Masaryk University
Faculty of Informatics

GPU-based speedup of EACirc project

Bachelor Thesis

Jiří Novotný

Brno, Spring 2015

Contents

1 Introduction

Random data and the concept of randomness are used in many branches of informatics. However one of the most fundamental use of these principles is in cryptography and IT security. For instance let there be an communication among several entities. The main content of the communication is meant to stay hidden from others 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 individual messages the encryption could be broken and the hidden content of the communication could be read by the attacker. Thus the goal of encryption protocols is that the encrypted messages is not similar or does not have some characteristic traits. In other words, the encrypted messages must look like random data to the attacker. But these constraints are very difficult to provide.

That is why tools have been created to test randomness and thus quality of ciphers. One of these tools is called EACirc and is 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 a 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 much more detail. 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 against CPU is that the CPU is optimized to minimize latency whereas GPU is optimized to maximize throughput. Latency is a time that is needed to start and finish the execution of a single instruction including time ro load necessary data. On the other hand, throughput is a number meaning how much instructions can be processed per one time unit. [2, p. 7] Since some parts of EACirc processes a lot of data with algorithms, which does not need to be optimized for latency[3], the use of GPU's is suitable.

Because GPGPU programming needs a specially enhanced hardware from the manufacturer several different solutions exist on the market which can be used. The

---

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.

solution used for this thesis is called CUDA [3] which is a proprietary technology developed by NVIDIA. [4] The decision to use CUDA was made by my advisor.

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

To set the EACirc project to use the CUDA technology required non-trivial intervention into settings for building the project from the sources, i.e. into the makefiles. This intervention would have resulted in a long-term unmaintainable and chaotic project if the previous workflow was preserved. To prevent this the secondary objective of this thesis is to improve the previous build-system of the EACirc project b using the open-source CMake [5] supportive tool which was developed by Kitware corporation [6].

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. [7]

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 chips, 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.

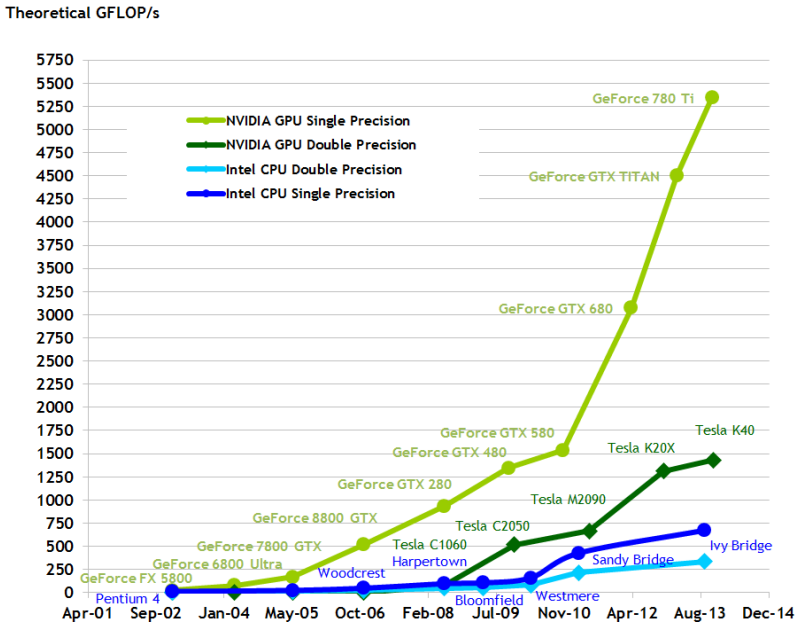

Figure 2.2: GPU vs CPU theoretical throughput. "Floating-Point Operations per Second for the CPU and GPU," NVIDIA. [7]

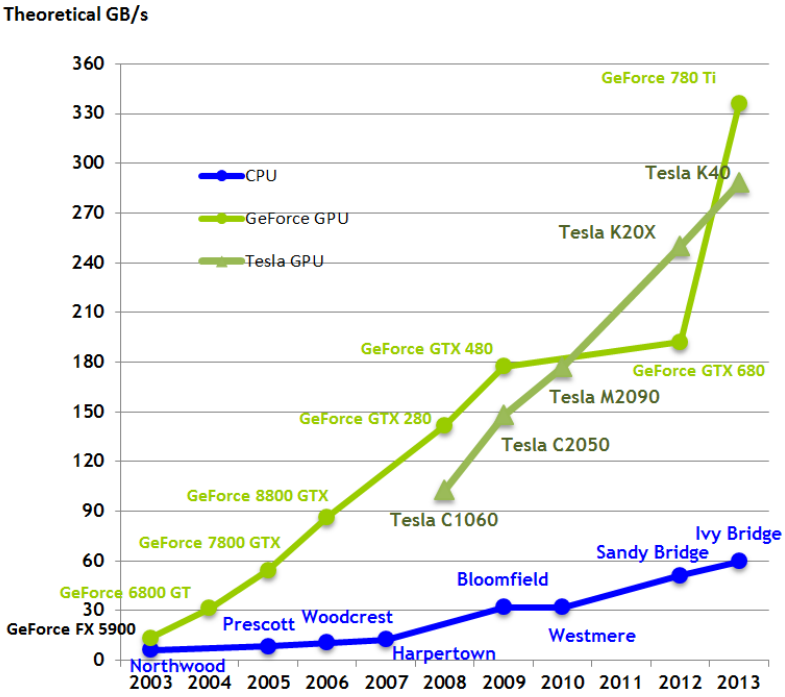

Figure 2.3: GPU vs CPU theoretical bandwidth. "Memory Bandwidth for the CPU and GPU." NVIDIA. [7]

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 is the number of ALUs must be running.

In CUDA terminology the control units are called *Streaming Multiprocessors* (SMs). Each SM has it's own ALUs referred to as *cores*. The single thread vector composes of 32 threads which is called a *warp*. From the SIMT architecture imply that all threads of the warp are performing the same instruction simultaneously at the same time. [7]

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. [8] 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. [7] It is recommended that the kernel processes only the smallest portion of input data that can be processed separately. [8] For instance, when adding two vectors the kernel should just add two corresponding scalars of the vector.

For each kernel that is being run 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 it's position in the block and each block has unique ID dependant on it's position in the grid.[6]

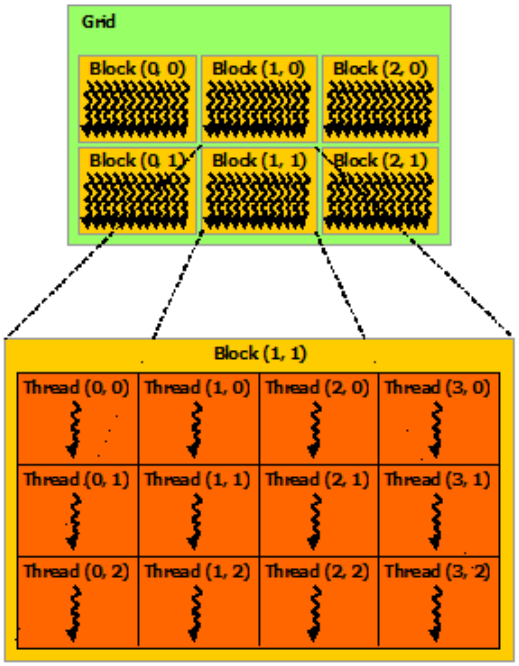

Figure 2.4: "Grid of Thread Blocks." NVIDIA. [7]

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 what is the position of the result in 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] [7] 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. [7]
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. [9] 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 multi level memory model. Each level differs in size, speed, and accessibility. The main levels are *global memory*, *shared memory*, and *local memory*.[13] [7]

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 to the different memory bank resulting into only one transaction. Otherwise the access will 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 [7] 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 [9]
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] [7] 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 a 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. [7]

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. [7]

16. There is also a *texture memory* but the its description is irrelevant for this thesis. For more informations see [7]
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 be overlapped with computation. The host program is then notified whether the asynchronous operation succeeded or failed. [10]

2.5 Compute capabilities

As producing an optimized code for CUDA devices is closely linked to their hardware parameters [9] and almost with every new product line of NVIDIA GPUs a new features are introduced, NVIDIA established a system for backward and a forward compatibility. The GPUs were divided to the classes reflecting the 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. [7]

The listing 1 shows a sample CUDA code of vector addition 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 will be serialized. The execution paths will merge after every thread of the warp executes its execution path. Fortunately some techniques exists to avoid this artefact. This principle also applies on other language control flow constructs as loops, goto statements, and etc. [7]

2.7 Tools

NVIDIA provide the necessary tools for developing on CUDA platform through CUDA Toolkit package. The source files of the CUDA programs must be compiled by a special compiler. The compiler that comes with the CUDA Toolkit is called `nvcc`. [11] 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 the 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 form of binary 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 the code for the host. It requires a special debugger. The one that comes with CUDA Toolkit is called `cuda-gdb`. [12] The interface of this tool is almost the same as the interface of 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 [7] 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

The EACirc sources mainly consists of *C* and *C++* code. The code was divided into reasonably logical sections but the overall structure and concept of the project were monolithic.[1] This led to compilation of all sources into one big executable of approximately 9 MB which took 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 [13] and Linux) special makefile or an IDE specific project file were used which described how to build the application. When a change in the build was introduced, e.g. a new source file was added, the change had to be manually implemented to all makefiles to provide consistency. This workflow was not easy to maintain as the violation of these rules could cause an uncomfortable pitfall.

To solve these problems the CMake [5] tool was integrated into the project of EACirc along with some changes to 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 provides better features they are not so widely supported. For instance CMake generates project files for almost every common IDE and some of those IDEs comes with a built-in support for CMake.

3.1 CMake toolset

The CMake is actually a set of several tools that are taking care of building, testing, and deploying a 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`.

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` [14], or `MSBuild`. [15] 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.

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

---

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 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. At first the user chooses the *source directory* and the *build directory*. Then (s)he 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 own generation of a makefile.[4]

The *source directory* is simply a directory where the project sources are located and 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`. When the user wants to generate project files to a specific IDE, he chooses the appropriate generator – e.g. Visual Studio 2013 [16] on Microsoft Windows [13]. 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 all right 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 as follows:

---

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.e. 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.

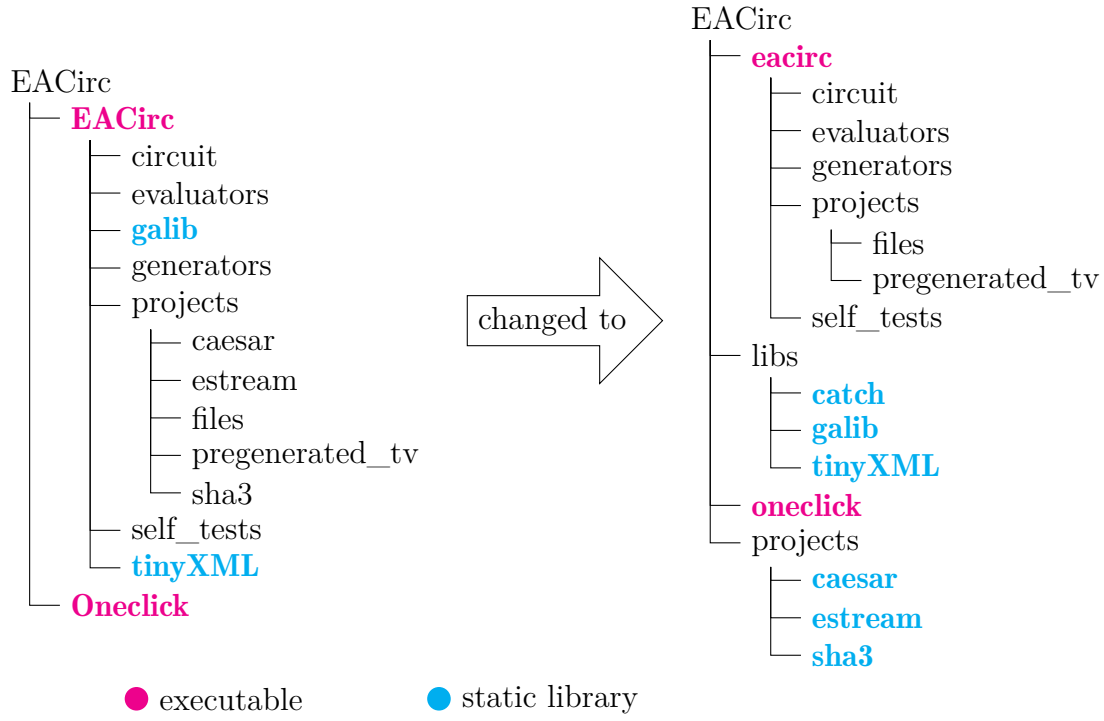

Figure 3.1: Old vs. new repository structure

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 from only one source file, it would be impractical to isolated them. Whereas the big modules `caesar`, `estream`, and `sha3` were moved to the the separate folder called the *projects* folder. Each of the isolated projects was remade to compile into a static library.[5]

The content od 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 automated task management developed by Ľubomír Obrátil. [17]

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. Here is a descriptive list of EACirc specific options:

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

`BUILD_CAESAR` enables building of the Caesar project.

`BUILD_ESTREAM` enables building of the Estream project.

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

`BUID_CUDA` enables to build the support for CUDA devices. This option is available only if the CUDA Toolkit [18] 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 are shown the dependencies of the all build targets.

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.

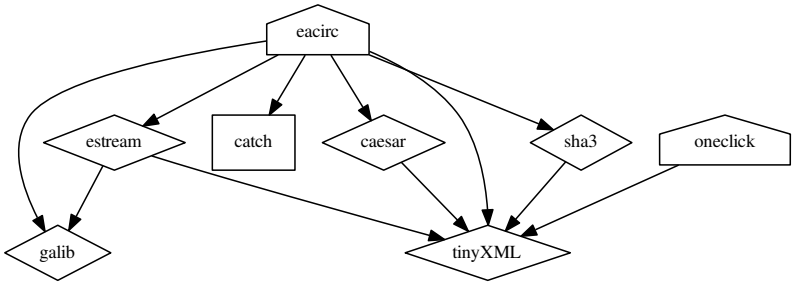

Figure 3.2: EAcirc dependency graph
The static libraries are shown in the rhombus. The executables have a house around them. The square represents an interface library.[7] The direction of the arrows represents that some build target depends on another one.

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 [19]. This means that for the correct build generation git tools must be properly installed on the build machine and found by CMake.[8]

The usage of CMake and the new options of building EACirc are 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 [18] 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 using Nvidia [4] `nvcc` compiler 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.

8. If git tools are installed and not found automatically by CMake then the path to git tools can be specified manually.
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 The Gate Circuit of EACirc

4.1 Functional specification

4.2 GPU implementation

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. (2015). About cuda, [Online]. Available: https://developer.nvidia.com/about-cuda.

[4] N. Corporation. (2015). Welcome to nvidia - world leader in visual computing technologies, [Online]. Available: http://www.nvidia.com.

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

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

[7] NVIDIA, *Cuda c programming guide*, Mar. 2015.

[8] 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.

[9] NVIDIA, *Cuda c best practices guide*, Mar. 2015.

[10] 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).

[11] NVIDIA, *Cuda compiler driver nvcc*, Mar. 2015.

[12] ——, *Cuda-gdb cuda debugger*, Mar. 2015.

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

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

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

[16] ——, (2015). Visual studio – microsoft developer tools, [Online]. Available: https://www.visualstudio.com/.

[17] L. Obrátil, "Automated task management for eacirc and boing", type, FI MUNI.

[18] N. Corporation. (2015). Cuda toolkit, [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

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

TODO Fix the autors of online resources
TODO Fix the titles in the bibliography to dislay big letters correctly.
TODO Cite Lobo's theses about oneclick and fix the source.
TODO Cite Martin Ukrop thesis in Introduction. What is EACirc?