SOCAO: Source-to-Source OpenCL Compiler for Intel-Altera FPGAs

Johanna Rohde\textsuperscript{1}, Marcos Martinez-Peiró\textsuperscript{2}, Rafael Gadea-Gironés\textsuperscript{2}

Date: 7.9.2017

\textsuperscript{1}Computer Systems Group, TU Darmstadt
\textsuperscript{2}Department of Electronic Engineering, Universitat Politècnica de València
Overview

• Introduction
• Background
• Design
• Implementation
• Evaluation
• Conclusion
Introduction

Problem: Accelerate a program with an FPGA

- How do I program the FPGA?
- How do I communicate with the FPGA?
- How much time do I need to rewrite the code?
Introduction

- ASIC
- FPGA Programmers
- Parallel Programmers
- Software Programmers

- Low Level Tools
- OpenCL for FPGA
- SOCAO Compiler for C to OpenCL
Background

- OpenCL
  - Open programming standard for heterogeneous parallel systems
  - Calculations are passed to external accelerator
  - Accelerator can be
    - CPU
    - GPU
    - FPGA
    - ...
Background
OpenCL

- Platform
  - Host
    - Manages the system
    - Is connected to one or more *compute devices*
  - Compute Device
    - Executes a kernel
    - Consists of multiple *compute units*
  - Compute Unit
    - Consists of multiple *processing elements*
Background
OpenCL

- Memory Model
  - Global Memory
    - Used to transfer data
    - Accessible by all work groups
    - Normally the slowest memory
  - Constant Memory
    - Used to save constants
  - Local Memory
    - Accessible by all work items of one work group
    - Not accessible by host
  - Private Memory
    - Accessible by one work item
    - Holds intermediate values
Background

OpenCL

- Host Program Flow
Background
OpenCL for FPGAs

- 2 additional forms of parallelism
- Instruction-level parallelism
  - Instructions that are independent of each other can be calculated at the same time

\[
x = (a+b) \times (c+d);
y = x - e;
z = x \ll 4;
\]
**Background**

OpenCL for FPGAs

- **Loop Pipelining**
  - Iterations are overlapped
  - Ideal case: One Iteration per clock cycle
  - Problem when the loop has loop-carried dependencies

```c
for(int a=0; a<4; a++)
{
    x = (a+b)*(c+d);
    y = x << 4;
    z = x - e;
}
```

<table>
<thead>
<tr>
<th></th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration 1</td>
<td>+/+</td>
</tr>
<tr>
<td>iteration 2</td>
<td>+/+</td>
</tr>
<tr>
<td>iteration 3</td>
<td>+/+</td>
</tr>
<tr>
<td>iteration 4</td>
<td>+/+</td>
</tr>
</tbody>
</table>
Background
OpenCL for FPGAs

- Intel’s SDK for OpenCL

**Host Code**

```c
void sum(int *A, int *B,
        int *res, int size)
{
    clEnqueueWriteBuffer(...);
    clEnqueueTask(...);
    clEnqueueReadBuffer(...);
}
```

**OpenCL Accelerator Code**

```c
__kernel void sum( __global int *A,
                  __global int *B,
                  __global int *res,
                  int size)
{
    for(int i=0; i<size; i++)
        res[i] = A[i] + B[i];
}
```

**Diagram**

The diagram illustrates the process of compiling and running OpenCL code on Intel FPGAs. It shows the flow from host code to cross compiler, offline compiler, and then to the .aocx file, which is run on the FPGA. The diagram also highlights the use of Intel's Offline Compiler and the ARM-powered processor.
Design

Input Program

```plaintext
//Altera_OpenCL_Accelerate
//Altera_OpcnCL_size A size
//Altera_OpenCL...
void sum(int *A, int *B, int *res, int size)
{
    for(int i=0; i<size; i++)
        res[i] = A[i] + B[i];
}
```

Host Code

```plaintext
void sum(int *A, int *B, 
    int *res, int size)
{
    clEnqueueWriteBuffer( ... ); 
    clEnqueueTask( ... );
    clEnqueueReadBuffer( ... );
}
```

SOCAO Compiler

OpenCL Accelerator Code

```plaintext
__kernel void 
aocl_generated_kernel(
    __global int *A,
    __global int *B,
    __global int *res,
    int size)
{
    for(int i=0; i < size; i++)
        res[i] = A[i] + B[i];
}
```
Implementation

<table>
<thead>
<tr>
<th>Front-end</th>
<th>Middle-end</th>
<th>Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Semantic Check</td>
<td>Unparse</td>
</tr>
<tr>
<td>Parse</td>
<td>Function Detection</td>
<td>Successful</td>
</tr>
<tr>
<td>Successful</td>
<td>Function Analysis &amp; Transformation</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

- ROSE Framework
  - Open-source compiler framework
  - Provides front-end, back-end and additional functionalities
Implementation

- ROSE Framework
  - Open-source compiler framework
  - Provides front-end, back-end and additional functionalities
Implementation

- The *Function Analysis & Transformation* phase is the most important
- All decisions are made during this phase
- Consists of 10 analysis/transformation steps
Implementation

```c
void vector_process(char *input, char value)
{
    int i;
    for(i = 0; i < 64; i++)
        input[i] += value;
}

void vector_update(char *input, int ilen)
{
    vector_process(input, 'c');
}
```

```c
void vector_update(char *input, int ilen)
{
    {
        char value_1 = 'c';
        int i;
        for(i = 0; i < 64; i++)
            input[i] += value_1;
    }
}
```
Implementation

```
const uint32_t K[] = {0x428A2F98, ...

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size K 64
//Altera_OpenCL_const_vec K
... void update_accelerated( ...

inline
```

```
aocl_kernel.cl

__constant const uint32_t K[] = {0x428A2F98, ...
__kernel void
aocl_generated_kernel( ...
{
    ...
}
```

```
program.cpp

```
#define WIDTH 512
int[5][WIDTH] A;

//Altera_OpenCL_Accelerate
//Altera_OpenCL_size A 2560
void func(...)
{
    int tmp = A[2][3];
}

#define WIDTH 512
__kernel void
aocl_generated_kernel(int
__global __restrict__ *A, ...)
{
    ...,
    int tmp = A[2*WIDTH + 3];
}
Implementation

```c
int *res;

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

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

Output Variables: \{i, res\}

Kernel Parameter: \{a, size, res\}

```c
__kernel void aocl_generated_kernel(
    int __global __restrict__ *a,
    const int size,
    int __global __restrict__ *res
);
```
Implementation

- Determine which memory buffer is used

<table>
<thead>
<tr>
<th>Transfer</th>
<th>Allocation</th>
<th>Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>Normal</td>
<td>No copy</td>
</tr>
<tr>
<td>Shared</td>
<td>No copy</td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td>Normal</td>
<td>No copy</td>
</tr>
<tr>
<td></td>
<td>Local copy</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>No copy</td>
<td>Local copy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation

- Inline Transformation
- Constant Value Transformation
- Constant Folding
- Constant Array Analysis
- 2D to 1D Array Transformation
- Loop Unrolling
- Memory Analysis
- Typedef Analysis
- Parameter Analysis
- In/Out Analysis

Flowchart:
- Shared memory allocation
- Is size constant?
  - Yes: Local internal copy
  - No: Normal memory allocation

- Is runtime constant?
  - Yes: Constant address space
  - No: Global address space
Implementation

- Insert `#pragma unroll` in front of a loop to unroll it
- Only unroll inner most loop
- Exclusion criteria
  - Number of iterations is not static
  - Number of iterations exceeds 16
  - The loop contains an operation that requires a lot of area

```c
#pragma unroll
for (j = 0; j < 8; j++)
    A[j] = state[j];
```
Evaluation

- DE1SoC evaluation board
  - ARM Cortex
    - Dual Core
    - 800 MHz
  - Cyclone V FPGA
Evaluation
Secure Hash Algorithm (SHA-256)

- Contains two nested for loops
- Both loops have loop-carried dependencies
- Hard to parallelize
- Has a lot of instruction parallelism

```c
//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 )
```
Evaluation
Secure Hash Algorithm (SHA-256)

- Inner loop is pipelined perfectly
- Outer loop is not pipelined well
- Maximum speedup-factor: 1.54
- Break even point: 10kBytes
Evaluation

Advanced Encryption Standard (AES-CTR)

- Contains two nested loops
- Inner loop has loop-carried dependencies and is unrolled
- Uses 10 constant arrays
- Loop Unrolling

```c
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 )
```
Evaluation

Advanced Encryption Standard (AES-CTR)

- Remaining loop can be pipelined perfectly
- Maximum speedup-factor: 3.78
- Break even point: 3.3 kBytes

![Graph showing evaluation of AES-CTR]
Conclusion

• C code → OpenCL for FPGAs

• Two test cases
  - SHA-256
    • Speedup of 1.54
  - AES-CTR
    • Speedup 3.78