_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Development of a Source-to-Source Compiler for Altera's SDK for OpenCL on FPGAs

Johanna Rohde

Tutor: Rafael Gadea Gironés

Cotutor: Marcos Martinez Peiró

Trabajo Fin de Máster presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Máster en Ingeniería de Telecomunicación

Curso 2015-16

Valencia, 4 de julio de 2016

Abstract:

This master thesis describes the *SOCAO* source-to-source compiler that translates C/C++ input sources into an OpenCL accelerated program. The concept is to accelerate a time consuming software function in a two step process. First it is transferred by the SOCAO compiler into an OpenCL kernel. Afterwards the user can use the *Altera SDK for OpenCL* to translate the kernel into a custom circuit that can be executed on an FPGA. By using the SOCAO compiler, the software engineer can accelerate his program without knowledge in the fields of parallel computing or hardware design. The compiler contains functions and algorithms that analyze the input sources and autonomously make the best suitable decisions for the user.

Results show a 50 % to 280 % increase of speed for three well-known algorithms. Nevertheless this work also addresses the challenges that have to be faced when a sequential program is translated into a parallel environment. Furthermore, the implementation of the compiler, the user-guide and the outcomes of the project are also explained in this work.

Resumen:

Este trabajo fin de máster describe el compilador fuente-a-fuente SOCAO, el cual convierte un programa escrito en C/C++ en un programa acelerado por OpenCL. El concepto es acelerar una función software en dos pasos: Primero SOCAO traslada la función en un kernel OpenCL. Después se usa el "Altera SDK for OpenCL" para crear un circuito integrado para un FPGA. Le ventaja del compilador SOCAO es que el usuario no necesita conocimientos en los campos de computación paralela o diseño de hardware. El compilador contiene funciones y algoritmos que analizan los fuentes de entrada para tomar los mejores decisiones de forma autónoma.

Los resultados muestran un incremento en tiempos de ejecución entre 50% y 280% para tres algoritmos bien conocidos. Sin embargo, este trabajo también explica los dificultades que aparecen cuando un programa secuencial es traducido a un entorno paralelo. Adicionalmente se explica la implementación del compilador, el manual de usuario y los resultados de este trabajo.

Resum:

Aquest treball fi de màster descriu el compilador font-a-font SOCAO, el qual converteix un programa escrit en C / C ++ en un programa accelerat per OpenCL. El concepte és accelerar una funció programari en dos passos: Primer SOCAO trasllada la funció en executar un nucli OpenCL. Després es fa servir el "Altera SDK for OpenCL" per crear un circuit integrat per a un FPGA. Li avantatge del compilador SOCAO és que l'usuari no necessita coneixements en els camps de computació paral·lela o disseny de maquinari. El compilador conté funcions i algoritmes que analitzen els fonts d'entrada per prendre els millors decisions de forma autònoma.

Els resultats mostren un increment en temps d'execució entre 50% i 280% per a tres algoritmes ben coneguts. No obstant això, aquest treball també explica els dificultats que apareixen quan un programa seqüencial és traduït a un entorn paral·lela. A més a s'explica la implementació del compilador, el manual d'usuari i els resultats d'aquest treball.

Contents

List of Figures

Chapter 1

Introduction

Great effort is put into the optimization of speed and energy consumption during the design and implementation of new software. Especially in embedded systems, FPGA accelerators can be used to implement time consuming calculations. Their advantage is their reprogrammability and the low power consumption compared to GPU accelerators. Furthermore, they exploit data parallelism on instruction level since execution is not limited to one *arithmetic logic unit (ALU)*. Nevertheless, their greatest disadvantage is that the accelerator has to be implemented in a hardware description language such as VHDL or Verilog.

In 2013 Altera introduced a software development kit (SDK) capable of synthesizing OpenCL kernels into hardware. This solution shifts the problem into a different domain: Programmers need knowledge in the field of parallel computing. Furthermore, already existing sequential programs have to be rewritten in order to include OpenCL kernels. This problem can be targeted by developing a source-to-source compilers that extracts OpenCL kernels automatically from C/C++ code.

The task of this master thesis was to develop a source-to-source compiler named *SOCAO* that incorporates OpenCL acceleration into a given C program by translating a target function which was marked by the user into an accelerator. The generated code targets Altera's SDK for OpenCL on FPGAs.

The rest of this master thesis is structured as follows: Chapter 2 describes the technical background. Afterwards, Chapter 3 explains the user interface and the over all design of the compiler. The implementation of all analysis and transformation steps is explained in Chapter 4. Finally, the SOCAO compiler is evaluated in Chapter 5.

Chapter 2

Technical Background

2.1 OpenCL

OpenCL is an open programming standard for heterogeneous parallel systems maintained by the Khronos group, [14]. It is designed to support a variety of different platforms such as CPUs, GPUs, DSPs and FPGAs and to simplify the process of integrating these as accelerators into a system. Therefore certain calculations can be passed to different processing devices depending on their characteristics and availability without having to change the code of the program, [12].

The key idea of the OpenCL standard is to provide a common interface and programming language while each vendor adapts the implementation and integration of the OpenCL standard on his own.

The OpenCL architecture can be explained with three models: The platform model, the execution model and the memory model [14, Chapter 3]. Section 2.1.1 to 2.1.3 will describe these models. Afterwards Section 2.1.4 will explain the OpenCL runtime and its usage in order to load and execute an OpenCL kernel.

2.1.1 Platform Model

The platform model describes a generic physical layout of a system using the OpenCL standard. Figure 2.1 shows the schematic.

The system consists of two parts. The *host* and a number of independent *computing devices*. These can be CPUs, GPUs, DSPs or FPGAs. Each device is connected directly with the host. Internally each computing device is assumed to consist of multiple *compute units* which for themselves contain various *processing elements*.

The task of the host is to manage the system. He has to send the data, invoke the kernel execution and retrieve the results.

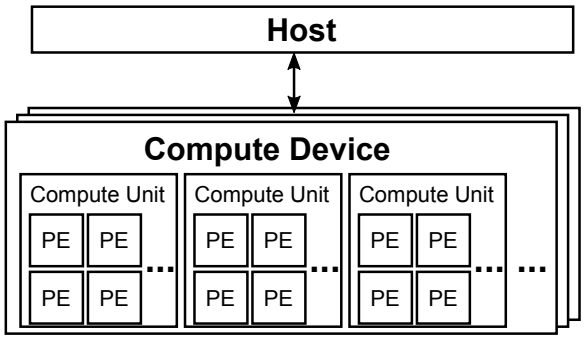


Figure 2.1: OpenCL platform model

2.1.2 Execution Model

Every OpenCL program consists of two parts: The host program executed on the host and kernels which are executed on one or more devices. The execution model describes best how kernels are executed in the OpenCL platform. Figure 2.2 shows this concept. The host program is implemented in any programming language for which an OpenCL API exist. In most cases this is C/C++ but it could also be Java or Python. The host program is compiled by a regular compiler and then executed. The code calls the OpenCL API in order to transfer data and invoke the kernel execution.

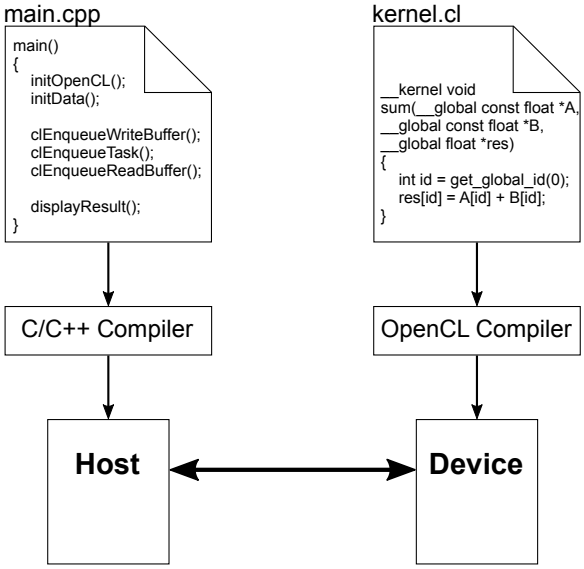


Figure 2.2: File level execution order

The kernel code can be either stored in a separate file or as a string within the host code. It can be compiled beforehand, thus offline, or at runtime, thus online. This can be used to keep the kernel implementation generic during the design phase and compile it when the type of device is known during runtime.

Before executing a kernel, the host has to create a context which contains all objects relevant to the execution. The OpenCL standard defines the content of a context as the following [14, Chapter 3.2.1]:

- **Devices:** The collection of OpenCL devices to be used by the host.
- **Kernels:** The OpenCL functions that run on OpenCL devices.
- **Program Objects:** The program source and executable that implement the kernels.
- **Memory Objects:** A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

Furthermore, the host has to create command queues in order to schedule the execution. The command queues accept three types of commands:

1. **Kernel execution:** Starts kernel execution on a given device.
2. **Memory transfer:** Before and after kernel execution data has to be transferred to and from the device using memory objects.
3. **Synchronization:** All functions concerning the command queues can be called as blocking or non-blocking. Blocking function calls do not return until the command is terminated while in the case of non-blocking calls, the call return directly and events have to be used to synchronize the command queue and to create dependencies between the commands.

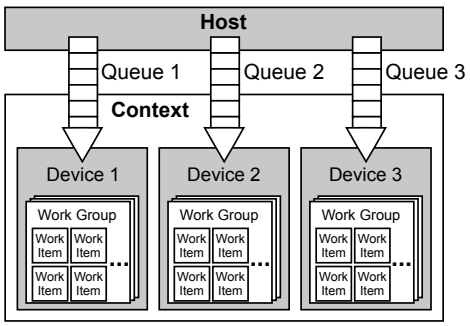


Figure 2.3: Relationship between host, queues, context and devices

Figure 2.3 shows the relationship between host, command queue, context and devices. It has to be noted that every device has its own command queue associated. OpenCL supports task parallel and data parallel programming as well as various combinations of these. A task parallel program is split into individual tasks that can be sent to different accelerators to be executed.

A data parallel program is split into work-groups and work-items in order to be executed on one device. The data can be arranged as an N-dimensional problem, a so called `NDRange`. N has to be a value between one and three. Figure 2.4 shows a two dimensional problem. Within the NDRange, work-groups are arranged in a matrix. According to the number of dimensions they are assigned x, y and z coordinates. Within the work-group work-items are arranged accordingly. A kernel for a data parallel problem will access its work-group and work-item id and do further calculation based on these values. The number of work-items and the group size is set by the host dynamically.

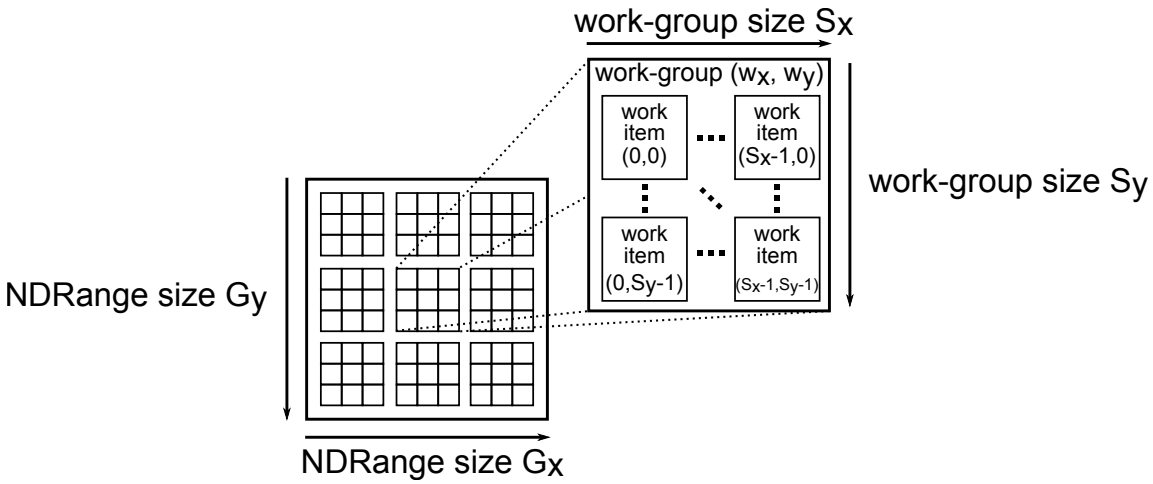


Figure 2.4: Two dimensional problem split into work-groups and work-items [14]

This scheme can be used to split code with nested `for`-loops into individual problems that can be executed in parallel.

```
1 void sum(int *A, int *B, int *↩
      C, int N)
{
3   for(int i =0; i<N; i++)
        C[i] = A[i] + B[i];
5 }
```

Listing (2.1) Serial program

```
1 __kernel void sum(
    __global const int *A,
3   __global const int *B,
    __global int *C)
5 {
    int idx = get_global_id(0);
7   C[idx] = A[idx] + B[idx];
}
```

Listing (2.2) Kernel program

Figure 2.5: Vector addition implemented as regular program and OpenCL kernel

Figure 2.5 shows an example implementation for a function adding two vectors. The regular implementation is shown on the left in Listing 2.1. The `for` loop does not show any data dependencies between the iterations. Therefore, they can be executed independently. Inside the OpenCL kernel the `for`-loop is replaced by calls to the OpenCL API in order to determine the position of the current work-item. Then a single addition is executed.

2.1.3 Memory Model

The memory model explains the different memory types that exist in the OpenCL standard. The memory space is split into four groups: The global, constant, local and private memory, [14, Chapter 6.5]. Furthermore, the host has its proper memory to which the computing devices have no access. Figure 2.6 shows how the address spaces are arranged.

The *global memory* is shared between the host and the computing device. It is the biggest memory but also the slowest since all work groups and work items within the same device have access to this memory space. Its space is allocated by the host system. The global memory is mainly used to transfer data from the host system to the kernel and vice

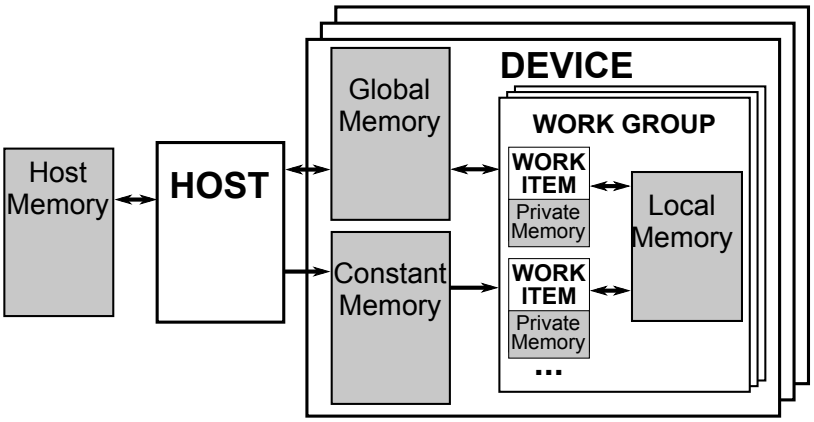


Figure 2.6: OpenCL memory model

versa. Each device has its own global memory. Global variables or parameter are declared with the `__global` modifier. They have to be either a function parameter, declared at file scope or a static variable inside a function.

The *constant memory* can only be written by the host and can be seen as a read-only memory from the perspective of the accelerator containing constant values that don't change over the time of execution. Pointer to constant memory can either be passed as a function parameter by the host or be declared in the outer most scope of a function or at file scope. If the pointer is not passed by the host, the variable marked as constant has to be initialized during declaration and the values have to be compile time constant. Constant variables are declared with the `__constant` modifier.

Each work group has access to its own *local memory*. This memory is not accessible by the host system but the host system can reserve local memory for the work groups before executing a kernel. Each work group is responsible on its own for transferring data from the global memory to the local memory and vice versa. The space of the local memory is usually limited. Local variables are declared with the `__local` modifier. They can be declared at three positions: As function paremeter, at file scope or at the outer most scope within a function definition. For the two later the space has to be compile time constant. Otherwise a pointer to the local memory can be passed as kernel argument by the host.

The smallest but also fasted memory is the *private memory*. It is associated with a single work item an saves all intermediate results needed by one thread. Private memory can be declared with the `__private` modifier when needed. Otherwise, all variables declared without address modifier within the kernel function are automatically marked as private. Private memory can not be allocated dynamically and its size therefore has to be constant.

2.1.4 OpenCL Runtime

The program flow of the host program can be split into three phases: The initiation of the OpenCL system, the execution of the kernel and the cleanup.

The steps taken during the initiation phase can be seen in figure 2.7 and are now explained to further detail:

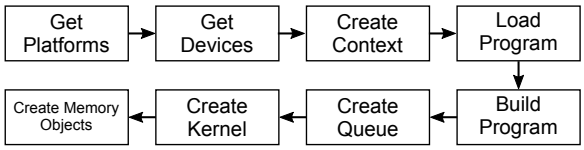


Figure 2.7: Program flow for OpenCL initiation

Get Platform First the host has to find all available OpenCL platforms using the `clGetPlatformIDs()` function.

Get Devices Using one of the platform ids the host can call the `clGetDeviceIDs()` function in order to find all computing devices available on this platform.

Create Context A context is created by passing one or multiple device ids to the `clCreateContext()` function. As described in section 2.1.2, the context is used to manage devices, programs, kernels and memory objects.

Load Program All OpenCL programs contain kernel and helper functions as well as constant data that is used by these functions. OpenCL programs can be compiled online or offline for the target device. Therefore, they can be loaded either as source or as a binary with the `clCreateProgramWithSource()` or `clCreateProgramWithBinary()` functions respectively.

Build Program The `clBuildProgram()` function is called in order to build the program executable.

Create Queue In Section 2.1.2 it was already described that command queues are used in order to schedule tasks such as transferring data, executing kernels and synchronizing tasks. They are assigned a context and a device when they are created with the `clCreateCommandQueue()` function.

Create Kernel Using a given program executable, a kernel object can be created by referring to the kernel function name. To do so, the OpenCL standard provides the `clCreateKernel()` function.

Create Memory Objects The last step is to create the memory objects necessary to transfer data to or from the computing device. Memory objects can be *buffer* or *images*. Buffers are used to store elements in a sequential order. Images are 4-component vectors. Their data storage format might vary from the format used inside the kernel. Since the compiler that is developed in this project only uses regular buffer objects, image objects are not explained to any further detail. Buffers are created with the `clCreateBuffer()` function. They have to be assigned a context and a size. A flag can be passed to specify allocation and usage information. Sometimes the creation of the memory objects is not

done until the execution of the kernel, especially when the size of the buffer is not known during the initiation phase.

Memory can be allocated as shared or as non-shared memory. In case of shared memory, an additional pointer has to be mapped to the buffer. This is done by enqueuing the `clEnqueueMapBuffer()` function. The return value is a pointer which can be parsed and assigned to a pointer in the host program.

Listing 2.3 shows an example for an initiation function under the assumption that there is only one platform with one device present. The source file *opencl_kernel.cl* is loaded and built at runtime and the kernel function *do_something_kernel* is created. An additional status value is checked after each step. If the execution was not successful, the host program is terminated with an error message.

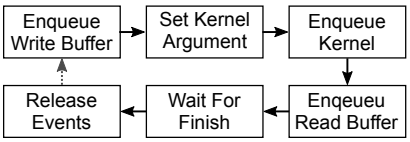


Figure 2.8: Program flow for kernel execution

The steps need in order to execute an OpenCL kernel can be seen in Figure 2.8 and are explained in the following:

Enqueue Write Buffer In the beginning the host has to transfer all data to the computing device. There are two options to do this, depending whether shared or non-shared memory is used:

For non shared memory the `clEnqueueWriteBuffer()` function is called. Among others, the function accepts a command queue reference, a memory object and a pointer to a local buffer in the host memory. The command queue determines which device the data will be written to. The data transfer can be executed as blocking or as non blocking operation. In the case of a non blocking function call, an event reference can be set as function parameter where the command queue will set the current execution status. Furthermore, in order to indicate dependencies to earlier command queue tasks, the function accepts a number of events which have to be finished before the transmission process can be started.

For shared memory a simple `memcpy` ca be used in order to copy the data from the host memory to the shared pointer.

Set Kernel Argument The function arguments of the OpenCL kernel are set by the `clSetKernelArg()` function. An argument can be either a memory object or a variable like an integer. In the case of the later, the value is transferred as constant value to the kernel.

Enqueue Kernel The execution of the kernel is started through the command queue. Depending whether a task parallel or a data parallel program was designed, a kernel can be executed as *task* by calling `clEnqueueTask()` or as *NDRange* by calling `clEnqueueNDRangeKernel()`. These functions accept like every process that is enqueued in

```
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
cl_mem input_buf, output_buf;
float input[100], output[100];

void init_opencl() {
  cl_int status;
  char *program_buffer;
  size_t program_size;

  status = clGetPlatformIDs(1, &platform, NULL);
  checkError(status,"Failed to load platform");

  status = clGetDeviceIDs(platforms,CL_DEVICE_TYPE_ALL,1,&device,NULL);
  checkError(status,"Failed to load device");

  context = clCreateContext(0,1,&device &oclContextCallback,0,&status);
  checkError(status,"Failed to create context");

  queue = clCreateCommandQueue(context,device,0,&status);
  checkError(status,"Failed to create command queue");

  program_size = getFileSize("opencl_kernel.cl");
  program_buffer = readSourceFile("opencl_kernel.cl");
  program = clCreateProgramWithSource(context,1,(const char **)&↩
    program_buffer,&program_size,&status);
  checkError(status,"Failed to create program");

  status = clBuildProgram(program,0,NULL,"",NULL,NULL);
  checkError(status,"Failed to build program");

  kernel = clCreateKernel(program,"do_something_kernel",&status);
  checkError(status,"Failed to create kerne");

  input_buf = clCreateBuffer(context,CL_MEM_READ_ONLY,100*sizeof(float),↩
    NULL,&status);
     checkError(status,"Failed to create buffer for input A");

  output_buf = clCreateBuffer(context,CL_MEM_WRITE_ONLY, 100*sizeof(float)↩
    ,NULL,&status);
     checkError(status,"Failed to create buffer for output");
}
```

Listing 2.3: Initiation Function for OpenCL

a command queue an array of events which have to be terminated before executing the current one. Usually this are the events associated with the data transmission to the memory buffer. The `clEnqueueNDRangeKernel()` function additionally accepts information about the number of work items that have to be created and how many of them have to be assigned to one work group. The execution of a kernel is always a non blocking command. Therefore an event has to be assigned in order to determine when the execution has finished.

Enqueue Read Buffer After the execution of the kernel has finished the results have to be transmitted back to the host memory. The `clEnqueueReadBuffer()` command behaves similar to the `clEnqueueWriteBuffer()` function explained earlier. In case of shared memory the result can be accessed directly via the shared memory pointer. That assumes that the user waits for the kernel to finish before accessing the memory.

Wait for Finish It is recommended to use the `clWaitForEvents()` functionality in order to await that all enqueued tasks have been completed.

Release Events All events have to be released after they are no longer needed in order to prevent unwanted behavior. Afterwards the execution process can start again.

Listing 2.5 shows an example function which runs the in Listing 2.3 loaded kernel as a task. The function prototype can be seen in Listing 2.4.

```
1 __kernel do_something_kernel(float *input, float *output, int size);
```

Listing 2.4: OpenCL function prototype

2.2 FPGA

Field Programmable Gate Arrays (FPGAs) are reprogrammable integrated circuits. They can be reconfigured at anytime to match any desired logical function or application. Thereby, they differ from application specific integrated circuits (ASICs) which are manufactured to perform one predefined task.

2.2.1 Physical Layout

The physical design of a FPGA is based on thousands of reconfigurable logic elements so called adaptive logic modules (ALMs) as well as DSP and RAM blocks, [5]. Figure 2.9 shows how the elements are placed in a matrix based arrangement and are surrounded by I/O logic that forms the interconnection between the FPGA and other external resources.

Every ALM has up to eight one bit inputs. The internal logic is reconfigurable to any logic function that has one or two output bits. Optionally these bits can be stored in registers before leaving the logic block. Therefore a group of ALMs can form arithmetic

```
cl_platform_id platform = NULL;
cl_device_id device = NULL;
cl_context context = NULL;
cl_command_queue queue = NULL;
cl_program program = NULL;
cl_kernel kernel;
cl_mem input_buf;
cl_mem output_buf;
float input[100], output[100];

...

void run_opencl()
{
  cl_int status;
  cl_event kernel_event, finish_event, write_event;

  status = clEnqueueWriteBuffer(queue, input_buf, CL_FALSE, 0, 100 * ↩
    sizeof(float), input, 0, NULL, &write_event);
  checkError(status,"Failed to transfer input");

  status = clSetKernelArg(kernel, 0, sizeof(cl_event), &input_buf);
  checkError(status,"Failed to set argument");

  status = clSetKernelArg(kernel, 1, sizeof(cl_event), &output_buf);
  checkError(status,"Failed to set argument");

  status = clSetKernelArg(kernel,2 ,sizeof(int), 100);
  checkError(status,"Failed to set argument");

  status = clEnqueueTask(queue, kernel, 1, &write_event, &kernel_event);
  checkError(status,"Failed to launch kernel");

  status = clEnqueueReadBuffer(queue, output_buf, CL_FALSE, 0, 100 * ↩
    sizeof(float), output, 1, &kernel_event, &finish_event);
  checkError(status,"Failed to transfer output");

  clWaitForEvents(1, &finish_event);

  clReleaseEvent(write_event);
  clReleaseEvent(kernel_event);
  clReleaseEvent(finish_event);
}
```

Listing 2.5: Execution of an OpenCL kernel

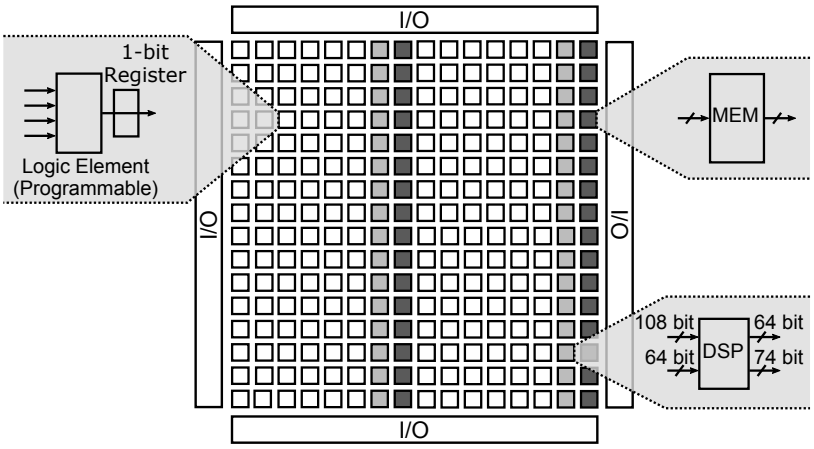


Figure 2.9: Basic FPGA layout

operations such as addition, multiplication or shifts as well as custom operations specified by the user.

DSP blocks are specialized blocks which accept a greater number of input bits and have a cascade of arithmetic functions preimplemented. They are used in order to speed up more complex calculations and in order to save area compared to an implementation using ALMs. The user can define the internal interconnects between the arithmetic function blocks in order to adapt the output function. Due to their size only a small number of DSP blocks is integrated into each FPGA.

Dedicated memory blocks serve as on-chip RAM or ROM in order to store greater amounts of data. They are also configurable and can behave as single-, simple-dual- or true-dual-port memory.

The blocks are interconnected with wires. It can be seen in Figure 2.10 that a horizontal and vertical net of routing resources lies between the grid of logic blocks. The interconnects between the wires are also programmable. This way the data bits are routed between logic cells.

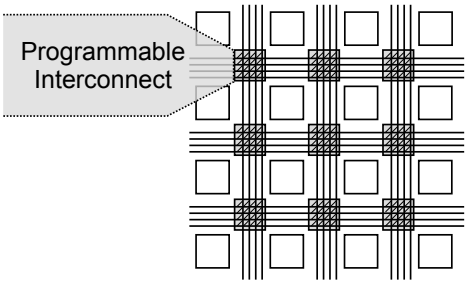


Figure 2.10: Reconfigurable interconnections between logic cells

2.2.2 Instruction-Level Parallelism

In the process of translating software code to hardware, every operation is mapped to an independent arithmetic module which for itself can be implemented using the resources

described in Section 2.2.1. These modules are connected in execution order to pass the results from one operation to the following one.

The advantage of executing calculations this way is that all modules can operate independently from each other. According, operations that are not directly depending on each other can be calculated at the same time. This form of parallelism is called instruction-level parallelism, [2].

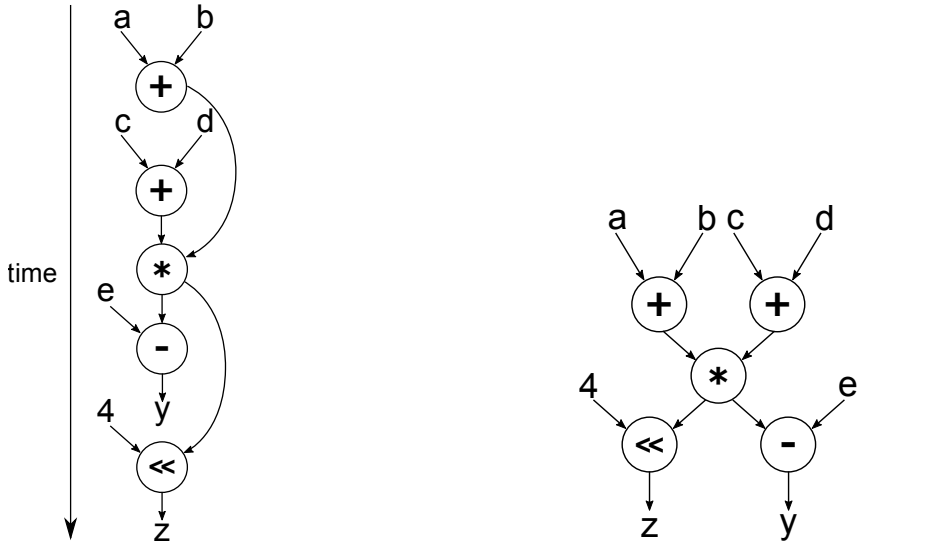
```
1 x = (a+b)*(c+d);
  y = x << 4;
3 z = x - e;
```

Listing 2.6: Example Program

Listing 2.6 shows an example code which uses four input values in order to calculate an intermediate variable. This value is used in order to calculate the result values y and z.

The execution order of a regular CPU can be seen in Figure 2.11a. Since every instruction has to be fetched separately, a bottleneck is created and the code has to be executed in a serial manner.

The data flow graph of the instruction-parallel execution order is displayed in Figure 2.11b. It can be seen that the addition of the variables a and b is independent of the addition of the variables c and d. Therefore, they can be executed at the same time. The same holds true for the shift operation in line 2 and the subtraction in line 3. They only depend on the result of the multiplication. Therefore they are also calculated in parallel. The direct comparison of both graphs shows that the calculation can be reduced from five to three steps.



(a) Serial execution order

(b) Instruction parallel execution order

Figure 2.11: Data flow graph to example program in Listing 2.6

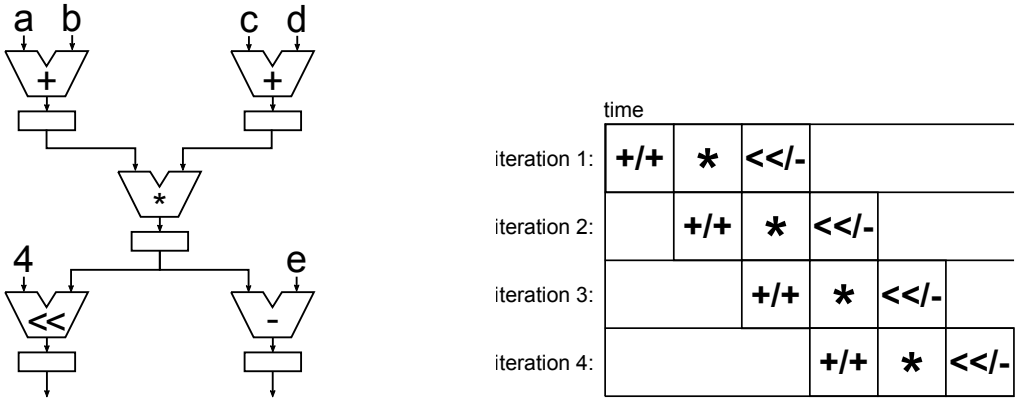
2.2.3 Pipelining

Most designs for FPGAs are implemented on the register transfer level (RTL). The *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis* defines this abstraction level as following: "Register transfer level is a level of description of a digital design in which the clocked behavior of the design is expressly described in terms of data transfers between storage elements in sequential logic, which may be implied, and combinational logic, which may represent any computing or arithmetic-logic-unit logic."[1]

This means that in the RTL the knots of the data flow graph are implemented as logic modules with tailing registers forming a pipeline of multiple stages. The data passes through one stage each clock cycle leaving the remaining stages idle in the meanwhile.

In situation where the same calculation has to be performed multiple times independently, a new set of data can be inserted into the pipeline each clock cycle.

Figure 2.12a shows the pipeline created from the data flow graph depicted in Figure 2.11b. It can be seen that the pipeline has three stages. The corresponding timing diagram for four iterations is shown in Figure 2.12b. After the pipeline is completely filled, a new result is produced every clock cycle.



(a) Pipeline at register transfer level

(b) Execution of filled pipeline

Figure 2.12: Pipelining the example shown in Listing 2.6

2.2.4 Altera's DE1-SoC

Altera's DE1-SoC board is a development board for system on chip (SoC) applications. It contains a hard processor system (HPS) and a Cyclone V FPGA. The characteristics of the HPS are summarized in Table 2.1. The processor is a dual core ARM Cortex-A9 which runs at a 800 MHz clock and has access to 1 GB RAM, [16]. The development board has a micro SD card slot which allows the user to boot an operating system.

The Cyclone V product family comes in different sizes and packages. Table 2.2 shows that the FPGA used in the DE1-SoC board has 85 k logic elements and 32 k adaptive logic modules (ALMs). It contains two types of memory cells. First, *M10K* cells that are dedicated memory resources that hold 10 kb each and can be configured as RAM or ROM. Second, *Memory logic array blocks (MLAB)* that are distributed memory cells which

Table 2.1: Hard processor system characteristics

Characteristic	
Processor	Dual-core ARM Cortex-A9
Clock	800 MHz
Memory	1 GB DDR3 SDRAM

Table 2.2: Cyclone V characteristics

Characteristic	
Ordering Code	5CSEMA5F31C6
Logic Elements	85000
ALM	32075
M10K	397
MLAB	768
Memory total	4,45 Mb
DSP Blocks	87

consists of 10 ALMs each that have been configured accordingly. This leads to a total of 4450 kb. Finally the FPGA has access to 87 DSP blocks [8].

There are four different possibilities to reconfigure the FPGA, [16]:

- From the quad serial configuration device (EPCQ256) which is a flash memory on the board.
- From the HPS system.
- While booting the HPS system with an image stored on the SD card.
- Via the JTAG chain using the *Quartus II programmer* on the development system.

2.3 OpenCL for FPGAs

FPGAs are usually programmed on register transfer level using a hardware description language such as Verilog or VHDL. Highly specialized tools translate the code into logic functions and map them onto the FPGA structure. A binary file is produced and can be used by the user to reconfigure the FPGA in the field.

Due to the highly fine grained and massive parallel nature of FPGAs, these programming languages follow their own logical order. This introduces a great barrier for developers who want to incorporate FPGA acceleration into their system.

The idea of programming FPGA accelerators with OpenCL and using a specialized compiler to automatically create hardware tries to close this gap. Nevertheless, it is important to understand how the high level description of an algorithm is translated into hardware in order to exploit the features of an FPGA and prevent bottlenecks.

The OpenCL compiler creates pipelines as described in Section 2.2.2 and 2.2.3 in order to exploit instruction parallelism. Section 2.1.2 explained that OpenCL kernel can be executed

as a range of work-items. These work items fill the pipeline exploiting pipeline parallelism. For small designs, the kernel pipeline can be duplicated two or more times in order to increase the number of work-items that are processed in parallel [6].

Not only work-items can be pipelined but also loops. The OpenCL compiler moves the evaluation of the exit condition to the beginning of the pipeline [7, Page 1-33]. In an ideal case, a new loop iteration is started every clock cycle but so call *loop-carried dependencies* can prevent this. These are situation in which the result of an earlier iteration is needed for a later one.

```
1 double A[10], B[10];
  A[0] = 1;
3 for(int i = 1; i < 10; i++)
  {
5   double x = B[i];
    double res = 0.3*x*x+0.8*x+1.2;
7   A[i] = (A[i-1]+res)/2;
  }
```

Listing 2.7: Example `for`-loop with loop-carried dependencies

An example can be seen in Listing 2.7. In line 7 the loop accesses the value `A[i-1]` which is produced in the previous iteration.

In order to assure that the variable is written and read in the correct order, a so called *initiation interval (II)* is introduced. The initiation interval is defined as the number of clock cycles between two consecutive iterations. It is obvious that it is desirable to keep the initiation interval as small as possible for good kernel performance. The ideal value is one.

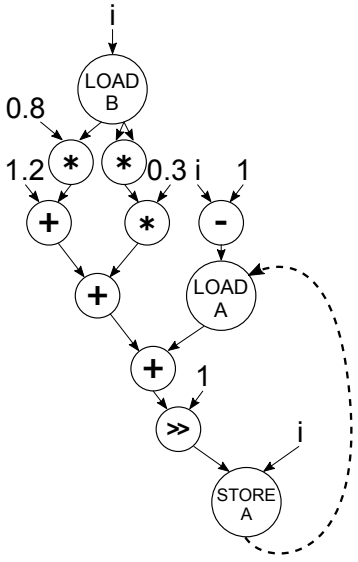


Figure 2.13: Data flow graph to example program in Listing 2.7

Figure 2.13 shows the data flow graph to the example in Listing 2.7. It can be seen, that the loop body can be translated into a pipeline with seven stages. The dashed line is not a data flow edge but rather indicates the dependency between the store and the load operation.

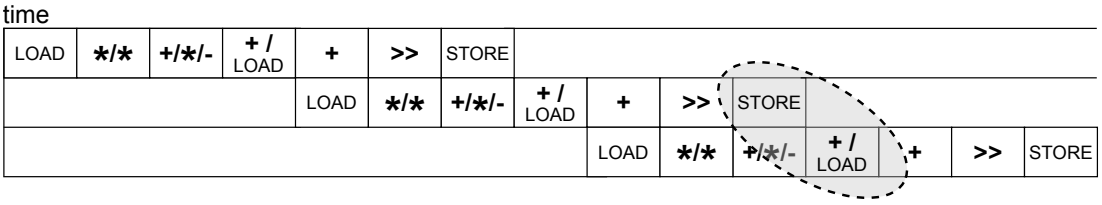


Figure 2.14: Pipeline to example program in Listing 2.7

The seven stages of the pipeline are shown in Figure 2.14. The loop-carried dependency is taken into account by delaying the second iteration by four clock cycles. This way that the load operation is executed after the store operation. Hence, the loop has an initiation interval of four. It is obvious that this pipeline works less efficient than the ideal pipeline in Figure 2.12b. It is important to remember that the II influences the execution time linearly with the number of iterations whereas the length of the pipeline only serves as an offset that is does not influence the execution time anymore once the pipeline is filled

2.4 Altera's SDK for OpenCL

Altera's software development kit (SDK) for OpenCL was first introduced in 2013 and it is conform to the OpenCL specification 1.0 [9, Appendix].

The tool flow of the SDK can be seen in Figure 2.15. The main part is the *Altera Offline Compiler (AOC)* which compiles and synthesizes the OpenCL kernel into a *xxx.aocx* FPGA configuration file that is loaded by the host program in order to reprogram the FPGA [9, Chapter 1]. Furthermore, it generates a number of log and report files which contain information about the created pipelines, the occupied area inside the FPGA and the achieved clock frequency. On the left side of the figure it can be seen that Altera's embedded design suit (EDS) cross compiler has to be used to compile the host program into an executable for ARM processors.

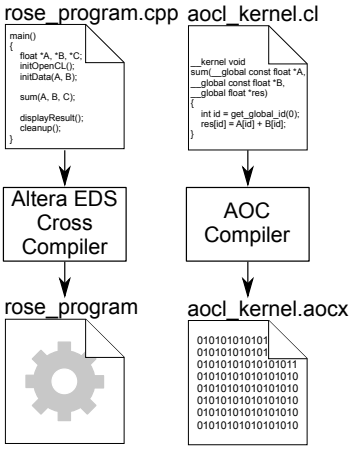


Figure 2.15: Altera SDK for OpenCL tool flow

Additionally, Altera provides a custom header files and the `aocl_utils` namespace in order to include system specific functions into the host program. These include routines in

order to load the *.aocx* file into the FPGA, check errors and to find a OpenCL platform by name.

2.5 Compiler

Compiler are specialized tools that are used to translate, thus *compile*, a program written in one programming language into another one. The input language are in most cases high level programming languages such as C or Java while the output language can either be another high level language or a low level machine language.

This section gives a conceptual overview over the design of compiler systems. In the beginning the general structure of a compiler is explained. Afterwards Section 2.5.2 and 2.5.3 focus on how programming languages are defined and what effect this has on the intermediate representation.

2.5.1 Structure of a Compiler

Compiler are complex systems that need good structuring in order to be manageable. Usually a compiler is split into three components which represent three phases of the compilation process and which are executed consecutively. They are called *front-end*, *middle-end* and *back-end*, [20, Chapter 1]. Figure 2.16 shows the conceptual structure of a compiler.

Front-end The front-end is used to interpret the input program and to transfer it into an intermediate representation. Furthermore, it it assures the syntactic correctness of the program.

Middle-end In the middle-end optimizations and transformations are taking place. Usually the aim of each compiler is to improve the efficiency of the program by reducing execution time and/or memory consumption. During the transformation, the intermediate representation is preserved and serves as interface to the back-end.

Back-end The back-end translates the program from its intermediate representation into an output language. In most cases this refers to machine code and the output file is an executable. In case the output is another high level programming language, the compiler is called a *source-to-source compiler*. If the aim is to compile output files in different output languages separate back-ends have to be implemented.

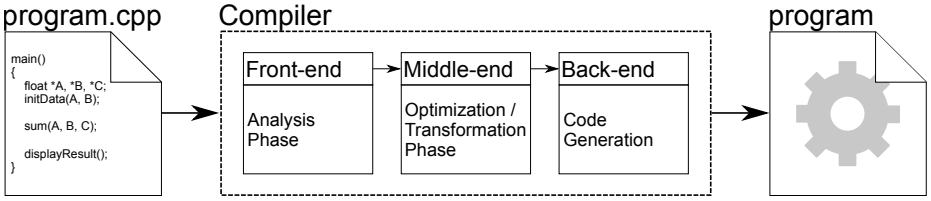


Figure 2.16: Structure of a compiler

The advantages of this structure are the clean separation of tasks and the precise interfaces. The intermediate representation is ideally chosen independently of the input and output language so that the front-end and back-end are replaceable. Nevertheless, the strict separation is often just the ideal case.

2.5.2 Context-Free Grammars

Before understanding how the intermediate representation of a compiler is chosen, it is important to understand how programming languages are designed.

The syntactic structure of a program is always described by a context-free grammar which defines a number of symbols, so called *nonterminals*. According to a given number of rules, they can be substituted by other symbols or regular expressions . Symbols that can not be replaced any further are called *terminals*. They are normally either keywords such as *if, =,* or *;* or they are references to variables and values.

R. Wilhelm gives in [20, Chapter 3.2.1] the following definition for a context-free grammar:

A context-free grammar (CFG for short) is a quadrupel G = (V_N, V_T, P, S) where V_N, V_t are disjoint alphabets, V_N is the set of *nonterminals*, V_T is the set of *terminals*, $P \subseteq V_N \times (V_N \cup V_T)^*$ is the finite set of *production rules* $S \in V_N$ is the *start symbol*.

In a simple example a context-free grammar could look like the following:

$$
\begin{array}{lcl}
S & \rightarrow & Stmt \\
Stmt & \rightarrow & \boldsymbol{id} = Exp; \mid Stat; \; Stat \mid \\
 & & if(Exp)\{Stmt\} \; else\{Stmt\} \\
Exp & \rightarrow & Exp + Exp \mid Exp - Exp \mid Exp * Exp \mid \\
 & & Exp \; / \; Exp \mid \boldsymbol{val} \mid (Exp) \mid \boldsymbol{id}
\end{array}
$$

Each line represents one production rule. The right hand side defines a nonterminal symbol while the left hand side shows a set of alternatives separated by the | character with can be used to substitute the symbol. The start symbol is S and the terminals are keywords or regular expressions for **id** in case of a variable name or **val** in case of a value.

Therefore, a simple expression might be derived in the following steps:

$$
\begin{array}{l}
S \\
Stmt \\
\boldsymbol{id} = Exp; \\
x = Exp * Exp; \\
x = (Exp) * \boldsymbol{id}; \\
x = (Exp + Exp) * b; \\
x = (\boldsymbol{id} + val) * b; \\
x = (a + 1) * b;
\end{array}
$$

The process starts with the start symbol `S` which is replaced by the `Stmt` symbol for which in turn the `id = Exp;` alternative is chosen. This process continues until all symbols

are replaced and the final expression `x=(a+1)*b;` is derived.

2.5.3 Intermediate Representation

The parser takes the input program and splits it into symbols representing keywords, variables and values. During the syntax analysis a tree is constructed by mapping the symbols onto the syntactic structure defined by the context-free grammar. The tree is called *syntax-tree* or *parse-tree*, [18, Chapter 2.2.1].

The root of the syntax-tree is the start node of the grammar. Descending from there each node represents a symbol of the left hand side choice made in the down casting process. Finally, the leafs are the terminals.

The `x=(a+1)*b` statement given as as example in Section 2.5.2 is parsed into the syntax-tree shown in Figure 2.17.

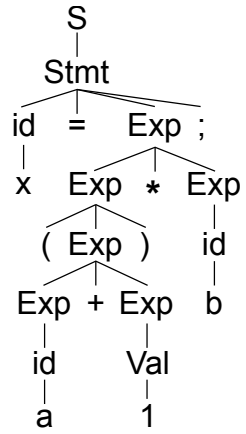


Figure 2.17: Syntax-tree for statement `x=(a+1)*b;`

The normal syntax-tree contains a lot of redundant and unnecessary information which is no longer important after parsing. Especially keywords like parentheses and semicolons are only useful during the syntax analysis in order to structure the syntax-tree itself but do not have any effect on the output of the compiler thereafter. Therefore, most compiler use a more dense form of the syntax-tree as intermediate representation, the so called *abstract-syntax-tree (AST)*, [18, Chapter 2.16.2].

The AST representation does not contain unnecessary nodes. Furthermore, nodes are grouped in order to compress the information they are holding. This facilitates iterating over the syntax-tree during the transformation and optimization phase in the middle-end. The syntax-tree shown in 2.17 can therefore be represented by the AST shown in Figure 2.18.

The example shows that the statement node and its sub nodes where reduced to one *AssignStmt* node. The same was done for the *MulExpr* node representing the multiplication and the *AddExpr* node representing the addition. The information about the actual variable identifier and the specific value were merged with the nodes identifying the type of expression to be an identifier or value respectively.

Most compiler which are written in object oriented programming languages group the

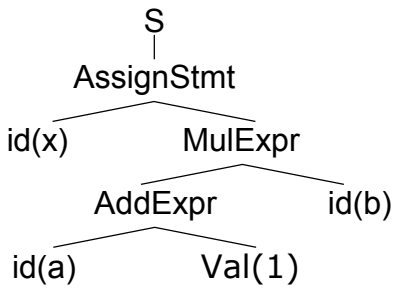


Figure 2.18: Abstract-syntax-tree for statement `x=(a+1)*b;`

different node types and implement them as inherited classes. In this case the *AssignStmt* node would be an object of a child class to the class representing the *Stmt* node.

The advantage of the AST representation is that it can be easily converted into the output language since it preserves the structure of the input grammar. Nevertheless it lacks information about the actual program flow. Optimizations which rely on the execution order of the statements are hard to implement using only the AST representation. Therefore, most compiler switch for these type of analysis to a control-flow based representation.

For the analysis and optimization algorithms implemented in this work, the AST representation is sufficient. For more information on control-flow-graphs see [15, Chapter 5]

2.5.4 ROSE Framework

The ROSE framework is an open-source compiler framework developed at the Lawrence Livermore National Laboratory. It is written specifically for the development of source-to-source compilers and supports a number of different input language, among others C, C++, Python and Java [19].

Most importantly the ROSE framework contains implementations of the front- and back-end as well as a variety of tools for transformation and analysis of the AST. The user can include the library into his own project and only has to implement the routines that analyze and modify the AST.

The intermediate representation (IR) is called SageIII and is auto-generated by the ROSETTA tool which is distributed with the ROSE Framework. This tool generates classes for all IR nodes that are necessary to support the variety of input and output languages. The edges between the nodes as depicted in Figure 2.18 are implemented as pointer references within the nodes. Furthermore the SageIII classes already implement a number of functionalities needed to store or retrieve information as well as to copy nodes with all their sub-nodes. A full list and description of all classes can be found in [4].

Figure 2.19 shows the AST from the example in Section 2.5.3 when parsed into the SageII intermediate representation. It can be seen that all class names start with `Sg` for Sage. The expression statement class `SgExprStatement` encapsulates an expression. Therefore the edge has to be pointing to an `SgExpression` object. All binary operations such as assigns, multiplications and additions are called `Sg`*XXX*`Op` whereas the *XXX* is substituted by the operation. They are derived from the `SgBinaryOp` class which in its turn is a child of the `SgExpression` class. The `SgVarRefExp` class represents a variable reference while the

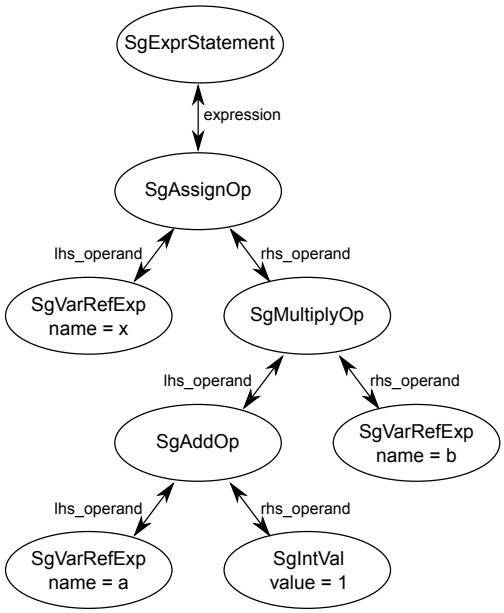


Figure 2.19: Syntax-tree for statement `x=(a+1)*b;`

`SgIntVal` class stands for an integer value.

A complete AST contains more complex constructs as the example shown in Figure 2.19. Besides the edges that are determined by the grammar, a number of additional edges and nodes are added in order to link information. This might be information about the data type of a variable as well as a reference to the declaration statement or the parent node. Incorrect or missing references can lead to segmentation faults during further AST processing or can prevent the back-end from compiling the program. Therefore the ROSE framework provides the `SageBuilder` and `SageInterface` namespaces. The `SageBuilder` namespace provides high level functions that create additional AST nodes and which set the references automatically according to information passed via the function arguments. The `SageInterface` on the other hand contains functions which can be used to insert or delete statements or replace whole sub-trees.

Chapter 3

Design

The SOCAO compiler adds an additional layer on top of the development flow described in Section 2.3. As shown in Figure 3.1, the developed source-to-source compiler is a tool that takes a C/C++ source file as input and automatically generates an Opencl host program and a kernel. The kernel is designed to be executed as a task. The program structure of the input program is automatically adapted to include the OpenCL routines described in Section 2.1.4 to load and run the kernel. The output files can be compiled and executed as if they were handwritten. The compiler targets specifically Altera's SDK for OpenCL.

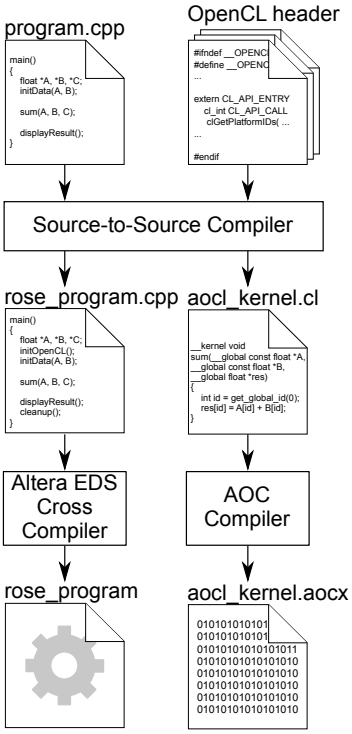


Figure 3.1: File input and output

The extraction of the kernel is function based. This allows better handling within the compiler because a function declaration is always a root node to an AST containing all further information like the function arguments and body. Therefore, only the root node

has to be passed to the analysis and transformations.

The idea is to move the execution of a function that is considerably time consuming to an FPGA computing device. All function calls are redirected to calls to the OpenCL API. The decision of which function is to be targeted is made by the user.

It is important to note that the output name of the host program can be chosen freely whereas the kernel is always written to a file named *aocl_kernel.cl*. This behavior is intended since the compiled and synthesized kernel always has to be named *aocl_kernel.aocx* in order to be found and loaded by the host program.

3.1 User Interface

The user can use the SOCAO compiler following the steps shown in Figure 3.2. The process consists of two parts: The preparation of the source file and the execution of the source-to-source compiler using the console terminal. The preparation will be described in Section 3.1.1 while the required console commands are explained in Section 3.1.2.

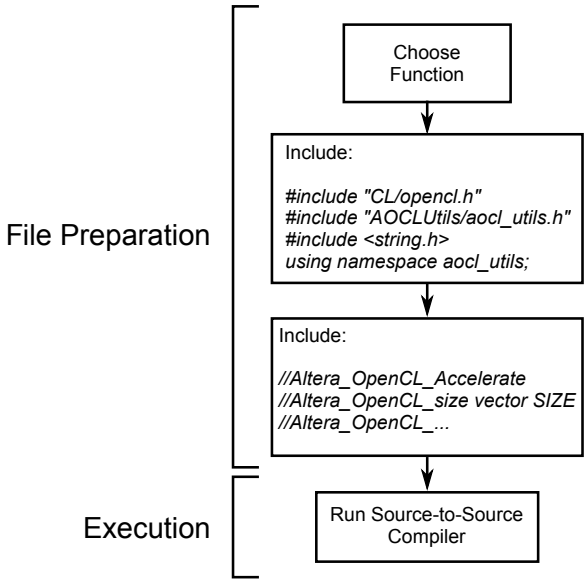


Figure 3.2: Use diagram

3.1.1 File Preparation

In the beginning the user has to choose which part of his program he wants to accelerate. He has to outline this code into a separate function and replace the affected lines with a function call. Furthermore, the user has to include the general and Altera specific OpenCL header files and the `aocl_utils` namespace. This is necessary for the compiler to find OpenCL specific structures and functions. Additionally the *string.h* header is needed in order to have access to the `memcpy()` function.

The compiler recognizes a number of comments that have to be placed in front of the target function. The user has to use these in order to pass necessary and optional

information that allow the compiler to optimize the kernel. The order of the comments has no importance as long as the same configuration is not made twice. In this case the first comment is ignored.

The list of the comments is:

Altera_OpenCL_Accelerate The `Altera_OpenCL_Accelerate` comment marks the function which is supposed to be accelerated. A program may only contain one target function.

Altera_OpenCL_size [NAME] [SIZE] As a default value the compiler assumes that every pointer used within the function points to an array with a single entry. In order to adjust the size this comment has to be used. `[NAME]` has to be the name of the pointer and `[SIZE]` has to be a single expression that can either be an integer number, a define identifier or a variable. The compiler therefore supports static and and dynamic vector sizes. Due to restrictions in the parser, a size given as a define identifier is not considered constant even though the identifier might be associated with a constant value. Better optimizations can always be achieved when the size is known at compile time.

Altera_OpenCL_max_size [NAME] [VALUE] The compiler can optimize the memory management within the OpenCL kernel when the size of an array is statically known. In some cases the size might be dynamic but an upper boundary is known to the user. In this case this information can be passed to the compiler using the `Altera_OpenCL_max_size` comment. The `[VALUE]` parameter has to be an integer value. Note that the actual vector size has to be known within the kernel. This is the case when the `Altera_OpenCL_size` comment is used with a variable name as size parameter and when the variable itself is read within the function. If this is not the case, the compiler issues a warning and ignores the comment. Further information about the memory management can be found in Section 4.1.9.

Altera_OpenCL_const_val [NAME] [VALUE] If the user is aware of any constant values that are defined outside of the target function, he can use this comment. `[VALUE]` can be a valid char, string, double or integer value as long as it fits the datatype of the variable.

Altera_OpenCL_const_vec [NAME] This comment announces a compile time constant array accessed by the function. In this case the whole array definition is copied into the OpenCL kernel file reducing the execution time since these values no longer have to be moved to the computing device memory by the host before kernel execution. The effect of copying constant arrays into the file scope of the kernel is evaluated in Section 5.7.

Altera_OpenCL_runtime_const [NAME] The compiler can reduce data transmission times when a buffer is not changed after the first call to the target function. Is this case, the `Altera_OpenCL_runtime_const` attribute can be assigned to an array. Note that the size of this array can be dynamically defined by using a variable. The change of its value

after the first call to the target function will have no effect. More information on how the compiler reduces the buffer transmission times can be found in Section 4.2.4

Altera_OpenCL_update_flag [NAME] [FLAG] This comment assigns a flag that indicates whether an array has been modified or not. In this case the elements of the buffer belonging to the `[NAME]` variable are only updated if `[FLAG]` evaluates to true. `[FLAG]` has to be a variable name or define identifier. No expressions are allowed due to parsing restrictions of the ROSE framework. If the user wishes to use an expression he has to assign this expression first to a define identifier and use this instead.

Altera_OpenCL_soc Indicates that the target platform is a system-on-chip platform like the DE1-SoC. This influences the memory allocation strategy explained in Section 4.1.9.

3.1.2 Console Interface

The source-to-source compiler is run via console terminal. As described in Section 2.5.4, the framework is compiled into a dynamic library that has to be loaded at runtime. Therefore, the user has to add the ROSE installation folder to the `PATH` and the `LD_LIBRARY_PATH` environment variables before running the compiler. He can do this using a script similar to the one shown in Listing 3.1 whereas he has to substitute `[path_to_installation]` with the path to the corresponding installation folder.

```
ROSE_INS=[path_to_installation]
PATH=$PATH:$ROSE_INS/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROSE_INS/lib
export PATH LD_LIBRARY_PATH
```

Listing 3.1: Example ROSE setup script

When calling the compiler the user has to make sure the compiler can find all header files especially those belonging to the OpenCL API. It is not necessary to include source files besides the main program since no actual compilation step is performed.

The ROSE framework itself proposes a variety of command line options but most of them are irrelevant to the user who wishes to perform an OpenCL transformation. Nevertheless, the following options might be helpful:

- **-rose:o [FILENAME]** sets the file name of the unparsed C++ code. This can also include a relative or absolute path. The default name is *rose_xxx.cpp* whereas *xxx* refers to the input file name.
- **-I[PATH]** include path indicating the compiler where to look for header files besides the source folder.

A whole list of command line options is available running the source-to-source compiler with the `--help` option.

A sample compile command can be seen in Listing 3.2. It is assumed that the environment variable `ALTERAOCLSDKROOT` points to the Altera SDK installation directory while all header files included as step two of the use diagram shown in Figure 3.2 are placed in a folder with the relative path *../common/inc*.

```
./socao -I../common/inc -I$ALTERAOCLSDKROOT/host/include program.cpp
```

Listing 3.2: Example compile command

3.1.3 Input File Restrictions

Even though the OpenCL language used to write the kernels is based on the C99 standard it includes several limitations. The SOCAO compiler is not able to overcome these without changing the outcome of the program. Moreover Altera's SDK for OpenCL and the design of the source-to-source compiler itself also imply certain limitations. This imposes several restrictions to the user who has to make sure that his input code is conform to the following rules:

Multiple files. In the current version the compiler assumes that the `main()` function and the target function are written in the same file.

C-Function. Altera SDK for OpenCL does not support C++ for OpenCL in their kernels. Therefore the target function may not use any C++ specific constructs such as objects or namespaces.

Function calls. The target function can contain function calls. Mathematical and `printf` function calls specified in [9] are detected automatically. All remaining function calls are inlined. Therefore, the function definition has to be accessible to the compiler and no recursive functions are allowed. In case the inlining process is not successful, the compile process is aborted. In accordance with the restriction on C++ constructs, object method calls are not allowed. Furthermore, OpenCL does not allow the kernel to allocate dynamic memory. Therefore calling `malloc()` is also forbidden.

Restricted Pointer. Restricted pointer management is key to a good kernel performance in OpenCL for FPGA. Therefore every pointer must be handled as if it was `restrict`. Any operation violating the `restrict` keyword is not allowed. This includes copying pointer such as shown in line 3 of Listing 3.3. Note that the data referred to by two individual pointer is transferred to two distinct buffer creating two equal copies. Furthermore, after kernel execution both buffer are copied back to the same address causing the later to overwrite the first. Nevertheless the user might take advantage of this if the values are read-only.

Pointer Arithmetic. Some users find it to be more efficient to use pointer manipulation within their code. Even though changing the value of a pointer does not necessarily violate the principle of restricted pointers, it might cause performance problems by creating

```
1 int A[15];
  int *B;
3 B = A;
```

Listing 3.3: Forbidden direct pointer assignment

loop-carried dependencies. Furthermore pointer arithmetic - such as incrementing or decrementing a pointer - is strictly not allowed within any inlined function. Further explanations can be found in Section 4.1.1.

Pointer to pointer. Pointer to pointer of any kind are not allowed since the actual value would not be copied to an address space accessible by the kernel. This causes the pointer to be invalid.

Two dimensional arrays. Two dimensional arrays are allowed as long as the values are saved in consecutive order in the memory and the width of the array is set during function declaration. OpenCL does not allow two dimensional arrays as function parameter. Therefore any two dimensional array is flattened and their references are converted to one dimension. Two dimensional arrays implemented as array of pointers are not supported as this is a case of the previously mentioned pointer to pointer restriction. The implementation of this transformation can be found in Section 4.1.2.

Structures and typedefs. Structures and typedefs are allowed with some restrictions. The declaration has to be accessible to the compiler. Structures may not contain pointers, since these would point to an address space not accessible to the kernel. Furthermore, the Altera SDK does only support pointer to a single structure element as kernel argument, [6, Chapter 1.52].

Function pointer. Function pointers are not allowed as they are not supported by OpenCL [14, Chapter 6.8].

Reserved names. The SOCAO compiler adds two functions and various file scope variables to the source files. Their names are reserved.

Long and long long variables. The OpenCL standard defines the bitlength of `long` variables to be 64 bit, [14, Chapter 6.1.1], whereas the ARM processor of the DE1-SoC defines them to be 32 bit. This leads to malfunction. The user is advised to use the `int64_t` data type instead since its size is equal in both systems.

`size_t` function parameter. OpenCL does not allow function parameter to be of type `size_t`.

3.2 System Design

As described in Section 2.5, every compiler consist of three major parts: The front-end, the middle-end and the back-end. Figure 3.3 shows that the SOCAO compiler can also be split into these three blocks.

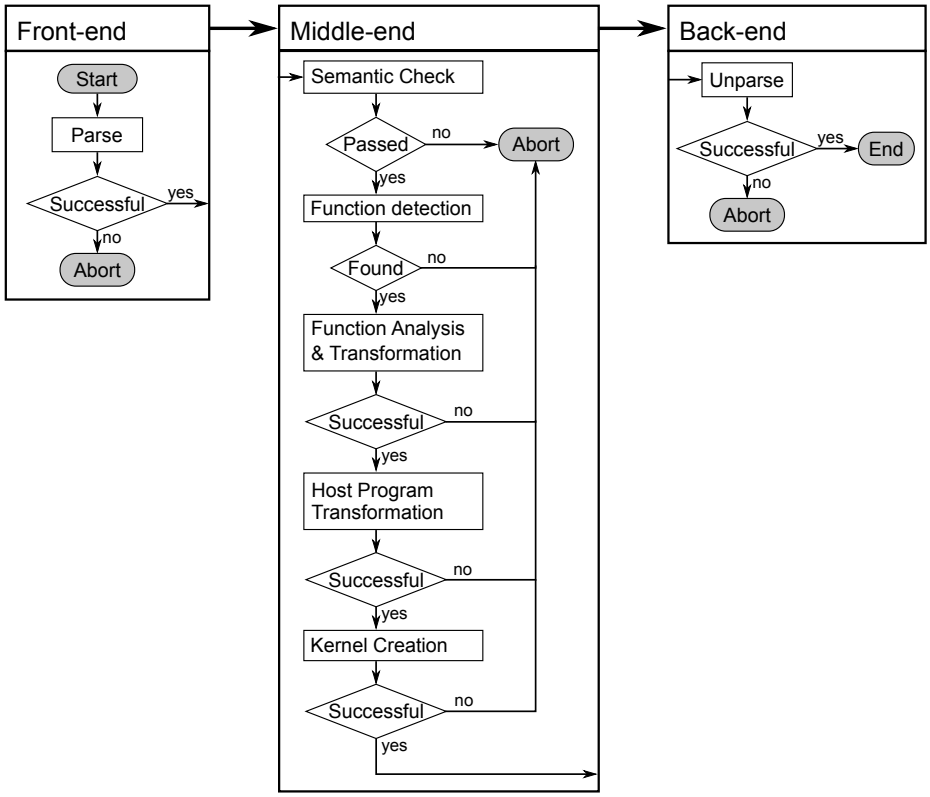


Figure 3.3: Design diagram of the SOCAO compiler

The compile process starts with the front-end by parsing the input code into an AST. In case of syntax errors, the parsing can not be terminated successfully and the compile process is aborted.

Afterwards the AST is analyzed and modified in the middle-end. The first step is to run an semantic check in order to assure that the code is free of static semantic errors. Afterwards the function marked by the `Altera_OpenCL_Accelerate` comment is detected.

During the *function analysis & transformation* phase, all information necessary to build an OpenCL kernel and a corresponding host program are collected. At this point most of the information given by the user are taken into account in order to make the best decisions on the structure of the kernel.

The next step is the *Host Program Transformation* phase. The original code of the target function is removed from the AST in order to be replaced by the kernel call. The AST is then further modified by inserting all variables and function calls described in Section 2.1.4.

In the final step of the middle-end, the kernel is created. The formerly removed code is inserted into a new function and modified where needed. This function is unparsed separately from the rest of the project and written to the *aocl_kernel.cl* file.

The transformation process can only pass from one phase to another if the previous one was completed successfully. Is this not the case, an error message is issued to the user, the whole AST is dumped and the compile process is stopped.

In the back-end the unparser finally reverts the conversion made by the front-end. In contrast to regular compilers, this source-to-source compiler does not compile the AST into an executable but generates new source code. Nevertheless some information such as spacing and formatting are lost during the AST generation process and therefore do not appear in the output file.

Chapter 4

Implementation

The system described in Chapter 3 is implemented using the ROSE framework described in Section 2.5.4. This library has all functionalities required by the front- and the back-end as well as the semantic check already implemented. This chapter focuses therefore on the manipulation and transformation of the abstract-syntax-tree(AST).

Figure 3.3 showed three major blocks in the middle-end: First the analysis and transformation of the target function, followed by the transformation of the host program and the creation of the OpenCL kernel. The implementations of these tasks are explained in this chapter to further detail.

4.1 Analysis and Transformations

Ten analysis and transformation steps are performed on the target function. Each one of them is explained in a separate section. Section 4.1.11 finally describes the dependencies between the steps and shows the final program flow.

4.1.1 Inline Transformation

The inline transformation is used to inline all function calls within the target function. It is the transformation that does the greatest changes to the AST of the target function.

The ROSE Framework already provides an inliner that can be called using the `doInline()` function. This function accepts a reference to a function call expression and returns a boolean value that indicates whether the inlining was successful or not.

The preimplemented inliner copies the AST of the function body into a separate basic block at the position of the function call and adds local variables for the function parameter. Afterwards these variables are renamed by adding a trailing number to their name in order to prevent errors from similar naming. Every function call generates copies of the function parameters, therefore the newly created variables are assigned the values from the function call expression. In order to implement various return statements, a label is added to the end of the function basic block and every return statement is replaced by a `goto` statement. If the function has a return expression, an assign statement is added before the `goto` statement.

The preimplemented inliner showed one problem: If the function header contained pointers, their values would be copied into internal variables. As explained in Section 3.1.3 this is against the principle of restricted pointer management.

In order to fix this problem, an additional restriction was opposed to the user by forbidding changes to the pointer itself within any inlined function. Now, assuming that the pointer are read-only values, the pointer passed by the function call can be used directly within the basic block of the inlined function code.

Since the inliner had to be modified, the original code was copied from the ROSE framework's sources to a separate file which was compiled with the sources of the SOCAO compiler. All relevant functions were renamed in order to prevent conflicts and the corresponding lines were adapted in order to implement the above explained behavior.

```
1  void vector_process(char *input,↩
       char value)
   {
3    int i;
     for(i = 0; i < 64; i++)
5      input[i] += value;
   }
7
9  void vector_update(char *input, ↩
     int ilen)
   {
11 ...
     while( ilen >= 64 )
13   {
       vector_process(input, 'c');
15     input += 64;
       ilen  -= 64;
17   }
   ...
19 }
```

Listing (4.1) Original program

```
1  void vector_update(char *input, ↩
     int ilen)
   {
3  ...
     while( ilen >= 64 )
5    {
       {
7        char value_1 = 'c';
         int i;
9        for(i = 0; i < 64; i++)
           input[i] += value_1;
11     }
       input += 64;
13     ilen  -= 64;
     }
15 ...
   }
```

Listing (4.2) Result after inlining function call

Figure 4.1: Example of an inlined function

An example of the functionality of the inliner can be seen in Figure 4.1. On the left it shows the original program where the `vector_update()` function contains a call to the `vector_process()` function. After the inlining, the function is located between the brackets in lines 7 to 12. The char value is copied into a scope variable whereas the pointer is not copied in order to prevent multiple pointer from pointing to the same data. No `goto` statements were added since the original function does not contain any return statements.

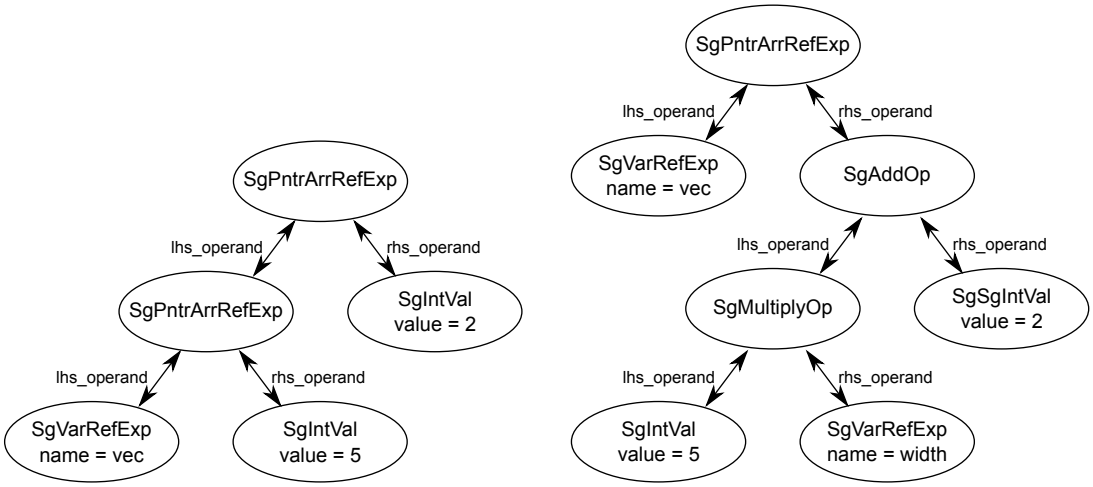
4.1.2 2D to 1D Transformation

The OpenCL standard does not allow two dimensional arrays as function parameters. Therefore, every two dimensional array has to be transformed to a one dimensional array. According to the restrictions made to the user in section 3.1.3, it can be assumed that

every two dimensional array is stored in a sequential manner. Therefore, expressions like `vec[a][b]` can be replaced by `vec[a*WIDTH + b]`.

In the AST, array reference expression nodes are implemented by the `SgPntrArrRefExp` class. They are binary operations. Therefore, the left hand side operand of this objects always points to the variable expression while the right hand side operand points to the index expression. For a two dimensional array reference, the left hand side points to another array reference expression. Figure 4.2a shows the AST of the example expression `vec[5][2]`.

The transformation is implemented as a separate pass over the target function. Every node is compared to the above described pattern. In case of a match a new `SgPntrArrRefExp` object with the pattern shown in figure 4.2b is created. The left hand operand now points directly to the variable reference node while the index is composed of an addition and a multiplication operator.



(a) AST for expression vec[5][2] (b) AST for expression vec[5*width + 2]

Figure 4.2: Access pattern to data arranged in two dimensions

The information about the width of a vector is obtained by analyzing the array type object corresponding to the variable. The helper function `getArrayDimensions()` returns a vector of expressions used in the declaration of the variable. It is assumed that the user has declared at least the second dimension of the array, otherwise the transformation can not be completed successfully.

The `SageInterface` provides a `replaceExpression()` function which replaces one expression node and its sub tree with another expression node.

Constant arrays are excluded from the transformation since their declaration is copied directly to the file scope of the kernel function and they are therefore never passed as function parameter.

4.1.3 Constant Value Transformation

The constant value transformation uses the information given by the `Altera_OpenCL_const_val` comments to substitute variable reference with their constant values.

The first step is to parse the comments into their components. A vector of pairs mapping the names of the variables to their values is then passed to the `replaceConstCommentValues()` function of the SOCAO compiler.

The transformation pass traverses over the AST of the target function and tries to downcast every node into a variable reference. If it succeeds the variable name is compared to the variable names in the list. When a corresponding entry is found, the value string is parsed into a value object. The subtype is determined by the type of the variable.

This optimization was implemented in order to give the user the chance to notify the compiler about constant values which are declared in the file scope or as function parameter. Since the kernel is compiled separately from the host code, the AOC can not find these constant variables by running constant propagation.

Listing 4.3 shows an example of a function which sums two entries of an array and divides the result by a factor. The factor is a constant declared in line 1. The `Altera_OpenCL_const_val` can be seen in line 6. Listing 4.4 shows the result. In this particular case the transformation has great impact on the performance of the kernel since the AOC will implement a division by two as a right shift which can be executed in less time and with less resources.

```
1  const int factor = 2;
2
3  //Altera_OpenCL_Accelerate
4  //Altera_OpenCL_size in size
5  //Altera_OpenCL_size out size
6  //Altera_OpenCL_const_val factor↩
       2
7  void do_something(int *in, int *↩
       out, int size)
8  {
9    for(int i=1; i<size; i++)
10   {
11     out[i-1]=(in[i-1]+in[i])/↩
       factor;
12   }
13 }
```

Listing (4.3) Original function

```
1  const int factor = 2;
2
3  //Altera_OpenCL_Accelerate
4  //Altera_OpenCL_size in size
5  //Altera_OpenCL_size out size
6  //Altera_OpenCL_const_val factor↩
       2
7  void do_something(int *in, int *↩
       out, int size)
8  {
9    for(int i=1; i<size; i++)
10   {
11     out[i-1]=(in[i-1]+in[i])/2;
12   }
13 }
```

Listing (4.4) Result after transformation

Figure 4.3: Example for constant value transformation

4.1.4 Constant Folding Transformation

During the constant folding transformation, expressions containing only constant values are evaluated, thus folded. This is normally the second step after constant propagation where references of variables are replaced by their constant value.

The transformation is already implemented by the ROSE Framework and can be accessed through the `SageInterface`. The `SageInterface::constantFolding()` function accepts an AST node as parameter and starts from there descending downwards to look for unary and binary expressions that can be folded.

This transformation is only applied to the target function and its outcome is limited since the aoc compiler itself has constant folding implemented too. Nevertheless, some other analysis and transformation work more efficiently when constant expressions are already folded limiting the amount of possible cases that have to be matched in order to search for constant expressions.

4.1.5 Input/Output Analysis

The input/output analysis determines which variables are read and written within the target function. The results are used in order to determine all function arguments of the kernel function and to manage the data transfer to and from the kernel.

The analysis is implemented using the visitor pattern [13, Chapter 5].Different visit functions are implemented for the main child classes of the `SgNode` class. Depending on the exact subclass of a node it can be determined what kind of expression is present and what operands are read or written.

For example, a `SgAssignOp` node can always be split into the left hand side operand which is written and the right hand side operand which is read. On the other hand most `SgBinaryOp` nodes such as additions and subtractions only contain operands that are read. Nevertheless, every operand itself can be an expression. Therefore, the helper function `findBaseVariable()` was implemented. This function filters expressions that are variable references and strips array, dot and arrow operands.

The ROSE Framework provides the `varID` class in order to identify variables based on their `SgInitializedName` object. This class links a `SgVarRefExp` node to the variable's definition. The `varID` found in the visitor functions are written into two different vectors: One for variables that are read, thus input variables, and one for variables that are written, thus output variables. The same variable can appear in both vectors if it is read and written.

It has to be noted that direct read and write operations to pointer are ignored. This is caused by the fact, that only the data stored to an address can be returned by the target function but never the pointer itself since it would point to a not accessible address space.

Figure 4.5 shows an example function which accumulates a given vector onto a result vector. The result vector is declared at file scope and therefore not an input argument of the target function. The results of the input/output analysis are the following:

Input Variables: {`i, size, res, a`}

Output Variables: {`i, res`}

It can be seen that the variables `size` and `a` are only read within the function. The variable `i` is the iteration variable and therefore read and written. Line 9 also contains a read and write access to the variable `res` which therefore is also listed in both lists.

4.1.6 Constant Array Analysis

The constant array analysis, searches the comments placed over the target function for the `Altera_OpenCL_const_vec` keyword. Then a list is created containing the variable symbols of all constant arrays.

```
double *res;

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size a size
//Altera_OpenCL_size res size
void vec_accumulate(double *a, int size)
{
  for(int i = 0; i < size; i++)
    res[i] = res[i]+a[i];
}
```

Listing 4.5: Vector accumulate example function for input/output analysis.

In order to check whether a string with a variable name coming from a comment line actually matches any constant variable, the `lookupVariableSymbolInParentScope()` function is called and the variable symbol is retrieved. The array declaration will be copied into the *aocl_kernel.cl* file during the phase of kernel creation. This assumes that the array is initialized at the declaration point. Therefore, the symbol is only added to the list of constant arrays when the pointer to the initializer object is set to a non null location.

Constant arrays have to be treated differently by a number of analysis. For example, they are excluded from the parameter list of the function header as well as from the 2D to 1D array transformation.

4.1.7 Typedef Analysis

The typedef analysis checks the type of all referenced variables within the target function and detects those whose type pointer points to a `SgTypedefType` node. Then the definitions of these types are added to the kernel file.

The analysis itself returns a string that contains all typedef declarations. This string is later prepended to the unparsed kernel before it is written to the file.

A special case are the integer types defined in the *cstdint* header such as `int32_t` or `uint64_t`. Their definitions depend on the target system in order to assure that the base type has the correct bit width. The OpenCL standard itself defines the bit width of every build-in scalar data type [14, Chapter 6]. This information was used to create corresponding definitions to all data types from the *cstdint* header. Since they are used quite frequently it was decided to include them in every OpenCL file per default. The full list of definitions can be seen in Listing 4.6.

4.1.8 Parameter Analysis

The parameter analysis determines all function parameters of the kernel and generates for each an `OpenCL_InOut` object that contain all information. There exist three positions which have to be analyzed for function parameter: the function parameter from the target function, the variables used in the return statement and the results of the input/output analysis described in Section 4.1.5. The input/output analysis does not distinguish between variables declared inside and outside the target function. It is obvious that only the later

```
typedef long         intmax_t;
typedef ulong        uintmax_t;
typedef char         int_least8_t;
typedef uchar        uint_least8_t;
typedef short        int_least16_t;
typedef ushort       uint_least16_t;
typedef int          int_least32_t;
typedef uint         uint_least32_t;
typedef long         int_least64_t;
typedef ulong        uint_least64_t;
typedef char         int_fast8_t;
typedef uchar        uint_fast8_t;
typedef short        int_fast16_t;
typedef ushort       uint_fast16_t;
typedef int          int_fast32_t;
typedef uint         uint_fast32_t;
typedef long         int_fast64_t;
typedef ulong        uint_fast64_t;

typedef char         int8_t;
typedef uchar        uint8_t;
typedef short        int16_t;
typedef ushort       uint16_t;
typedef int          int32_t;
typedef uint         uint32_t;
typedef long         int64_t;
typedef ulong        uint64_t;
```

Listing 4.6: Typedef definitions to match data types declared in the *cstdint* header.

has to be added to the list of function parameters.

```
struct OpenCL_InOut{
  const char *var_name;
  SgType *type;
  bool is_buffer;
  const char *buffer_name;
  unsigned long flags;
  bool size_is_const;
  size_t size;
  bool size_is_define;
  const char *size_param;
  bool has_max_size;
  size_t max_size;
  bool is_global_ref;
  bool is_trans_pntr;
  bool local_cpy;
    bool use_shared_memory;
  bool is_runtime_const;
  bool has_update_flag;
  const char *update_flag;
};
```

Listing 4.7: OpenCL_InOut struct used to collect information of the used function parameter

The definition of the `OpenCL_InOut` struct can be seen in listing 4.7 and contains the following fields:

var_name The name of the original variable.
type Reference to the data type information.
is_buffer This flag is set when the variable is a pointer or an array and the data has to be transferred using a memory object.
buffer_name This field is set in combination with the `is_buffer` flag. It identifies the name of the memory object used in the host code. The name is composed of a prefix, the name of the variable and a suffix. The prefix can be either `input_`, `output_` or `inout_` depending on the results of the input/output analysis. The suffix is always `_buf`.
flags The flag describes whether a variable is read-only, write-only or read-write. As described in Section 2.1.4 OpenCL defines a number of flags which categorize this behavior and that can be used when creating memory objects. Even though the compiler uses the flags defined by the OpenCL standard, this value is set not only for buffer but also for scalar variables. The information is taken from the input/output analysis.
size_is_const Set to true when the value passed by the `Altera_OpenCL_size` comment is an integer value.
size Static number of elements stored in a pointer or array.
size_param Name of the size parameter if it is not an integer value.
has_max_size Set to true when the user passes an `Altera_OpenCL_max_size` comment.
max_size The maximum size of a vector.
is_global_ref Set to true when the array has to be declared at file scope of the kernel. This is the case for constant vectors.
is_trans_pntr In some cases the target function writes scalar variables defined in the file scope of the input source file. In order to retrieve the updated value from the kernel,

a memory object with only one entry has to be created. This leads to inconsistencies within the function statements, since the variable now has to be treated as a pointer. In order to avoid replacing all variable references, the transparent pointer concept was created. A transparent pointer is a function parameter whose first element is copied to and from a local variable at the beginning and at the end of the kernel. Therefore the pointer is not visible to the original function code. Is such a situation detected, the `is_trans_pntr` flag together with the `is_buffer` flag are set to true.

use_const_memory If set to true, the memory object will be stored to the `__constant` address space.

local_cpy Is set when the data of the buffer can be copied into the local memory of the computing device. This flag is evaluated during the memory analysis explained in Section 4.1.9.

use_shared_memory If set tu true, shared memory between host and OpenCL has to be used to transfer the data.

is_runtime_const Is set to true if the user passes a `Altera_OpenCL_runtime_const` comment with name of the variables.

has_update_flag Is set to true if the user passes a `Altera_OpenCL_update_flag` comment associated with this variable name.

update_flag Value passed by the `Altera_OpenCL_update_flag` comment.

4.1.9 Memory Analysis

The memory analysis determines for every buffer which address space is used to transfer the data, how the buffer is allocated and how the data is saved inside the kernel.

The data can be transferred using either the global or the constant address space. Altera states that the global address space implements dedicated logic in order to compensate for long access delays and to exploit consecutive memory access. On the other hand, data located in the constant address space is loaded into a cache the first time they are read in order to reduce further access times. This cache even persists between kernel executions [7, Page 1-66].

The memory in both address spaces can be allocated either normally or as shared memory. For normal memory the data is transferred as described in Section 2.1.4 by using the `clEnqueueWriteBuffer()` function while shared memory is mapped to a pointer inside the host program which can be written using `memcpy()`. Altera recommends to use shared memory for kernels running on one of their SoC platforms since this leads to a higher bandwidth [9, Page 1-66]. Nevertheless, constant memory that is allocated as shared memory can only be written once by the host and a new memory object has to be created every time the data set changes.

Inside the kernel, the accessing global memory is more time consuming than accessing local or private memory. Therefore, data should be transferred to a local or private copy on which the calculations are performed. Using private memory increases the kernel size. Therefore it is hard to control how much private memory can be declared within a kernel before the limits of the FPGA are reached. The local memory is also limited but its size is predefined for every OpenCL device. A test program can call the `clGetDeviceInfo()` function with the `CL_DEVICE_LOCAL_MEM_SIZE` flag in order to retrieve the local memory size. For the DE1-SoC the its is 16384 Byte. As described in Section 2.1.3, local memory

has to be declared with a static constant size within the kernel. Therefore, only buffer with a know size or maximum size can be copied.

Table 4.1: Address space, allocation type and copy combinations

Transfer	Allocation	Internal
constant	normal	no copy
	shared	no copy
global	normal	no copy
		local copy
	shared	no copy
		local copy

Table 4.1 shows all valid combinations for transfer, allocation and internal representation. Data that is transferred in the constant address space is never copied whereas for data in the global address space all four combinations are possible. The option to use constant memory with internal copies is excluded since the data would be copied in a consecutive order making the global memory the better choice.

Figure 4.4 shows the decision diagram that leads to the best adjustments for each buffer. In the beginning it has to be determined whether a SoC is targeted or not. As described in Section 3.1.1, this information is passed by the user via the `Alter_OpenCL_soc` comment. The `use_shared_memory` field is set to true in every `OpenCL_InOut` object if the comment is present.

Afterwards it has to be determined, whether the `local_cpy` field in the `OpenCL_InOut` object hast to be set. Arrays with a dynamic number of elements and without a maximum size are excluded directly. A function to determine the static or maximum size of a buffer was implemented. It has to be taken into account that the `size` and `max_size` fields only hold the number of elements but not the size in bytes. Therefore the function has to analyze the data type of the buffer. The size for preimplemented data types is defined by the OpenCL standard [14, Chapter 6.1] whereas typedef and struct types have to be split into their base types. If it is not possible to determine the size, the function will return a number higher than the limit of the local buffer. The result has to be compared to the remaining size of the local memory. If the buffer fits, the `local_cpy` field is set and the size of the local memory is reduced accordingly. The analysis implements a *first-come-first-serve* technique.

Finally, runtime constant buffer and buffer with update flag that do not have an internal copy are stored in the constant address space while the rest is stored in the global address space.

The effects of the decisions made by this analysis are evaluated in Section 5.5

4.1.10 Loop Unrolling

Altera's SDK for OpenCL provides a `unroll` pragma which can be used in front of a `for`-loop in order to indicate to the AOC that the loop has to be unrolled. It can be used with a factor in order to unroll only the given number of iterations.

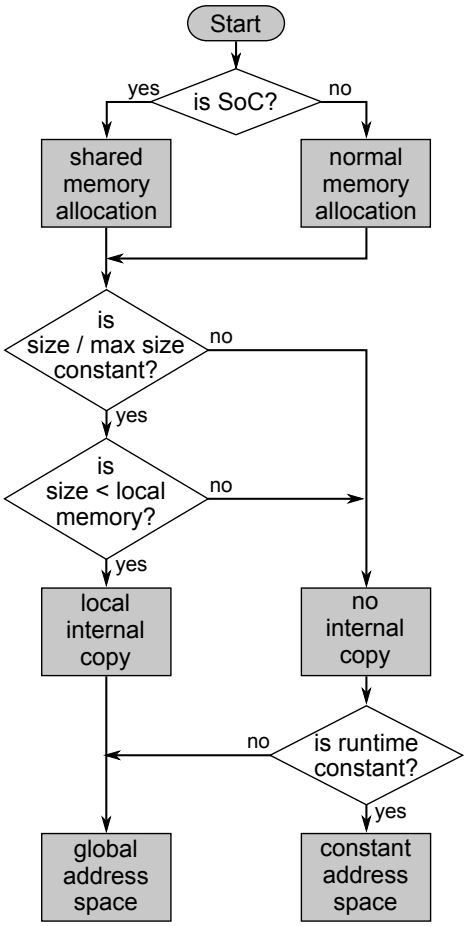


Figure 4.4: Memory analysis decision diagram

This technique is useful in order to increase the pipeline size or to exploit continuous memory access. For example, nested loops with loop-carried dependencies often have significant initiation intervals and slow down the execution time of the surrounding loop. Unrolling them, helps to exploit instruction level parallelism across iterations and to reduce the initiation interval of the outer loop.

The loop unrolling analysis determines which loops shall be unrolled during the compile process. The analysis implements an *all-or-nothing* approach. This means that either the loop is unrolled as a whole or not at all. Furthermore, only the innermost `for`-loops in a kernel are taken into account .

Therefore, in the beginning the analysis searches the target function for all `for`-loops that do not contain any further loops. Afterwards a number of disqualification criteria are checked:

- The number of iterations is not statically determinable
- The number of iterations exceeds 16.
- The loop body contains a call to a mathematical function that requires a lot of space. This includes all trigonometric functions such as `cos` and `sin` as well as logarithmic functions.
- The loop body contains a division with a divisor other than a value that is a power of two.

If non of these criteria is met, the `#pragma unroll` statement is inserted in front of the `for`-statement. The effect of the unroll analysis is evaluated in Section 5.6.

4.1.11 Inter Analysis and Transformation Dependencies

Figure 4.5 shows the dependency diagram between the analysis and transformation steps explained in Sections 4.1.1 to 4.1.10.

It can be seen that most steps depend directly or indirectly on the inline transformation. This is caused by the fact, that the inline transformation adds additional statements to the target function and with that more variables and expressions. Each of them has to by analyzed or might be part of a transformation. Only the constant array analysis is independent of the inline transformation since it does not access the statement list of the target function but the declaration statements of arrays declared at a higher scope.

It is the natural order that the constant folding transformation is executed after the constant variables have been replaced by their constant value. The loop unrolling step benefits from the constant folding since it tries to find loops with a constant number of iterations.

As explained in Section 4.1.2, constant arrays are excluded from the 2D to 1D array transformation. Therefore the names of the constant arrays have to be evaluated before the transformation.

The input/output analysis explained in Section 4.1.5 is a crucial step in the analysis process since it analyzes which variables are read and written within the target function.

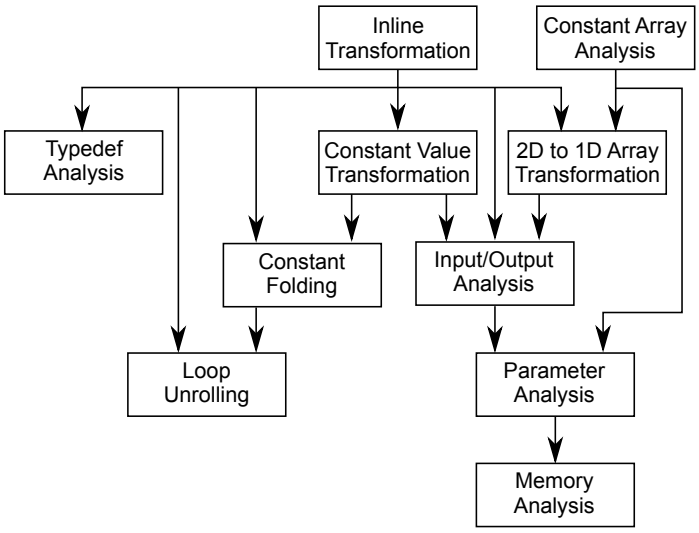


Figure 4.5: Dependency diagram between analysis and transformations

Therefore, all transformations which could add new variable references to the target function have to be done beforehand.

The parameter analysis described in Section 4.1.8 depends on the information of the input/output analysis. Furthermore it needs the results of the constant array analysis in order to exclude all pointer elements accessible by the kernel through the file scope.

The memory analysis needs the vector of `OpenCL_InOut` objects representing the function parameter of the kernel function. Therefore the only direct dependency is to the parameter analysis.

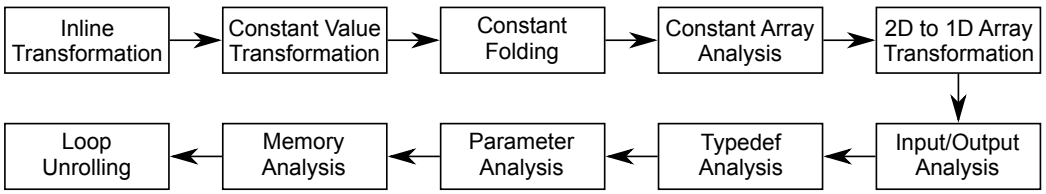


Figure 4.6: Flow diagram of all analysis and transformation steps

The final program flow can be seen in figure 4.6. It was decided to start with the inline transformation since this transformation is responsible for the greatest changes to the target function. The loop unrolling step was implemented last since it adds OpenCL specific pragmas to the code and could therefore also be moved to the kernel creation step. The rest of the analysis and transformations are grouped by topics such as constant values or arrays.

4.2 Host Program Transformation

The transformation of the input source file can be split into four tasks: The insertion of the global variables, the insertion of an initialization and a cleanup function, the adjustment of the main function and the modification of the target function.

4.2.1 Global Variables

Section 2.1.4 defines a number of objects that have to be initialized before a kernel can be executed. They are defined in the file scope of the output source file since the initialization and cleanup phase is separated from the execution phase. The initialization takes time and therefore it is not desirable to recreate these object for each kernel execution. The prototypes of the initialization and cleanup functions have to be inserted at the beginning of the source file as well in order to insert the function definition without regard of the position where these functions are called within the source code.

The first step of the process is to determine the insertion point within the statement list of the program. They can not be simply written to the beginning of the file since the OpenCL specific data types have to be read from the OpenCL header files first. Taking into account the common practice of arranging C/C++ source files, it was decided to chose the position above the first function definition as insertion point.

```
void cleanup();
bool init_opencl(const char *file_name,const char *kernel_name);
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
cl_mem output_output_buf = 0;
uint32_t *output_output_ptr = 0;
cl_mem input_input_buf = 0;
uint32_t *input_input_ptr = 0;
int aocl_input_old_size = 0;
```

Listing 4.8: Global variable and function declarations

The list of all global variables can be seen in Listing 4.8. The declarations in lines 1 to 8 are identical for every compile process, whereas the lower part depends on the structure of the target function.

A `cl_mem` object is created for every input and output buffer identified by the parameter analysis described in section 4.1.8. The name of the buffer is taken from the `buffer_name` entry of the corresponding `OpenCL_InOut` object. The memory objects are initialized with `0` in order to identify the first execution of the target function in which they have to be initialized. Buffer that have the `use_shared_memory` flag set need an additional pointer to safe the address of the shared memory. In the example shown in Listing 4.8 one input and one output buffer are created. Both use shared memory hence two additional pointers are added. An integer value like the one seen in line 13 is created for every buffer of variable size. This way it can be determined if the memory object from the previous execution is big enough to hold the new data.

4.2.2 Initialization and Cleanup Function

The initialization and cleanup functions are in most parts identical for all output programs. They are named `init_opencl()` and `cleanup()`. The name of the cleanup function is

predetermined by the `checkError()` function which internally calls a function with this name.

The initialization of the memory objects is moved to the point right before kernel execution leaving only the static objects to be initialized. Listing 4.9 shows the resulting `init_opencl()` function.

The cleanup function is used to delete all OpenCL related objects. Pointer to shared memory have to be unmapped before the buffer is deleted. The statements in lines 11 to 23 of Figure 4.10 are implemented in every program whereas lines 3 to 9 depend on the list of `OpenCL_InOut` objects.

```
1  bool init_opencl(const char *file_name,const char *kernel_name)
   {
3    cl_int status;
     cl_device_id *curr_devices;
5    cl_uint num_devices;
     cl_device_type cl_device_type_all = 0xffffffffUL;
7    cl_command_queue_properties cl_queue_profiling_enable = 0x2;
     std::string binary_file;
9    if (!setCwdToExeDir()) {
       return false;
11   }
     platform = findPlatform("Altera");
13   if (platform == 0) {
       printf("ERROR: Unable to find Altera OpenCL platform.\n");
15     return false;
     }
17   curr_devices = getDevices(platform,cl_device_type_all,&num_devices);
     device = curr_devices[0];
19   context = clCreateContext(0,1,&device,&oclContextCallback,0,&status);
     checkError(status,"Failed to create context");
21   queue = clCreateCommandQueue(context,device,cl_queue_profiling_enable,&↩
       status);
     checkError(status,"Failed to create command queue");
23   binary_file = getBoardBinaryFile(file_name,device);
     program = createProgramFromBinary(context,(binary_file.c_str()),&device↩
       ,1);
25   status = clBuildProgram(program,0,0,"",0,0);
     checkError(status,"Failed to build program");
27   kernel = clCreateKernel(program,kernel_name,&status);
     checkError(status,"Failed to create kerne");
29   return true;
   }
```

Listing 4.9: Initiation function

4.2.3 Main Function

The main function has to be modified in order to call the initialization and the cleanup function.

Section 3 explained that the kernel is always written to the same output file. Therefore, the strings passed to the parameter of the `init_opencl()` function are static constant.

```
void cleanup()
{
  if (output_output_buf) {
    clEnqueueUnmapMemObject(queue,output_output_buf,output_output_ptr↩
    ,0,0,0);
    clReleaseMemObject(output_output_buf);
  }
  if (input_input_buf) {
    clEnqueueUnmapMemObject(queue,input_input_buf,input_input_ptr,0,0,0);
    clReleaseMemObject(input_input_buf);
  }
  if (kernel) {
    clReleaseKernel(kernel);
  }
  if (program) {
    clReleaseProgram(program);
  }
  if (queue) {
    clReleaseCommandQueue(queue);
  }
  if (context) {
    clReleaseContext(context);
  }
  return;
}
```

Listing 4.10: Cleanup function

A successful execution of the program is only possible if the initialization was successful. Therefore the return value of the function call is checked and a return statement is inserted in case the evaluation fails.

The cleanup function is called at the end of the program. This function call is straight forward since the function does not accept any parameter.

The modified `main()` function can be seen in listing 4.11.

```
int main()
{
  if (!init_opencl("aocl_kernel","aocl_generated_kernel")) {
    printf("WARNING: COULDN'T INITIALIZE OPENCL\n");
    return -1;
  }

  // original code

  cleanup();
}
```

Listing 4.11: Changes made do the main Function

4.2.4 Target Function

The greatest manipulations have to be made to the target function: First, the function body is stripped. This means that all statements are removed from the statement list. Only variable declarations and the return statement are left. Afterwards statements implementing the kernel execution routine explained in Section 2.1.4 are inserted. Since the initialization of the memory buffer is moved from the initialization function to the target function, this step is prepended to the other tasks.

The first statements created are variable declarations. Besides the status variable already known from the `init_opencl()` and `cleanup()` functions, events have to be created in order to track the data transfer to the kernel, the execution and the data transfer back from the kernel. The number of events needed is determined by iterating the vector containing the `OpenCL_InOut` objects and evaluating the `flags` field.

The information is used to create two event arrays called `write_event` and `finish_event` for the transfer to and from the kernel respectively. The execution of the kernel is a single event and is therefore implemented as a variable called `kernel_event`.

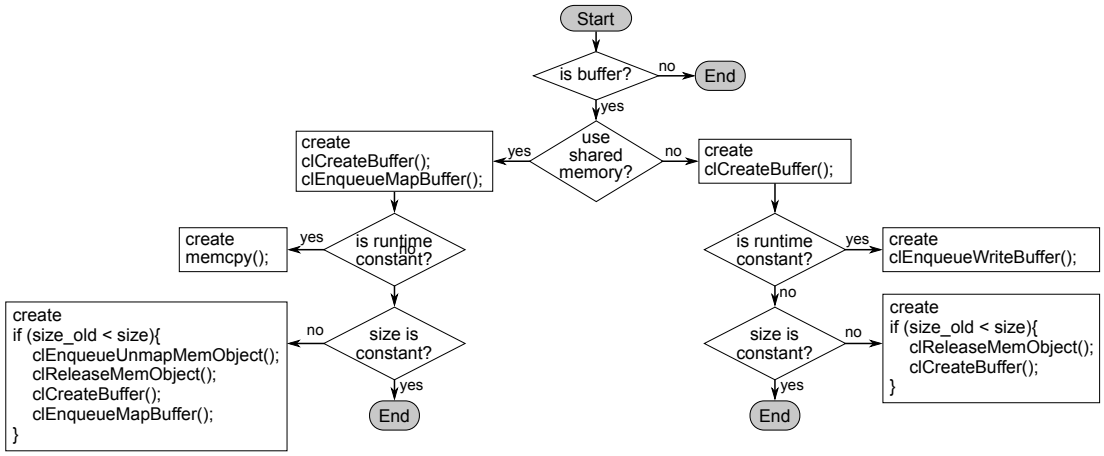


Figure 4.7: Decision diagram for different buffer types

In the next step the initialization of the buffer is implemented. Figure 4.7 shows the decision diagram the SOCAO compiler uses to generate the right code. For every `OpenCL_InOut` it is checked whether the `is_buffer` flag is set. If not, the compiler continues to the next object. It has to be distinguish whether the memory analysis described in Section 4.1.9 has set the `use_shared_memory` flag or not. Shared memory uses a pointer to access the data. Therefore the `clEnqueueMapBuffer()` function is called in order to map the pointer to the shared memory.

On both sides of the decision diagram three buffer types can exist: runtime constant buffers, variable size buffers and constant size buffers. For runtime constant buffer the data has to be copied right after the creation point. This is either done using the `memcpy()` function from the `string.h` header or the `clEnqueueWriteBuffer()` from the OpenCL header. Constant size buffer have to be released and recreated when the size of the existing buffer has become to small. In case of shared memory, the pointer has to be unmapped before the memory can be freed. Constant size buffers do not need any further treatment.

Figure 4.12 shows the initiation code for the three buffer types under the assumption that shared memory is used. The first buffer in lines 1 to 8 has a constant size. The second buffer in lines 11 to 27 has a variable size determined by the `size` variable. Therefore the size has to be checked every time the target function is called. The last buffer in lines 29 to 36 is runtime constant. The data is copied immediately after the buffer was created. The size variable is not reevaluated since its value is not supposed to change.

In accordance with the program flow explained in section 2.1.4, the next step is the transfer of the input data. In case shared memory is used, this is done by adding a `memcpy()` function call expression. For non-shared memory the `clEnqueueWriteBuffer()` function is used.

The compiler differentiates between two update behaviors: the first is updating the data in a buffer every execution, the second is updating the data only if the update flag is set. For the later the copy statement is inserted into an `if`-condition. In this case the `clEnqueueWriteBuffer()` function is used, the `blocking_write` parameter is set to `true` because no event was created to track the termination of the write process. Afterwards the SOCAO compiler inserts a `clSetKernelArg()` call for every `OpenCL_InOut` object in the function parameter vector.

The `clEnqueueTask()` function call is almost identical in every case. Only the number of write events which have to be awaited before executing the command changes.

The number of function calls to copy the results of the OpenCL kernel also depends on the parameter list. In order to insert them, the vector with `OpenCL_InOut` elements is evaluated one more time and a `clEnqueueReadBuffer()` or `memcpy()` function call expression is created for every output variable.

Finally the function call to await the termination of all `finish_event` is added as well as one `clReleaseEvent()` function call for every event.

An examples of an altered target function can be found in the appendix in Listing A.2 lines 53 to 101.

4.3 Kernel Creation

In order to successfully create the *aocl_kernel.cl* file, the compiler needs the list with the extracted statements, the vector of `OpenCL_InOut` elements and the results of the typedef and constant array analysis. The whole process is implemented in the `createOpenCLKernel()` function.

The function is split into three tasks: The creation of the function header with the parameter list, the creation of the function body and the unparsing of the code into the destination file.

4.3.1 Function Header

A function is defined by its name, a list of function parameter, a return value and optionally a number of modifier keywords. Therefore, the first step is to create the function parameter list from the vector of `OpenCL_InOut` objects. The vector is traversed and for every entry

```
//constant size buffer
if (!input_constant_size_buf) {
  input_constant_size_buf = clCreateBuffer(context, 0x4UL| ↩
    CL_MEM_ALLOC_HOST_PTR, 8*sizeof(float), 0, &status);
  checkError(status, "Failed to create buffer");
  input_constant_size_ptr = (float*)clEnqueueMapBuffer(queue,↩
    input_constant_size_buf,CL_TRUE,0,0,8*sizeof(float),0,0,0,&status);
  checkError(status, "Failed to map buffer");

}

//variable size buffer
if (!output_variable_buf) {
  output_variable_buf = clCreateBuffer(context, 0x2U| ↩
    CL_MEM_ALLOC_HOST_PTRL, size*sizeof(float), 0, &status);
  checkError(status, "Failed to create buffer");
  output_variable_ptr = (float*)clEnqueueMapBuffer(queue,↩
    output_variable_buf,CL_TRUE,0,0,size*sizeof(float),0,0,0,&status);
  checkError(status, "Failed to map buffer");
  aocl_variable_old_size = size;
}
else {
  if (aocl_variable_old_size < size) {
    clReleaseMemObject(output_variable_buf);
    output_variable_buf = clCreateBuffer(context, 0x2UL| ↩
    CL_MEM_ALLOC_HOST_PTR, size*sizeof(float), 0, &status);
    checkError(status, "Failed to create buffer");
    output_variable_ptr = (float*)clEnqueueMapBuffer(queue,↩
    output_variable_buf,CL_TRUE,0,0,size*sizeof(float),0,0,0,&status);
    checkError(status, "Failed to map buffer");
    aocl_variable_old_size = size;
  }
}

//runtime constant buffer
if (!input_const_buf) {
  input_const_buf = clCreateBuffer(context,0x4UL| CL_MEM_ALLOC_HOST_PTR,↩
    size * sizeof(float ),0,&status);
  checkError(status,"Failed to create buffer");
  input_const_ptr = (float*)clEnqueueMapBuffer(queue,input_const_buf,↩
    CL_TRUE,0,0,size*sizeof(float),0,0,0,&status);
  checkError(status, "Failed to map buffer");
  memcpy(input_const_ptr, const, size);
}
```

Listing 4.12: Initiation code for different buffer types

the process shown in Figure 4.8 is followed to generate a `SgInitializedName` object.

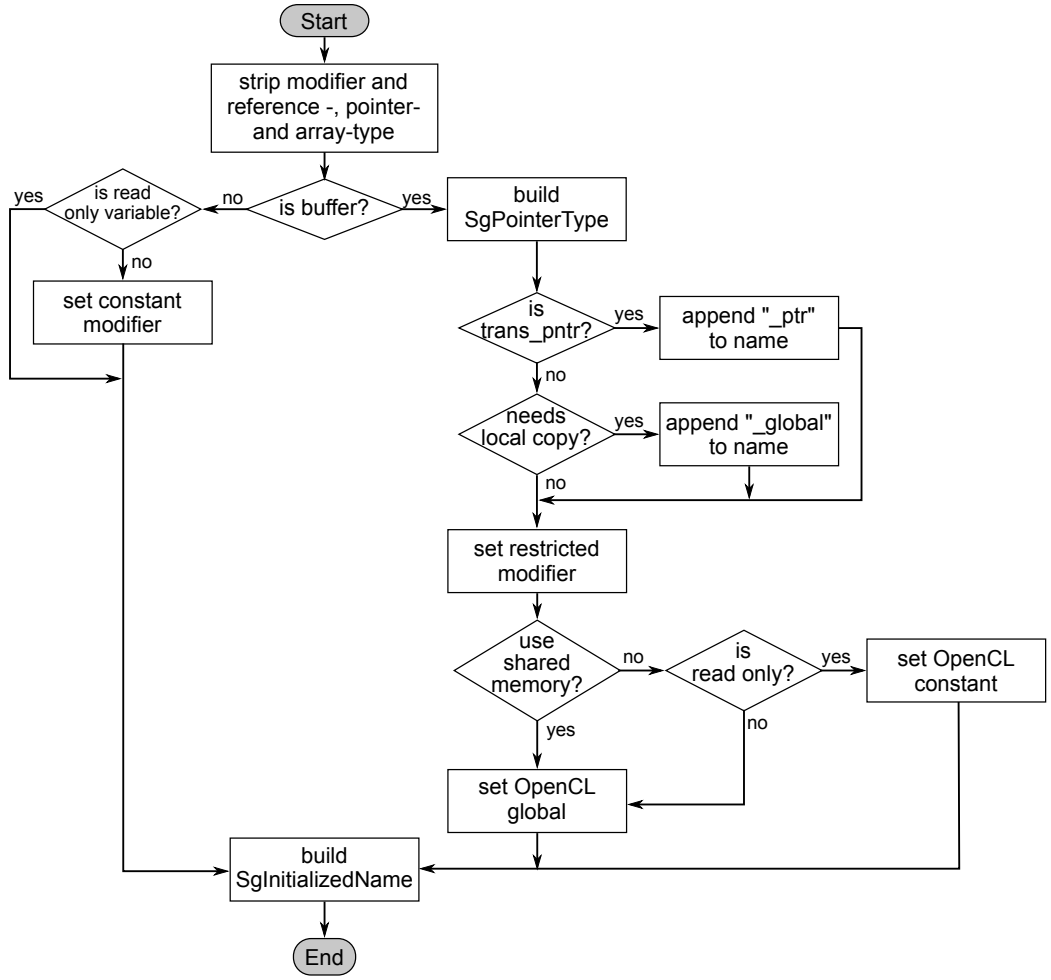


Figure 4.8: Decision diagram for the creation of parameter list

In the beginning the data type of the variable is striped to its base type. This means removing all `SgModifierType`, `SgReferenceType`, `SgPointerType` and `SgArrayType` class encapsulations. The `stripType()` function used to do so can strip various layers of encapsulation classes until only the base type is left.

The rest of the process is decided by the `is_buffer` flag. A scalar variable does not need to be further modified except of the `const` modifier in case the variable is read-only. It has to be noted, that this is the case for most scalar variables.

Buffers are always passed as pointer. Therefore the first step after detecting a buffer parameter is to create a new `SgPointerType` object with the stripped data type as base. Afterwards the name of the buffer is changed where needed. The suffix `_ptr` is appended at the name for transparent pointers in order to distinguish the name from the original variable which is used within the function body. Variables which have the `local_copy` flag set are copied later into an array with the original name located in the `__local` address space of the computing device. Therefore the name of the function parameter has to be altered by appending the `_global` suffix.

All buffers are restricted. Therefore a new `SgModifierType` encapsulating the pointer is build and the restrict option is set. Finally the address space of the parameter is set in accordance with the `use_const_memory` flag. The ROSE framework provides the functions `setOpenclConstant()` and `setOpenclGlobal()` as part of the `SgModifierType` class.

With the variable type and name fully build and configured, a `SgInitializedName` object can be created and appended at the parameter list.

The parameter list is used in the `buildDefiningFunctionDeclaration()` function to build the function definition of the kernel. The name is always set to `aocl_generated_kernel` and the return type is `void`.

Every `SgFunctionDeclaration` object is linked to a modifier object that holds all information about the modifier keywords that can prepended to the function header. By calling the `setOpenclKernel()` function, the `__kernel` keyword is set in front of the function header.

Listing 4.13 shows a function prototype with three arguments generated to target a SoC platform. Therefore, the input data that is passed as pointer is located in the global address space.

```
__kernel void aocl_generated_kernel(uint32_t __global * __restrict__ ↩
    state_global,unsigned char __global * __restrict__ input,const uint64_t↩
     len);
```

Listing 4.13: OpenCL kernel header

4.3.2 Function Body

The function body is composed of three major blocks. In the beginning data is copied from the global address space into the private or the local address space. Then the original function body is executed. At the end the data is copied back.

In the beginning the compiler adds the statements which were striped from the target function into the new function body and removes all preprocessing info and comments.

Afterwards the copy statements are created. First the compiler deals with the variable which need a transparent pointer. A variable declaration has to be inserted in the beginning of the function body, since the variable itself is no long declared in the function header. For read-write variables an initiation value has to be copied from the transparent pointer. A second assign statement is added at the end of the function body in order to copy the result back into the global address space. The result can be seen in listing 4.14.

Finally the declaration and the copy statements for the data that has to be copied to and from the local address space are generated. The results are similar to the example shown in 4.15. OpenCL does not support arrays of dynamic size, therefore, the arrays are declared using either the constant size value from the `size` field or the maximum size value from the `max_size` field. This can be seen in line 4 of the example. Afterwards the `for`-loops used to copy the data are created. For arrays with `has_max_size` property not the `size` field is used as upper bound but the `size_param` field containing the variable name which holds the actual size. Variables that are write-only do not need the loop at the beginning of the

```
__kernel void aocl_generated_kernel(float __global * __restrict__ ↩
    error_ptr, ...)
{
  float error = *error_ptr;

  ...

  error_ptr[0] = error;
}
```

Listing 4.14: Copy statements for transparent pointer

function for initialization while read-only variable do not need the loop at the end.

```
_kernel void aocl_generated_kernel(int __global * __restrict__ ↩
    state_global, ...)
{
  int rose_it;
  __local int state[8];
  for (rose_it = 0; rose_it < 8; ++rose_it) {
    state[rose_it] = state_global[rose_it];
  }

  ...

  for (rose_it = 0; rose_it < 8; ++rose_it) {
    state_global[rose_it] = state[rose_it];
  }
}
```

Listing 4.15: Copy statements for variables safed to the local address space

4.3.3 Unparsing

Unparsing and writing the *aocl_kernel.cl* file are the final steps of the kernel creation.

In the beginning the file is opened and the typedef definitions found by the typedef analysis are written into it. This also includes the definitions of the *cstdint* header file which are always added to the output file. Afterwards the arrays identified by the constant array analysis added.

The kernel function is first unparsed into a string by calling the `unparseToCompleteString()` function on the `SgFunctionDeclaration` object. Unfortunately, the ROSE framework makes a mistake when unparsing the pointer function arguments: The unparser attaches the star indicating a pointer type in front of the address space identifier creating expressions like `*__global`. These are detected by the aoc compiler as syntax errors. Therefore, the unparsed string ins searched for these expressions and the position of the star is moved to the end of the address space qualifier creating an expression like `__global *`.

Finally the string is written to the file to finish the process of creating the OpenCL kernel. Example kernel can be found in Listings A.3, B.3, C.3 and D.3 in the appendix.

Chapter 5

Evaluation

The SOCAO compiler is evaluated by compiling four example programs and running them on Altera's DE1-SoC evaluation board.

Sections 5.1 to 5.4 describe the input test programs, their characteristics and the performance results. Section 5.5 evaluated the performance impact of the memory types by compiling different versions of the final OpenCL kernel and comparing them with each other. The same is done in Sections 5.6 and 5.7 to show the effect of loop unrolling and the use of constant file scope arrays.

5.1 SHA-256

The Secure Hash Algorithm (SHA) standard is defined by the National Institute of Standards and Technology (NIST) in [10]. In general hashing functions map a string of arbitrary length onto a string of fixed length. They are used in the field of cryptography in order to generate and verify digital signatures and message authentication codes. The SHA standard most widely used is the SHA-2 standard. It also defines the SHA-256 algorithm which is named after the number of bits the final hash consists of.

The calculation is split into two major steps: The preprocessing of the message and the actual calculation of the hash tag.

5.1.1 Compilation

The implementation of the algorithm was taken from the *mbed TLS* library [17]. This library provides cryptography functionalities for embedded systems. A target function was extracted which takes the preprocessed message and calculates the hash tag. The source code can be seen in Listing A.1 in the appendix.

It has to be noted that the algorithm contains two nested `for`-loops. Both show loop-carried dependencies. The outer loop splits the input message in blocks of 512 bit and depends on the accumulation of the `state` variable in line 65 of Listing A.1. The inner loop executes the 64 iterations defined by the SHA-256 algorithm and shows read-write dependencies for the `A` array in lines 59 to 61. The algorithm can therefore not be split

into work-items that work independently of each other. Hence, it is hard to find a good way to accelerate the execution of the algorithm. Nevertheless the algorithm has a lot of instruction parallelism an FPGA can exploit.

The header of the target function can be seen in Listing 5.1. Besides the obligatory `Altera_OpenCL_Accelerate`, the `Altera_OpenCL_size` comment is used to indicate the element count of the `input` and the `state` pointer. The array `K` contains the constant values defined in the SHA specification [10, Chapter 4.2.2]. Therefore, it is marked as constant array. The last comment indicates that a SoC platform is targeted.

```
//Altera_OpenCL_Accelerate
//Altera_OpenCL_size input len
//Altera_OpenCL_size K 64
//Altera_OpenCL_size state 8
//Altera_OpenCL_const_vec K
//Altera_OpenCL_soc
void mbedtls_sha256_update_accelerated( uint32_t *state, const unsigned ↩
    char *input, uint64_t len )
```

Listing 5.1: SHA-256 target function header with Altera_OpenCL comments

The resulting host program and OpenCL kernel can be seen in Listings A.2 and A.3. The AOC compile log shows that the inner loop can be pipelined without initiation interval despite the loop-carried dependencies. The outer loop has an initiation interval of two cycles but the iterations have to be executed serially across the inner loop because each iteration depends directly on the results of the previous iteration. This means that the outer loop is de facto executed sequentially since it contains little more beside the inner loop.

5.1.2 Area

The area consumption within the FPGA after synthesis can be seen in Table 5.1. It shows, that the kernel requires only 27 % of the total logic cells, 8 % of the available memory bits and 17 % of the M10K memory blocks. No DSP blocks are needed as they are typically used to implement floating point operations and complex mathematical functions from the math library. The design runs with a clock frequency of 112MHz.

Table 5.1: Resources occupied by the SHA-256 OpenCL kernel

	SHA-256
ALUTs	8,720
Registers	19,036
Logic utilization/[10^3]	8.72 / 32.0 (27 %)
Memory bits/[10^3]	327 / 4,065 (8 %)
M10K blocks	69 / 397 (17 %)
DSP blocks	0 / 87 (0 %)
Clock freq	112 MHz

The kernel is relatively small. This is caused by the fact that most operations within the kernel are logic function such as *AND* and *OR*.

5.1.3 Execution Time

The execution time of the SHA-256 algorithm for different input sizes can be seen in Figure 5.1. It is obvious that the execution time increases linearly with the size of the input message. This can be explained by the dependency of the outer `for`-loop on the message size, whereas the number of iterations of the inner loop is constant.

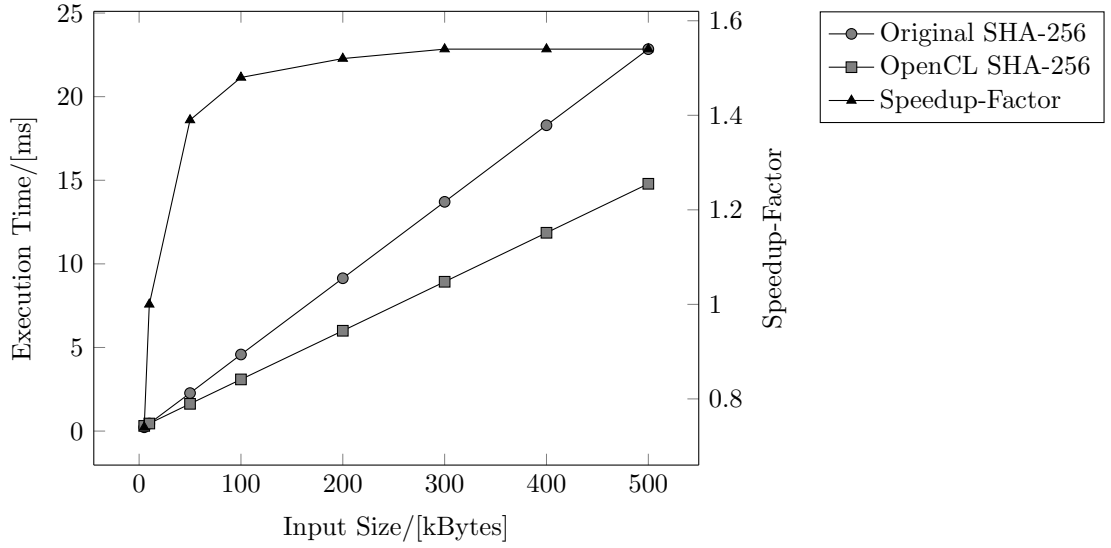


Figure 5.1: Execution time SHA-256

The speedup is also shown in Figure 5.1. It is calculated as the execution time of the original code divided by the execution time of the OpenCL optimized version. It can be seen that the speedup depends greatly on the message size. For small messages the OpenCL version runs slower than the original program. This is caused by the OpenCL overhead and the time consumed to transfer the data. The maximum speedup is 1.54 and is reached for a message size of around 300 kByte. The break even point is reach for a message size of 10 kBytes.

5.2 AES

The advances encryption standard (AES) was introduced by the NIST in 2011 and belongs to the class of symmetric block cipher algorithms that can encrypt and decrypt data [11].

5.2.1 Compilation

The implementation was taken from the *mbed TLS* library as well [17]. The extracted function encrypts the AES standard using counter mode (CTR) and a 128 bit key. CTR refers to a block cipher operation mode where the key and a counter are used as input values to the AES algorithm that calculates a unique bitstream. The data is encrypted and decrypted by performing a bitwise xor operation with the bitstream.

Listing 5.2 shows the function header with the user comments for the SOCAO compiler.

The element count is only given for the pointer `input`, `output` and `RK`. The SOCAO compiler is able to detect the static size of the remaining arrays. Furthermore, the AES algorithm uses various constants which are marked as constant arrays in order to have them declared directly in the OpenCL file.

```
1  //Altera_OpenCL_Accelerate
   //Altera_OpenCL_size input length
3  //Altera_OpenCL_size output length
   //Altera_OpenCL_size RK 68
5  //Altera_OpenCL_const_vec FT3
   //Altera_OpenCL_const_vec FT2
7  //Altera_OpenCL_const_vec FT1
   //Altera_OpenCL_const_vec FT0
9  //Altera_OpenCL_const_vec RT3
   //Altera_OpenCL_const_vec RT2
11 //Altera_OpenCL_const_vec RT1
   //Altera_OpenCL_const_vec RT0
13 //Altera_OpenCL_const_vec FSb
   //Altera_OpenCL_const_vec RSb
15 //Altera_OpenCL_soc
   int mbedtls_aes_crypt_ctr_nr10( uint32_t *RK, uint64_t length, unsigned ↩
       char nonce_counter[16], unsigned char stream_block[16], const unsigned ↩
       char *input, unsigned char *output )
```

Listing 5.2: AES-CTR target function header with Altera_OpenCL comments

The complete input code can be seen in Listing B.1 in the appendix. It shows that the target function has equally to the SHA-256 algorithm two nested `for`-loops. The outer loop splits the message into blocks. The calculation of the AES algorithm depends only on the key and the counter but not on the result of previous calculations. Therefore the bitstream for each input block can be calculated independently. The inner `for`-loop shows loop-carried dependencies since the variables `X1`, `X2`, `X3` and `X4` are read before they are written.

The output from the SOCAO compiler can be seen in Listings B.2 and B.3. The loop unroll analysis described in Section 4.1.10 added an `unroll` pragma in front of the `for` statement in line 86 because of it is the inner loop and has a constant number of iterations.

The compile log of the AOC shows that the outer loop can be pipelined perfectly.

5.2.2 Area

Table 5.2 shows the resources occupied by the kernel. It can be seen that the design needs 31 % of the logic resources, 42 % of the memory bits and 78 % of the M10K memory blocks. The kernel runs on a clock frequency of 141 MHz.

The AES kernel is significantly bigger than the SHA-256 kernel analyzed in Section 5.1.2. The memory increase can be explained by the number of constant arrays which are needed for the compilation of the bitstream. Eight arrays with 256 32 bit values and two arrays with 256 8 bit values are declared in the file scope of the OpenCL program. This alone equals 70 kbit of data that have to be stored. As stated in Section 5.2.1, the algorithm is translated into a perfect pipeline. This means that a separate copy of the `stream_block`, `nonce_count`

Table 5.2: Resources occupied by the AES-CTR OpenCL kernel

	AES-CTR
ALUTs	10,888
Registers	22,956
Logic utilization/[10^3]	10.1 / 32.0 (31 %)
Memory bits/[10^3]	1,709 / 4,065 (42 %)
M10K blocks	311 / 397 (78 %)
DSP blocks	0 / 87 (0 %)
Clock freq	141 MHz

and `RK` arrays declared at the beginning of the kernel function exists in every stage of the pipeline. This increases the memory requirements with the length of the algorithm.

5.2.3 Execution Time

The execution times of the original and optimized AES-CTR program can be seen in Figure 5.2. The graph shows a linear dependency between input size and execution time because the unrolled kernel only consists of one loop that depends on the message size.

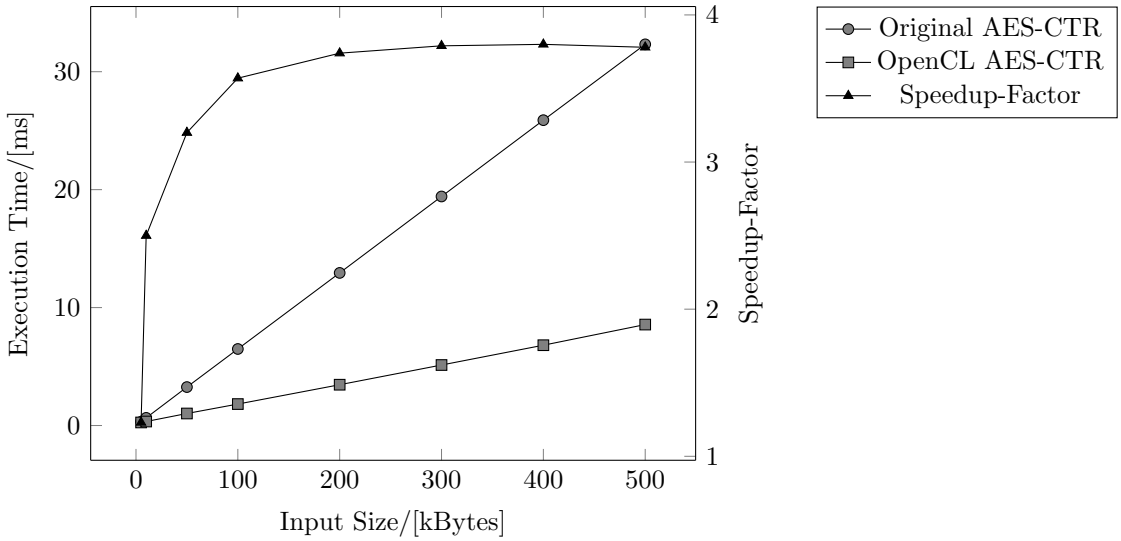


Figure 5.2: Execution time AES-CTR

The maximum speedup-factor is 3.78. This means that the original program needs 3.78 times the time of the optimized program in order to execute one AES encryption. The maximum speedup is reached at a message size of 300 kBytes. The OpenCL program executes faster than the original version once the message size exceeds 3.3 kBytes.

It is obvious that the performance of the AES kernel is better than the SHA-256 kernel seen in Section 5.1.3 because the outer loop is a perfect pipeline whereas in the SHA-256 algorithm it is only possible to pipeline the inner loop. Another factor is the clock frequency with which the kernel runs. The SHA-256 kernel runs at 112 MHz whereas the AES-CTR kernel uses a clock of 141 MHz.

5.3 FFT

The Fast Fourier Transformation (FFT) is an algorithm used to translate a discrete signal from the time domain into the frequency domain.

5.3.1 Compilation

The implementation was taken from the *Worst-Case Execution Time (WCET)* research group of the Mälardalen Real-Time Research Center which provides a large number of benchmarks [3].

The implemented algorithm takes a fixed number of 2048 input samples in order to calculate the FFT. The extracted target function contains the main loop of the FFT calculation. The prototype can be seen in Listing 5.3. The function arguments are a flag that indicates whether the normal or inverted version of the FFT is calculated and the `pi_frac` pointer which holds the necessary fractions of π. The `ar` and `ai` arrays that hold the real and imaginary input data are not shown in the function header. They are detected by the parameter analysis explained in Section 4.1.8. Nevertheless it is obligatory to indicate their size with the `Altera_OpenCL_size` comment. The fractions of π are calculated once before the first execution of the FFT. Therefore, the array is marked as runtime constant.

```
//Altera_OpenCL_Accelerate
//Altera_OpenCL_size ar 2048
//Altera_OpenCL_size ai 2048
//Altera_OpenCL_size pi_frac ITER
//Altera_OpenCL_runtime_const pi_frac
//Altera_OpenCL_soc
void accelerate(int flag, float *pi_frac)
```

Listing 5.3: FFT target function header with Altera_OpenCL comments

The complete input file can be seen in Listing C.1 in the Appendix. It is important to notice that the target function displayed in lines 69 to 104 contains three nested `for`-loops. Furthermore the inner most loop creates a loop-carried dependency by reading the arrays `ar` and `ai` in lines 91 to 94 and then writing back to them in lines 97 to 100. Furthermore the function calls the trigonometric functions `sin` and `cos` in lines 84 and 85.

The output file of the compilation is shown in Listing C.2. It can be seen in lines 96 to 103 that the `pi_frac` pointer is initialized, mapped and copied only once.

The FFT kernel is displayed in Listing C.3. The `ar` and `ai` arrays occupy together 16284 Bytes which is exactly the size of the local memory. Therefore copies are created for both of them. No loop fulfills the criteria for the loop unrolling analysis. Therefore no `#pragma unroll` statement was added.

The AOC log shows that the `for`-loops can not be pipelined very effectively. The inner most loop has an initiation interval of 24 cycled due to the loop-carried dependencies. The middle loop is listed with an initiation interval of 2 cycles but the iterations are executed serially over the region of the inner loop which will produce a lot of stalls in the pipeline. Finally the outer loop is not pipelined at all.

5.3.2 Area

Table 5.2 shows the resource report of the FFT kernel. It can be seen that around one third of logic and M10K blocks are occupied. Furthermore, the FFT kernel uses 14 DSP blocks. They are mainly used to implement the `sin` and `cos` function calls in lines 84 and 85 of Listing C.1.

Table 5.3: Resources occupied by the FFT OpenCL kernel

	FFT
ALUTs	13,030
Registers	23,203
Logic utilization/$[10^3]$	10.8 / 32.0 (34 %)
Memory bits/$[10^3]$	628 / 4,065 (16 %)
M10K blocks	131 / 397 (33 %)
DSP blocks	14 / 87 (16 %)
Clock freq	125 MHz

5.3.3 Execution Time

No input size sweep can be made to evaluate the execution time since the implementation uses a fix sample size.

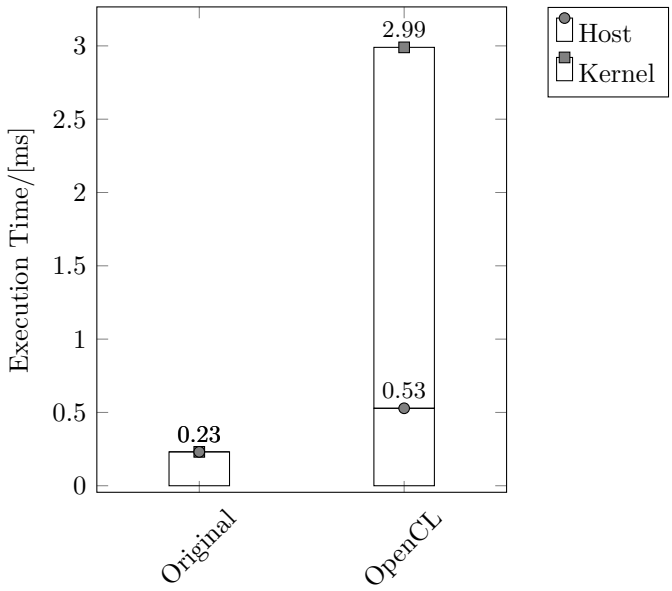


Figure 5.3: Breakdown execution time for the FFT example

Figure 5.3 compares the execution time of the original program and the OpenCL enhanced program. It can be seen that the execution time of the modified version exceeds the original program by a factor of 12.5. The breakdown even shows that the time spend to execute the host program already exceeds the total execution time of the original algorithm. This indicates that the break-even point has not been exceeded. In comparison, for an in-

put size of 16384 Byte the original SHA-256 algorithm needs 0.73 ms and the original AES encryption needs 1.07 ms which equals three to four times the execution time of the original FFT algorithm. This leaves enough time to compensate for the time needed to transfer the data. Unfortunately, if the number of input samples in the FFT was increased any further, the data would not fit into the local address space. The impact of this is discussed in Section 5.5.2.

Another reason for the bad timing result can be found in the initiation interval of the inner `for`-loop. The 24 clock cycles which have to pass between consecutive iterations are multiplied by the total number of iterations that have to be performed. Since this loop is enclosed by two more loops this number grows big quickly.

It has to be concluded that not every program is suitable for OpenCL acceleration. First, programs which use the same array as input and output pointer should be considered carefully since they are more likely to create loop-carried dependencies which result in extensive initiation intervals. Second, programs which contain more than two nested loops can result in bad performance since the AOC compiler has more problems pipelining these. Last, the target function has to include sufficient operations in order to compensate for the delay of the data transfer.

Nevertheless, the FFT example has certain characteristics which show effect once the impact of internal copies are evaluated in Section 5.5.2.

5.4 CRC

The cyclic redundancy check (CRC) is a procedure used to detect errors during data transmission. The algorithm calculates a checksum using a polynomial and appends this to the data stream. The receiver repeats the calculation including the crc. In case of correct data transmission the resulting value should be zero.

5.4.1 Compilation

Like the FFT, the implementation of the CRC was taken from the WCET benchmark project [3]. The prototype of the target function can be seen in Listing 5.4. The algorithm in the function body calculates the CRC for a char array of arbitrary length. The `lin` array that is holding the data stream is a file scope variable and therefore not part of the function header. The `icrctb` and `rchr` arrays contain the coefficients of the polynomial function. Their values are initialized at the beginning of the main function. Therefore, they are indicated as runtime constant.

The full input source code is printed in the appendix in Listing D.1. It can be seen in lines 36 to 44 that the algorithm only contains one `for`-loop which iterates over the size of the input message. The calculation consists mostly of *xor* and *shift* operations on the input data. This can normally be done more efficiently by an FPGA than by an ALU. Nevertheless, the access to the `cword` variable causes a loop-carried dependency.

The host program and OpenCL kernel generated by the SOCAO compiler can be seen in listings D.2 and D.3. It has to be noted that the `icrctb` and `rchr` arrays are copied into the local array since they fit the criteria of the memory analysis described in Section 4.1.9.

```
//Altera_OpenCL_Accelerate
//Altera_OpenCL_size lin len
//Altera_OpenCL_size icrctb 256
//Altera_OpenCL_size rchr 256
//Altera_OpenCL_runtime_const icrctb
//Altera_OpenCL_runtime_const rchr
//Altera_OpenCL_soc
unsigned short icrc(unsigned short crc, unsigned int len, short jinit, int↩
    jrev)
```

Listing 5.4: CRC target function prototype with Altera_OpenCL comments

The AOC compile log shows that the loops can be pipelined perfectly even though the loop has read-write dependencies for the `cword` variable.

5.4.2 Area

The results of the resource report can be seen in Table 5.4. In comparison to the kernel seen in the previous sections, the CRC kernel occupies relatively few space. Only 27 % of the logic and 16 % of the M10K memory blocks are utilized. The clock frequency is 143 MHz which is comparable to the AES example.

Table 5.4: Resources occupied by the CRC OpenCL kernel

	CRC
ALUTs	7,785
Registers	10,162
Logic utilization/$[10^3]$	5.52 / 32.0 (17 %)
Memory bits/$[10^3]$	258 / 4,065 (6 %)
M10K blocks	65 / 397 (16 %)
DSP blocks	0 / 87 (0 %)
Clock freq	143 MHz

5.4.3 Execution Time

The execution times for the original and optimized CRC program are shown in Figure 5.4. A sweep from 5 to 500 kBytes input size is made in order to document the effect the OpenCL overhead has on the execution time. It can be seen that the maximum speedup-factor is 3.74 and is reached at around 300 kBytes. Thereby it is similar to the AES example. The break-even point is 4 kBytes which is slightly higher than the AES example but lower than the SHA-256 algorithm.

5.5 Memory Performance

In this chapter the performances of the different address spaces and the different forms of memory allocation are evaluated in order to justify the decisions implemented in the

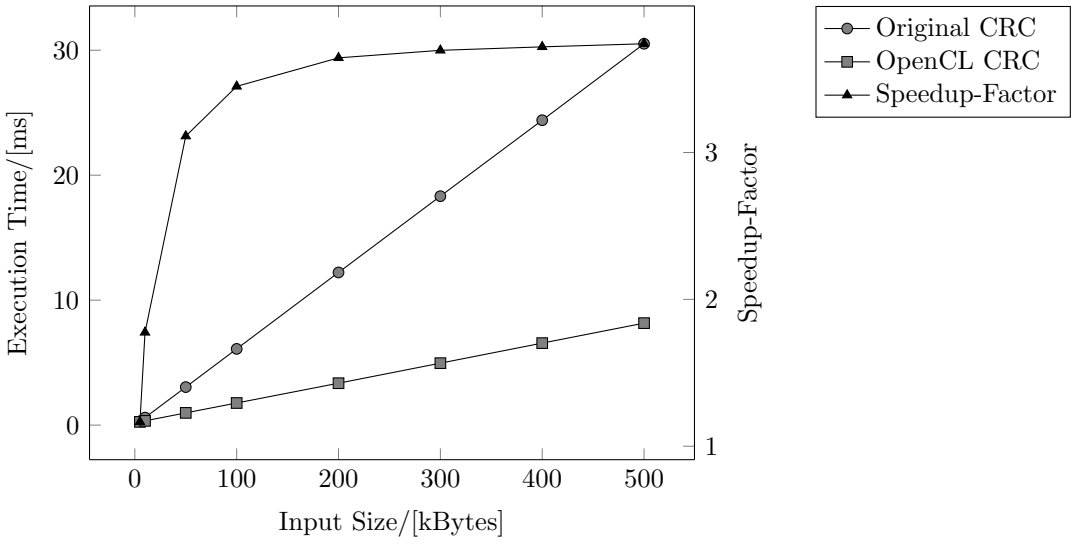


Figure 5.4: Execution time CRC

memory analysis which was explained in Section 4.1.9. Therefore, different versions of the test programs explained in Sections 5.1 and 5.2 are created and compared with each other.

5.5.1 Memory Allocation

As explained in Section 4.1.9, the first property evaluated by the memory analysis is whether to use shared or non-shared memory allocation. Unfortunately, only the performance for SoC platforms can be evaluated because hardware-wise only the DE1-SoC was available.

For the test, two new versions of the AES and SHA-256 program were created by deleting the `//Altera_OpenCL_soc` comment in front of the target-function declaration. Thereby the SOCAO compiler is forced to implement non-shared memory allocation.

The speedup is calculated for all four programs by dividing the execution time of the original program by the execution time of the optimized program. The results are shown in Figure 5.5.

It can be seen, that the maximum speedup is equal for both versions of the SHA-256 program. Nevertheless, for small message sizes, the speedup increases when shared memory allocation is used. This also influences on the break even point which is 10 kBytes for shared memory allocation and 15 kBytes for non-shared memory allocation.

For the AES-CTR algorithm, the maximum speedup increases from 3.7 to 3.8 when shared memory allocation is used. The The break-even point is also reduced from 6 kBytes for non-shared memory allocation to 3.8 kBytes for shared memory allocation.

The decrease in execution time is caused to the fact that the time for data transfer is reduced. Especially the AES-CTR algorithm that has to transfer as many bytes from the kernel back to the host system as it copies to the OpenCL device, profits from this technique.

It can be concluded that shared memory allocation should be used whenever a SoC

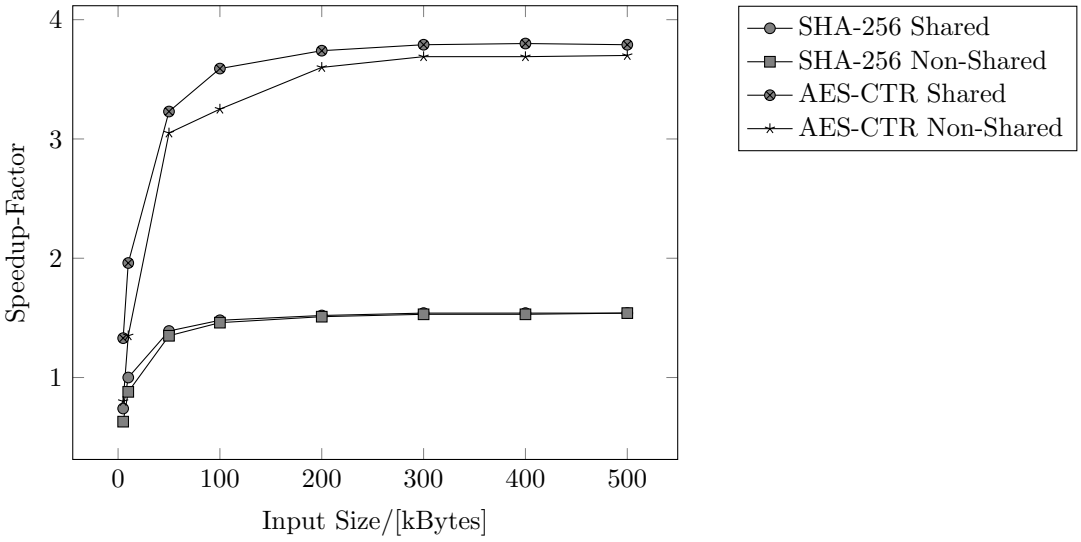


Figure 5.5: Speedup-factor for different types of memory allocation

platform is targeted. Even though it might not necessary result in a higher speed-up it might increase the range of input data size for which the optimized program can be used.

5.5.2 Internal Copy

In this section the effect of copying data to the local memory is evaluated.

The SHA-256 uses the `state` array to transfer the results of an earlier function call in order to resume a hashing process. This array holds exactly 256 bit divided into 8 32 bit integer values. This array fulfills the requirements to create an internal copy that is used inside the kernel function. Nevertheless, the kernel code in Listing A.3 shows that after copying the data from the global to the local address space in lines 35 to 37 the data is copied again to the private memory in lines 48 and 49. The AOC recognized the local copy as unnecessary and optimized the code. The log states the warning: *'Compiler Warning: Aggressive compiler optimization: removing unnecessary storage to local memory'.*

The AES-CTR kernel in Listing B.3 has three array for which local copies are created. The `stream_block` and `nonce_counter` arrays are used to like the `state` array in the SHA-256 kernel in order to resume a previous calculation. The `RK` array contains the key and is therefore constant. A closer look shows that the `nonce_counter` is copied into the `int_counter` variable in line 57. Therefore the log of the AOC compiler states the same warning as for the SHA-256 example. Furthermore, the kernel of the AES algorithm forms a perfect pipeline. Longer access times to the memory will enlarge the pipeline but not affect the initiation interval. In terms of execution time this results in an offset that becomes unimportant once the pipeline is filled.

Figure 5.6 shows the speed-up factor for kernels that use local copies and those not. As expected the difference is negligible.

A better example is the CRC algorithm. Once the internal copies are removed, the polynomial coefficients have to be read from the global memory. As described in Section

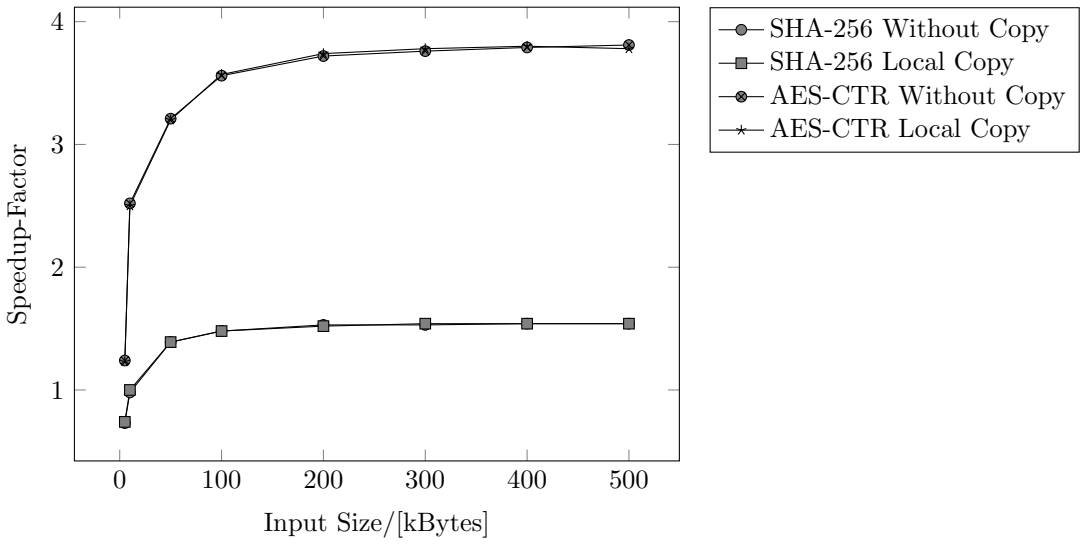


Figure 5.6: Speedup-factor for kernels with or without internal copy

2.1.3, the global memory is the slowest memory available. Furthermore, Altera states that their implementation is optimized for consecutive memory access. However the CRC algorithm accesses the `rchr` and `icrctb` arrays randomly. The AOC compile log indicates that once the copy statements are removed, the main loop has an initiation interval of 165 clock cycles caused by the load operations.

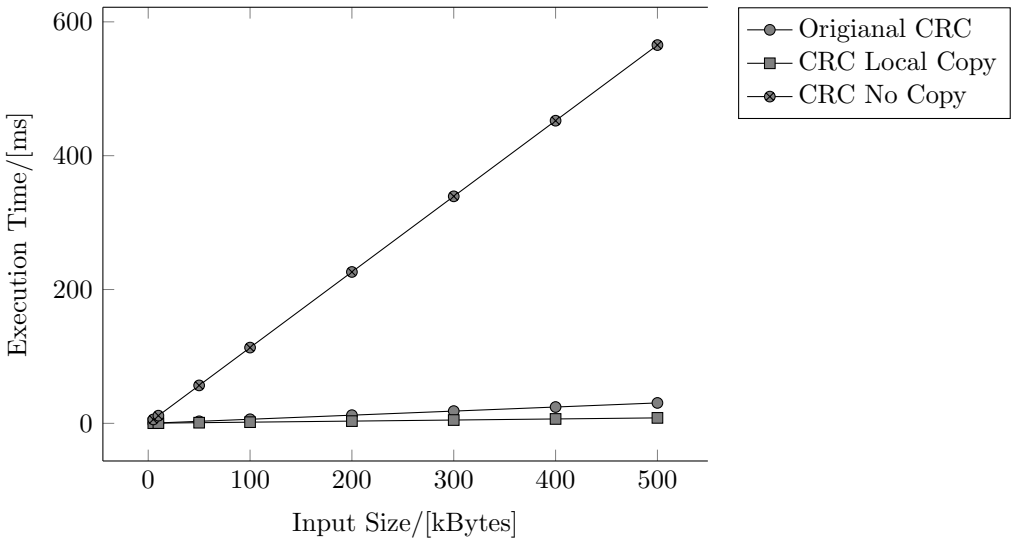


Figure 5.7: Execution time of the CRC algorithm with and without copy

Figure 5.7 compares the execution times for the calculation on the ARM core, the OpenCL kernel using local copies and the OpenCL kernel not using any internal copies. The unoptimized kernel needs up to 18.5 times the time the ARM core needs and 69 times the time the optimized kernel needs.

Another example that shows the difference between local and global memory performance is the FFT explained in Section 5.3. While the AES and SHA-256 algorithms have

dedicated arrays for input and output, this algorithm reads and writes the same array creating a dependency between the statements in lines 91 to 94 and 97 to 100 in Listing C.3. The compile log shows that the inner loop has an initiation interval of 24 cycles. Once the local copies are removed the initiation interval increases to 470 cycles. The timing results are shown in Figure 5.8. It can be seen that the execution time increases by a factor of 14.7 from 2.9 ms to 42.5 ms. It is interesting to notice that the execution time of the host stays the same and only the kernel time increases. This is the case because the changes are only inside the kernel and do not have any effect on the data that has to be transferred.

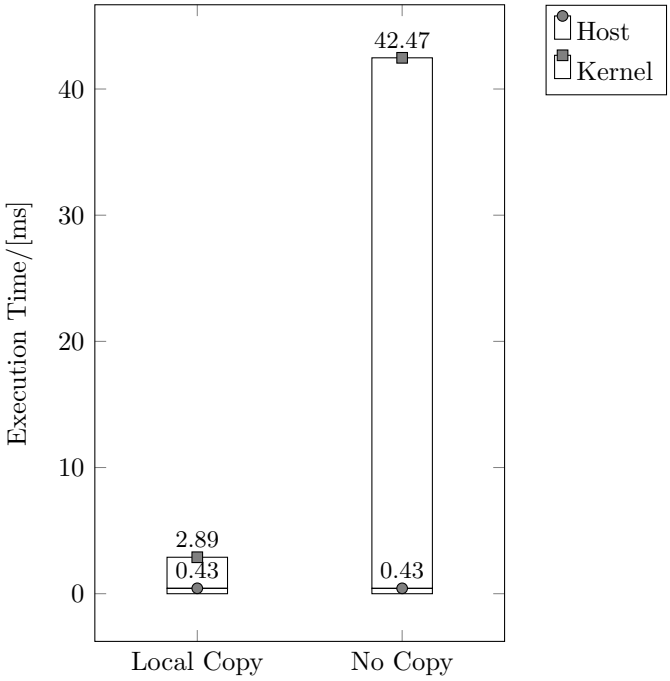


Figure 5.8: Breakdown execution time for the FFT example

It can be concluded that local copies should be created whenever possible. In cases the algorithm includes data dependencies or random memory access, this might have a great effect on reducing the initiation interval. In cases the AOC detects them to be obsolete they are optimized away. As explained in Section 4.1.9, the decision on which variables are copied to the local memory, is implemented as *first-come-first-serve*. In the future this could be optimized by taking into account which arrays are effected by data dependencies or random memory access.

5.5.3 Global and Constant Address Space

Section 4.1.9 explained that runtime constant variables can be saved to the constant address space. The only example that uses this concept with sufficient data is the CRC program. Nevertheless, in the final version, the data is loaded into the local memory which performs even better than the constant cache. Therefore, in order to compare the global and the constant address space, the OpenCL kernel without local copies from Section 5.5.2 is used as basis and modified further by changing the address space of the `icrctb` and the `rchr` array to constant.

The AOC compile log of the new version shows that the initiation interval is reduced from 165 to 11 clock cycles.

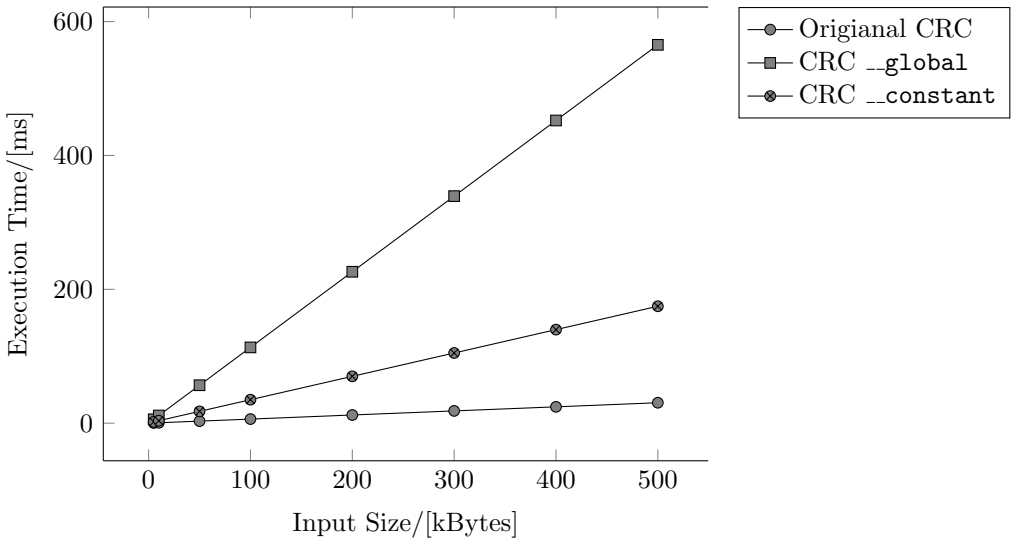


Figure 5.9: Execution time of the CRC algorithm using the constant or global address space for runtime constant arrays

Figure 5.9 compares the execution times of all three versions. It can be seen that the execution time of the OpenCL version using constant cache drops drastically. Compared to the OpenCL program using the global address space, the kernel performs 3.2 times faster. Nevertheless, the sequential program still runs faster than both OpenCL accelerated versions.

Therefore, it can be concluded, that constant memory can make a significant change to the kernel performance. Nevertheless, only a few situations are suitable. For once the data has to be accessed more than once and be ideally even runtime constant in order to exploit the constant cache that persists between kernel executions. On the other hand, local memory should always be preferred and therefore, the number of situations is limited further to those that use data too big to fit into the local address space or of unknown size.

5.6 Loop Unrolling

Section 4.1.10 explained the loop unrolling analysis that determines which loops should be unrolled by the AOC and which not.

From the examples explained previously, only the AES algorithm contains a loop that fits the requirements of the analysis. In order to evaluate the effect, a version of the kernel is created that does not contain the `#pragma unroll` statement in line 86 of Listing B.3.

The new compile log shows that the inner loop has an initiation interval of 4 clock cycles and the outer loop has one of 2 clock cycles. The reason for the initiation interval is the access to the ROM memory.

Figure 5.10 shows the change in the speedup factor. It can be seen that the maximum

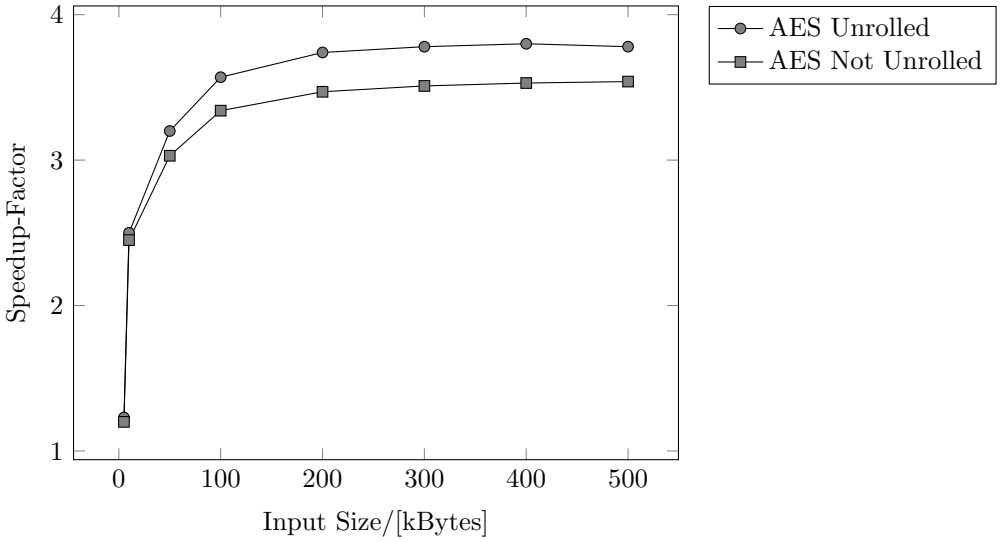


Figure 5.10: Speedup-factor for kernels with or without loop unrolling

speedup factor drops from 3.78 to 3.54. Furthermore, the break-even point increases from 3.3 kBytes to 3.6 kBytes.

Table 5.5: Comparison of resources occupied by OpenCL kernels with or without loop unrolling

	AES-CTR Not Unrolled/Unrolled
ALUTs	10,375 / 10,888
Registers	24,623 / 22,956
Logic utilization	32 % / 31 %
Memory bits	23 % / 42 %
M10K blocks	47 % / 78 %
DSP blocks	0 % / 0 %
Clock freq	134 MHz / 141 MHz

Table 5.5 compares the resource needed by the two OpenCL kernels. The amount of memory bits increases from 23 % to 42 % and the amount of M10K blocks increases from 47 % to 78 % when loop unrolling is activated. This can be explained by the increase of the pipeline structure.

The results show that loop unrolling is a powerful tool to increase the kernel performance. Nevertheless, it has to be taken into account that including the `unroll`-pragma will increase the kernel size and even might result in a solution that does not fit any longer into the FPGA. Furthermore, an unrolled loop only exploit instruction parallelism. Therefore unrolling perfectly pipelined loop might even reduce kernel performance. It can be concluded, that further research has to be spend to improve the loop unrolling analysis in order to include factors such as data dependencies, loop size and loop nesting.

5.7 File Scope Arrays

Section 3.1.1 explained that the user can declare static constant arrays by using the `Altera_OpenCL_const_vec` comment. These arrays are copied to the file scope of the OpenCL kernel. This section evaluated the effect on the execution time and kernel size.

The AES and the SHA-256 examples use constant arrays. In the SHA-256 the `K` array contains 64 values of 32 bit. By declaring them as constant arrays the 2 kbit of data are stored into a ROM once the kernel is loaded into the FPGA instead of transferring them every time the kernel is executed. The AES algorithm uses ten different constant arrays leading to a total of 70 kbit. The FFT and CRC examples do not use any static constant arrays. Therefore, they are not considered in this evaluation.

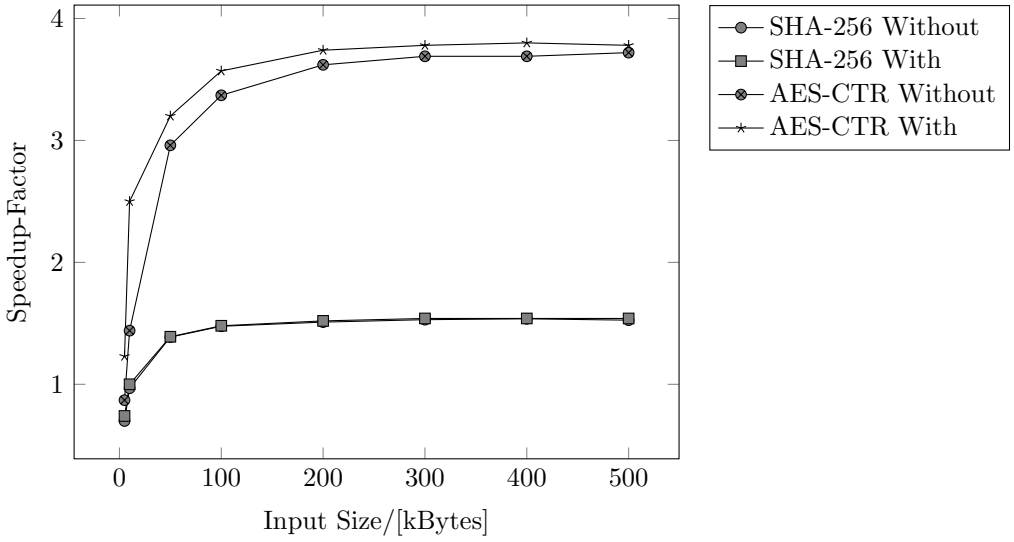


Figure 5.11: Speedup-factor for kernels with or without internal copy

A new version of each program was created by deleting the `Altera_OpenCL_const_vec` statements from line 40 seen in Listing A.1 and lines 35 to 44 in Listing B.1. Figure 5.11 shows the change in the speedup-factor for both examples. It can be seen that the maximum speedup-factor for the AES drops from 3.80 to 3.0 once the option is turned off. Furthermore, the break-even point increases from 3.3 kBytes input size to 5.8 kBytes input size. The effect on the SHA-256 algorithm is barely noticeable since the amount of data declared as constant array is smaller than in the AES example.

Table 5.6 compares the change in resources. It can be seen that the implementation without file scope arrays occupies more resources for both examples. Especially the number of registers increases. The reason for this is that accessing the global address space requires more logic than accessing the internal ROM memory. Furthermore, the maximum cock frequency is decreasing.

It can be concluded that the `Altera_OpenCL_const_vec` comment should be used whenever possible. Nevertheless the effect depends highly on the amount of data that is copied.

Table 5.6: Comparison of resources occupied by OpenCL kernels with or without file scope arrays

	SHA-256 Without/With	**AES-CTR** Without/With
ALUTs	10,481/ 8,720	15,601 / 10,888
Registers	20,742/ 19,036	52,342 / 22,956
Logic utilization	29 % / 27 %	58 % / 31 %
Memory bits	10 % / 8 %	32 % / 42 %
M10K blocks	21 % / 17 %	81 % / 78 %
DSP blocks	0 % / 0 %	0 % / 0 %
Clock freq	112 % / 112 MHz	128 MHz / 141 MHz

Chapter 6

Conclusion

A C/C++ source-to-source-compiler has been developed and evaluated successfully in this work. It was shown that the SOCAO compiler is able to generate a working OpenCL accelerated program which produces a noticeable speedup compared to the original version. In the evaluated cases the maximum speedup was between 1.54 and 3.78. Nevertheless, this project also showed that not every algorithm is suitable to be implemented as single-work-item kernel.

The main challenge was to implement analysis and transformations that make the best possible choices for any input program. Therefore, future work should focus on the improvement of the existing analysis and transformations and the implementation of new ones. At the current point no form of data dependency analysis is implemented. Nevertheless, loop-carried dependencies effect the kernel performance more than any other characteristic. Therefore, a number of analysis and transformations could benefit when they are taken into account.

One example is the loop unrolling analysis. The current analysis does not evaluate enough characteristics. The size of the loop body, data dependencies, operation types and the overall structure of the kernel have effects which determine whether unrolling a loop might results into better or worse kernel performance. A bad choice here might result into a kernel that is not compilable, to big to fit into the FPGA or runs even slower than the original algorithm. Furthermore, the current analysis implements a *all-or-nothing* approach. In some cases it might be useful to unroll a loop only to a certain extend in order exploit consecutive memory access.

The other example is the memory analysis. Currently the data copied to the local memory is chosen without any priority. Nevertheless, arrays that create a loop-carried dependency should be preferred in order to reduce the initiation interval of the loop.

The SOCAO compiler only generates single-work-item kernel which are executed as tasks. This introduces limitations on the algorithms that can be translated successfully into OpenCL. Future work should therefore also focus on the automated generation of NDRange kernel. Nevertheless, this also relies on a data dependency analysis.

Last, the introduced implementation focuses on the Altera's SDK for Opencl 15.1. The newer 16.0 version has some improvements for single work-item kernels implemented such as the `ivdep` pragma that indicates that the access to memory does not conflict with

loop-carried dependencies. This introduces some interesting possibilities to improve the automatically generated kernel. Nevertheless, the 16.0 version does not work out of the box on the DE1-SoC platform since it requires newer libraries. Therefore it wasn't taken into account in this thesis.

Appendix A

SHA-256 Source Code

The SHA-256 input source code is part of the *mbed TLS* library [17]. Listing A.1 shows the input source file and Listing A.2 the host program generated by the SOCAO compiler. The OpenCL kernel can be found in Listing A.3.

For reasons of clarity, Listings A.1 and A.2 show only the relevant parts. The missing functions can be downloaded directly from the homepage of the *mbed TLS* library [17].

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
#include "mbedtls/sha256.h"
#include "mbedtls/platform.h"
#include "mbedtls/config.h"
using namespace aocl_utils;

#define  SHR(x,n) ((x & 0xFFFFFFFF) >> n)
#define ROTR(x,n) (SHR(x,n) | (x << (32 - n)))

#define S0(x) (ROTR(x, 7) ^ ROTR(x,18) ^  SHR(x, 3))
#define S1(x) (ROTR(x,17) ^ ROTR(x,19) ^  SHR(x,10))

#define S2(x) (ROTR(x, 2) ^ ROTR(x,13) ^ ROTR(x,22))
#define S3(x) (ROTR(x, 6) ^ ROTR(x,11) ^ ROTR(x,25))

#define F0(x,y,z) ((x & y) | (z & (x | y)))
#define F1(x,y,z) (z ^ (x & (y ^ z)))

#define R(t)                                        \
(                                                   \
  W[t] = S1(W[t -  2]) + W[t -  7] +                \
    S0(W[t - 15]) + W[t - 16]                       \
)

#define P(a,b,c,d,e,f,g,h,x,K)                      \
{                                                   \
  temp1 = h + S3(e) + F1(e,f,g) + K + x;            \
  temp2 = S2(a) + F0(a,b,c);                        \
  d += temp1; h = temp1 + temp2;                    \
}
```

```
36 //Altera_OpenCL_Accelerate
   //Altera_OpenCL_size input len
38 //Altera_OpenCL_size K 64
   //Altera_OpenCL_size state 8
40 //Altera_OpenCL_const_vec K
   //Altera_OpenCL_soc
42 void mbedtls_sha256_update_accelerated( uint32_t *state, const unsigned ↩
       char *input, uint64_t len )
   {
44   uint64_t ilen = len;
     for(int k = 0; (k+64) <= ilen; k+=64)
46   {
       uint32_t temp1, temp2, W[64];
48     uint32_t A[8];
       int j;
50     for( j = 0; j < 8; j++ )
         A[j] = state[j];
52     for( int i = 0; i < 64; i++ )
       {
54       if( i < 16)
           W[i] = ((uint32_t )input[k+4 * i]) << 24 | ((uint32_t )input[k+4 *↩
        i + 1]) << 16 | ((uint32_t )input[k+4 * i + 2]) << 8 | ((uint32_t )↩
       input[k+4 * i + 3]);
56       else
           R( i );
58
         P( A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], W[i], K[i] );
60       temp1 = A[7]; A[7] = A[6]; A[6] = A[5]; A[5] = A[4]; A[4] = A[3];
         A[3] = A[2]; A[2] = A[1]; A[1] = A[0]; A[0] = temp1;
62     }
       for( j = 0; j < 8; j++ )
64       state[j] += A[j];
       }
66 }

68 void mbedtls_sha256_update( mbedtls_sha256_context *ctx, const unsigned ↩
       char *input,  size_t ilen )
   {
70   size_t fill;
     uint32_t left;
72
     if( ilen == 0 )
74     return;

76   left = ctx->total[0] & 0x3F;
     fill = 64 - left;
78
     ctx->total[0] += (uint32_t) ilen;
80   ctx->total[0] &= 0xFFFFFFFF;

82   if( ctx->total[0] < (uint32_t) ilen )
       ctx->total[1]++;
84   if( left && ilen >= fill )
     {
86     for(int i = 0; i < fill; i++)
         ctx->buffer[left+i] = input[i];
88     mbedtls_sha256_process( ctx, ctx->buffer );
       input += fill; ilen  -= fill; left = 0;
```

```
  }

  if(ilen > 64)
    mbedtls_sha256_update_accelerated(ctx->state,input,(uint64_t)ilen);

  if( ilen % 64 > 0 )
  {
    input += (ilen/64)*64;
    for(int i = 0; i < ilen%64; i++)
      ctx->buffer[left+i] = input[i];
  }
}

void mbedtls_sha256( const unsigned char *input, size_t ilen, unsigned char output[32], int is224 )
{
  mbedtls_sha256_context ctx;
  mbedtls_sha256_init( &ctx );
  mbedtls_sha256_starts( &ctx, is224 );
  mbedtls_sha256_update( &ctx, input, ilen );
  mbedtls_sha256_finish( &ctx, output );
  mbedtls_sha256_free( &ctx );
}


int main (int argc, char *argv[])
{
  int len;
  if(argc > 1)
    len= atoi(argv[1]);
  else
    len =64000;

    unsigned char *max_message = (unsigned char*)malloc(len*sizeof(unsigned char));
  memset( max_message, 'a', len );
  unsigned char output[32];
  double timecounter;
  timer_clear(0); timer_start(0);
  for(int i = 0; i < 100; i++)
  {
    mbedtls_sha256( max_message, len*sizeof(unsigned char), output, false );
  }
  timer_stop(0);
  timecounter = timer_read(0);
  printf("\tresult: %X %X %X\n", output[0], output[1], output[2]);
  printf("\tTime [ms] \t= \t %f \n",timecounter*10);
}
```

Listing A.1: SHA-256 SOCAO input file

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
#include "mbedtls/sha256.h"
#include "mbedtls/platform.h"
```

```
#include "mbedtls/config.h"
using namespace aocl_utils;

#define  SHR(x,n) ((x & 0xFFFFFFFF) >> n)
#define ROTR(x,n) (SHR(x,n) | (x << (32 - n)))
#define S0(x) (ROTR(x, 7) ^ ROTR(x,18) ^  SHR(x, 3))
#define S1(x) (ROTR(x,17) ^ ROTR(x,19) ^  SHR(x,10))
#define S2(x) (ROTR(x, 2) ^ ROTR(x,13) ^ ROTR(x,22))
#define S3(x) (ROTR(x, 6) ^ ROTR(x,11) ^ ROTR(x,25))
#define F0(x,y,z) ((x & y) | (z & (x | y)))
#define F1(x,y,z) (z ^ (x & (y ^ z)))
#define R(t)                                    \
(                                               \
  W[t] = S1(W[t -  2]) + W[t -  7] +            \
    S0(W[t - 15]) + W[t - 16]                   \
)

#define P(a,b,c,d,e,f,g,h,x,K)                  \
{                                               \
  temp1 = h + S3(e) + F1(e,f,g) + K + x;        \
  temp2 = S2(a) + F0(a,b,c);                    \
  d += temp1; h = temp1 + temp2;                \
}

void cleanup();
bool init_opencl(const char *file_name,const char *kernel_name);
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
cl_mem inout_state_buf = 0;
uint32_t *inout_state_ptr = 0;
cl_mem input_input_buf = 0;
int aocl_input_old_size = 0;
unsigned char *input_input_ptr = 0;

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size input len
//Altera_OpenCL_size K 64
//Altera_OpenCL_size state 8
//Altera_OpenCL_const_vec K
//Altera_OpenCL_soc

void mbedtls_sha256_update_accelerated(uint32_t *state,const unsigned char↩
    *input,uint64_t len)
{
  uint64_t ilen = len;
  cl_int status;
  cl_event kernel_event;
  unsigned int argi = 0;
  if (!inout_state_buf) {
    inout_state_buf = clCreateBuffer(context,0x1UL |      ↩
   CL_MEM_ALLOC_HOST_PTR,8 * sizeof(uint32_t ),0,&status);
    checkError(status,"Failed to create buffer");
    inout_state_ptr = ((uint32_t *)(clEnqueueMapBuffer(queue,↩
   inout_state_buf,CL_TRUE,0,0,8 * sizeof(uint32_t ),0,0,0,&status)));
    checkError(status,"Failed to map buffer");
```

```
  }
  if (!input_input_buf) {
      input_input_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR,len * sizeof(unsigned char ),0,&status);
      checkError(status,"Failed to create buffer");
      input_input_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,input_input_buf,CL_TRUE,CL_MAP_READ,0,len * sizeof(unsigned char ),0,0,0,&status)));
      checkError(status,"Failed to map buffer");
      aocl_input_old_size = len;
  }
  else {
    if (aocl_input_old_size < len) {
        clEnqueueUnmapMemObject(queue,input_input_buf,input_input_ptr,0,0,0);
      clReleaseMemObject(input_input_buf);
      input_input_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR,len * sizeof(unsigned char ),0,&status);
      checkError(status,"Failed to create buffer");
      input_input_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,input_input_buf,CL_TRUE,CL_MAP_READ,0,len * sizeof(unsigned char ),0,0,0,&status)));
      checkError(status,"Failed to map buffer");
  aocl_input_old_size = len;
    }
  }
  memcpy(inout_state_ptr,state,8 * sizeof(uint32_t ));
  memcpy(input_input_ptr,input,len * sizeof(unsigned char ));
  status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&inout_state_buf);
  checkError(status,"Failed to set argument");
  status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&input_input_buf);
  checkError(status,"Failed to set argument");
  status = clSetKernelArg(kernel,argi++,sizeof(uint64_t ),&len);
  checkError(status,"Failed to set argument");
  status = clEnqueueTask(queue,kernel,0,0,&kernel_event);
  checkError(status,"Failed to launch kernel");
  clWaitForEvents(1,&kernel_event);
  memcpy(state,inout_state_ptr,8 * sizeof(uint32_t ));
  clReleaseEvent(kernel_event);
}

int main (int argc, char *argv[])
{

  if (!init_opencl("aocl_kernel","aocl_generated_kernel")) {
    printf("WARNING: COULDN'T INITIALIZE OPENCL\n");
    return -1;
  }
  int len;
  if(argc > 1)
    len= atoi(argv[1]);
  else
    len =64000;

    unsigned char *max_message = (unsigned char*)malloc(len*sizeof(unsigned char));
  memset( max_message, 'a', len );
```

```
113   unsigned char output[32];
      double timecounter;
115   timer_clear(0); timer_start(0);
      for(int i = 0; i < 100; i++)
117   {
        mbedtls_sha256( max_message, len*sizeof(unsigned char), output, false );
119   }
      timer_stop(0);
121   timecounter = timer_read(0);
      printf("\tSize = \t%d\n", len);
123   printf("\tresult: %X %X %X\n", output[0], output[1], output[2]);
      printf("\tTime [ms] \t= \t %f \n",timecounter*10);
125   printf("\tTime kernel [ms] \t= \t %f \n",total_time_kernel/1000000.0);
      cleanup();
127 }

129 bool init_opencl(const char *file_name,const char *kernel_name)
    {
131   cl_int status;
      cl_device_id *curr_devices;
133   cl_uint num_devices;
      cl_device_type cl_device_type_all = 0xffffffffUL;
135   cl_command_queue_properties cl_queue_profiling_enable = 0x2;
      std::string binary_file;
137   if (!setCwdToExeDir()) {
        return false;
139   }
      platform = findPlatform("Altera");
141   if (platform == 0) {
        printf("ERROR: Unable to find Altera OpenCL platform.\n");
143       return false;
      }
145   curr_devices = getDevices(platform,cl_device_type_all,&num_devices);
      device = curr_devices[0];
147   context = clCreateContext(0,1,&device,&oclContextCallback,0,&status);
      checkError(status,"Failed to create context");
149   queue = clCreateCommandQueue(context,device,cl_queue_profiling_enable,&
        status);
      checkError(status,"Failed to create command queue");
151   binary_file = getBoardBinaryFile(file_name,device);
      program = createProgramFromBinary(context,(binary_file .  c_str ()),&
        device,1);
153   status = clBuildProgram(program,0,0,"",0,0);
      checkError(status,"Failed to build program");
155   kernel = clCreateKernel(program,kernel_name,&status);
      checkError(status,"Failed to create kerne");
157   return true;
    }
159

    void cleanup()
161 {
      if (inout_state_buf) {
163       clEnqueueUnmapMemObject(queue, inout_state_buf, inout_state_ptr
        ,0,0,0);
        clReleaseMemObject(inout_state_buf);
165   }
      if (input_input_buf) {
167     clEnqueueUnmapMemObject(queue, input_input_buf, input_input_ptr, 0, 0,
```

```
    0);
    clReleaseMemObject(input_input_buf);
  }
  if (kernel) {
    clReleaseKernel(kernel);
  }
  if (program) {
    clReleaseProgram(program);
  }
  if (queue) {
    clReleaseCommandQueue(queue);
  }
  if (context) {
    clReleaseContext(context);
  }
  return ;
}
```

Listing A.2: SHA-256 host program

```
typedef long       intmax_t;
typedef ulong      uintmax_t;
typedef char     int_least8_t;
typedef uchar    uint_least8_t;
typedef short      int_least16_t;
typedef ushort     uint_least16_t;
typedef int        int_least32_t;
typedef uint       uint_least32_t;
typedef long       int_least64_t;
typedef ulong      uint_least64_t;
typedef char     int_fast8_t;
typedef uchar    uint_fast8_t;
typedef short      int_fast16_t;
typedef ushort     uint_fast16_t;
typedef int        int_fast32_t;
typedef uint       uint_fast32_t;
typedef long       int_fast64_t;
typedef ulong      uint_fast64_t;

typedef char     int8_t;
typedef uchar    uint8_t;
typedef short      int16_t;
typedef ushort     uint16_t;
typedef int        int32_t;
typedef uint       uint32_t;
typedef long       int64_t;
typedef ulong      uint64_t;


__constant const uint32_t K[] = { ... };

__kernel void aocl_generated_kernel(uint32_t __global * __restrict__ ↩
    state_global,unsigned char __global * __restrict__ input,const uint64_t↩
     len)
{
  int rose_it;
  __local uint32_t state[8];
  for (rose_it = 0; rose_it < 8UL; ++rose_it) {
    state[rose_it] = state_global[rose_it];
```

```
  }
  uint64_t ilen = len;
  for (int k = 0; (k + 64) <= ilen; k += 64) {
    uint32_t temp1;
    uint32_t temp2;
    uint32_t W[64];
    uint32_t A[8];
    int j;

    #pragma unroll
    for (j = 0; j < 8; j++)
      A[j] = state[j];
    for (int i = 0; i < 64; i++) {
      if (i < 16) {
        W[i] = ((uint32_t )input[k + 4 * i]) << 24 | ((uint32_t )input[k +↩
     4 * i + 1]) << 16 | ((uint32_t )input[k + 4 * i + 2]) << 8 | ((↩
    uint32_t )input[k + 4 * i + 3]);
      }
      else {
        W[i] = (((W[i - 2] & 0xffffffffu) >> 17 | W[i - 2] << 15) ^ ((W[i ↩
    - 2] & 0xffffffffu) >> 19 | W[i - 2] << 13) ^ (W[i - 2] & 0xffffffffu) ↩
    >> 10) + W[i - 7] + (((W[i - 15] & 0xffffffffu) >> 7 | W[i - 15] << 25)↩
     ^ ((W[i - 15] & 0xffffffffu) >> 18 | W[i - 15] << 14) ^ (W[i - 15] & 0↩
    xffffffffu) >> 3) + W[i - 16];
      }
      {
        temp1 = A[7] + (((A[4] & 0xffffffffu) >> 6 | A[4] << 26) ^ ((A[4] ↩
    & 0xffffffffu) >> 11 | A[4] << 21) ^ ((A[4] & 0xffffffffu) >> 25 | A[4]↩
     << 7)) + (A[6] ^ A[4] & (A[5] ^ A[6])) + K[i] + W[i];
        temp2 = (((A[0] & 0xffffffffu) >> 2 | A[0] << 30) ^ ((A[0] & 0↩
    xffffffffu) >> 13 | A[0] << 19) ^ ((A[0] & 0xffffffffu) >> 22 | A[0] <<↩
     10)) + (A[0] & A[1] | A[2] & (A[0] | A[1]));
    A[3] += temp1;
    A[7] = temp1 + temp2;
    }
    ;
    temp1 = A[7];
    A[7] = A[6];
    A[6] = A[5];
    A[5] = A[4];
    A[4] = A[3];
    A[3] = A[2];
    A[2] = A[1];
    A[1] = A[0];
    A[0] = temp1;
  }

  #pragma unroll
  for (j = 0; j < 8; j++)
    state[j] += A[j];
  }
  for (rose_it = 0; rose_it < 8UL; ++rose_it) {
    state_global[rose_it] = state[rose_it];
  }
}
```

Listing A.3: SHA-256 kernel

Appendix B

AES Source Code

The AES input source code is part of the *mbed TLS* library [17]. Listing B.1 shows the input source file and Listing B.2 the host program generated by the SOCAO compiler. The OpenCL kernel can be found in Listing B.3.

For reasons of clarity, Listings B.1 and B.2 show only the relevant parts. The missing functions can be downloaded directly from the homepage of the *mbed TLS* library [17]. Furthermore, the constant values of the OpenCL function in Listing B.3 were removed in order to reduce the print size.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
#include "mbedtls/config.h"
#include "mbedtls/platform.h"
#include "mbedtls/aes.h"
using namespace aocl_utils;

#ifndef GET_UINT32_LE
#define GET_UINT32_LE(n,b,i)                            \
{                                                       \
    (n) = ( (uint32_t) (b)[(i)    ]       )             \
        | ( (uint32_t) (b)[(i) + 1] <<  8 )             \
        | ( (uint32_t) (b)[(i) + 2] << 16 )             \
        | ( (uint32_t) (b)[(i) + 3] << 24 );            \
}
#endif

#ifndef PUT_UINT32_LE
#define PUT_UINT32_LE(n,b,i)                                    \
{                                                               \
    (b)[(i)    ] = (unsigned char) ( ( (n)       ) & 0xFF );    \
    (b)[(i) + 1] = (unsigned char) ( ( (n) >>  8 ) & 0xFF );    \
    (b)[(i) + 2] = (unsigned char) ( ( (n) >> 16 ) & 0xFF );    \
    (b)[(i) + 3] = (unsigned char) ( ( (n) >> 24 ) & 0xFF );    \
}
#endif

//Altera_OpenCL_Accelerate
```

```
//Altera_OpenCL_size input length
//Altera_OpenCL_size output length
//Altera_OpenCL_size RK 68
//Altera_OpenCL_const_vec FT3
//Altera_OpenCL_const_vec FT2
//Altera_OpenCL_const_vec FT1
//Altera_OpenCL_const_vec FT0
//Altera_OpenCL_const_vec RT3
//Altera_OpenCL_const_vec RT2
//Altera_OpenCL_const_vec RT1
//Altera_OpenCL_const_vec RT0
//Altera_OpenCL_const_vec FSb
//Altera_OpenCL_const_vec RSb
//Altera_OpenCL_soc
int mbedtls_aes_crypt_ctr_nr10( uint32_t *RK, uint64_t length, unsigned char nonce_counter[16], unsigned char stream_block[16], const unsigned char *input, unsigned char *output )
{
  uint32_t int_counter = ((uint32_t )nonce_counter[12]) << 24 | ((uint32_t)nonce_counter[13]) << 16 | ((uint32_t )nonce_counter[14]) <<8 | ((uint32_t )nonce_counter[15]) << 0;
  uint64_t k;
  for(k = 0; k < length; k+=16)
  {

    uint32_t  X0, X1, X2, X3, Y0, Y1, Y2, Y3;
    GET_UINT32_LE( X0, nonce_counter,  0 ); X0 ^= RK[0];
    GET_UINT32_LE( X1, nonce_counter,  4 ); X1 ^= RK[1];
    GET_UINT32_LE( X2, nonce_counter,  8 ); X2 ^= RK[2];
    X3 = ((int_counter>> 24) & 0xff) | ((int_counter<<8) & 0xff0000) | ((int_counter >> 8) & 0xff00) | ((int_counter<<24) & 0xff000000);
    X3 ^= RK[3];
    int i;
    for(i = 0; i < 4; i++)
    {
      int idx = i*8;
      //printf("idx = %d\t", idx);
      Y0 =  RK[idx + 4] ^ FT0[X0 & 0xFF] ^ FT1[X1 >> 8 & 0xFF] ^ FT2[X2 >> 16 & 0xFF] ^ FT3[X3 >> 24 & 0xFF];
      Y1 =  RK[idx + 5] ^ FT0[X1 & 0xFF] ^ FT1[X2 >> 8 & 0xFF] ^ FT2[X3 >> 16 & 0xFF] ^ FT3[X0 >> 24 & 0xFF];
      Y2 =  RK[idx + 6] ^ FT0[X2 & 0xFF] ^ FT1[X3 >> 8 & 0xFF] ^ FT2[X0 >> 16 & 0xFF] ^ FT3[X1 >> 24 & 0xFF];
      Y3 =  RK[idx + 7] ^ FT0[X3 & 0xFF] ^ FT1[X0 >> 8 & 0xFF] ^ FT2[X1 >> 16 & 0xFF] ^ FT3[X2 >> 24 & 0xFF];

      X0 =  RK[idx + 8] ^ FT0[Y0 & 0xFF] ^ FT1[Y1 >> 8 & 0xFF] ^ FT2[Y2 >> 16 & 0xFF] ^ FT3[Y3 >> 24 & 0xFF];
      X1 =  RK[idx + 9] ^ FT0[Y1 & 0xFF] ^ FT1[Y2 >> 8 & 0xFF] ^ FT2[Y3 >> 16 & 0xFF] ^ FT3[Y0 >> 24 & 0xFF];
      X2 =  RK[idx + 10] ^ FT0[Y2 & 0xFF] ^ FT1[Y3 >> 8 & 0xFF] ^ FT2[Y0 >> 16 & 0xFF] ^ FT3[Y1 >> 24 & 0xFF];
      X3 =  RK[idx + 11] ^ FT0[Y3 & 0xFF] ^ FT1[Y0 >> 8 & 0xFF] ^ FT2[Y1 >> 16 & 0xFF] ^ FT3[Y2 >> 24 & 0xFF];

    }
    int idx = 36;

    Y0 =  RK[idx] ^ FT0[X0 & 0xFF] ^ FT1[X1 >> 8 & 0xFF] ^ FT2[X2 >> 16 & 0xFF] ^ FT3[X3 >> 24 & 0xFF];
    Y1 =  RK[idx + 1] ^ FT0[X1 & 0xFF] ^ FT1[X2 >> 8 & 0xFF] ^ FT2[X3 >> 16 & 0xFF] ^ FT3[X0 >> 24 & 0xFF];
    Y2 =  RK[idx + 2] ^ FT0[X2 & 0xFF] ^ FT1[X3 >> 8 & 0xFF] ^ FT2[X0 >> 16 & 0xFF] ^ FT3[X1 >> 24 & 0xFF];
    Y3 =  RK[idx + 3] ^ FT0[X3 & 0xFF] ^ FT1[X0 >> 8 & 0xFF] ^ FT2[X1 >> 16 & 0xFF] ^ FT3[X2 >> 24 & 0xFF];

    PUT_UINT32_LE( Y0, stream_block,  0 );
    PUT_UINT32_LE( Y1, stream_block,  4 );
    PUT_UINT32_LE( Y2, stream_block,  8 );
    PUT_UINT32_LE( Y3, stream_block, 12 );

    int_counter++;

    int i;
    for(i = 0; i < 16; i++)
      output[i] = input[i] ^ stream_block[i];

    output += 16;
    input += 16;
  }
  return 0;
}
```

```
    0xFF] ^ FT3[X3 >> 24 & 0xFF];
78  Y1 =  RK[idx + 1] ^ FT0[X1 & 0xFF] ^ FT1[X2 >> 8 & 0xFF] ^ FT2[X3 >> ↩
    16 & 0xFF] ^ FT3[X0 >> 24 & 0xFF];
    Y2 =  RK[idx + 2] ^ FT0[X2 & 0xFF] ^ FT1[X3 >> 8 & 0xFF] ^ FT2[X0 >> ↩
    16 & 0xFF] ^ FT3[X1 >> 24 & 0xFF];
80  Y3 =  RK[idx + 3] ^ FT0[X3 & 0xFF] ^ FT1[X0 >> 8 & 0xFF] ^ FT2[X1 >> ↩
    16 & 0xFF] ^ FT3[X2 >> 24 & 0xFF];

82

    X0 = RK[idx + 4] ^ \
84      ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
86      ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );
88
    X1 = RK[idx + 5] ^ \
90      ( (uint32_t) FSb[ ( Y1       ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( Y2 >>  8 ) & 0xFF ] <<  8 ) ^
92      ( (uint32_t) FSb[ ( Y3 >> 16 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( Y0 >> 24 ) & 0xFF ] << 24 );
94
    X2 = RK[idx + 6] ^ \
96      ( (uint32_t) FSb[ ( Y2       ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( Y3 >>  8 ) & 0xFF ] <<  8 ) ^
98      ( (uint32_t) FSb[ ( Y0 >> 16 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( Y1 >> 24 ) & 0xFF ] << 24 );
100
    X3 = RK[idx + 7] ^ \
102     ( (uint32_t) FSb[ ( Y3       ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( Y0 >>  8 ) & 0xFF ] <<  8 ) ^
104     ( (uint32_t) FSb[ ( Y1 >> 16 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( Y2 >> 24 ) & 0xFF ] << 24 );
106
    PUT_UINT32_LE( X0, stream_block,  0 );
108 PUT_UINT32_LE( X1, stream_block,  4 );
    PUT_UINT32_LE( X2, stream_block,  8 );
110 PUT_UINT32_LE( X3, stream_block, 12 );

112

    int_counter++;
114
    int c = input[k];
116 output[k] = ((unsigned char )(c ^ stream_block[0]));
    if (k + 1 < length) {
118   c = input[k + 1];
      output[k+1] = ((unsigned char )(c ^ stream_block[1]));
120 }
    if (k + 2 < length) {
122   c = input[k + 2];
      output[k+2] = ((unsigned char )(c ^ stream_block[2]));
124 }
    if (k + 3 < length) {
126   c = input[k + 3];
      output[k+3] = ((unsigned char )(c ^ stream_block[3]));
128 }
    if (k + 4 < length) {
130   c = input[k + 4];
      output[k+4] = ((unsigned char )(c ^ stream_block[4]));
132 }
```

```
    if (k + 5 < length) {
      c = input[k + 5];
      output[k+5] = ((unsigned char )(c ^ stream_block[5]));
    }
    if (k + 6 < length) {
      c = input[k + 6];
      output[k+6] = ((unsigned char )(c ^ stream_block[6]));
    }
    if (k + 7 < length) {
      c = input[k + 7];
      output[k+7] = ((unsigned char )(c ^ stream_block[7]));
    }
    if (k + 8 < length) {
      c = input[k + 8];
      output[k+8] = ((unsigned char )(c ^ stream_block[8]));
    }
    if (k + 9 < length) {
      c = input[k + 9];
      output[k+9] = ((unsigned char )(c ^ stream_block[9]));
    }
    if (k + 10 < length) {
      c = input[k + 10];
      output[k+10] = ((unsigned char )(c ^ stream_block[10]));
    }
    if (k + 11 < length) {
      c = input[k + 11];
      output[k+11] = ((unsigned char )(c ^ stream_block[11]));
    }
    if (k + 12 < length) {
      c = input[k + 12];
      output[k+12] = ((unsigned char )(c ^ stream_block[12]));
    }
    if (k + 13 < length) {
      c = input[k + 13];
      output[k+13] = ((unsigned char )(c ^ stream_block[13]));
    }
    if (k + 14 < length) {
      c = input[k + 14];
      output[k+14] = ((unsigned char )(c ^ stream_block[14]));
    }
    if (k + 15 < length) {
      c = input[k + 15];
      output[k+15] = ((unsigned char )(c ^ stream_block[15]));
    }
  }

  nonce_counter[12] = (unsigned char) (int_counter >> 24);
  nonce_counter[13] = (unsigned char) (int_counter >> 16);
  nonce_counter[14] = (unsigned char) (int_counter >> 8);
  nonce_counter[15] = (unsigned char) (int_counter);
  return( 0 );
}

int main (int argc, char *argv[])
{
  int len;
  if(argc > 1)
    len= atoi(argv[1]);
  else
```

```
192     len =64000;

194   printf("\tSize = \t%d\n", len);
      unsigned char *max_message = (unsigned char*)malloc(len*sizeof(unsigned ↩
        char));
196   unsigned char *output_message = (unsigned char*)malloc(len*sizeof(↩
        unsigned char));
      memset( max_message, 'a', len );
198   memset( output_message, 0, len );

200   mbedtls_aes_context ctx;
      mbedtls_aes_init( &ctx );
202   unsigned char key[32];

204   unsigned char nonce_counter[16];
      unsigned char stream_block[16];
206   memset( key, 0, 32 );
      memcpy( key, aes_test_ctr_key[0], 16 );
208   memset( nonce_counter, 0, 16 );

210   uint64_t offset = 0;
      mbedtls_aes_setkey_enc( &ctx, key, 128 );
212   double timecounter;
      timer_clear(0); timer_start(0);
214   for(int i = 0; i < 100; i++)
      {
216     mbedtls_aes_crypt_ctr_nr10( ctx.rk, (uint64_t)len, nonce_counter, ↩
        stream_block, max_message, output_message );
      }
218   timer_stop(0);
      timecounter = timer_read(0);
220   printf("\tresult: %X %X %X\n", output_message[0], output_message[len/2],↩
        output_message[len-1]);
      printf("\tTime [ms] \t= \t %f \n",timecounter*10);
222   return 0;
}
```

Listing B.1: AES SOCAO input file

```
1  #include <stdlib.h>
   #include <stdio.h>
3  #include <string.h>
   #include "CL/opencl.h"
5  #include "AOCLUtils/aocl_utils.h"
   #include "mbedtls/config.h"
7  #include "mbedtls/platform.h"
   #include "mbedtls/aes.h"
9  using namespace aocl_utils;

11 #ifndef GET_UINT32_LE
   #define GET_UINT32_LE(n,b,i)                                \
13 {                                                           \
       (n) = ( (uint32_t) (b)[(i)    ]       )               \
15         | ( (uint32_t) (b)[(i) + 1] <<  8 )               \
           | ( (uint32_t) (b)[(i) + 2] << 16 )               \
17         | ( (uint32_t) (b)[(i) + 3] << 24 );              \
   }
19 #endif
```

```
#ifndef PUT_UINT32_LE
#define PUT_UINT32_LE(n,b,i)                                            \
{                                                                       \
    (b)[(i)    ] = (unsigned char) ( ( (n)       ) & 0xFF );     \
    (b)[(i) + 1] = (unsigned char) ( ( (n) >>  8 ) & 0xFF );     \
    (b)[(i) + 2] = (unsigned char) ( ( (n) >> 16 ) & 0xFF );     \
    (b)[(i) + 3] = (unsigned char) ( ( (n) >> 24 ) & 0xFF );     \
}
#endif

void cleanup();
bool init_opencl(const char *file_name,const char *kernel_name);
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
cl_mem input_RK_buf = 0;
uint32_t *input_RK_ptr = 0;
cl_mem inout_nonce_counter_buf = 0;
unsigned char *inout_nonce_counter_ptr = 0;
cl_mem inout_stream_block_buf = 0;
unsigned char *inout_stream_block_ptr = 0;
cl_mem input_input_buf = 0;
int aocl_input_old_size = 0;
unsigned char *input_input_ptr = 0;
cl_mem output_output_buf = 0;
int aocl_output_old_size = 0;
unsigned char *output_output_ptr = 0;

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size input length
//Altera_OpenCL_size output length
//Altera_OpenCL_size RK 68
//Altera_OpenCL_const_vec FT3
//Altera_OpenCL_const_vec FT2
//Altera_OpenCL_const_vec FT1
//Altera_OpenCL_const_vec FT0
//Altera_OpenCL_const_vec RT3
//Altera_OpenCL_const_vec RT2
//Altera_OpenCL_const_vec RT1
//Altera_OpenCL_const_vec RT0
//Altera_OpenCL_const_vec FSb
//Altera_OpenCL_const_vec RSb
//Altera_OpenCL_soc
int mbedtls_aes_crypt_ctr_nr10(uint32_t *RK,uint64_t length,unsigned char ↩
    nonce_counter[16],unsigned char stream_block[16],const unsigned char *↩
    input,unsigned char *output)
{
  uint32_t int_counter = ((uint32_t )nonce_counter[12]) << 24 | ((uint32_t↩
     )nonce_counter[13]) << 16 | ((uint32_t )nonce_counter[14]) << 8 | ((↩
    uint32_t )nonce_counter[15]) << 0;
  uint64_t k;
  cl_int status;
  cl_event kernel_event;
  unsigned int argi = 0;
  if (!input_RK_buf) {
    input_RK_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR,68↩
```

```
    * sizeof(uint32_t ),0,&status);
    checkError(status,"Failed to create buffer");
    input_RK_ptr = ((uint32_t *)(clEnqueueMapBuffer(queue,input_RK_buf,↩
  CL_TRUE,CL_MAP_READ,0,68 * sizeof(uint32_t ),0,0,0,&status)));
    checkError(status,"Failed to map buffer");
  }
  if (!inout_nonce_counter_buf) {
    inout_nonce_counter_buf = clCreateBuffer(context,0x1UL | ↩
  CL_MEM_ALLOC_HOST_PTR,16 * sizeof(unsigned char ),0,&status);
    checkError(status,"Failed to create buffer");
    inout_nonce_counter_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
  inout_nonce_counter_buf,CL_TRUE,0,0,16 * sizeof(unsigned char ),0,0,0,&↩
  status)));
    checkError(status,"Failed to map buffer");
  }
  if (!inout_stream_block_buf) {
    inout_stream_block_buf = clCreateBuffer(context,0x1UL | ↩
  CL_MEM_ALLOC_HOST_PTR,16 * sizeof(unsigned char ),0,&status);
    checkError(status,"Failed to create buffer");
    inout_stream_block_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
  inout_stream_block_buf,CL_TRUE,0,0,16 * sizeof(unsigned char ),0,0,0,&↩
  status)));
    checkError(status,"Failed to map buffer");
  }
  if (!input_input_buf) {
    input_input_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR↩
  ,length * sizeof(unsigned char ),0,&status);
    checkError(status,"Failed to create buffer");
    input_input_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
  input_input_buf,CL_TRUE,CL_MAP_READ,0,length * sizeof(unsigned char )↩
  ,0,0,0,&status)));
    checkError(status,"Failed to map buffer");
    aocl_input_old_size = length;
  }
  else {
    if (aocl_input_old_size < length) {
      clEnqueueUnmapMemObject(queue,input_input_buf,input_input_ptr,0,0,0)↩
  ;
      clReleaseMemObject(input_input_buf);
      input_input_buf = clCreateBuffer(context,0x4UL | ↩
  CL_MEM_ALLOC_HOST_PTR,length * sizeof(unsigned char ),0,&status);
      checkError(status,"Failed to create buffer");
      input_input_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
  input_input_buf,CL_TRUE,CL_MAP_READ,0,length * sizeof(unsigned char )↩
  ,0,0,0,&status)));
      checkError(status,"Failed to map buffer");
      aocl_input_old_size = length;
    }
  }
  if (!output_output_buf) {
      output_output_buf = clCreateBuffer(context,0x2UL | ↩
  CL_MEM_ALLOC_HOST_PTR,length * sizeof(unsigned char ),0,&status);
    checkError(status,"Failed to create buffer");
    output_output_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
  output_output_buf,CL_TRUE,CL_MAP_WRITE,0,length * sizeof(unsigned char ↩
  ),0,0,0,&status)));
    checkError(status,"Failed to map buffer");
    aocl_output_old_size = length;
  }
```

```
117   else {
        if (aocl_output_old_size < length) {
119       clEnqueueUnmapMemObject(queue,output_output_buf,output_output_ptr↩
      ,0,0,0);
          clReleaseMemObject(output_output_buf);
121       output_output_buf = clCreateBuffer(context,0x2UL | ↩
      CL_MEM_ALLOC_HOST_PTR,length * sizeof(unsigned char ),0,&status);
          checkError(status,"Failed to create buffer");
123       output_output_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
      output_output_buf,CL_TRUE,CL_MAP_WRITE,0,length * sizeof(unsigned char ↩
      ),0,0,0,&status)));
          checkError(status,"Failed to map buffer");
125       aocl_output_old_size = length;
        }
127   }
      memcpy(input_RK_ptr,RK,68 * sizeof(uint32_t ));
129   memcpy(inout_nonce_counter_ptr,nonce_counter,16 * sizeof(unsigned char )↩
        );
      memcpy(inout_stream_block_ptr,stream_block,16 * sizeof(unsigned char ));
131   memcpy(input_input_ptr,input,length * sizeof(unsigned char ));
      status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&input_RK_buf);
133   checkError(status,"Failed to set argument");
      status = clSetKernelArg(kernel,argi++,sizeof(uint64_t ),&length);
135   checkError(status,"Failed to set argument");
      status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&↩
        inout_nonce_counter_buf);
137   checkError(status,"Failed to set argument");
      status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&↩
        inout_stream_block_buf);
139   checkError(status,"Failed to set argument");
      status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&input_input_buf↩
        );
141   checkError(status,"Failed to set argument");
      status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&↩
        output_output_buf);
143   checkError(status,"Failed to set argument");
      status = clEnqueueTask(queue,kernel,0,0,&kernel_event);
145   checkError(status,"Failed to launch kernel");
      clWaitForEvents(1,&kernel_event);
147   memcpy(nonce_counter,inout_nonce_counter_ptr,16 * sizeof(unsigned char )↩
        );
      memcpy(stream_block,inout_stream_block_ptr,16 * sizeof(unsigned char ));
149   memcpy(output,output_output_ptr,length * sizeof(unsigned char ));
      cl_ulong time_start, time_end;
151   clReleaseEvent(kernel_event);
      return 0;
153 }

155 int main (int argc, char *argv[])
    {
157   if (!init_opencl("aocl_kernel","aocl_generated_kernel")) {
        printf("WARNING: COULDN'T INITIALIZE OPENCL\n");
159     return -1;
      }
161   int len;
      if(argc > 1)
163     len= atoi(argv[1]);
      else
165     len =64000;
```

```
167     unsigned char *max_message = (unsigned char*)malloc(len*sizeof(↩
        unsigned char));
    unsigned char *output_message = (unsigned char*)malloc(len*sizeof(↩
        unsigned char));
169   memset( max_message, 'a', len );
    memset( output_message, 0, len );
171
    mbedtls_aes_context ctx;
173   mbedtls_aes_init( &ctx );
    unsigned char key[32];
175
    unsigned char nonce_counter[16];
177   unsigned char stream_block[16];
    memset( key, 0, 32 );
179   memcpy( key, aes_test_ctr_key[0], 16 );
    memset( nonce_counter, 0, 16 );
181
183   uint64_t offset = 0;
    mbedtls_aes_setkey_enc( &ctx, key, 128 );
185   double timecounter;
    timer_clear(0); timer_start(0);
187   for(int i = 0; i < 100; i++)
    {
189     mbedtls_aes_crypt_ctr_nr10( ctx.rk, (uint64_t)len, nonce_counter, ↩
        stream_block, max_message, output_message );
    }
191   timer_stop(0);
    timecounter = timer_read(0);
193   printf("\tresult: %X %X %X\n", output_message[0], output_message[16000],↩
        output_message[len-1]);
    printf("\tTime total [ms] \t= \t %f \n",timecounter*10);
195   printf("\tTime kernel [ms] \t= \t %f \n",total_time_kernel/1000000.0);
    cleanup();
197   return 0;
}
199
bool init_opencl(const char *file_name,const char *kernel_name)
201 {
    cl_int status;
203   cl_device_id *curr_devices;
    cl_uint num_devices;
205   cl_device_type cl_device_type_all = 0xffffffffUL;
    cl_command_queue_properties cl_queue_profiling_enable = 0x2;
207   std::string binary_file;
    if (!setCwdToExeDir()) {
209     return false;
    }
211   platform = findPlatform("Altera");
    if (platform == 0) {
213     printf("ERROR: Unable to find Altera OpenCL platform.\n");
        return false;
215   }
    curr_devices = getDevices(platform,cl_device_type_all,&num_devices);
217   device = curr_devices[0];
    context = clCreateContext(0,1,&device,&oclContextCallback,0,&status);
219   checkError(status,"Failed to create context");
    queue = clCreateCommandQueue(context,device,cl_queue_profiling_enable,&↩
```

```
    status);
  checkError(status,"Failed to create command queue");
  binary_file = getBoardBinaryFile(file_name,device);
  program = createProgramFromBinary(context,(binary_file .  c_str ()),&
    device,1);
  status = clBuildProgram(program,0,0,"",0,0);
  checkError(status,"Failed to build program");
  kernel = clCreateKernel(program,kernel_name,&status);
  checkError(status,"Failed to create kerne");
  return true;
}

void cleanup()
{
  if (input_RK_buf) {
    clEnqueueUnmapMemObject(queue,input_RK_buf,input_RK_ptr,0,0,0);
    clReleaseMemObject(input_RK_buf);
  }
  if (inout_nonce_counter_buf) {
    clEnqueueUnmapMemObject(queue,inout_nonce_counter_buf,
    inout_nonce_counter_ptr,0,0,0);
    clReleaseMemObject(inout_nonce_counter_buf);
  }
  if (inout_stream_block_buf) {
    clEnqueueUnmapMemObject(queue,inout_stream_block_buf,
    inout_stream_block_ptr,0,0,0);
    clReleaseMemObject(inout_stream_block_buf);
  }
  if (input_input_buf) {
      clEnqueueUnmapMemObject(queue,input_input_buf,input_input_ptr,0,0,0)
    ;
    clReleaseMemObject(input_input_buf);
  }
  if (output_output_buf) {
    clEnqueueUnmapMemObject(queue,output_output_buf,output_output_ptr
    ,0,0,0);
    clReleaseMemObject(output_output_buf);
  }
  if (kernel) {
    clReleaseKernel(kernel);
  }
  if (program) {
    clReleaseProgram(program);
  }
  if (queue) {
    clReleaseCommandQueue(queue);
  }
  if (context) {
    clReleaseContext(context);
  }
  return;
}
```

Listing B.2: AES SOCAO output file

```

typedef long      intmax_t;
typedef ulong     uintmax_t;
typedef char    int_least8_t;
```

```
typedef uchar   uint_least8_t;
typedef short     int_least16_t;
typedef ushort    uint_least16_t;
typedef int       int_least32_t;
typedef uint      uint_least32_t;
typedef long      int_least64_t;
typedef ulong     uint_least64_t;
typedef char    int_fast8_t;
typedef uchar   uint_fast8_t;
typedef short     int_fast16_t;
typedef ushort    uint_fast16_t;
typedef int       int_fast32_t;
typedef uint      uint_fast32_t;
typedef long      int_fast64_t;
typedef ulong     uint_fast64_t;

typedef char    int8_t;
typedef uchar   uint8_t;
typedef short     int16_t;
typedef ushort    uint16_t;
typedef int       int32_t;
typedef uint      uint32_t;
typedef long      int64_t;
typedef ulong     uint64_t;


__constant const uint32_t FT3[256] = { ... };
__constant const uint32_t FT2[256] = { ... };
__constant const uint32_t FT1[256] = { ... };
__constant const uint32_t FT0[256] = { ... };
__constant const uint32_t RT3[256] = { ...};
__constant const uint32_t RT2[256] = { ... };
__constant const uint32_t RT1[256] = { ... };
__constant const uint32_t RT0[256] = { ... };
__constant const unsigned char FSb[256] = { ... };
__constant const unsigned char RSb[256] = { ... };

__kernel void aocl_generated_kernel(uint32_t __global * __restrict__ ↩
    RK_global,const uint64_t length,unsigned char __global * __restrict__ ↩
    nonce_counter_global,unsigned char __global * __restrict__ ↩
    stream_block_global,unsigned char __global * __restrict__ input,↩
    unsigned char __global * __restrict__ output)
{
  int rose_it;
  __local unsigned char stream_block[16];
  for (rose_it = 0; rose_it < 16UL; ++rose_it) {
    stream_block[rose_it] = stream_block_global[rose_it];
  }
  __local unsigned char nonce_counter[16];
  for (rose_it = 0; rose_it < 16UL; ++rose_it) {
    nonce_counter[rose_it] = nonce_counter_global[rose_it];
  }
  __local uint32_t RK[68];
  for (rose_it = 0; rose_it < 68UL; ++rose_it) {
    RK[rose_it] = RK_global[rose_it];
  }
  uint32_t int_counter = ((uint32_t )nonce_counter[12]) << 24 | ((uint32_t↩
    )nonce_counter[13]) << 16 | ((uint32_t )nonce_counter[14]) << 8 | ((↩
    uint32_t )nonce_counter[15]) << 0;
```

```
58  uint64_t k;
    for (k = 0; k < length; k += 16) {
60    uint32_t X0;
      uint32_t X1;
62    uint32_t X2;
      uint32_t X3;
64    uint32_t Y0;
      uint32_t Y1;
66    uint32_t Y2;
      uint32_t Y3;
68    {
        X0 = ((uint32_t )nonce_counter[0]) | ((uint32_t )nonce_counter[1]) ↩
      << 8 | ((uint32_t )nonce_counter[2]) << 16 | ((uint32_t )nonce_counter↩
      [3]) << 24;
70    }
      ;
72    X0 ^= RK[0];
      {
74      X1 = ((uint32_t )nonce_counter[4]) | ((uint32_t )nonce_counter[5]) ↩
      << 8 | ((uint32_t )nonce_counter[6]) << 16 | ((uint32_t )nonce_counter↩
      [7]) << 24;
        }
76    ;
      X1 ^= RK[1];
78    {
        X2 = ((uint32_t )nonce_counter[8]) | ((uint32_t )nonce_counter[9]) ↩
      << 8 | ((uint32_t )nonce_counter[10]) << 16 | ((uint32_t )nonce_counter↩
      [11]) << 24;
80    }
      ;
82    X2 ^= RK[2];
      X3 = int_counter >> 24 & 0xFF | int_counter << 8 & 0xff0000 | ↩
      int_counter >> 8 & 0xff00 | int_counter << 24 & 0xff000000;
84    X3 ^= RK[3];
      int i;
86    #pragma unroll
      for (i = 0; i < 4; i++) {
88      int idx = i * 8;
        Y0 = RK[idx + 4] ^ FT0[X0 & 0xFF] ^ FT1[X1 >> 8 & 0xFF] ^ FT2[X2 >> ↩
      16 & 0xFF] ^ FT3[X3 >> 24 & 0xFF];
90      Y1 = RK[idx + 5] ^ FT0[X1 & 0xFF] ^ FT1[X2 >> 8 & 0xFF] ^ FT2[X3 >> ↩
      16 & 0xFF] ^ FT3[X0 >> 24 & 0xFF];
        Y2 = RK[idx + 6] ^ FT0[X2 & 0xFF] ^ FT1[X3 >> 8 & 0xFF] ^ FT2[X0 >> ↩
      16 & 0xFF] ^ FT3[X1 >> 24 & 0xFF];
92      Y3 = RK[idx + 7] ^ FT0[X3 & 0xFF] ^ FT1[X0 >> 8 & 0xFF] ^ FT2[X1 >> ↩
      16 & 0xFF] ^ FT3[X2 >> 24 & 0xFF];
        X0 = RK[idx + 8] ^ FT0[Y0 & 0xFF] ^ FT1[Y1 >> 8 & 0xFF] ^ FT2[Y2 >> ↩
      16 & 0xFF] ^ FT3[Y3 >> 24 & 0xFF];
94      X1 = RK[idx + 9] ^ FT0[Y1 & 0xFF] ^ FT1[Y2 >> 8 & 0xFF] ^ FT2[Y3 >> ↩
      16 & 0xFF] ^ FT3[Y0 >> 24 & 0xFF];
        X2 = RK[idx + 10] ^ FT0[Y2 & 0xFF] ^ FT1[Y3 >> 8 & 0xFF] ^ FT2[Y0 >>↩
       16 & 0xFF] ^ FT3[Y1 >> 24 & 0xFF];
96      X3 = RK[idx + 11] ^ FT0[Y3 & 0xFF] ^ FT1[Y0 >> 8 & 0xFF] ^ FT2[Y1 >>↩
       16 & 0xFF] ^ FT3[Y2 >> 24 & 0xFF];
      }
98    int idx = 36;
      Y0 = RK[idx] ^ FT0[X0 & 0xFF] ^ FT1[X1 >> 8 & 0xFF] ^ FT2[X2 >> 16 & 0↩
      xFF] ^ FT3[X3 >> 24 & 0xFF];
100   Y1 = RK[idx + 1] ^ FT0[X1 & 0xFF] ^ FT1[X2 >> 8 & 0xFF] ^ FT2[X3 >> 16↩
```

```
    & 0xFF] ^ FT3[X0 >> 24 & 0xFF];
    Y2 = RK[idx + 2] ^ FT0[X2 & 0xFF] ^ FT1[X3 >> 8 & 0xFF] ^ FT2[X0 >> 16↩
    & 0xFF] ^ FT3[X1 >> 24 & 0xFF];
102 Y3 = RK[idx + 3] ^ FT0[X3 & 0xFF] ^ FT1[X0 >> 8 & 0xFF] ^ FT2[X1 >> 16↩
    & 0xFF] ^ FT3[X2 >> 24 & 0xFF];
    X0 = RK[idx + 4] ^ ((uint32_t )FSb[Y0 & 0xFF]) ^ ((uint32_t )FSb[Y1 >>↩
    8 & 0xFF]) << 8 ^ ((uint32_t )FSb[Y2 >> 16 & 0xFF]) << 16 ^ ((uint32_t↩
    )FSb[Y3 >> 24 & 0xFF]) << 24;
104 X1 = RK[idx + 5] ^ ((uint32_t )FSb[Y1 & 0xFF]) ^ ((uint32_t )FSb[Y2 >>↩
    8 & 0xFF]) << 8 ^ ((uint32_t )FSb[Y3 >> 16 & 0xFF]) << 16 ^ ((uint32_t↩
    )FSb[Y0 >> 24 & 0xFF]) << 24;
    X2 = RK[idx + 6] ^ ((uint32_t )FSb[Y2 & 0xFF]) ^ ((uint32_t )FSb[Y3 >>↩
    8 & 0xFF]) << 8 ^ ((uint32_t )FSb[Y0 >> 16 & 0xFF]) << 16 ^ ((uint32_t↩
    )FSb[Y1 >> 24 & 0xFF]) << 24;
106 X3 = RK[idx + 7] ^ ((uint32_t )FSb[Y3 & 0xFF]) ^ ((uint32_t )FSb[Y0 >>↩
    8 & 0xFF]) << 8 ^ ((uint32_t )FSb[Y1 >> 16 & 0xFF]) << 16 ^ ((uint32_t↩
    )FSb[Y2 >> 24 & 0xFF]) << 24;
    {
108   stream_block[0] = ((unsigned char )(X0 & 0xFF));
      stream_block[1] = ((unsigned char )(X0 >> 8 & 0xFF));
110   stream_block[2] = ((unsigned char )(X0 >> 16 & 0xFF));
      stream_block[3] = ((unsigned char )(X0 >> 24 & 0xFF));
112 }
    ;
114 {
      stream_block[4] = ((unsigned char )(X1 & 0xFF));
116   stream_block[5] = ((unsigned char )(X1 >> 8 & 0xFF));
      stream_block[6] = ((unsigned char )(X1 >> 16 & 0xFF));
118   stream_block[7] = ((unsigned char )(X1 >> 24 & 0xFF));
    }
120 ;
    {
122   stream_block[8] = ((unsigned char )(X2 & 0xFF));
      stream_block[9] = ((unsigned char )(X2 >> 8 & 0xFF));
124   stream_block[10] = ((unsigned char )(X2 >> 16 & 0xFF));
      stream_block[11] = ((unsigned char )(X2 >> 24 & 0xFF));
126 }
    ;
128 {
      stream_block[12] = ((unsigned char )(X3 & 0xFF));
130   stream_block[13] = ((unsigned char )(X3 >> 8 & 0xFF));
      stream_block[14] = ((unsigned char )(X3 >> 16 & 0xFF));
132   stream_block[15] = ((unsigned char )(X3 >> 24 & 0xFF));
    }
134     ;
    int_counter++;
136 int c = input[k];
    output[k] = ((unsigned char )(c ^ stream_block[0]));
138 if (k + 1 < length) {
      c = input[k + 1];
140   output[k + 1] = ((unsigned char )(c ^ stream_block[1]));
    }
142 if (k + 2 < length) {
      c = input[k + 2];
144   output[k + 2] = ((unsigned char )(c ^ stream_block[2]));
    }
146 if (k + 3 < length) {
      c = input[k + 3];
148   output[k + 3] = ((unsigned char )(c ^ stream_block[3]));
```

```
    }
    if (k + 4 < length) {
      c = input[k + 4];
      output[k + 4] = ((unsigned char )(c ^ stream_block[4]));
    }
    if (k + 5 < length) {
      c = input[k + 5];
      output[k + 5] = ((unsigned char )(c ^ stream_block[5]));
    }
    if (k + 6 < length) {
      c = input[k + 6];
      output[k + 6] = ((unsigned char )(c ^ stream_block[6]));
    }
    if (k + 7 < length) {
      c = input[k + 7];
      output[k + 7] = ((unsigned char )(c ^ stream_block[7]));
    }
    if (k + 8 < length) {
      c = input[k + 8];
      output[k + 8] = ((unsigned char )(c ^ stream_block[8]));
    }
    if (k + 9 < length) {
      c = input[k + 9];
      output[k + 9] = ((unsigned char )(c ^ stream_block[9]));
    }
    if (k + 10 < length) {
      c = input[k + 10];
      output[k + 10] = ((unsigned char )(c ^ stream_block[10]));
    }
    if (k + 11 < length) {
      c = input[k + 11];
      output[k + 11] = ((unsigned char )(c ^ stream_block[11]));
    }
    if (k + 12 < length) {
      c = input[k + 12];
      output[k + 12] = ((unsigned char )(c ^ stream_block[12]));
    }
    if (k + 13 < length) {
      c = input[k + 13];
      output[k + 13] = ((unsigned char )(c ^ stream_block[13]));
    }
    if (k + 14 < length) {
      c = input[k + 14];
      output[k + 14] = ((unsigned char )(c ^ stream_block[14]));
    }
    if (k + 15 < length) {
      c = input[k + 15];
      output[k + 15] = ((unsigned char )(c ^ stream_block[15]));
    }
  }
  nonce_counter[12] = ((unsigned char )(int_counter >> 24));
  nonce_counter[13] = ((unsigned char )(int_counter >> 16));
  nonce_counter[14] = ((unsigned char )(int_counter >> 8));
  nonce_counter[15] = ((unsigned char )int_counter);
  for (rose_it = 0; rose_it < 16UL; ++rose_it) {
    nonce_counter_global[rose_it] = nonce_counter[rose_it];
  }
  for (rose_it = 0; rose_it < 16UL; ++rose_it) {
    stream_block_global[rose_it] = stream_block[rose_it];
```

```
208     }
    }
```

Listing B.3: AES OpenCL kernel

Appendix C

FFT

The FFT input source code was taken from the WCET research group which provides a large number of benchmarks [3]. Listing C.1 shows the input source file and Listing C.2 the host program generated by the SOCAO compiler. The OpenCL kernel can be found in Listing C.3.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"

using namespace aocl_utils;

#define SIZE 2048
#define PI 3.14159
#define ITER 11
float ar[SIZE];
float ai[SIZE] = {0.,  };

void fft1(int flag,float *pi_frac);
void timer_clear( int n );
void timer_start( int n );
void timer_stop( int n );
float timer_read( int n );


int main()
{
  int  i, n = SIZE, flag, chkerr1, chkerr2, iter = ITER, xp2 = SIZE;
  float* pi_frac;
  double timecounter;

  pi_frac = (float*)malloc(iter*sizeof(float));
  for(i = 0; i < iter; i++)
  {
    xp2 /= 2;
    pi_frac[i] = PI/xp2;
  }

  for(i = 0; i < n; i++)
    ar[i] = cos(2*M_PI*i/n);
```

```
37
   timer_clear(0);
39 timer_start( 0 );
   for(i = 0; i < 100; i++)
41 {
     /* forward fft */
43   flag = 0;
     chkerr1 = fft1(flag, pi_frac);
45   /* inverse fft */
     flag = 1;
47   chkerr2 = fft1(flag, pi_frac);
   }
49
   timer_stop( 0 );
51 timecounter = timer_read( 0 );

53 printf("Results: \n");
   printf("\tSIZE = \t\t%d\n",2048);
55 for (i = 0; i < 8; i++)
     printf("\tar[%d] = \t\t%f\n",i,ar[i]);
57 printf("\tTime [ms] = \t%f\n",timecounter*5);
   delete pi_frac;
59 return 0;
}
61

63 //Altera_OpenCL_Accelerate
   //Altera_OpenCL_size ar 2048
65 //Altera_OpenCL_size ai 2048
   //Altera_OpenCL_size pi_frac ITER
67 //Altera_OpenCL_runtime_const pi_frac
   //Altera_OpenCL_soc
69 void accelerate(int flag, float *pi_frac)
   {
71   int  j, k, it;

73   /*  Main FFT Loops  */
     float sign = ((flag == 1) ? 1.0 : -1.0);
75   int xp2 = SIZE;
     for(it = 0; it < ITER; it++)
77   {
       int xp = xp2;
79     xp2 /= 2;
       float w = pi_frac[it];
81     for(k = 0; k < xp2; k++)
       {
83       float arg = k * w;
         float wr = cos(arg);
85       float wi = sign * sin(arg);
         int i = k - xp;
87       for(j = xp; j <= SIZE; j += xp)
         {
89         int j1 = j + i;
           int j2 = j1 + xp2;
91         float dr1 = ar[j1];
           float dr2 = ar[j2];
93         float di1 = ai[j1];
           float di2 = ai[j2];
95         float tr = dr1 - dr2;
```

```
        float ti = di1 - di2;
        ar[j1] = dr1 + dr2;
        ai[j1] = di1 + di2;
        ar[j2] = tr * wr - ti * wi;
        ai[j2] = ti * wr + tr * wi;
      }
    }
  }
}

void fft1( int flag, float *pi_frac)
{
  int i, j;
  accelerate(flag, pi_frac);

  /*  Digit Reverse Counter  */
  int j1 = SIZE / 2;
  int j2 = SIZE - 1;
  j = 1;
  for(i = 1; i <= j2; i++)
  {
    if(i < j)
    {
      float tr = ar[j-1];
      float ti = ai[j-1];
      ar[j-1] = ar[i-1];
      ai[j-1] = ai[i-1];
      ar[i-1] = tr;
      ai[i-1] = ti;
    }
    int tmp = j1;
    while(tmp < j)
    {
      j -= tmp;
      tmp /= 2;
    }
    j += tmp;
  }

  if(flag)
  {
    int w = SIZE;
    for(i = 0; i < SIZE; i++)
    {
      ar[i] /= w;
      ai[i] /= w;
    }
  }
}
```

Listing C.1: FFT SOCAO input file

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
using namespace aocl_utils;
```

```
#define SIZE 2048
#define PI 3.14159
#define ITER 11
float ar[2048];
float ai[2048] = {(0.)};
void fft1(int flag,float *pi_frac);
void timer_clear(int n);
void timer_start(int n);
void timer_stop(int n);
double timer_read(int n);
void cleanup();
bool init_opencl(const char *file_name,const char *kernel_name);

cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
cl_mem input_pi_frac_buf = 0;
float *input_pi_frac_ptr = 0;
cl_mem inout_ar_buf = 0;
float *inout_ar_ptr = 0;
cl_mem inout_ai_buf = 0;
float *inout_ai_ptr = 0;

int main()
{
  if (!init_opencl("aocl_kernel","aocl_generated_kernel")) {
  printf("WARNING: COULDN'T INITIALIZE OPENCL\n");
  return -1;
  }
  int i;
  int n = 2048;
  int flag;
  int chkerr1;
  int chkerr2;
  int iter;
  int xp2 = 2048;
  float *pi_frac;
  double timecounter = 0;
  iter = 11;
  pi_frac = ((float *)(malloc(iter * sizeof(float ))));
  for (i = 0; i < iter; i++) {
    xp2 /= 2;
    pi_frac[i] = (3.14159 / xp2);
  }
  for (i = 0; i < n; i++)
    ar[i] = (cos(2 * 3.14159265358979323846 * i / n));
  timer_clear(0);
  timer_start(0);
  for(i = 0; i < 100; i++)
  {
    /* forward fft */
    flag = 0;
    fft1(flag,pi_frac);
    /* inverse fft */
    flag = 1;
    fft1(flag,pi_frac);
```

```
  }
  timer_stop(0);
  timecounter = (timer_read(0));
  printf("Results: \n");
  printf("\tSIZE = \t\t%d\n",2048);
  for (i = 0; i < 8; i++)
    printf("\tar[%d] = \t\t%f\n",i,ar[i]);
  printf("\tTime [ms] = \t%f\n",timecounter*5);
  delete pi_frac;
  cleanup();
  return 0;
}

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size ar 2048
//Altera_OpenCL_size ai 2048
//Altera_OpenCL_size pi_frac ITER
//Altera_OpenCL_runtime_const pi_frac
//Altera_OpenCL_soc
void accelerate(int flag,float *pi_frac)
{
  int j;
  int k;
  int it;
  /*  Main FFT Loops  */
  float sign = (flag == 1?1.0 : -1.00000);
  int xp2 = 2048;
  cl_int status;
  cl_event kernel_event;
  unsigned int argi = 0;
  if (!input_pi_frac_buf) {
    input_pi_frac_buf = clCreateBuffer(context,0x4UL | ↩
    CL_MEM_ALLOC_HOST_PTR,ITER * sizeof(float ),0,&status);
    checkError(status,"Failed to create buffer");
    input_pi_frac_ptr = ((float *)(clEnqueueMapBuffer(queue,↩
    input_pi_frac_buf,CL_TRUE,CL_MAP_READ,0,ITER * sizeof(float ),0,0,0,&↩
    status)));
    checkError(status,"Failed to map buffer");
    memcpy(input_pi_frac_ptr,pi_frac,ITER * sizeof(float ));
  }
  if (!inout_ar_buf) {
    inout_ar_buf = clCreateBuffer(context,0x1UL | CL_MEM_ALLOC_HOST_PTR↩
    ,2048 * sizeof(float ),0,&status);
    checkError(status,"Failed to create buffer");
    inout_ar_ptr = ((float *)(clEnqueueMapBuffer(queue,inout_ar_buf,↩
    CL_TRUE,0,0,2048 * sizeof(float ),0,0,0,&status)));
    checkError(status,"Failed to map buffer");
  }
  if (!inout_ai_buf) {
    inout_ai_buf = clCreateBuffer(context,0x1UL | CL_MEM_ALLOC_HOST_PTR↩
    ,2048 * sizeof(float ),0,&status);
    checkError(status,"Failed to create buffer");
    inout_ai_ptr = ((float *)(clEnqueueMapBuffer(queue,inout_ai_buf,↩
    CL_TRUE,0,0,2048 * sizeof(float ),0,0,0,&status)));
    checkError(status,"Failed to map buffer");
  }
  memcpy(inout_ar_ptr,ar,2048 * sizeof(float ));
  memcpy(inout_ai_ptr,ai,2048 * sizeof(float ));
  status = clSetKernelArg(kernel,argi++,sizeof(int ),&flag);
```

```
  checkError(status,"Failed to set argument");
  status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&↩
    input_pi_frac_buf);
  checkError(status,"Failed to set argument");
  status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&inout_ar_buf);
  checkError(status,"Failed to set argument");
  status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&inout_ai_buf);
  checkError(status,"Failed to set argument");
  status = clEnqueueTask(queue,kernel,0,0,&kernel_event);
  checkError(status,"Failed to launch kernel");
  clWaitForEvents(1,&kernel_event);
  memcpy(ar,inout_ar_ptr,2048 * sizeof(float ));
  memcpy(ai,inout_ai_ptr,2048 * sizeof(float ));
  clReleaseEvent(kernel_event);
}

void fft1(int flag,float *pi_frac)
{
  int i;
  int j;
  accelerate(flag,pi_frac);

  /*  Digit Reverse Counter  */
  int j1 = 2048 / 2;
  int j2 = 2048 - 1;
  j = 1;
  for (i = 1; i <= j2; i++)   {
    if (i < j) {
      float tr = ar[j - 1];
      float ti = ai[j - 1];
      ar[j - 1] = ar[i - 1];
      ai[j - 1] = ai[i - 1];
      ar[i - 1] = tr;
      ai[i - 1] = ti;
    }
    int tmp = j1;
    while(tmp < j){
      j -= tmp;
      tmp /= 2;
    }
    j += tmp;
  }

  if (flag) {
    int w = 2048;
    for (i = 0; i < 2048; i++) {
      ar[i] /= w;
      ai[i] /= w;
    }
  }
}

bool init_opencl(const char *file_name,const char *kernel_name)
{
  cl_int status;
  cl_device_id *curr_devices;
  cl_uint num_devices;
  cl_device_type cl_device_type_all = 0xffffffffUL;
  cl_command_queue_properties cl_queue_profiling_enable = 0x2;
```

```
  std::string binary_file;
  if (!setCwdToExeDir()) {
    return false;
  }
  platform = findPlatform("Altera");
  if (platform == 0) {
    printf("ERROR: Unable to find Altera OpenCL platform.\n");
    return false;
  }
  curr_devices = getDevices(platform,cl_device_type_all,&num_devices);
  device = curr_devices[0];
  context = clCreateContext(0,1,&device,&oclContextCallback,0,&status);
  checkError(status,"Failed to create context");
  queue = clCreateCommandQueue(context,device,cl_queue_profiling_enable,&↩
    status);
  checkError(status,"Failed to create command queue");
  binary_file = getBoardBinaryFile(file_name,device);
  program = createProgramFromBinary(context,(binary_file .  c_str ()),&↩
    device,1);
  status = clBuildProgram(program,0,0,"",0,0);
  checkError(status,"Failed to build program");
  kernel = clCreateKernel(program,kernel_name,&status);
  checkError(status,"Failed to create kerne");
  return true;
}

void cleanup()
{
  if (input_pi_frac_buf) {
    clEnqueueUnmapMemObject(queue,input_pi_frac_buf,input_pi_frac_ptr↩
    ,0,0,0);
    clReleaseMemObject(input_pi_frac_buf);
  }
  if (inout_ar_buf) {
    clEnqueueUnmapMemObject(queue,inout_ar_buf,inout_ar_ptr,0,0,0);
    clReleaseMemObject(inout_ar_buf);
  }
  if (inout_ai_buf) {
    clEnqueueUnmapMemObject(queue,inout_ai_buf,inout_ai_ptr,0,0,0);
    clReleaseMemObject(inout_ai_buf);
  }
  if (kernel) {
    clReleaseKernel(kernel);
  }
  if (program) {
    clReleaseProgram(program);
  }
  if (queue) {
    clReleaseCommandQueue(queue);
  }
  if (context) {
    clReleaseContext(context);
  }
  return ;
}
```

Listing C.2: FFT SOCAO output file

```
typedef long      intmax_t;
```

```
typedef ulong     uintmax_t;
typedef char    int_least8_t;
typedef uchar   uint_least8_t;
typedef short     int_least16_t;
typedef ushort    uint_least16_t;
typedef int       int_least32_t;
typedef uint      uint_least32_t;
typedef long      int_least64_t;
typedef ulong     uint_least64_t;
typedef char    int_fast8_t;
typedef uchar   uint_fast8_t;
typedef short     int_fast16_t;
typedef ushort    uint_fast16_t;
typedef int       int_fast32_t;
typedef uint      uint_fast32_t;
typedef long      int_fast64_t;
typedef ulong     uint_fast64_t;

typedef char    int8_t;
typedef uchar   uint8_t;
typedef short     int16_t;
typedef ushort    uint16_t;
typedef int       int32_t;
typedef uint      uint32_t;
typedef long      int64_t;
typedef ulong     uint64_t;

__kernel void aocl_generated_kernel(const int flag,float __constant * ↩
    __restrict__ pi_frac,float __global * __restrict__ ar_global,float ↩
    __global * __restrict__ ai_global)
{
  int rose_it;
  __local float ai[2048];
  for (rose_it = 0; rose_it < 2048UL; ++rose_it) {
    ai[rose_it] = ai_global[rose_it];
  }
  __local float ar[2048];
  for (rose_it = 0; rose_it < 2048UL; ++rose_it) {
    ar[rose_it] = ar_global[rose_it];
  }
  int j;
  int k;
  int it;
  float sign = (flag == 1?1.0 : -1.00000);
  int xp2 = 2048;
  for (it = 0; it < 11; it++)   {
    int xp = xp2;
    xp2 /= 2;
    float w = pi_frac[it];
    for (k = 0; k < xp2; k++) {
      float arg = k * w;
      float wr = (cos(arg));
      float wi = (sign * sin(arg));
      int i = k - xp;
      for (j = xp; j <= 2048; j += xp) {
      int j1 = j + i;
        int j2 = j1 + xp2;
        float dr1 = ar[j1];
        float dr2 = ar[j2];
```

```
          float di1 = ai[j1];
          float di2 = ai[j2];
          float tr = dr1 - dr2;
          float ti = di1 - di2;
          ar[j1] = dr1 + dr2;
          ai[j1] = di1 + di2;
          ar[j2] = tr * wr - ti * wi;
          ai[j2] = ti * wr + tr * wi;
        }
      }
    }
    for (rose_it = 0; rose_it < 2048UL; ++rose_it) {
      ar_global[rose_it] = ar[rose_it];
    }
    for (rose_it = 0; rose_it < 2048UL; ++rose_it) {
      ai_global[rose_it] = ai[rose_it];
    }
}
```

Listing C.3: FFT OpenCL kernel

Appendix D

CRC

The CRC input source code was taken from the WCET research group [3]. Listing D.1 shows the input source file and Listing D.2 the host program generated by the SOCAO compiler. The OpenCL kernel can be found in Listing D.3.

For reasons of clarity, only the relevant parts are printed. The missing functions can be downloaded directly from the homepage of the *WCET* [3].

```
#include <stdlib.h>
#include <stdio.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
using namespace aocl_utils;

#define LOBYTE(x) ((unsigned char)((x) & 0xFF))
#define HIBYTE(x) ((unsigned char)((x) >> 8))

unsigned short icrctb[256];
unsigned char rchr[256];

void    timer_clear( int n );
void    timer_start( int n );
void    timer_stop( int n );
double  timer_read( int n );

unsigned char *lin;

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size lin len
//Altera_OpenCL_size icrctb 256
//Altera_OpenCL_size rchr 256
//Altera_OpenCL_runtime_const icrctb
//Altera_OpenCL_runtime_const rchr
//Altera_OpenCL_soc
unsigned short icrc(unsigned short crc, unsigned int len, short jinit, int↩
    jrev)
{
  unsigned short tmp1, tmp2, cword=crc;
  unsigned int j;

  if (jinit >= 0) cword=((unsigned char) jinit) | (((unsigned char) jinit)↩
    << 8);
```

```
  else if (jrev < 0)
    cword=rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;

  for (j=1;j<len;j++) {
    if (jrev < 0) {
      tmp1 = rchr[lin[j]]^ HIBYTE(cword);
    }
    else {
      tmp1 = lin[j]^ HIBYTE(cword);
    }
    cword = icrctb[tmp1] ^ LOBYTE(cword) << 8;
  }
  if (jrev >= 0) {
    tmp2 = cword;
  }
  else {
    tmp2 = rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;
  }
  return tmp2;
}

int main (int argc, char *argv[])
{
  int n;
  if(argc > 1)
    n= atoi(argv[1]);
  else
    n = 64000;

  unsigned short i1,i2;
  double timecounter;

  lin = (unsigned char*)malloc((n+2)*sizeof(unsigned char));
  memset( lin, 'a', n*sizeof(unsigned char));
  lin[n]=0;
  init();
  timer_clear( 0 );
  timer_start( 0 );
  for(int i = 0; i < 100; i++)
  {
    i1=icrc(0,n,(short)0,1);
    lin[n]=HIBYTE(i1);
    lin[n+1]=LOBYTE(i1);
    i2=icrc(i1,n+2,(short)0,1);
  }

  timer_stop( 0 );
  timecounter = timer_read( 0 );
  printf("Results: \n");
  printf("\ti1 = \t\t%d\n", i1);
  printf("\ti2 = \t\t%d\n", i2);
  printf( "\tTime [ms] = \t%f\n", timecounter*5 );
  return 0;
}
```

Listing D.1: CRC SOCAO input file

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
using namespace aocl_utils;
#define LOBYTE(x) ((unsigned char)((x) & 0xFF))
#define HIBYTE(x) ((unsigned char)((x) >> 8))
unsigned short icrctb[256];
unsigned char rchr[256];
void timer_clear(int n);
void timer_start(int n);
void timer_stop(int n);
double timer_read(int n);
unsigned char *lin;
void cleanup();
bool init_opencl(const char *file_name,const char *kernel_name);
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
cl_mem output_tmp2_buf = 0;
unsigned short *output_tmp2_ptr = 0;
cl_mem input_rchr_buf = 0;
unsigned char *input_rchr_ptr = 0;
cl_mem input_lin_buf = 0;
int aocl_lin_old_size = 0;
unsigned char *input_lin_ptr = 0;
cl_mem input_icrctb_buf = 0;
unsigned short *input_icrctb_ptr = 0;

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size lin len
//Altera_OpenCL_size icrctb 256
//Altera_OpenCL_size rchr 256
//Altera_OpenCL_runtime_const icrctb
//Altera_OpenCL_runtime_const rchr
//Altera_OpenCL_soc
unsigned short icrc(unsigned short crc,unsigned int len,short jinit,int
    jrev)
{
unsigned short tmp1;
unsigned short tmp2;
unsigned short j;
unsigned short cword = crc;
cl_int status;
cl_event kernel_event;
unsigned int argi = 0;
if (!output_tmp2_buf) {
  output_tmp2_buf = clCreateBuffer(context,0x2UL | CL_MEM_ALLOC_HOST_PTR,1
    * sizeof(unsigned short ),0,&status);
  checkError(status,"Failed to create buffer");
  output_tmp2_ptr = ((unsigned short *)(clEnqueueMapBuffer(queue,
    output_tmp2_buf,CL_TRUE,CL_MAP_WRITE,0,1 * sizeof(unsigned short )
    ,0,0,0,&status)));
  checkError(status,"Failed to map buffer");
}
if (!input_rchr_buf) {
  input_rchr_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR
```

```
    ,256 * sizeof(unsigned char ),0,&status);
  checkError(status,"Failed to create buffer");
  input_rchr_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
    input_rchr_buf,CL_TRUE,CL_MAP_READ,0,256 * sizeof(unsigned char )↩
    ,0,0,0,&status)));
  checkError(status,"Failed to map buffer");
  memcpy(input_rchr_ptr,rchr,256 * sizeof(unsigned char ));
}
if (!input_lin_buf) {
  input_lin_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR,len↩
     * sizeof(unsigned char ),0,&status);
  checkError(status,"Failed to create buffer");
  input_lin_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
    input_lin_buf,CL_TRUE,CL_MAP_READ,0,len * sizeof(unsigned char )↩
    ,0,0,0,&status)));
  checkError(status,"Failed to map buffer");
  aocl_lin_old_size = len;
}
else {
  if (aocl_lin_old_size < len) {
    clEnqueueUnmapMemObject(queue,input_lin_buf,input_lin_ptr,0,0,0);
    clReleaseMemObject(input_lin_buf);
    input_lin_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR,↩
    len * sizeof(unsigned char ),0,&status);
    checkError(status,"Failed to create buffer");
    input_lin_ptr = ((unsigned char *)(clEnqueueMapBuffer(queue,↩
    input_lin_buf,CL_TRUE,CL_MAP_READ,0,len * sizeof(unsigned char )↩
    ,0,0,0,&status)));
    checkError(status,"Failed to map buffer");
    aocl_lin_old_size = len;
  }
}
if (!input_icrctb_buf) {
  input_icrctb_buf = clCreateBuffer(context,0x4UL | CL_MEM_ALLOC_HOST_PTR↩
    ,256 * sizeof(unsigned short ),0,&status);
  checkError(status,"Failed to create buffer");
  input_icrctb_ptr = ((unsigned short *)(clEnqueueMapBuffer(queue,↩
    input_icrctb_buf,CL_TRUE,CL_MAP_READ,0,256 * sizeof(unsigned short )↩
    ,0,0,0,&status)));
  checkError(status,"Failed to map buffer");
  memcpy(input_icrctb_ptr,icrctb,256 * sizeof(unsigned short ));
}
memcpy(input_lin_ptr,lin,len * sizeof(unsigned char ));
status = clSetKernelArg(kernel,argi++,sizeof(unsigned short ),&crc);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(unsigned int ),&len);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(short ),&jinit);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(int ),&jrev);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&output_tmp2_buf);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&input_rchr_buf);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&input_lin_buf);
checkError(status,"Failed to set argument");
status = clSetKernelArg(kernel,argi++,sizeof(cl_event ),&input_icrctb_buf)↩
    ;
```

```
104 checkError(status,"Failed to set argument");
    status = clEnqueueTask(queue,kernel,0,0,&kernel_event);
106 checkError(status,"Failed to launch kernel");
    clWaitForEvents(1,&kernel_event);
108 memcpy(&tmp2,output_tmp2_ptr,1 * sizeof(unsigned short ));
    clReleaseEvent(kernel_event);
110 return tmp2;
    }
112
    int main (int argc, char *argv[])
114 {
      if (!init_opencl("aocl_kernel","aocl_generated_kernel")) {
116     printf("WARNING: COULDN'T INITIALIZE OPENCL\n");
        return -1;
118   }
      int n;
120   if(argc > 1)
        n= atoi(argv[1]);
122   else
        n = 64000;
124   unsigned short i1;
      unsigned short i2;
126   double timecounter;
      lin = (unsigned char*)malloc((n+2)*sizeof(unsigned char));
128   memset(lin,'a',n * sizeof(unsigned char ));
      lin[n] = 0;
130   init();
      timer_clear(0);
132   timer_start(0);
      for (int i = 0; i < 100; i++) {
134     i1 = icrc(0,n,((short )0),1);
        lin[n] = ((unsigned char )(i1 >> 8));
136     lin[n + 1] = ((unsigned char )(i1 & 0xFF));
        i2 = icrc(i1,n + 2,((short )0),1);
138   }
      timer_stop(0);
140   timecounter = timer_read(0);
      printf("Results: \n");
142   printf("\ti1 = \t\t%d\n",i1);
      printf("\ti2 = \t\t%d\n",i2);
144   printf("\tTime [ms] = \t%f\n",timecounter*5);
      cleanup();
146 return 0;
    }
148
    bool init_opencl(const char *file_name,const char *kernel_name)
150 {
      cl_int status;
152   cl_device_id *curr_devices;
      cl_uint num_devices;
154   cl_device_type cl_device_type_all = 0xffffffffUL;
      cl_command_queue_properties cl_queue_profiling_enable = 0x2;
156   std::string binary_file;
      if (!setCwdToExeDir()) {
158     return false;
      }
160   platform = findPlatform("Altera");
      if (platform == 0) {
162     printf("ERROR: Unable to find Altera OpenCL platform.\n");
```

```
    return false;
  }
  curr_devices = getDevices(platform,cl_device_type_all,&num_devices);
  device = curr_devices[0];
  context = clCreateContext(0,1,&device,&oclContextCallback,0,&status);
  checkError(status,"Failed to create context");
  queue = clCreateCommandQueue(context,device,cl_queue_profiling_enable,&↩
    status);
  checkError(status,"Failed to create command queue");
  binary_file = getBoardBinaryFile(file_name,device);
  program = createProgramFromBinary(context,(binary_file .  c_str ()),&↩
    device,1);
  status = clBuildProgram(program,0,0,"",0,0);
  checkError(status,"Failed to build program");
  kernel = clCreateKernel(program,kernel_name,&status);
  checkError(status,"Failed to create kerne");
  return true;
}

void cleanup()
{
  if (output_tmp2_buf) {
    clEnqueueUnmapMemObject(queue,output_tmp2_buf,output_tmp2_ptr,0,0,0);
    clReleaseMemObject(output_tmp2_buf);
  }
  if (input_rchr_buf) {
    clEnqueueUnmapMemObject(queue,input_rchr_buf,input_rchr_ptr,0,0,0);
    clReleaseMemObject(input_rchr_buf);
  }
  if (input_lin_buf) {
    clEnqueueUnmapMemObject(queue,input_lin_buf,input_lin_ptr,0,0,0);
    clReleaseMemObject(input_lin_buf);
  }
  if (input_icrctb_buf) {
    clEnqueueUnmapMemObject(queue,input_icrctb_buf,input_icrctb_ptr,0,0,0)↩
    ;
    clReleaseMemObject(input_icrctb_buf);
  }
  if (kernel) {
    clReleaseKernel(kernel);
  }
  if (program) {
    clReleaseProgram(program);
  }
  if (queue) {
    clReleaseCommandQueue(queue);
  }
  if (context) {
    clReleaseContext(context);
  }
  return ;
}
```

Listing D.2: CRC SOCAO output file

```
typedef long      intmax_t;
typedef ulong     uintmax_t;
typedef char    int_least8_t;
typedef uchar   uint_least8_t;
```

```
typedef short      int_least16_t;
typedef ushort     uint_least16_t;
typedef int        int_least32_t;
typedef uint       uint_least32_t;
typedef long       int_least64_t;
typedef ulong      uint_least64_t;
typedef char     int_fast8_t;
typedef uchar    uint_fast8_t;
typedef short      int_fast16_t;
typedef ushort     uint_fast16_t;
typedef int        int_fast32_t;
typedef uint       uint_fast32_t;
typedef long       int_fast64_t;
typedef ulong      uint_fast64_t;

typedef char     int8_t;
typedef uchar    uint8_t;
typedef short      int16_t;
typedef ushort     uint16_t;
typedef int        int32_t;
typedef uint       uint32_t;
typedef long       int64_t;
typedef ulong      uint64_t;

__kernel void aocl_generated_kernel(const unsigned short crc,const unsigned int len,const short jinit,const int jrev,unsigned short __global * __restrict__ tmp2_ptr,unsigned char __global * __restrict__ rchr_global,unsigned char __global * __restrict__ lin,unsigned short __global * __restrict__ icrctb_global)
{
  int rose_it;
  __local unsigned short icrctb[256];
  for (rose_it = 0; rose_it < 256UL; ++rose_it) {
    icrctb[rose_it] = icrctb_global[rose_it];
  }
  __local unsigned char rchr[256];
  for (rose_it = 0; rose_it < 256UL; ++rose_it) {
    rchr[rose_it] = rchr_global[rose_it];
  }
  unsigned short tmp1;
  unsigned short tmp2;
  unsigned int j;
  unsigned short cword = crc;
  if (jinit >= 0) {
    cword = (((unsigned char )jinit) | ((unsigned char )jinit) << 8);
  }
  else {
    if (jrev < 0) {
      cword = (rchr[(unsigned char )(cword >> 8)] | rchr[(unsigned char )(cword & 0xFF)] << 8);
    }
  }
  for (j = 1; j < len; j++) {
    if (jrev < 0) {
      tmp1 = (rchr[lin[j]] ^ ((unsigned char )(cword >> 8)));
    }
    else {
      tmp1 = (lin[j] ^ ((unsigned char )(cword >> 8)));
    }
```

```
59      cword = (icrctb[tmp1] ^ ((unsigned char )(cword & 0xFF)) << 8);
    }
61  if (jrev >= 0) {
      tmp2 = cword;
63  }
    else {
65    tmp2 = (rchr[(unsigned char )(cword >> 8)] | rchr[(unsigned char )(↩
      cword & 0xFF)] << 8);
    }
67  tmp2_ptr[0] = tmp2;
}
```

Listing D.3: CRC OpenCL kernel

Bibliography

[1] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)*, pages 1–112, 2004.

[2] Timothy J. Callahan and John Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, FPL '98, pages 248–257, London, UK, UK, 1998. Springer-Verlag.

[3] Mälardalen Real-Time Research Centre. WCET project / Benchmarks [Online]. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[4] ROSE compiler infrastructure. ROSE HTML References [online]. `http://rosecompiler.org/ROSE_HTML_Reference/index.html`. Accessed: Jun 2016.

[5] Altera Corp. White Paper - FPGA Architecture [online]. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf`, 2006.

[6] Altera Corp. White Paper - Implementing FPGA Design with the OpenCL Standard [online]. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf`, Nov 2013.

[7] Altera Corp. Altera SDK for OpenCL Best Practices Guide [online]. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf`, May 2015.

[8] Altera Corp. Cyclone V Device Overview [online]. `https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf`, Dec 2015.

[9] Altera Corp. Altera SDK for OpenCL - Programming Guide [online]. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf`, May 2016.

[10] Secure Hash Standard (SHS). Federal Information Processing Standards Publication FIPS Pub 180-4, March 2012.

[11] Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication FIPS Pub 197, November 2011.

[12] J. A. Fraire, A. Ferreyra, and C. Marques. OpenCL Overview, Implementation, and Performance Comparison. *IEEE Latin America Transactions*, 11(1):274–280, Feb 2013.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[14] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.0, Document Revision 48. `https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`, Oct 2009.

[15] D. Grune, K. van Reeuwijk, H.E. Bal, C.J.H. Jacobs, and K. Langendoen. *Modern Compiler Design*. Springer New York, 2012.

[16] Terasic Inc. DE1-SoC User Manual (rev.F Board). `http://www.terasic.com`, Aug 2015.

[17] ARM Limited. mbed TLS [Online]. `https://tls.mbed.org/`.

[18] T.Æ. Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, 2011.

[19] Dan Quinlan and Chunhua Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[20] R. Wilhelm, H. Seidl, and S. Hack. *Compiler Design: Syntactic and Semantic Analysis*. Springer Berlin Heidelberg, 2013.