Using SDAccel for Host and Accelerator Code Optimizations

Presented By

Peter Frey
October 2, 2018
SDAccel Performance Optimization Agenda

> SDAccel Overview
> Host Code Optimization
> Kernel Code Optimization
> Topological Optimization
> Implementation Optimization
> Performance Profiling with SDAccel
> Summary
Architecture of an FPGA Accelerated Application

- **x86 CPU**
  - Host Application
  - Acceleration API
  - Runtime Library
  - Drivers

- **FPGA**
  - Accelerated Functions
  - AXI Interfaces
  - Global Memory
  - DMA Engine

- **PCIe**

- **User Application Code**

- **Acceleration Platform**

© Copyright 2018 Xilinx
Hardware Acceleration

> **When to USE**
  >>> Algorithm allows for parallelization
  >>> Many similar tasks

> **When May Not be beneficial**
  – Small problem size
  – Cost of Host to Device transfers outweighs benefit

> **When NOT beneficial**
  – Little to no parallelism
    • Algorithm is highly sequential over multiple data
    • Tasks are highly dependent

Amdahl’s Law: If the hardware is 50% of the time, you can accelerate the hardware to zero and you only get 2X
Overview of SDAccel
Flow Overview

C/C++ with OpenCL API

x86 Build Steps

Host Application Executable (.exe)

RTL, C/C++ or OpenCL C

Build Target Selection

FPGA Build Steps

FPGA Binary (.xclbin)
The FPGA Kernel Compilation Flow

OpenCL

xocc -c

.xo

C/C++

xocc -c

.xo

RTL

package_xo

.xo

xocc -link

.xclbin

DSA
## SDAccel Execution Modes

<table>
<thead>
<tr>
<th>Software Emulation</th>
<th>Hardware Emulation</th>
<th>Hardware Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host application runs with a C/C++ or OpenCL model of the Kernels</td>
<td>Host application runs with a simulated RTL model of the Kernels</td>
<td>Host application runs with actual FPGA implementation of the Kernels</td>
</tr>
<tr>
<td>Confirm functional correctness of the system</td>
<td>Test the host / kernel integration, get performance estimates</td>
<td>Confirm system runs correctly and with desired performance</td>
</tr>
<tr>
<td>Fastest turnaround time</td>
<td>Best debug capabilities</td>
<td>Accurate performance results</td>
</tr>
</tbody>
</table>
SDAccel Development, Debug & Analysis

➢ Designed to develop and integrate FPGA based acceleration technology into general software solutions

➢ Fully integrated Eclipse based development environment

➢ Automatic hardware execution flows support

➢ Provides software and acceleration debugging capabilities

➢ Enables detailed system performance analysis
Areas for Performance Optimization

> Host program optimizations
  >> Asynchronous programming, SW pipelining
  >> Optimizing transfer sizes

> Kernel Code optimizations
  >> Interface Specification (512-bit, bursting interfaces)
  >> Dataflow
  >> Pipelining
  >> Memory Optimization

> Topological optimizations
  >> Multiple CUs
  >> DDR mapping

> Implementation optimizations
  >> SLR
  >> Other Vivado P&R controls
Host Code Optimization
Overview Host Code Optimization

Aim to reduce CPU idle time

Aim to maximize kernel utilization

Optimize data transfer sizes
OpenCL Command Queue Optimization

commands = clCreateCommandQueue(context, device_id, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

Command Queue Creation
- CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

CommandQueue

- R
- E
- W
- R
- E
- W

Ordered

Time

Out of Order

© Copyright 2018 Xilinx
OpenCL Buffer Allocation and Transfers

Buffers are used to exchange data between the host and the device.

Aim to reuse available buffers instead of constantly allocating and deallocating new ones.
- Reduce the overhead of DDR memory management.

Aim for 1 or 2MBytes transfers.
- Host <-> Device effective bandwidth varies with transfer size.
- Allocate optimally sized buffers.
- Group several small buffers in a single transaction.

READ_WRITE buffer types can create additional dependencies impacting parallel compute unit execution.
- Only use them when necessary.

```c
cl_mem d_p_A = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(int) * number_of_words, NULL, NULL);
```
Task Synchronization

for(int i=0; i < 2; i++) {
    d_p_A[i] = clCreateBuffer(..., CL_MEM_READ_ONLY |..., ...);
    d_p_B[i] = clCreateBuffer(..., CL_MEM_WRITE_ONLY |..., ...);
    clEnqueueMigrateMemObjects(commands, 1, &d_p_A[i], ..., 0, NULL, &writeevent[i]);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_p_A[i]);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_p_B[i]);
    clEnqueueTask(commands, kernel, 1, &writeevent[num], 0);
} 
clFinish(commands);

clEnqueueMigrateMemObjects(commands, 1, &d_p_B[0], ..., 0, NULL, &readevent[0]);
clEnqueueMigrateMemObjects(commands, 1, &d_p_B[1], ..., 0, NULL, &readevent[1]);
clWaitForEvents(1, &readevent[0]);
clWaitForEvents(1, &readevent[1]);

Example:
- One compute unit
- Single Out of Order Command Queue
- Two parallel tasks
Software Pipelining

for(int i=0; i < MAX; i++) {
    d_p_A[i] = clCreateBuffer(..., CL_MEM_READ_ONLY |..., ...);
    d_p_B[i] = clCreateBuffer(..., CL_MEM_WRITE_ONLY |..., ...);
    clEnqueueMigrateMemObjects(commands, 1, &d_p_A[i], ..., 0, NULL, &writeevent[i]);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_p_A[i]);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_p_B[i]);
    clEnqueueTask(commands, kernel, 1, &writeevent[num], &runevent[i]);
    clEnqueueMigrateMemObjects(commands, 1, &d_p_B[i], ..., 1, &runevent[i], 0);
} clFinish(commands);

Example:
- One compute unit
- Single Out of Order Command Queue
- MAX = 4

Schedule all tasks for execute
Use events to synchronize
Wait for all tasks to complete
Kernel Code Optimization
Key Techniques to Develop High Performance C Kernel

1. Improving Computation efficiency : Parallelize
   - Customized data type adjusted to requirement
   - Pipeline and Dataflow
   - Unroll (Not always required)

2. Memory Configuration
   - Memory customization by array partition
   - Reduce memory access by using local caches, shift registers

3. Interface and Datatype Optimization
   - Interface bandwidth consideration
   - Memory Burst Read and Write
Datawidth Optimization: Bit Accurate Datatypes

> Leverage Arbitrary precision datatypes from HLS library
  >> AP_INT, AP_UINT
  >> AP_FIXED, AP_UFIXED

> Using exact bit width helps to reduce the resource and achieve better performance
  >> Practical example: Floating point to Fixed point conversation improve performance
  >> A whitepaper: Deep Learning with INT8 Optimization

```c
ap_uint<5> last_i; // 5 bits unsigned
ap_uint<2> tu_size; // 2 bits unsigned

switch (tu_size) {
  case 0:
    last_i = 0;
    break;
  case 1:
    last_i = 7;
    break;
  case 2:
    last_i = 15;
    break;
  case 3:
    last_i = 31;
    break;
}
```

Example code shows using bit-accurate integer datatype instead of native short, int etc
Improve Compute Efficiency: Dataflow

> Dataflow = Task level parallelism

```c
void top(a,b,c,d) {
    ...  
    func_A(a,b,i1);
    func_B(c,i1,i2);
    func_C(i2,d);
    return d;
}
```

#include <iostream>

int main() {
    int a, b, c, d;
    a = 1;
    b = 2;
    c = 3;
    d = 4;
    top(a, b, c, d);
    std::cout << d << std::endl;
    return 0;
}

#pragma HLS dataflow

```c
func_A(a,b,i1);
func_B(c,i1,i2);
func_C(i2,d);
```

- initiation interval: 3 cycles
- latency: 5 cycles
Improve Compute Efficiency: Pipeline

> Pipeline = Instruction level parallelism

```c
void func(m,n,o) {
    for (int i=2; i>=0; i--) {
        op_read;
        op_compute;
        op_write;
    }
}
```

initiation interval
1 cycle

3 cycles latency
Improve Compute Efficiency: Unrolling Loops

> For smaller body loops with limited number of iterations, unrolling improve performance

```c
... add: for (int i=0; i<=3; i++) {
    b = a[i] + b;
}
...```

Unroll: 1 cycle

> If complete unrolling is not feasible, exploit partial unrolling

> Unrolling loops with large number of iterations and large body significantly increases resource usage and slows down compilation
Key Techniques to Develop High Performance C Kernel

1. Improving Computation efficiency : Parallelize
   - Customized data type adjusted to requirement
   - Pipeline and Dataflow
   - Unroll (Not always required)

2. Memory Configuration
   - Memory customization by array partition
   - Reduce memory access by using local caches, shift registers

3. Interface and Datatype Optimization
   - Interface bandwidth consideration
   - Memory Burst Read and Write
Large Arrays get placed in Memory

> Mapping the array ‘a’ to a single memory will force a sequential implementation of the algorithm

```
for ( int i=0; i<512; i++) {
    b = a[i] + b;
}
```

At most 2 Ports each permitting one read per cycle
Memory Access crucial for Performance

> Understand the Array access pattern
> Use several BRAMS or Registers to implement arrays (parallel access)

Array Partitioning

- block
- cyclic
- complete
Key Techniques to Develop High Performance C Kernel

1. **Improving Computation efficiency : Parallelize**
   - Customized data type adjusted to requirement
   - Pipeline and Dataflow
   - Unroll (Not always required)

2. **Memory Configuration**
   - Memory customization by array partition
   - Reduce memory access by using local caches, shift registers

3. **Interface and Datatype Optimization**
   - Interface bandwidth consideration
   - Memory Burst Read and Write
Interface Bandwidth Optimization – 512 bits

> Kernels use AXI4 MM Master ports to connect with DDR banks over an AXI interconnect

> AXI Interconnect supports up to 512 bit wide transfers

> For maximum throughput, the kernel should use the full 512 bits of the AXI interface

```c
void vadd(
    const ap_uint<512> *in1,
    const ap_uint<512> *in2,
    ap_uint<512>       *out,
    int                size )
{
}
```

Use `ap_uint<512>` types to create 512-bit wide AXI_M ports
Interface Bandwidth Optimization – Number of Ports

> Number of AXI_M ports impacts kernel performance
  >> Maximum theoretical bandwidth per AXI_M port is 512bits @ 300MHz (based on platform clock)

> By default, SDAccel creates a single AXI_M port per kernel
  >> Different I/O processes will have to access the AXI_M port sequentially

```c
void K_VADD( dType *A, dType *B, dType*R) {
  #pragma HLS INTERFACE m_axi port=A offset=slave
  #pragma HLS INTERFACE m_axi port=B offset=slave
  #pragma HLS INTERFACE m_axi port=R offset=slave
}
```

By default, SDAccel maps all pointer arguments to the same AXI_M interface
> **Number of AXI_M ports impacts kernel performance**
  >> Maximum theoretical bandwidth per AXI_M port is 512bits @ 300MHz (based on platform clock)

> **By default, SDAccel creates a single AXI_M port per kernel**
  >> Different I/O processes will have to access the AXI_M port sequentially

> **Adding extra AXI_M ports increases kernel bandwidth**
  >> With at least two ports, a kernel can read inputs and write outputs simultaneously

```c
void K_VADD( dType *A, dType *B, dType*R) {
    #pragma HLS INTERFACE m_axi port=A offset=slave bundle=p0
    #pragma HLS INTERFACE m_axi port=B offset=slave bundle=p1
    #pragma HLS INTERFACE m_axi port=R offset=slave bundle=p2
}
```

Use the “bundle” property on the INTERFACE pragma to create and name AXI_M ports
Interface Bandwidth Optimization – Bursting

> Read/Write accesses to DDR cause have a long latency overhead
> Random sequences of individual accesses are bad for performance
> Bursting is the most efficient way to access DDR as it hides latency
> To ensure bursting behavior, create a dedicated dataflow function in which a pipelined loop reads or writes from an AXI_M port

```cpp
template<typename out_t>
void read_blocks(const out_t *in, hls::stream<out_t> &out, unsigned int blocks) {
    for(unsigned int i = 0; i < blocks*2; i++) {
        #pragma HLS loop_tripcount min=2048 max=2048
        #pragma HLS PIPELINE
        out.write(in[i]);
    }
}
```

Enable burst transfers from global memory

Sequential data access enables streaming data between blocks
Sustaining Interface Throughput in the Kernel

> Top function with Read, Compute, Write Dataflow blocks

```c
#include <cassert>

#pragma HLS DATAFLOW

void top_function(datatype_t *m_in, // Memory data Input
datatype_t *m_out, // Memory data Output
int inp1, // Other Input
int inp2) { // Other Input

    #pragma HLS dataflow
    // Read function contains pipelined for loop
    // to infer burst
    read_function(m_in, inp1);
    execute_function(m_in, inp1, inp2);
    write_function(m_in, m_out);
}
```

Concurrent execution of read, execute, write

initiation interval

3 cycles

read read
exec exec
write write

5 cycles latency
Topological Optimizations
Multiple Compute Units (Kernel Instances)

- By default, SDAccel generates 1 instance of each kernel

- Use multiple instances when the same function is performed on independent blocks of data (data-level parallelism)

- Example: 2D Image Filter
  - Use 1 CU to process Y, U and V color planes sequentially
  - Use 3 CUs to process Y, U and V color planes parallel
  - Use 6 CUs to process two images in parallel

```
# Create 3 CUs for kernel “foo”
xocc -l --nk foo:3 <other options>
```

Use the xocc --nk option during the link phase to specify number of CUs for each kernel.
Kernel Port Connections to DDR Banks

> SDAccel platforms typically contain 4 DDR banks

> By default all kernel AXI_M ports are mapped to the same DDR bank
  >> DDR bandwidth is shared, multiple AXI requests are arbitrated

> Careful mapping of kernel ports to specific DDR banks improves performance
  >> Take advantage of full DDR bandwidth
  >> Simultaneous transfers to each DDR
  >> Physical proximity of kernel and DDR improves Fmax

```
xocc -l --sp kernel_top_1.m_axi_p0.bank0 \  
  --sp kernel_top_1.m_axi_p1.bank1 \  
  --sp kernel_top_1.m_axi_p2.bank2 \  
  <other options>
```

Use the xocc --sp option during the link phase to specify desired mapping
Update OpenCL Buffers properties in the host program
Implementation Optimization
Today’s largest FPGA are stacked silicon devices with several SLRs (super logic regions).

Connections between SLRs incur a greater delay than standard intra-SLR routing.

By default, kernels are placed in the same SLR as the Shell.

Careful placement of kernels in SLRs will improve Fmax:

- Aim to place kernels in the same SLR as the DDR they interface with.
- Aim to minimize SLR congestion and cross-SLR connections.
Understand the target DSA

> Review Documentation
  >> SDAccel Release Notes (UG 1238)
    - DSA Released 2017.4
  >> DSA Specifications

> Shell
  >> Consumes Resources to implement
  >> Removes available resources from the dynamic Region
  >> Understand which SLRs are impacted

SLR resources and DDR assignment can impact performance
Kernel Placement Control

> Specify Kernel Locations
  >> Provided by XOCC command line arguments
  xocc <arguments> --xp param:compiler.userPostSysLinkTcl=<path>/place_krnl.tcl
    - Auto-executes a Vivado Tcl file
  >> Kernel locations specified by Vivado Tcl script
  ```
  place_krnl.tcl
  set_property CONFIG.SLR_ASSIGNMENTS SLR0 [get_bd_cells /krnl_0]
  set_property CONFIG.SLR_ASSIGNMENTS SLR2 [get_bd_cells /krnl_1]
  ```

> Command line option provided in 2018.3
  >> Requires a new DSA revision (5.2)
  >> Also ensures local SLR reset is used
  xocc <arguments> --slr krnl_0:SLR0 --slr krnl_1:SLR2
Performance Profiling with SDAccel
Main Report Files

> SDAccel generates important report files that help to improve performance

> Reports from Hardware Emulation are most useful for performance improvement
  > Guidance Report

> Reports to analyze overall system performance (combining Host and Kernel)
  > Profile Summary Report
  > Timeline Trace
  > Waveform

> Reports to understand/improve Kernel performance
  > HLS report
  > Schedule Viewer
Design Guidance

> Expert system built-in the tool
  » Analysis of build results and emulation runs

> Guidance window with feedback by category
  » Host code
  » Kernels
  » Data transfers (host to DDR, DDR to kernels)

> Explicit and actionable hints
  » How to improve the design
  » Links to detailed explanation and solutions
  » HTML report (available for makefile runs as well)
Profile Summary

> Top Operations
   >> Activity summary

> Kernel & Compute Units
   >> Detailed execution statistics

> Data Transfer
   >> Global Memory access statistics from host and from kernels

> Host Code OpenCL API statistics
Annotated Waveform Viewer

> Shows task-level parallelism in action
> Show how many tasks overlap and for how long
HLS Report

> Static Performance Estimates:
  ▸ Timing
  ▸ Latency
  ▸ Hierarchical contribution

> Utilization Estimates:
  ▸ Summary
  ▸ Detail analysis

### Performance Estimates

- **Timing (ns)**
  - **Summary**
    - Clock | Target | Estimated | Uncertainty
    - `ap_clk` | 5.00 | 3.123 | 0.62

- **Latency (clock cycles)**
  - **Summary**
    - | min | max | min | max | Type
    - 257 | 257 | 257 | 257 | none

### Utilization Estimates

- **Summary**
  - | Name | BRAM_18K | DSP48E | FF | LUT
  - | DSP | - | - | - | -
  - | Expression | - | - | 0 | 53
  - | FIFO | - | - | - | -
  - | Instance | - | - | 0 | 1362
  - | Memory | - | - | - | -
  - | Multiplexer | - | - | - | 1173
  - | Register | - | - | 47 | -
  - | Total | 0 | 0 | 47 | 2588
  - | Available | 4320 | 5520 | 1326720 | 663360
  - | Available SLR | 2160 | 2760 | 663360 | 331680
  - | Utilization (%) | 0 | 0 | -0 | -0
  - | Utilization SLR (%) | 0 | 0 | -0 | -0
HLS Schedule Viewer

- Shows in which cycle operations are scheduled
- Shows operator timing and clock margin
- Shows data dependencies
- Cross-probing from operations to source code
- Supports specific focus on:
  - II Violation
  - Timing Violation
Summary

- First address Guidance Suggestions provided by SDAccel

- Use performance analysis viewers to identify further optimization opportunities

- Consider all areas for Performance Optimization
  - Host program optimizations
  - Kernel Code optimizations
  - Topological optimizations
  - Implementation optimizations
More Details

> UG1207: SDAccel Environment Optimization Guide

> SDAccel Examples:
  >> https://github.com/Xilinx/SDAccel_Examples