Masaryk University
Faculty of Informatics

GPU-based speedup of EACirc project

Bachelor Thesis

Jiří Novotný

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jiří Novotný

Advisor: RNDr. Petr Švenda, Ph.D.

Acknowledgement

I thank CRoCS laboratory for provision of workspace and hardware resources to make this thesis happen. I would like to thank Petr Švenda for leading my work and Martin Ukrop for endurance to answering my numerous question about EACirc insides.

Last but not least, I would like to thank my family for continuous support and for help with language aspects of my bachelor thesis.

Abstract

The main objective of this thesis is to bring support for GPU acceleration into EACirc project in order to speed-up the most computationally demanding part. The chosen part is implemented using GPGPU and the achieved speed-up is measured. The thesis also elaborates in build task automation and refactoring of the basic project structure.

Keywords

acceleration, CUDA programming, optimization, build configuration, GPU, GPGPU, EACirc, software circuit, randomness testing

Contents

1 Introduction

Random data and the concept of randomness are used in many branches of informatics. However, one of the fundamental ways how to use these principles is in cryptography and IT security. For instance, let there be a communication among several entities. The main content of the communication is meant to stay hidden from other entities, thus the communication needs to be encrypted by a chosen encryption protocol. The potential attacker[1] could intercept some encrypted messages and subject them to analysis. On the basis of certain traits of the protocol or similarities among the individual messages the encryption could be broken and the hidden content of the communication could be read by the attacker. Therefore, the goal of encryption protocols is that the encrypted messages are not similar or do not have some characteristic traits. In other words, the encrypted messages must look like random data to the attacker. However, this is very difficult to be ensured.

That is why tools have been created to test randomness, and thus quality, of ciphers. One of these tools is called EACirc and is being developed at the Faculty of Informatics of Masaryk University in CRoCS laboratory (Centre for Research on Cryptography and Security). It can tell how much the input data are close to the referential random data.[2] To achieve this EACirc uses raw computation power. However, the computations are not run in parallel. If it was the case the whole process would be significantly speeded-up. Faster evaluation could advance capabilities of EACirc and help to test the randomness in a more detailed manner. Speeding-up the EACirc is the primary objective of this thesis.

The speed-up of EACirc is achieved with running some chosen computations on a GPU. The GPU must have got a build-in support of a general purpose programming (GPGPU). Such chip can perform not only algorithms used in rendering of computer graphics but also almost every other algorithm that is runnable on a CPU. The main difference in comparison with CPU is that the CPU is optimized to minimize latency whereas GPU is optimized to maximize throughput. Latency is the time needed to start and finish the execution of a single instruction including time to load necessary data. On the other hand, throughput is a number which means how many instructions can be processed per one time unit. [2] Since some parts of EACirc process a lot of data with algorithms which does not need to be optimized for latency[3] the use of GPU's is suitable.

Several different solutions exist on the market which can be used for GPGPU programming. Each solution provides different special features. The solution used

[1] The one who wants to know the hidden content of the encrypted communication without permission of legal participants.

[2] This is only an approximative explanation. The exact definition and meaning of EACirc results are described in Martin Ukrop's thesis Usage of evolvable circuit for statistical testing of randomness [1] since the accurate understanding of EACirc tool is not relevant for this thesis.

[3] An algorithm that needs to be optimized for latency in order to maximize performance is that one which has lots of edges in it's control flow graph.

for this thesis is called CUDA [3] and it is a proprietary technology developed by NVIDIA. [4] The decision to use CUDA was taken by my advisor.

Since the performance of GPGPU is dependable on the used hardware the achieved speed-up was measured by an experimental method. The benchmarks took place particularly on machines which are being used at CRoCS laboratory for their own computations and are capable to run CUDA programs.

To set EACirc project for using CUDA, a non-trivial intervention into the settings of the project was needed. If the original settings of EACirc had been applied then this intervention would have resulted into a chaotic and further unmaintainable project. Therefore, in addition to the primary objective of this thesis to speed-up EACirc, the secondary objective is to remade the settings completely from scratch in order to prevent the problems that emerged by using CUDA. For this purpose an open-source tool CMake [5] was used that was developed by Kitware corporation [6].

The text of this thesis was typeset in LaTeX using *fithesis2* package created by Stanislav Filipčík [7]. The codes fragments which appear in the thesis text were typeset using LaTeX package called *minted* [8], version 2.0.

2 CUDA

As stated on the NVIDIA website [3], "CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)." The strengths and weaknesses of GPU lie in it's architecture and in differences from CPU. A GPU that is able to execute CUDA programs is addressed as *CUDA capable device* or simply as the *device*.

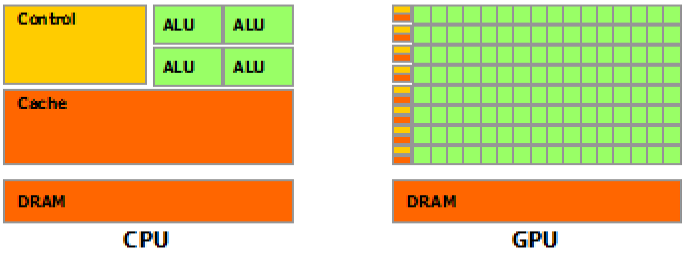

Figure 2.1: "The GPU Devotes More Transistors to Data Processing." NVIDA. [9]

The figure 2.1 shows a high level view of the CPU and GPU architecture. They both contain the same parts: DRAM, Cache, Control, and ALU. DRAM and Cache are memory units, the difference is that Cache is much smaller but significantly faster. The Control unit is responsible mainly for instruction fetching, decoding, etc. ALU is simply a worker that processes the input data. CPU's Control unit and Cache is much bigger and focuses on flow control and data caching in order to reduce the latency. The GPU's counterparts are simpler but multiplied, which allows to focus on data processing (throughput) and data parallelism. It is worth mentioning that the DRAM of a GPU is significantly faster in order to supply enough data to the big number of ALUs and to keep them busy.[1]

The performance of GPU is mainly measured by two variables: throughput and memory bandwidth. Since GPU is mainly used to compute with real numbers throughput[2] is expressed in units of floating-point operations per second (FLOPS)[3]. Memory bandwidth is the amount of data that can be processed per unit of time which is expressed in units of bytes per second (Bps). [2] The figures 2.2 and 2.3 show the theoretical maximum performance achievable on NVIDIA GPU's in contrast to Intel CPU's in terms of throughput and bandwidth.

[1] The memory model of GPU is described in section 2.3.

[2] For definition of throughput see section 1.

[3] Floating-point operation is an operation on floating-point numbers which in informatics represents real numbers according to the international standard IEEE 754.

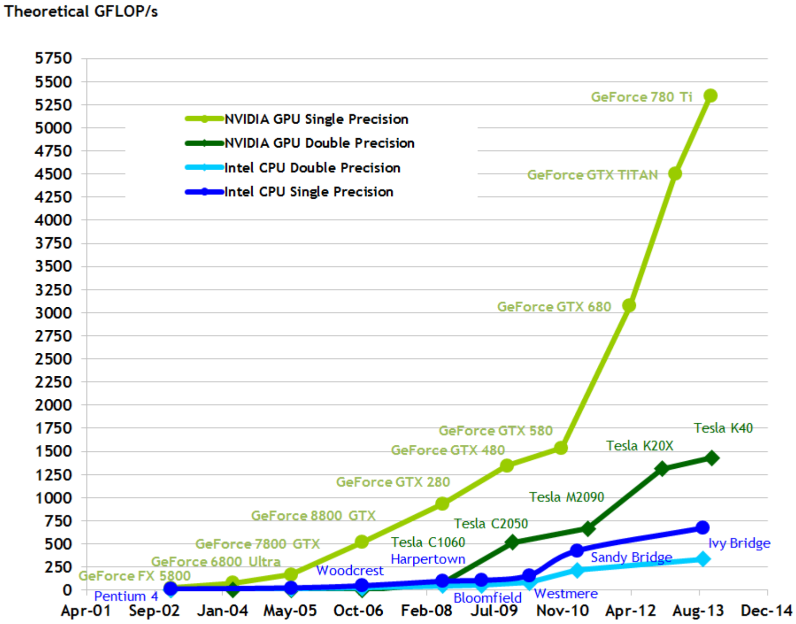

Figure 2.2: GPU vs CPU theoretical throughput. "Floating-Point Operations per Second for the CPU and GPU," NVIDIA. [9]

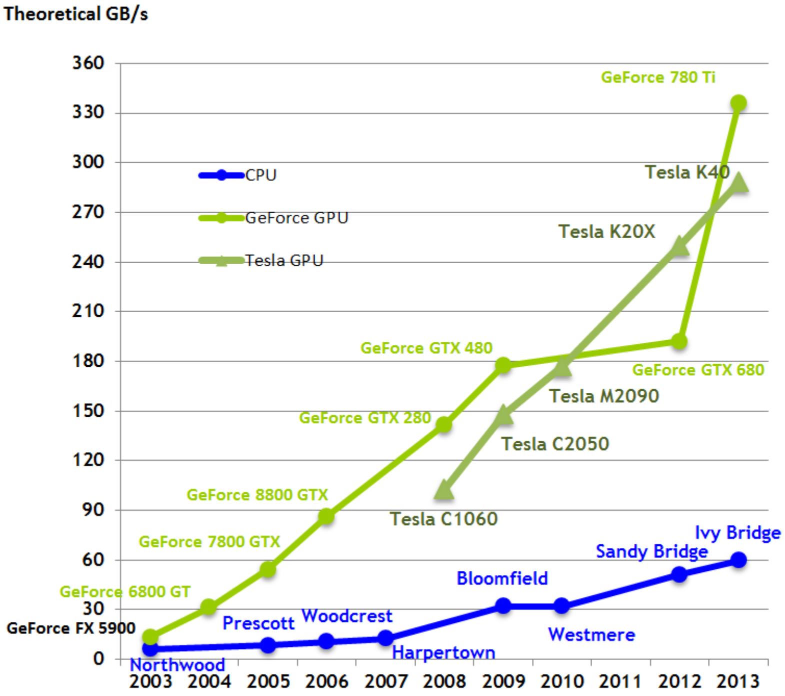

Figure 2.3: GPU vs CPU theoretical bandwidth. "Memory Bandwidth for the CPU and GPU." NVIDIA. [9]

2.1 Hardware architecture

In both, CPUs and GPUs, DRAM (figure 2.1) is significantly slower than ALU. If an ALU requires some data from DRAM, the ALU must wait hundreds of clock cycles for the data to became available (see latency). The waiting is highly ineffective and it is usually solved with some technique for latency-hiding. One of these techniques is executing another thread's instruction which has its data available. The difference between CPU and GPU is how often happens that an instruction wants data from DRAM.

Current CPUs use SIMD (Single Instruction, Multiple Data) execution model. [2] It processes a vector of data with only one instruction. The data in CPU are cached massively to reduce latency[4] and so the ALU does not need to wait. Thus, if big data are not accessed in a wrong way the probability of cache miss is low and switching context to a different thread can be relatively expensive operation.

The execution model of CUDA is called SIMT (Single Instruction, Multiple Threads). [2] Instead of vector of data, a *vector of threads* is executed with one instruction simultaneously. The vector of threads resides in one of the control units. Each thread of the execution vector is then mapped to an ALU related to the control unit. Each control unit has it's own cache. Since the cache is smaller, the cache miss is going to happen more often and another vector of threads, which has all its resources available, is executed. The switching of a thread context is done instantly with null overhead.[5] Therefore, to keep the GPU busy, more threads than the number of ALUs must be running.

In CUDA terminology the control units are called *Streaming Multiprocessors* (SMs). Each SM has its own ALUs referred to as *cores*. The single thread vector is composed of 32 threads and this is called *warp*. The SIMT architecture implies that all threads of the warp are performing the same instruction simultaneously at the same time. [9]

2.2 Thread hierarchy

The SIMT architecture of GPU is well suited and designed for computational problems that can be optimized using data parallelism. [2] Data parallelism is a parallelization technique that divides the input data to the independent parts and executes them separately (but evenly) on parallel computing nodes. The final result is then composed from each sub-result. [10] CUDA platform supports this technique through kernels and thread hierarchy.

[4] Data caching is a technique to avoid waiting for data which are stored in a slow storage by introducing memory hierarchy. When data are requested they are firstly searched for in faster memory. When they are not found (cache miss) then a slower memory is searched as long as they are found. Then they are promoted to the faster memory to become available to subsequent requests (cache hit). The data stored in cache which were not accessed for a long time are replaced with a new ones.

[5] The section 2.3 describes how is this achieved.

In CUDA context a *kernel* is a top-level function that is runnable on CUDA capable devices. [9] It is recommended that the kernel processes only the smallest portion of input data that can be processed separately. [10] For instance, when adding two vectors the kernel should just add two corresponding scalars of the vector.

For each kernel that is running a separate lightweight thread is created on the device. As shown in figure 2.4 a group of threads is forming a *block* and a group of blocks is forming a *grid*. Each thread has got unique ID dependant on its position in the block and each block has unique ID dependant on its position in the grid[6].

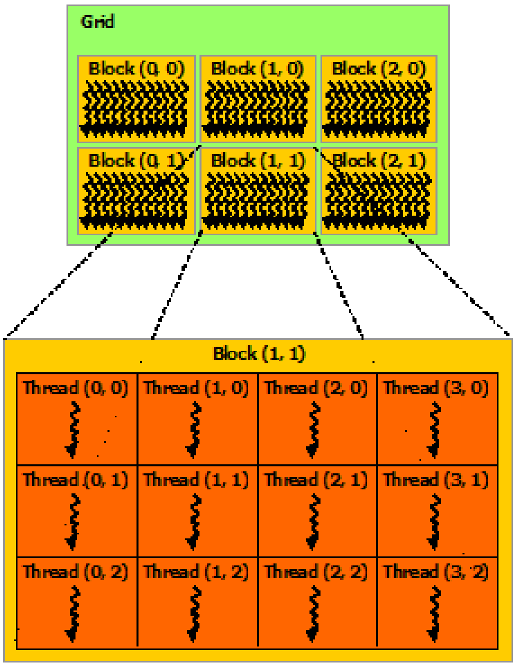

Figure 2.4: "Grid of Thread Blocks." NVIDIA. [9]

The dimensions of the grid should correspond to the dimensions of the computational problem. In the example of adding vectors the grid should have the same width as the number of scalars in the vector. Since each thread has a unique ID the kernel knows which scalars to add and where to put the result into the final vector.

The execution of a single kernel is initiated as soon as the device has enough available resources to run a whole block.[7] [9] This constraint allows that the threads of one block can communicate with each other (see section 2.3) and that the computational problem can be scaled across different types of CUDA GPUs that disposes with other hardware capabilities[8].[9] The execution order of the blocks is not defined.

[6] The grid might have up to 3 dimensions. [9]

[7] Threads of the same block are running concurrently.

[8] For instance the number of cores or available memory.

[9] Any program scales if and only if a similar performance is achieved for arbitrary size of input data or across different hardware. [11] It is recommended that any well optimized program for CUDA scales.

To fully utilize the device the size of the block should be multiple of a warp size[10]. Each block is mapped to a single SM[11]. On a single SM several blocks may be active but the exact number is dependant on the GPU hardware parameters. Depending of the number of SMs several block may be run in parallel. This is fully done automatically by the CUDA platform. However, the programmer should know these constraints to produce optimized code for each device.[12]

2.3 Memory hierarchy

CUDA devices dispose with a multi level memory model. Each level differs in size, speed, and accessibility. The main levels are *global memory*, *shared memory*, and *local memory*.[13] [9]

The global memory is the slowest and the biggest. The data living in the global memory are accessible everywhere in the device. The access to the data is done via 32-, 64-, or 128-byte transactions. The access should fulfil several constraints to achieve maximum performance. One of these restrictions is a coalesced access.[14] The coalesced access appears if every thread in the block order reads or writes on the subsequent address simultaneously. The global memory should be mainly used for kernel's input/output data and the number of accesses should be minimal.

The shared memory is fast. It is almost as fast as registers and hundreds of times faster than the global memory. It resides in Streaming Multiprocessor's (SM) cache. The data are accessible only by threads of the same block. The number of running blocks on SM is mainly determined by the size of the shared memory required for a single block, which should be known before the block starts executing. Again, there are access restriction for maximum performance. The address space of the memory is alternately divided into 32 (warp size) memory banks of size 32- or 64-bytes. Each thread of the warp should access the different memory bank, which will result in only one transaction. Otherwise the access would be serialized to the number of transactions depending on the maximum number of accesses to one memory bank.[15]

The performance of the local memory is almost the same as of the global memory which is slow. The data lives only for the lifespan of a thread and is accessible only by the owning thread. The local memory is only used in a few scenarios described in CUDA C Programming guide [9] in section Device Memory Accesses.

[10] For current devices the warp size is 32 (see. section 2.1).

[11] For definition of Streaming multiprocessor (SM) see 2.1.

[12] For various optimization techniques for CUDA platform see [11].

[13] In heterogeneous programming the GPU global memory is denoted as subset of *device memory* and CPU memory is denoted as *host memory*. More about heterogeneous programming is in section 2.4.

[14] Other constraints are not relevant for this thesis.

[15] If multiple threads are requesting value from one address the access will not be serialized but the value will be broadcast resulting only in one transaction.

Besides global, shared and local levels of memory a *constant memory* exists.[16] [9] It is a special kind of memory that is meant for constants and is implemented in almost the same way as the global memory. A single access to the constant memory is as slow as an access to the global memory. Since the data in this memory are constants (read-only values) they can be massively cached and the subsequent readings are as fast as readings from shared memory. Lifespan of the data living in the constant memory is the same as of the data living in the global memory.

Every other suitable variable of the scope of a kernel is placed into register. Registers are very fast. The number of registers for each Streaming Multiprocessor (SM) is large so that each SM may have a lot of active threads and to switch their context easily.[17] The number of active blocks and threads is determined by the count of used registers by one kernel.

2.4 Heterogeneous programming

A heterogeneous system is a system composed of more than one kind of processor. [2] The CUDA philosophy supposes that CUDA executes on a physically separate *device* (GPU) that acts as a coprocessor to the program that is running on the *host* (CPU). The state of the device and the host is stored in their own physically separate memory space referred to as the *device memory* and the *host memory*. Therefore, the host program is in charge of the device resources and manages launching of kernels. [9]

The most often used scenario of kernel launch including device and host resource management is as follows:

1. Host program allocates the space for the input and output data in the host memory.
2. Host program allocates the space for the input and output data in the device memory.
3. Host program populates the space for input data in the host memory.
4. Host program sends the input data to the device.
5. Host program configures and launches the kernel on the device.
6. Device program is executed.
7. Host program copies the final results from the the device to the host.
8. Host program processes the results.

Since most of these actions are input/output operations, they may be performed either synchronously or asynchronously.[18] The execution of the kernel on the device is always asynchronous. [9]

[16] There is also a *texture memory* but its description is irrelevant for this thesis. For more informations see [9]

[17] See SIMT architecture and switching of thread context in section 2.1.

[18] A synchronous operation blocks execution of the host program for the duration of the operation. An asynchronous operation does not block the execution of the host program and the incoming/outgoing transmission can overlap with computation. The host program is then notified whether the asynchronous operation succeeded or failed. [12]

2.5 Compute capabilities

As producing an optimized code for CUDA devices is closely linked to their hardware parameters [11] and because with almost every new product line of NVIDIA GPUs new features are introduced, NVIDIA established a system for backward and forward compatibility. The GPUs were divided to the classes reflecting technical parameters and runtime features of CUDA platform. These classes are referred to as compute capabilities.[19].

2.6 Programming language

The main programming languages for CUDA are C and C++ which are enriched by several CUDA specific keywords. Since C++ is a high-level language, some C++ features are not supported in the device code. The source code for the device and for the host may be mixed together and placed into one source file jointly. [9]

The listing 1 shows a sample of CUDA code which produces a sum of two vectors on the device. The sample follows the scenario of launching the kernel as described in section 2.4.

However, one caution should be remembered before porting an algorithm to the CUDA platform. If an algorithm contains an *if-then-else* statement and any thread of the warp evaluates the condition differently then the different execution paths are serialized. The execution paths merge after every thread of the warp finished its execution path. Fortunately some techniques exists to avoid this artefact. This principle also applies on other language control flow constructs such as loops, goto statements, etc. [9]

2.7 Tools

NVIDIA provides the necessary tools for the development on CUDA platform through CUDA Toolkit package. [13] The source files of CUDA programs must be compiled by a special compiler. The compiler that comes with the CUDA Toolkit is called `nvcc`. [14] The compiler supports mixing the code for the host with the code for the device. The `nvcc` identifies the code for the device and compiles it to an intermediate object file. The remaining code for the host is then forwarded to the ordinary compiler for C/C++ like `gcc`, `clang`, or `cl`. The object files from the `nvcc` and the host compiler are then linked into the binary form using ordinary linker or by `nvcc`. Therefore, the formed binary includes both the code for the host and for the device.

Debugging of the device code is slightly different from debugging of the code for the host. It requires a special debugger. The one that comes with CUDA Toolkit is called `cuda-gdb`. [15] The interface of this tool is almost the same as the interface of the known Linux compiler `gdb`.

[19] For full list of compute capabilities corresponding to the date of release of this thesis see CUDA C Programming Guide [9] section G. Compute Capabilities

```
// kernel definition (the __global__ keyword declares the kernel)
__global__ void vector_add_kernel( float* a, float* b, float* c )
{
        // the id of this thread in the block as a local variable
        int i = threadIdx.x;

        // add corresponding scalars of the vector and store the result
        // vectors a, b, and c are stored in the global memory
        c[i] = a[i] + b[i];
}

int main()
{
        float* host_a, * host_b, * host_c;
        float* device_a, * device_b, * device_c;

        // allocate vectors on the host
        cudaMallocHost( &host_a, SIZE * sizeof( float ) );
        cudaMallocHost( &host_b, SIZE * sizeof( float ) );
        cudaMallocHost( &host_c, SIZE * sizeof( float ) );

        // allocate vectors on the device
        cudaMalloc( &device_a, SIZE * sizeof( float ) );
        cudaMalloc( &device_b, SIZE * sizeof( float ) );
        cudaMalloc( &device_c, SIZE * sizeof( float ) );

        // copy input vectors from host to device
        cudaMemcpy( device_a, host_a, size, cudaMemcpyHostToDevice );
        cudaMemcpy( device_b, host_b, size, cudaMemcpyHostToDevice );

        // launch kernel with only 1 block of size SIZE
        // a special CUDA syntax is used
        vector_add_kernel<<< 1, SIZE >>>( device_a, device_b, device_c );

        // retrieve the result from device
        // although the launch of the kernel is asynchronous this function
        // waits untill the execution of the kernel is not finished
        cudaMemcpy( host_c, device_c, SIZE, cudaMemcpyDeviceToHost );

        // free memory on the device
        cudaFree( device_a );
        cudaFree( device_b );
        cudaFree( device_c );

        // free memory on the host
        cudaFreeHost( host_a );
        cudaFreeHost( host_b );
        cudaFreeHost( host_c );

        return 0;
}
```

Listing 1: A sample program of vector addition on CUDA platform.

3 CMake

To set EACirc project for using CUDA, an intervention into the settings of the project was needed. Below the original settings and workflow of EACirc are described. However, this settings had to be remade from scratch.

The EACirc sources consist mainly of C and C++ code. The code is divided into reasonably logical sections, but the overall structure and concept of the project are monolithic[1]. This leads to compilation of all sources into one big executable of approximately 9 MB, which takes some non-trivial time.

On top of this EACirc is developed as a cross-platform application. To provide native builds for each supported platform (Windows [16] and Linux) a special makefile or an IDE specific project file are used simultaneously. They describe how to build the application. When a change in the build is introduced, e.g. a new source file is added, the change has to be manually implemented to all makefiles to provide consistency. This workflow is not easy to maintain as the violation of these rules can cause an uncomfortable pitfall.

To solve these problems the CMake [5] tool was integrated into the project of EACirc along with some changes of the basic structure of EACirc. The CMake tool is developed and maintained by Kitware, Inc. [6] as an open-source software. The main purpose of this tool is to provide native builds of cross-platform applications and to minimize the effort to maintain the project.

Although there are many similar tools as CMake and some of them provide better features they are not so widely supported. For instance, CMake generates project files for almost every common IDE and some of these IDEs come with a built-in support for CMake.

3.1 CMake toolset

The CMake is actually a set of several tools that provide for building, testing, and deploying of user's C or C++ project. These tools can be installed on Linux, Windows, or MacOSX. The CMake toolkit consists of the main tool `cmake` and the supportive `ccmake` (or `cmake-gui`), `ctest`, and `cpack`. [17]

The `cmake` tool takes a configuration file called `CMakeLists.txt` distributed with the project source files and generates the platform specific makefiles as an output. Then the user invokes a platform specific tool for building – usually `make`, `ninja` [18], or `MSBuild`. [19] If the process is successful the native binaries of the project are now made.

The `ctest` tool provides a simple platform for project testing. If the build is successful the user can run some custom made tests on the binaries.

[1] A monolithic binary is an executable that does not need any other dependencies or resources at a runtime. In other words, the binary is independent.

The cpack tool provides a cross-platform mean to deploy your application on the target system.

The remaining ccmake and cmake-gui are just more convenient ways to use a cmake tool since cmake has only a command line interface. The former provides a TUI[2] and the latter provides GUI[3].

3.2 A closer look at the cmake executable

The cmake executable is not just a dummy build-system. The process of generating a makefile is quite sophisticated. First, the user chooses the *source directory* and the *build directory* and then invokes the cmake command in a *build directory* with appropriate parameters. The subsequent process consists of several phases – selection of a native build-system (in a CMake terminology referenced as a *generator*), configuration based on a user-specific input, and the generation of a makefile itself.[4] [17]

The *source directory* is simply a directory where the project sources are located as well as the top-level CMakeLists.txt file which is distributed with the sources. The build directory is an empty user-created directory in which the user wants the binaries to be build.

The selection of the *generator* depends on the user's platform, on the user-installed native build-systems, and on the user's intentions. The generator used on Linux is usually make or ninja. If the user wants to generate project files to a specific IDE, appropriate generator can be chosen – e.g. Visual Studio 2013 [20] on Microsoft Windows [16]. Usually the selection of the appropriate generator is done by CMake automatically.

The subsequent phase is configuration. Here the user specifies variable options for the build that the project supports. For instance, some features of the application can be switched on/off or the location of a third party dependencies can be specified. Also the different build configuration can be switched, i.e. release or debug.

If the configuration is correct then the makefile is successfully created in the *build directory*. Then the user just invokes the appropriate tool to execute the makefile and the binaries are build.

It is worth mentioning that the makefile automatically detects any changes made in the *source directory*. So the user invokes the cmake executable just once to generate the makefile or to change the variable options of the build. The makefile also provides a way to install the application and/or to test it.

The minimal and the most common sequence of commands to build and install a project on Linux using the CMake is shown in listing 2.

[2] Text-based user interface (TUI)

[3] Graphical user interface (GUI)

[4] Note that the exact scheme of this process can differ according to which interface of CMake is used – i.e. cmake, ccmake, or cmake-gui.

```
mkdir <build_directory>
cd <build_directory>
cmake <path_to_source_directory>
make
make install
```

Listing 2: The minimal CMake workflow.

Note that the `make` is chosen as a default generator. In addition, the default project settings and configurations are applied. The binaries are installed to the platform specific location, i.g. on Linux it is `/usr/share/local`.

3.3 Changes made to the EACirc repository structure

There were several changes made to the EACirc repository structure. The new folder design reflects the logical structure of the EACirc philosophy. The changes are shown in the figure 3.1.

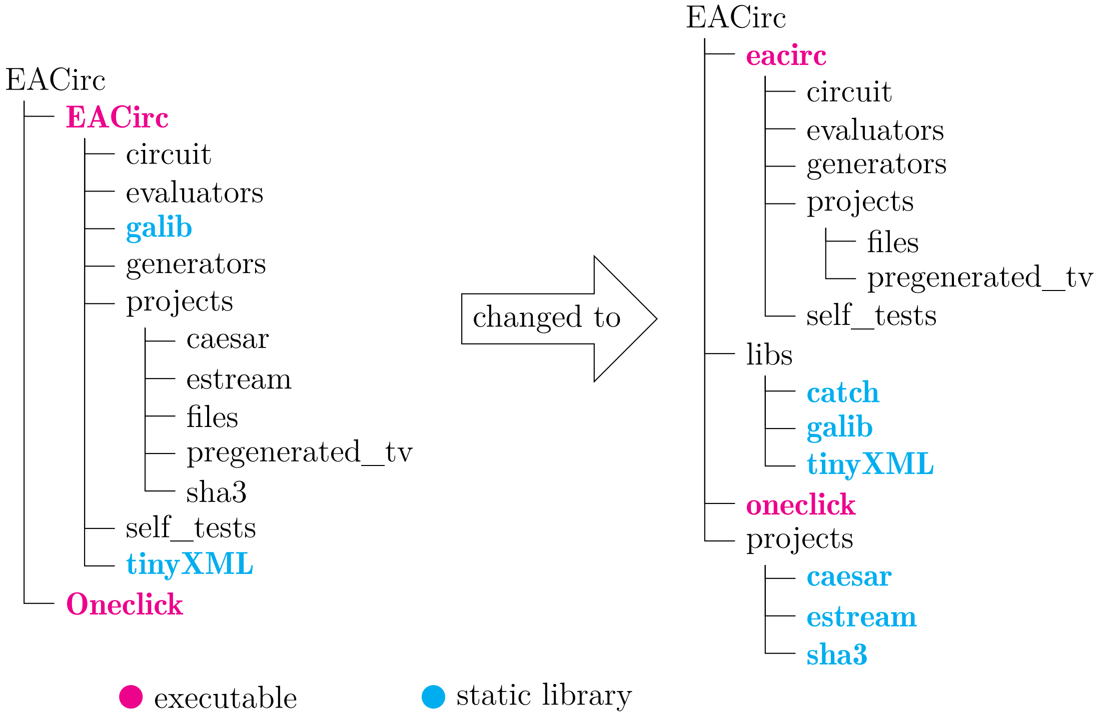

Figure 3.1: Old vs. new repository structure

The first and also the smallest change was to name all source folders with only small letters. Next the libraries from 3rd party providers `catch`, `galib`, and `tinyXML` were moved into the separate folder – the *libs* directory.

Then the so called *projects* were isolated. A *project* in EACirc terminology means a problem solving module. These *projects* are `caesar`, `estream`, `sha3`, `files` and `pregenerated_tv`. Since `files` and `pregenerated_tv` are both just small modules consisting of only one source file, it would be impractical to isolate them. Whereas the big modules `caesar`, `estream`, and `sha3` were moved to the separate folder called the *projects* folder. Each of the isolated projects was remade to compile into a static library.[5]

The content of folders `eacirc` and `oneclick` is build into executables which are named accordingly to their corresponding folder. The *projects* which are now compiled into the static libraries are now statically linked to the `eacirc` executable representing the EACirc tool as a whole. The `oneclick` executable is a supportive tool for the automated task management developed by Ľubomír Obrátil. [21]

3.4 The new build-system of EACirc

The new build-system is written on the CMake platform. This platform allows to define custom options for generating the build. A descriptive list of EACirc specific options is given below:

`BUILD_ONECLICK`
Enable building of Oneclick, the supportive tool for EACirc.

`BUILD_CAESAR`
Enable building of the Caesar project.

`BUILD_ESTREAM`
Enable building of the Estream project.

`BUILD_SHA3`
Enable building of the SHA-3 project.

`BUILD_CUDA`
Enable to build the support for CUDA devices. This option is available only if the CUDA Toolkit [13] is installed on the build machine[6] and found by the CMake.

Since the *projects* are build into static libraries they must be linked to the `eacirc` executable at the compile time. This is done automatically when the option for the specific *project* is enabled. In the figure 3.2 dependencies of all the build targets are shown.

The build-system is also version aware. The current version is stored in the `eacirc/Version.h` header file. The version corresponds to git commit hash [22]. This means that for the correct build generation git tools must be properly installed on the build machine and found by CMake.[8]

[5] There is a plan to remake the projects to modules loaded dynamically at runtime. This would require to compile them separately into the dynamic libraries.

[6] A build machine is a physical or a virtual machine that is used to build the project.

[8] If git tools are installed and not found automatically by CMake then the path to git tools can be specified manually.

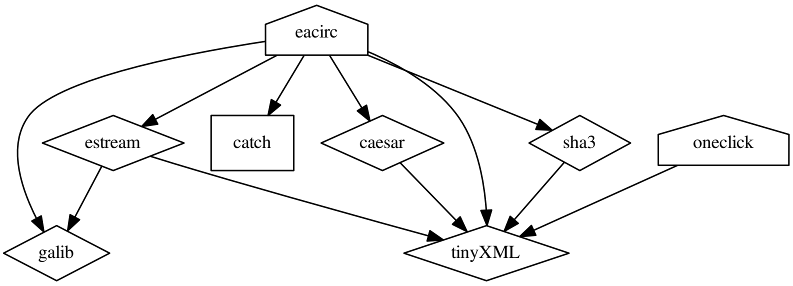

Figure 3.2: EAcirc dependency graph

The static libraries are shown in the rhombus. The executables have a house around them, squares represent interface libraries[7] and the direction of arrows describes dependency of build targets.

The use of CMake and the new options of building EACirc is explained in detail on the Github wiki project page under the Building EACirc section.

3.5 Project settings for CUDA

It is now much easier to set the project for CUDA support with CMake than with ordinal makefiles. When the CUDA Toolkit [13] is installed and automatically found by CMake[9] then the option `BUILD_CUDA` becomes available. If this option is enabled then the `eacirc` executable is build by using Nvidia `nvcc` compiler [14] and the C preprocessor macro `CUDA` is defined causing that the executable will be runnable on CUDA capable devices. When writing a code for CUDA the preprocessor macro `CUDA` can be queried.

[9] If CUDA Toolkit is installed on the build machine but not found by CMake automatically then the path to CUDA Toolkit can be specified manually.

4 Gate Circuit of EACirc

The purpose of EACirc is to differentiate input data and referential random data from one another without any context. The input data consists of thousands of sample outputs from an arbitrary random generator. These sample outputs are referred to as *input vectors*. The distinction is done by creating a gate circuit which processes the input data and outputs the result. The internal structure of the optimal gate circuit is unknown at the moment of invocation of the EACirc program. However, the optimal form of the circuit is constructed via a genetic algorithm. [1]

Genetic algorithms are inspired by processes of reproduction and natural selection in living systems. They search for a satisfactory optimal solution for a given problem. The process of producing the acceptable solution consists of creating a random initial generation of individuals and then iteratively applying evaluation of the generation, selection of the best individuals for the next generation, and creating next generation by applying genetic operators. [23]

The functionality of the gate circuit in EACirc is the same as an ordinary digital circuit that is composed from electronic components (*gates*) connected with wires (*connectors*). Each gate is performing some logical function on its input defined by connectors. The whole circuit takes one input vector, which is a fixed amount of logical data, and outputs if the input is random or not.[1] Since the gate circuit, in the context of EACirc, is stored in the memory of a computer, the implementation differs from the real world.

Producing the final gate circuit via genetic algorithm is an iterative process. In each iteration the whole generation of circuits must be evaluated. The evaluation of a single circuit is done by processing all the input vectors. All of these parts of the process must be done in large quantities to ensure quality, which is a computationally intensive task. Fortunately, the evaluation of a single circuit and even the evaluation of the whole generation can be run effectively in parallel. This thesis focuses mainly on parallelization of evaluation of a single circuit.

The evaluation of a single circuit is suitable for parallelization, particularly for data parallelism on GPGPU and CUDA platform. The size of the input data is relatively small (usually 16 KiB or more)[2] The input data are a set of input vectors (usually 1000 vectors). Each input vector is evaluated separately by the same circuit which usually outputs only 1 byte of data. Therefore, the data transfers between host and device via a *slow* PCIe bus will be small.[3] The data parallelism can be achieved by mapping of execution of single input data to separate device thread. The single

[1] Actually the output of a circuit is not only a binary (random/non-random) but contains other information that is further processed.

[2] The exact parameters of the circuit are runtime variables. The reported values in parenthesis are just the most used but if running on CUDA, bigger values are expected.

[3] The PCIe bus has got a big latency, thus the number of transfers must be minimized or the transfers must be buffered.

circuit processes different data equally, i.e. the data is different but the execution paths of multiple running circuits are the same.

4.1 Software representation of hardware circuit

The following text describes the CPU implementation and functionality of the gate circuit designed by Martin Ukrop. [1] This representation served as a starting point for the GPU implementation.

From informatics point of view, the gate circuit can be described as a directed acyclic graph (DAG) with some other restrictions described further. Each node represents a logical function (*gate*), the edges (*connectors*) of the vertices represents inputs for the individual logical functions and the flow of data. The graph is divided into several layers. Every node in each layer is connected only to nodes which belong to the previous layer. For a graphical presentation of the circuit see figure 4.1. [24]

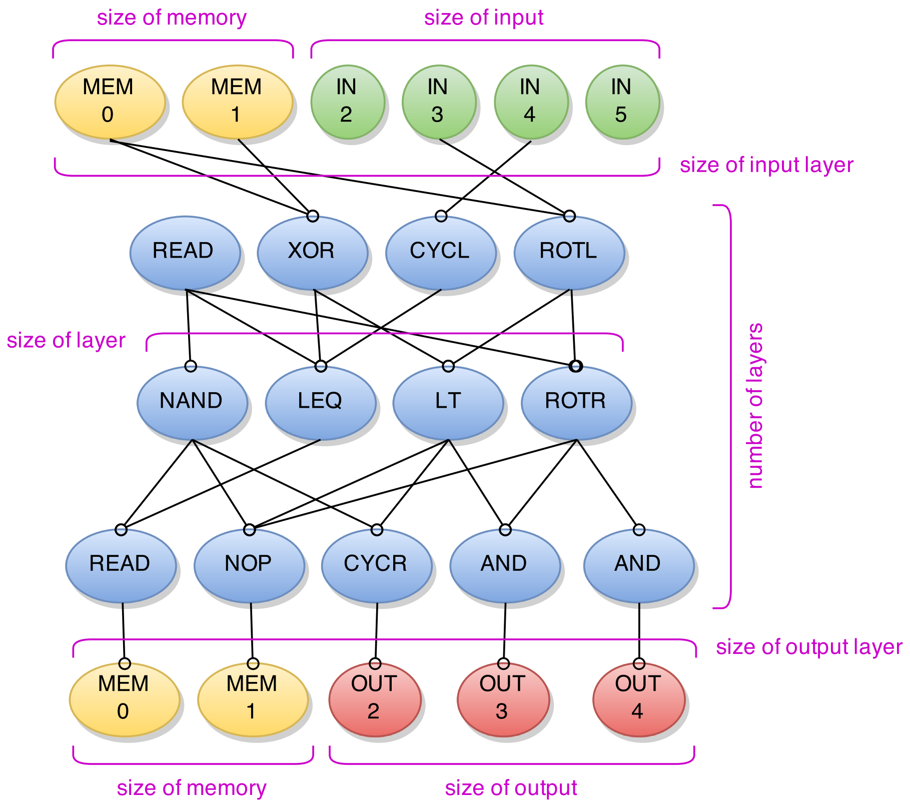

Figure 4.1: The dimensions of the gate circuit

The gates are in blue, the connectors are the black lines. The yellow, green, and red nodes show mapping of data to the gates. The flow of the data is from top to bottom.

4.1.1 Use memory feature

The gate circuit has a special feature of using a memory. It allows to process larger input vectors than the maximal width of the circuit input (see figure 4.1 and `size_of_input`). The input vector is divided into several parts of size `size_of_input` and each part is processed sequentially. A part of output (`size_of_memory`) from current run of the circuit is transfused to the input of the next run (see figure 4.1 and the yellow nodes). If this feature is not used then the `size_of_memory` is equal to 0. Unfortunately, the feature of using a memory was newer used in production. Therefore, the feature is not implemented in GPU version of the circuit.

4.1.2 Circuit dimensions

The circuit dimensions (see figure 4.1) are loaded at the EACirc invocation from settings and fixed for the whole runtime of the program. The circuit has got `num_of_layers` layers. All layers except the last one have `size_of_layer` nodes. The last layer has `size_of_output_layer` nodes. Each node is limited by the same maximum number of allowed connectors.

4.2 CPU implementation

The understanding of CPU implementation of the circuit by Martin Ukrop [1] is necessary in order to provide a bridge interface to the GPU implementation and to preserve functionality.

In computer memory the gate circuit is represented as an 1-dimensional array of 32-bit unsigned integers. Each integer has its own meaning. The array is alternately divided into connection layers and function layers as shown in the figure 4.2. Each layer is aligned to a multiple of `genome_width`.

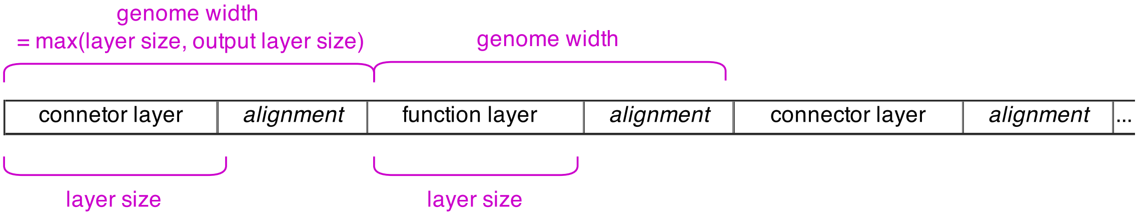

Figure 4.2: The representation of gate circuit in CPU memory.

4.2.1 Connector layers

The connection layers contain connector masks which are represented as 32-bit unsigned integers. Each node has assigned its own connector mask. If the i-th bit of the connector mask is `true` then connection exists from the n-th node of the previous layer to the node the connector is assigned to, where n depends on the connector type. There are two types of connectors which are convertible to one another. The types are *absolute connectors* and *relative connectors*.

Relative connector is a representation of a connection which is defined through relative positioning to the node it belongs to. This representation is used in genetic algorithm when creating new generation, i.e. if the circuit is cut in half and crossed with another individual[4] the connector remains still valid.

Absolute connector is a representation of a connection by absolute positioning. This positioning is used when the circuit is being executed because then the process of resolving the absolute connector is much faster then resolving the relative connector.

4.2.2 Function layers

The function layers contain function masks which are represented as 32-bit unsigned integers. To each node a special function mask is assigned. The mask contains information about the function type and arguments. For the list of possible function types see EACirc wiki section Gate circuits. [24]

4.2.3 Circuit interpreter

The circuit interpreter is responsible for executing the circuit. The interpreter processes each layer of the circuit sequentially. The CPU version uses the dynamic allocation of memory for the inputs and outputs of the currently processed layer. Each invocation of the CPU circuit interpreter on the same circuit (the same circuit is invoked thousand times and more) converts the relative connectors to the absolute connectors. This is a major performance drawback and it could be done only once for each circuit to speed the execution.

4.3 GPU implementation

The GPU implementation source files of the gate circuit is placed in the directory `eacirc/circuit/gpu_gate`. The sources are written mainly in CUDA C++[5] or C++ programming language using the standard *C++11*. [9], [25] The GPU implementation provides the proper interface for communication with the rest of the program. The GPU circuit representation and the interpreter are written using C++ templates [25] in order to provide multiple sizes of connector and function masks and thus larger circuits.

The GPU implementation is written with the idea of running the code on GPU and on CPU, too. The GPU support is the primary one. Running the code on CPU should by possible after a minimal intervention into the EACirc source codes.[6] The reason to make the new implementation capable running on GPU and CPU is to

[4] Crossing of individuals is one of the types of genetic operators used for creating a new generation of individuals.

[5] For the description of CUDA C++ language see section 2.6

[6] Running the code on CPU has not been tested yet, but should be possible.

maintain only one implementation that is runnable on every machine and provides GPU acceleration if the machine hardware supports it.

4.3.1 Circuit representation

The GPU implementation cannot use the storage layout of the circuit as the CPU version described in section 4.2, because of the CUDA memory restriction for faster data access.

The circuit is stored as 1-dimensional array of instances of C++ template class `gate_circuit::node` in a *constant memory* of the device. The nodes are stored similarly as layers in CPU implementation using alignment (see figure 4.3).

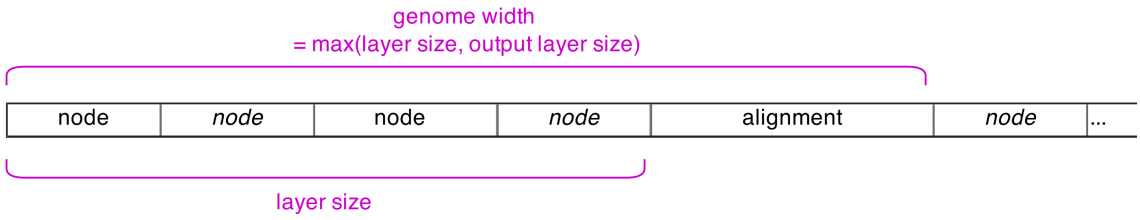

Figure 4.3: The gate circuit stored in GPU constant memory

The `node` class represents circuit nodes that have its own function and connector masks. The reasons of this data layout are to use coalesced memory access for faster data transfers and to better use the caching capabilities of GPU.[7] The whole node always resides in one cache line and therefore, only one transaction to load data from constant memory is used. The listing 3 shows the basic structure of the node using C++ code.

```
template <typename T>
class gate_circuit {
public:
        using value_type = T;

        class node {
                value_type _func_mask;
                value_type _conn_mask;
        };
};
```

Listing 3: The GPU representation of circuit node

[7] For coalesced access and caching see section 2.3.

4.3.2 Converting between CPU and GPU representation

The template class gate_helper is responsible for conversion between CPU and GPU representation of the circuit. The conversion is performed on CPU before the circuit definition is sent to device and executed. The gate helper also transfers the relative connectors to absolute connectors, because the GPU interpreter works only with absolute connectors.

4.3.3 Interpreter

The GPU circuit interpreter is contained in template class gate_interpreter. This class is capable of executing the circuit on GPU and CPU. The reason why this class is written as a template is to support different types of circuits (see 4.3.1). One invocation of the circuit interpreter processes only one input vector.

The GPU interpreter processes the data similarly as the CPU version – one circuit layer after another. The state of the execution is maintained per instance and thus multiple instances must be created for concurrent execution. The inputs and outputs from the currently processed layer are stored in a memory that is allocated outside the class scope. Therefore, different storage spaces may be used when running the interpreter on CPU or GPU. The input and output vectors are passed through a pointer for each circuit execution.

4.3.4 Kernel

The kernel function[8] accepts the following necessary parameters to run the circuit on GPU: pointer to the storage of circuit inputs and outputs, the circuit definition, and the size of the device's shared memory bank[9]. The listing 4 shows the code.

The kernel computes its ID and the exact position of input and output vectors which are located in the global memory and which the kernel is supposed to process. Then the kernel allocates the space for interpreter layers in a shared memory for the fastest data access. The reason to use shared memory over registers is that the size of shared memory may be specified at the runtime during kernel launch. On the contrary, the size of registers must be known at the compile time. Since the size of the circuit layer is not know at the compile time, the shared memory is used for inputs and outputs of the currently processed layer.

The allocation is done by the CUDA platform automatically when the execution of the block starts. Since the shared memory belongs to the whole block of threads, each kernel must ensure a space for its own execution layers. However, each thread of the warp accesses the data in the shared memory at different addresses simultaneously and the bank conflict[10] can emerge. The bank conflicts are prevented using a technique called striding. The technique is based on allocation of an extra space for each thread.

[8] For definition of kernel see section 2.2.

[9] For the description of the shared memory bank see section 2.3.

[10] For the description of bank conflict see 2.3.

The size of the extra space is equal to the bank size and is never accessed by any thread, thus each thread of the warp accesses a different bank every time.

```
template <class T>
__global__ void kernel(const byte* ins, byte* outs,
                       const gate_circuit<T> circuit,
                       const size_t bank_size)
{
        const int id = blockIdx.x * blockDim.x + threadIdx.x;

        const byte* in = ins + (id * circuit.in_size);
        byte* out = outs + (id * circuit.out_size);


        extern __shared__ byte memory[];

        byte* layers =
                 memory + ((2 * circuit.in_size + bank_size) * threadIdx.x);

        gate_interpreter<T> interpreter(layers, &circuit);
        interpreter.execute(in, out);
}
```

Listing 4: The interpreter kernel

4.3.5 Job dispatching

A class called `gpu_task` was created. This class is responsible for launching the kernel, managing device memory and copying the data from CPU and GPU. It uses host page locked memory [9] for faster data transfers between the host and the device. The number of launched kernel threads is calculated as follows:

```
std::ceil( float( vec_count ) / block_size ) * block_size
```

The `block_size` is 128 and `vec_count` is the number of input vectors of EACirc.

5 Testing and benchmarking

5.1 Testing

5.2 Benchmarking

6 Conclusions and future work

The automated building of the EACirc project from the source files was unified across multiple platforms. For this purpose a CMake tool was added to the project dependencies. Despite the added dependency the project is now easier to be maintained. The previous automated build system suffered from code duplication, which is now solved through the CMake tool. During the CMake integration into the project the basic structure of EACirc was refactored in order to improve the compilation time and to introduce conditional building of distinct parts of EACirc.

After the new build system had been properly tested, it was integrated into the EACirc development branch `eacirc-dev` of the project repository located on GitHub. The new build system is now being used by every EACirc developer.

Now the EACirc program supports GPU accelerated computations using CUDA platform in order to speed-up the program execution. The support of the GPU acceleration may be turned on/off during the project compilation time.

Since the GPU acceleration of an computation is beneficial only if the computation fulfils certain criteria, the most suitable part of EACirc was chosen to be accelerated. The chosen part represents the execution of the software circuit. The circuit is run independently and repeatedly in large quantities, which allows to optimize it by using data parallelism required for GPGPU.

The original CPU version of the circuit served as a starting point for the newly created implementation of the circuit that is runnable on GPU in line with the goal of this thesis. Some unused features of the CPU version were not implemented into the created GPU version. However, the basic functionality of the circuit remained unchanged.

The original goal of this thesis was to develop another implementation of the circuit which would be runnable only on GPU. In addition to this, the created implementation should be capable of running not only on GPU but on CPU, too. This additional feature will be possible after a minor code adjustments. It is worth to do so, because it will solve the current issue of code duplication of the circuit.

The created implementation of the circuit execution was validated via integration testing. The tests that were made did not detected any severe errors.

TODO results of speed-up and the hardware of machines used for benchmarking

The GPU implementation of the circuit is expected to be integrated into the EACirc development branch `eacirc-dev` of the project repository located on GitHub and to be used by most of the EACirc users.

6.1 Future work

There are several options how to focus future work in the field of speeding-up the EACirc project using parallelism. These options are mentioned below in an ascending order based on their complexity.

The most trivial option is a multi-device execution of the created circuit. It would bring the possibility of processing even larger amount of data on machines that are equipped with multiple CUDA capable GPUs. However, this solution would pay off only if enough data are available to be processed in order to harness the power of SIMT architecture of the devices.

The next option is to work on the GPU implementation of the circuit to be runnable on CPU. This will lead to only one implementation of the circuit in project source files, which will prevent code duplication. To achieve this, the CUDA code must be adjusted so that it can be build via regular C++ compiler, and also a CPU job dispatcher must be implemented. This dispatcher could use the current multi-core CPU and run the execution of the circuit in parallel.

The last but not least option is the evaluation of the circuit on CPU while another circuit is executed on GPU simultaneously. This would require a large intervention into the project design and the program execution path.

Bibliography

[1] M. Ukrop, “Usage of evolvable circuit for statistical testing of randomness”, bachelor thesis, FI MU, 2013.

[2] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA ® C Programming*. Wrox, Sep. 9, 2014, ISBN: 978-1118739327.

[3] Nvidia Corporation. (2015). About CUDA, [Online]. Available: `https://developer.nvidia.com/about-cuda`.

[4] ——, (2015). Welcome to NVIDIA - World Leader in Visual Computing Technologies, [Online]. Available: `http://www.nvidia.com`.

[5] Kitware Inc. (2015). CMake, [Online]. Available: `http://www.cmake.org/` (visited on 03/08/2015).

[6] ——, (2015). Kitware Inc. – leading edge, high-quality software, [Online]. Available: `http://www.kitware.com/` (visited on 03/08/2015).

[7] S. Filipčík, “LaTeX Thesis Style”, bachelor thesis, Faculty of Informatics Masaryk University, 2009. [Online]. Available: `http://is.muni.cz/th/173173/fi_b/` (visited on 05/03/2015).

[8] G. M. Poore, *The minted package: highlighted source code in LaTeX*, Jan. 31, 2015. [Online]. Available: `https://github.com/gpoore/minted/blob/master/source/minted.pdf`.

[9] Nvidia Corporation, *CUDA C Programming Guide*, Mar. 2015.

[10] W. D. Hillis and G. L. Steele Jr., “Data parallel algorithms”, *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986, ISSN: 0001-0782. DOI: `10.1145/7902.7903`. [Online]. Available: `http://doi.acm.org/10.1145/7902.7903`.

[11] Nvidia Corporation, *CUDA C Best Practices Guide*, Mar. 2015.

[12] Microsoft. (2015). Synchronous and Asynchronous I/O (Windows), [Online]. Available: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365683(v=vs.85).aspx` (visited on 05/06/2015).

[13] NVIDIA Corporation. (2015). CUDA Toolkit, [Online]. Available: `https://developer.nvidia.com/cuda-toolkit`.

[14] Nvidia Corporation, *CUDA Compiler Driver NVCC*, Mar. 2015.

[15] ——, *CUDA-GDB CUDA Debugger*, Mar. 2015.

[16] Microsoft. (2015). Windows – Microsoft Windows, [Online]. Available: `http://windows.microsoft.com`.

[17] Kitware, Inc. (2014). CMake Reference Documentation – CMake 3.0.2 Documentation, [Online]. Available: `http://www.cmake.org/cmake/help/v3.0/`.

[18] E. Martin. (Nov. 24, 2014). Ninja, a small build system with a focus on speed, [Online]. Available: `https://martine.github.io/ninja/`.

[19] Microsoft. (2015). MSBuild, [Online]. Available: `https://msdn.microsoft.com/en-us/library/dd393574.aspx`.

[20] ——, (2015). Visual Studio – Microsoft Developer Tools, [Online]. Available: `https://www.visualstudio.com/`.

[21] Ľ. Obrátil, "Automated task management for EACirc and BOING", bachelor thesis, FI MUNI, 2015.

[22] S. Chacon and B. Straub, *Pro Git*, 2nd editon. Apress, Dec. 24, 2014, ISBN: 978-1484200773.

[23] N. Pillay, "A genetic programming system for the induction of iterative solution algorithms to novice procedural programming problems", in *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, ser. SAICSIT '05, White River, South Africa: South African Institute for Computer Scientists and Information Technologists, 2005, pp. 66–77, ISBN: 1-59593-258-5. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1145675.1145683`.

[24] CRoCS. (Nov. 1, 2014). Gate circuits · crocs-muni/EACirc Wiki, [Online]. Available: `https://github.com/crocs-muni/EACirc/wiki/Gate-circuits`.

[25] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st ed. O'Reilly Media, Dec. 5, 2014, 336 pp., ISBN: 978-1491903995.