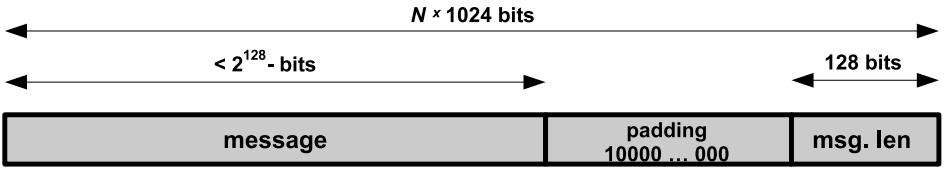

Figure 1. Structure of SHA-512 message block

A detailed description of SHA-512 can be found in [11]; for the purpose of clarity, we provide only a brief description here. The processing of a single 1024-bit block is presented in Figure 2. The algorithm's input is a 1024-bit message and 512-bit *digest*. The message is organized as 16 64-bit words (w_i; $0 \leq i \leq 15$). The *digest* is the algorithm output for the preceding message block or the value specified by the standard for the first block. In our case, the algorithm is executed only once, so the *digest* is a constant value.

The algorithm consists of 80 similar rounds. Each round runs with the unique constant K_i (defined by the standard) and value W_i (computed out of the message block by the *message expansion* function).

$$W_i = \begin{cases} w_i; & i \leq 15 \\ W_{i-16} \oplus \mathrm{RotShift}_{1-8-7}(W_{i-15}) \oplus W_{i-7} \oplus \mathrm{RotShift}_{19-61-6}(W_{i-2}); & i > 15 \end{cases},$$

where $\mathrm{RotShift}_{l-m-n}(x) = \mathrm{Ror}_l(x) \oplus \mathrm{Ror}_m(x) \oplus \mathrm{Lsr}_n(x)$, while $\mathrm{Ror}_k(x)$ is rotated x right by k bits, and $\mathrm{Lsr}_k(x)$ is shifted x right by k bits.

It is worth noting that the expansion function includes the bit-shift, rotate, and *exor* operations only. The *digest* ABCDEFGH of each iteration is calculated using the following functions:

$\mathrm{Ch(x,y,z)} = (x \text{ and } y) \oplus (\bar{x} \text{ and } z)$
$\mathrm{Ma(x,y,z)} = (x \text{ and } y) \oplus (y \text{ and } z) \oplus (z \text{ and } x)$
$\Sigma(x) = \mathrm{Ror}_{28}(x) \oplus \mathrm{Ror}_{34}(x) \oplus \mathrm{Ror}_{39}(x)$.

Please note again that the above functions use bitwise *and*, *not*, and *exor* operations as well as bit rotations only.

The dataflow and simple control of SHA-512 make the FPGA technology particularly fitting for its implementation. The FPGAs usually outperform CPUs for bitwise manipulation and logic operation, and they exploit pipelining efficiently. Additionally, the number of iterations in the outer loop is static (80 repetitions); therefore, it can be fully un-looped and pipelined to maximize the data throughput. Although pipelining consumes much hardware (each iteration stage requires separate resources) and makes sense only if a batch of data is processed (each pipeline stage executes a single element of the set at a time), it is worth a try in our case, where a big input set is expected – as in a brute force method, for example.

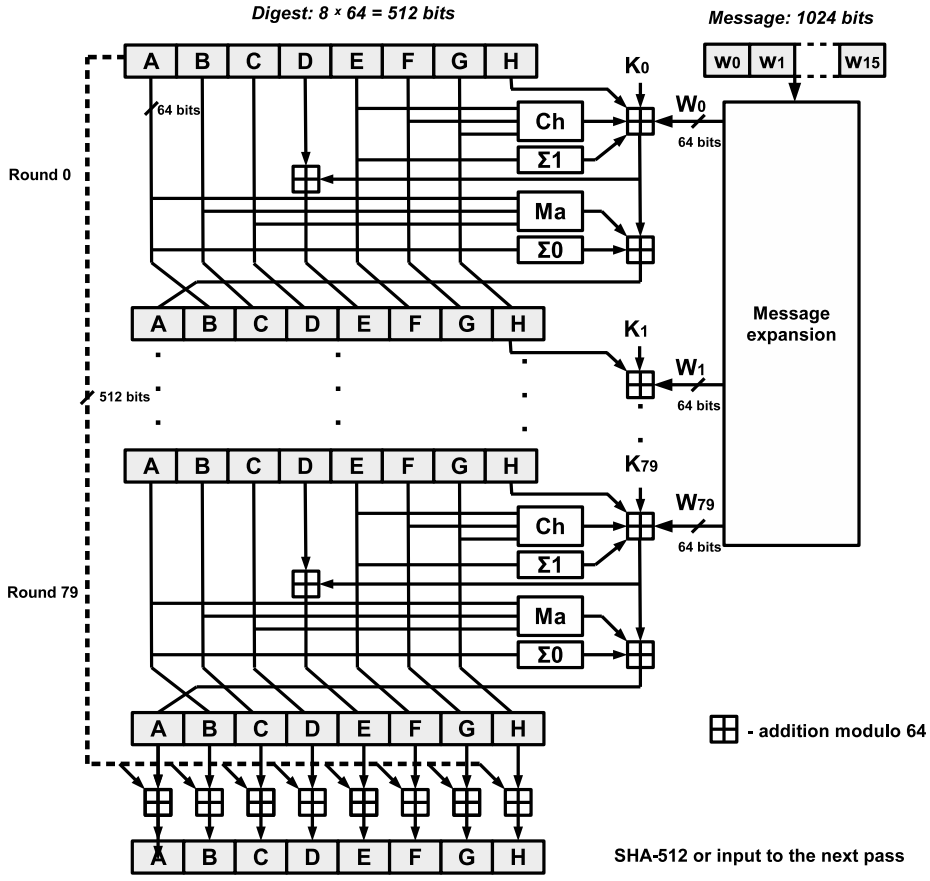

Figure 2. Dataflow of SHA-512 algorithm

3. AOCL architectures

In contrast to other OpenCL devices, FPGA devices do not have fixed architectures. The internal architecture of the processing units (*cores* are named *units* in AOCL manuals), number of units, and memory interface are hardwired for multi- and many-core CPUs, GPGPUs, and DSPs; these can be customized in the FPGA OpenCL technology.

This flexibility of FPGA allows AOCL to offer extra parametrization of the OpenCL project. The designer chooses between Single Work Item and NDRange kernels; he/she configures a number of the unit's SIMD lanes and selects the number of computing units (CU). These attributes are detailed below; a full reference for this section can be found in [4, 5].

3.1. Single work item

OpenCL is a language for massively parallel processing (MPP); thus, the number of units is expected to be high – like in the case of GPGPUs and many-core CPUs. Although OpenCL adheres to the MPP programming model, it is not restricted to MPP hardware architectures. Contrarily, many-core CPUs provide a few cores/units, while FPGA typically grants a single or couple of units – like in this study. A special case of CU when only one computing unit is invoked by AOCL is the Single Work Item (SWI) CU. The OpenCL kernels that are implemented as SWI units do not use work-item identifiers (returned by OpenCL's *get_global_id*() function, for example). Typically, the SWI kernel executes an outer loop that iterates throughout all elements of the input data set. The example SWI kernel is given in Listing 1.

Listing 1: The example SWI kernel

```
__kernel void SWIKernel(__global x_type * restrict x, ...)
 {
  #pragma unroll
  for (int id=0; id < nbrOfElements; id++){ /*outer loop*/
    y = doFirst(x[i]);    /* outer */
    z = doSecond(y);      /* loop  */
    res[i] = doThird(z); /* body  */
  }
}
```

The SWI model is a part of the OpenCL standard [10], but it is not an attractive configuration for CPUs nor GPUs, as the SWI kernel code is executed sequentially by a single CU on these devices. The FPGA devices are different because the hardware of CU is customized. The body of the outer loop can be pipelined [13]. In the Listing 1, variables y_i, z_{i-1}, and res_{i-2} would be calculated simultaneously (where i denotes the iteration number). Thanks to the pipelining, the CU is able to output the results at a high processing rate; however, the individual result can be delayed by many clock cycles with respect to its corresponding input. Consequently, the throughput rate of the SWI CU can be outstanding in FPGAs. The throughput rate is

$$R = \frac{clock \quad rate}{II},$$

where II is an iteration interval; *i.e.,* the number of clock cycles between reads of the consecutive inputs. The *unroll* directive controls the loop unrolling in AOCL. The kernels that are not SWI are called NDRange kernels.

3.2. Multiple compute units

The kernels that are not SWI are called NDRange kernels. Contrary to SWI, the scheduler distributes work-items to the NDRange computing units. However, it should

be considered that NDRange kernels require more FPGA resources, as the automatic organization of computing grid costs. The advantage of NDRange kernels is that they can be easily multiplied (within available FPGA resources) by AOCL. This increases the overall performance in general; however, the available bandwidth of the global memory should be weighted, as multiplied CUs share the memory bandwidth (Fig. 3a). It may also happen that the clock frequency will be reduced in the case of higher area utilization; then, more CUs are implemented.

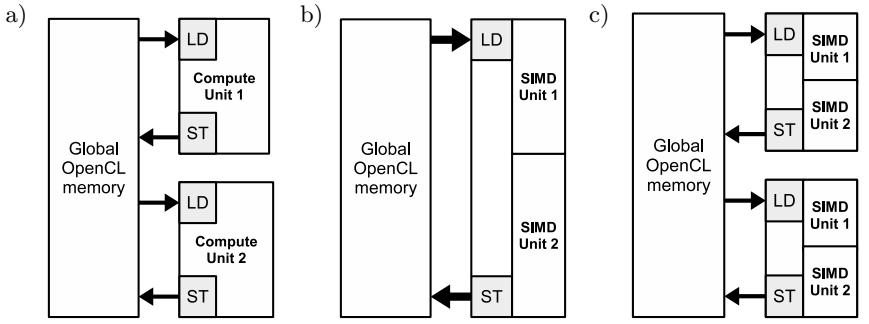

Figure 3. Methods to multiple performance of FPGA device in AOCL: a) Two CUs; b) Dual SIMD CU; c) Two SIMD CUs

Also, it is possible to apply multiple SWI-type kernels in the FPGA device. Nonetheless, it required manual replication of the kernel code in the OpenCL program – AOCL 16.0 does not multiply SWI kernels automatically. In practice, we duplicated the SWI kernel function, giving its clones different names; *e.g.,* using suffixes _0, _1, *etc.*. During host program execution, the kernels were enqueued separately with the input and output buffers containing only a fraction of the total workload. Additionally, the manual replication allows the designer to create kernels that perform different functions – they do not work in the SPMD manner.

3.3. SIMD computing units

An interesting way to multiply the performance is to aggregate multiple CUs into a unison Single Instruction Multiple Data (SIMD) block (Fig. 3b). The computing power of the SIMD CU block matches the combined power of all of the involved CUs. In the SIMD computing unit, individual CUs share a memory interface, and this pays off twofold: first, it introduces logic resource savings; second, the aggregated read or write from the global memory is typically faster than separate memory accesses.

It is important to consider that only work-items within the same work-group execute in the SIMD manner. The specification of a required work-group size (RWGS) is possible. AOCL relies on this specification to optimize the hardware usage of the kernel without involving excess logic. To take effect, the introduction of the SIMD size attribute in conjunction with the RWGS attribute is necessary. Also, the specified

SIMD size attribute must evenly divide the RWSG attribute. Additionally, the SIMD CU blocks can be combined into multiple NDRange kernels (Fig. 3c).

4. Experiments and results

In our experiments, we tried the SHA-512 kernel in the three computing unit configurations; *i.e.,* NDRange, SWI, and SIMD. Due to the available FPGA resources, we tested up to the two units at a time: two independent NDRange CUs, two SWI CUs, and one dual-SIMD CU. The performance tests were performed with an input data set that contained 2^{20} passwords.

In the first experiment, we studied the impact of the local work-group size (LWS) parameter on the overall performance of the system. LWS is a software parameter specified during the host program run; it dictates the size a work-group for the given thread execution – the work-group contains a set of work-items that must be able to make progress in the presence of barriers, and it must be mapped to a single compute unit. We have found that LWS only impacts the NDRange kernel that was compiled with no RWGS specified. The LWS parameter has no sense for SWI kernels, as they do not recognize threads. Also, we did not observe the significant impact of LWS on the performance of the SIMD kernels – they performed well for the small and large LWS values. The performance of the two NDRange CUs is given in Figure 4. The maximum performance was measured for LWS = 128; the average kernel time for a single hash was 7.13 ns, which corresponded to the throughput of *ca.* 140 million hashes per second.

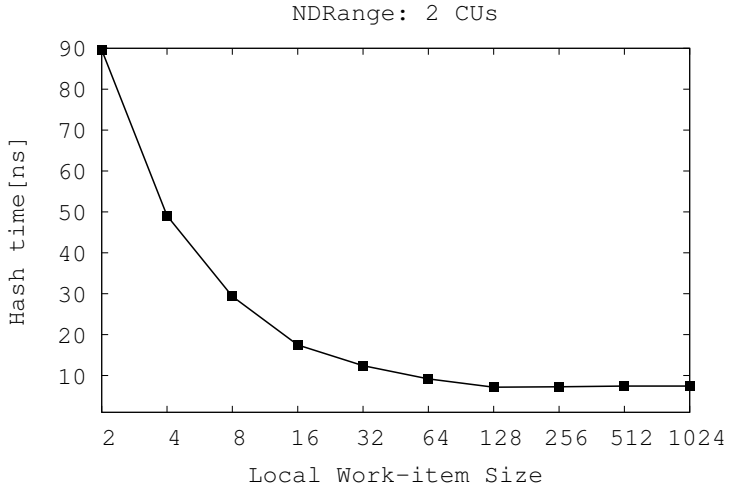

Figure 4. Performance of two computing units of NDRange type

In the next experiment, we compared the best results of the different kernel modes for the SHA-512 implementation. Table 1 summarizes the CU clock frequency

and average hash execution time for the NDRange, SWI, and SIMD kernels. The loop unrolling was tested for the NDRange and SWI kernels; the requested work group size equaled 64 or 256 for the SIMD kernel. The very best performance was achieved for the unrolled NDRange 2×CU kernel: the kernel hash time was 7.13 ns. It is worth noting that the total execution time includes both the kernel and host time. In our case, the best total execution time of a single hash value was 127 ns. In Table 1, we only provide the kernel time because it allows us to perform a clear comparison of different kernel implementations; the host time is constant and kernel-type independent.

Table 1
Result metrics of various kernel implementation modes

Kernel mode	Unrolled	Clock [MHz]	#CU	#SIMD	Hash kernel time [ns]
NDRange	no	175	1	1	531
SWI	no	208	1	1	399
NDRange	yes	236	1	1	8.13
NDRange	yes	236	2	1	7.13
SWI	yes	258	1	1	**8.22**
SWI	yes	237	2	1	**7.37**
SIMD; RWGS=64	yes	205	1	2	9.78
SIMD; RWGS=256	yes	205	1	2	9.81

The program source codes that were used in conducted experiments are available in repository [15].

5. Conclusions and contribution

The SIMD and SWI optimizations presented in Table 1 provide only a minuscule performance growth with respect to the single unrolled NDRange CU. The reason for this lies in the Global Memory access bottleneck. The memory access statistics are provided by the AOCL profiler; these are presented in Table 2. *Read Stall* gives the percentage of the time that the global memory causes pipeline stalls, and *Read Occupancy* is the fraction of the time the work-item executes the memory operation [5].

For all kernels given in Table 2, we see *Read Stall* that is close to 100%; this means a wait cycle for every memory read. For the NDRange kernel, *Read Occupancy* is close to 50%; as half of the execution time, the work-item is idle waiting for memory access completion. The *Read Occupancy* value is 25% for the SIMD kernels, as the performance could double due to the additional compute units; however, it does not due to the Global Memory bandwidth saturation.

Table 2
Global memory performance for unrolled kernels

Kernel	Read stall [%]	Read occupancy [%]	Total bandwidth [MB/s]	Hash kernel time [ns]
NDRange	93.99	48	23,822	8.13
SIMD, RWGS 64	93.75	25	19,697	9.78
SIMD, RWGS 256	93.75	25	19,695	9.81

The total Global Memory bandwidth reported by the profiler is 34 GBs. In Table 2, the bandwidth consumed by the unrolled NRDange CU is *ca.* 24 GBs. The hash time is approximately 8 ns – therefore, at a 250-MHz clock frequency, each hash takes two clock cycles. This leads to the conclusion that we need a 48-GBs bandwidth to fully utilize the computing performance of the single NDRange CU and 96 GBs to reach the maximum performance with the two SIMD units. It is worth mentioning that the four SIMD units do not fit the FPGA resources of Stratix V GS; therefore, further requests for higher memory bandwidth are not necessary.

In the presented experiments, the two NDRange CUs turned out to be the quickest solutions (7.13 ns per hash); this can be estimated as two clock cycles per hash at the 236 MHz operating frequency. In this configuration, each CU uses its own Global Memory interface. Apparently, a smaller transfer size allows it to transfer more data than in the aggregated SIMD optimization.

Our observations explain the reason – the GPGPUs are equipped with fast Global Memory. For example, the Nvidia GTX 1080 has 8 GB of memory, which offers 320 GBs of bandwidth [12]; this is four times smaller but more than nine times faster than the examined FPGA board.

The presented performance was also compared to the respective GPGPU results; these were compared in terms of performance and performance per watt. The results are summarized in Table 3. Despite the fact that the detailed power consumption was not measured in our experiment, we considered the power given by the manufacturers. The Radeon R9 is not the most power-efficient GPU, and we see that our FPGA represents a better performance-per-watt ratio. However, the Geforce GTX (which is the best choice for crypto-currency mining) is more than three times more energy efficient and nearly eight times faster than our FPGA accelerator.

Table 3
Performance and performance per watt of FPGA vs. GPGPUs

SHA-512 r accelerator	Power [W]	Performance [Mhash/s]	Performance [Mhash/s per W]
Nallatech PCIe395 FPGA	75	140	1.87
AMD Radeon R9 290 [7]	300	527.5	1.76
Nvidia Geforce GTX 1080 [12]	180	1088	6.04

Our results lead to conclusions that are similar to the opinions given by the authors of the papers in the topic of the FPGA-accelerated computing using OpenCL. As shown in [19], FPGA's results depend greatly on the applied optimizations. The execution of an unoptimized kernel may lead to extremely low performance; we observed this with the base implementation of the NDRange kernel. Even well-optimized kernels usually show lower performance when compared to GPUs and CPUs fabricated from the same semiconductor technology. Nevertheless, the benefit is often better power efficiency [19].

There are many advantages of FPGAs over CPU and GPU accelerators, which are the result of their high reconfigurability and availability of private block memory [17].

Tucci *et al.* implemented the Smith-Waterman algorithm on the Stratix V GX FPGA, which was 4.94 and 3.78 times more energy-efficient than the Nvidia Tesla K20 and Intel Xeon Phi 5110P, respectively [16]. There are many remarks about the efficiency of the compilation toolchain; as was observed, it has a great impact when compiling kernels for high clock rates. All of the generated kernels had frequencies that were above 200 MHz, which is a good result for complex FPGA designs. The work has shown that the same design can take significantly fewer hardware resources.

There is a claim that Altera trades off its sixfold development time reduction with a logic usage that is up to three times higher [8]. Another benefit of using OpenCL is the automatic integration of the PCIe communication interface (with which the developer does not have to bother). On the other hand, the abstraction provided by the OpenCL framework hides the information that is easily accessible by engineers using HDLs. The available AOCL reports are not enough to precisely identify the bottlenecks.

There is a need for works similar to the one performed by Wang *et al.* that concerns the FPGA optimization framework [17]. If manufacturers aim to make FPGA competitive with GPU-based accelerators, such tools must be a part of the FPGA tool-chains. Obviously, there are applications for FPGA accelerators (as was shown by Che *et al.*, who presented FPGA overtaking GPU thanks to the development of optimal HDL code [6]). However, OpenCL FPGA compilers should be able to make aggressive optimizations in order to deliver high-speed implementations, thus eliminating the need for the time-consuming HDL code development. Regardless, AOCL SDK is not such a tool as of yet; however, it is definitely a step towards efficient high-level synthesis.

Acknowledgements

Paweł Russek was supported by AGH University of Science and Technology Statutory Activity, Grant No. 11.11.230.017. Kazimierz Wiatr was supported by the National Science Center (NCN), Grant No. DEC-2011/01/B/ST6/03024.

References

[1] ACC Cyfronet AGH. http://www.cyfronet.pl/en/.

[2] Khronos Group. OpenCL overview. https://www.khronos.org/opencl/.

[3] SHA512AOCLStudy repository, 2018. https://git.plgrid.pl/scm/~plgrussek/sha512aoclstudy.git.

[4] Altera Corporation: *Altera SDK for OpenCL Getting Started, Version 15.0.0*, 2015.

[5] Altera Corporation: *Altera SDK for OpenCL Optimization Guide, Version 15.0.0*, 2015.

[6] Che S., Li J., Sheaffer J.W., Skadron K., Lach J.: Accelerating Compute-Intensive Applications with GPUs and FPGAs. In: *2008 Symposium on Application Specific Processors*, pp. 101–107, 2008. http://dx.doi.org/10.1109/SASP.2008.4570793.

[7] Ge C., Xu L., Qiu W., Huang Z., Guo J., Liu G., Gong Z.: Optimized Password Recovery for SHA-512 on GPUs. In: *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. 2, pp. 226–229, 2017. http://dx.doi.org/10.1109/CSE-EUC.2017.226.

[8] Hill K., Craciun S., George A., Lam H.: Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 189–193, 2015. http://dx.doi.org/10.1109/ASAP.2015.7245733.

[9] Janik I., Khalid M.A.S.: Synthesis and evaluation of SHA-1 algorithm using altera SDK for OpenCL. In: *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1–4, 2016. http://dx.doi.org/10.1109/MWSCAS.2016.7870147.

[10] Khronos OpenCL Working Group: The OpenCL Specification, Version 1.0, 2011. https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.

[11] National Institute of Standards and Technology: FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2, Tech. rep., 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[12] Nvidia: Geforce GTX 1080 specification, 2018. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/.

[13] Russek P.: Data-intensive processing on FPGAs, chap. 2.3, pp. 62–67, AGH University of Science and Technology Press, 2015.

[14] Russek P., Wiatr K.: The enhancement of a computer system for sorting capabilities using FPGA custom architecture, *Computing and Informatics*, vol. 32(4), pp. 859–876, 2014.

[15] Szkotak P., Russek P., Wiatr K.: SHA512AOCLStudy repository, 2018. `https://git.plgrid.pl/scm/~plgrussek/sha512aoclstudy.git`.

[16] Tucci L.D., O'Brien K., Blott M., Santambrogio M.D.: Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 716–721, 2017. `http://dx.doi.org/10.23919/DATE.2017.7927082`.

[17] Wang Z., He B., Zhang W., Jiang S.: A performance analysis framework for optimizing OpenCL applications on FPGAs. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 114–125, 2016. `http://dx.doi.org/10.1109/HPCA.2016.7446058`.

[18] Wielgosz M., Mazur G., Makowski M., Jamro E., Russek P., Wiatr K.: Analysis of the Basic Implementation Aspects of Hardware-Accelerated Density Functional Theory Calculations, *Computing and Informatics*, vol. 29(6), pp. 989–1000, 2010. `http://www.cai.sk/ojs/index.php/cai/article/viewArticle/125`.

[19] Zohouri H.R., Maruyama N., Smith A., Matsuda M., Matsuoka S.: Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 409–420, 2016. `http://dx.doi.org/10.1109/SC.2016.34`.

Affiliations

Piotr Szkotak
AGH University of Science and Technology, Krakow, Poland, piotrszkotak92@gmail.com

Paweł Russek
AGH University of Science and Technology, Krakow, Poland, russek@agh.edu.pl,
ORCID ID: https://orcid.org/0000-0003-3858-4278

Kazimierz Wiatr
AGH University of Science and Technology, Krakow, Poland, wiatr@agh.edu.pl,
ORCID ID: https://orcid.org/0000-0001-5959-0277

Received: 19.11.2018
Revised: 28.02.2019
Accepted: 28.02.2019