



OpenCL Design Flows for Intel and Xilinx FPGAs

Common Optimization Strategies, Design Patterns and
Vendor-specific Differences

Tobias Kenter

Paderborn Center for Parallel Computing &
Department of Computer Science
Paderborn University, Germany



Paderborn
Center for
Parallel
Computing

Part 1

Common Design Patterns
Key Differences

Introduction

- Our mission at PC²

Promote and Establish FPGAs as accelerators in HPC



Paderborn
Center for
Parallel
Computing

- Objectives for applications and libraries

Achieve Throughput Close to Architectural Limits

Use OpenCL as Performance Portable FPGA Design Tool

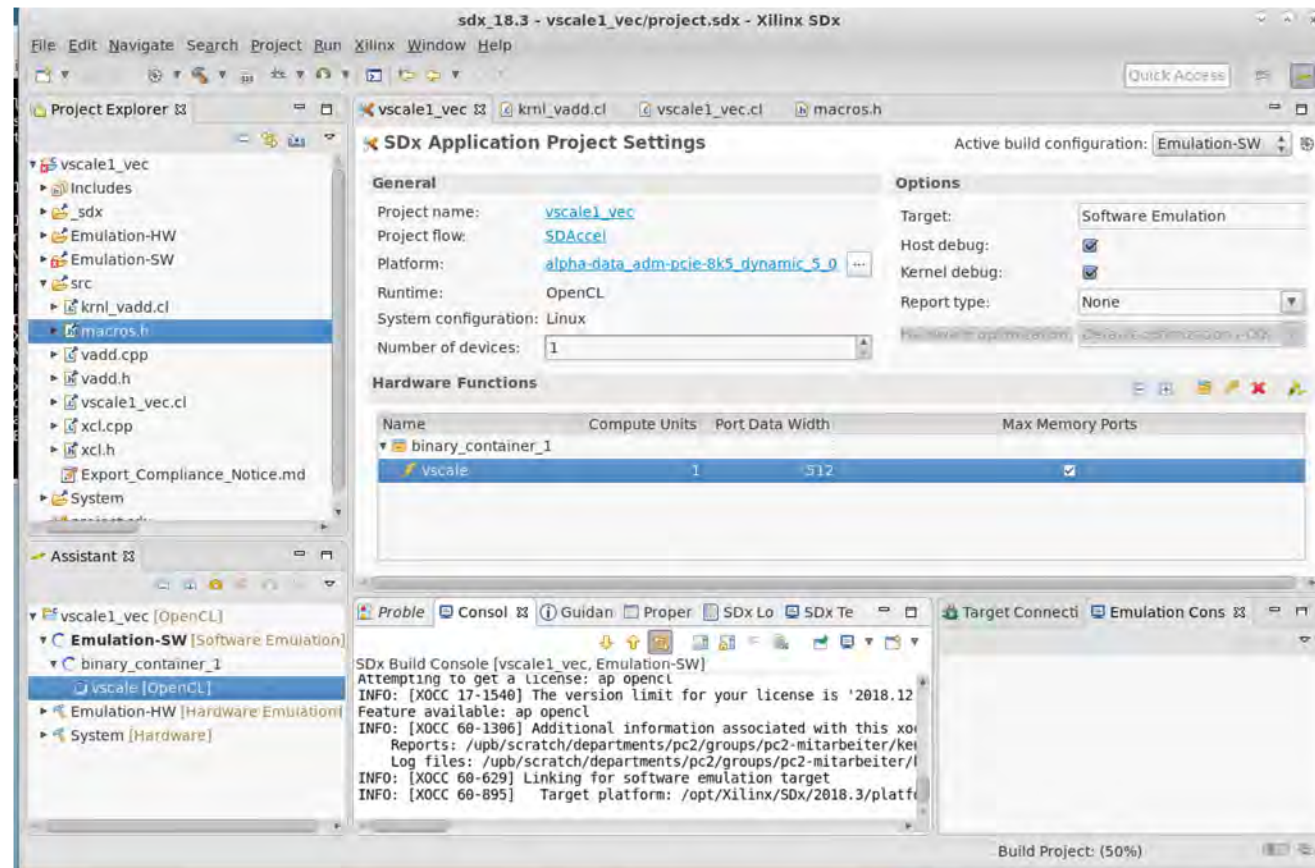
- How far can those coexist?

My Background

- **Research interests / background**
 - application acceleration
 - architecture exploration
 - compilation tools
 - tool user: OpenCL, Maxeler
 - compiler extensions: LLVM, Clang
- **Experience with OpenCL FPGA tool chains since 2016**
 - FDTD stencil computations with Xilinx and Intel
 - DG code with Intel
 - matrix multiplication with Intel and Xilinx
 - CNN, convolutions with Xilinx and Intel
 - FFT with Intel
 - image processing and generalization with Xilinx
 - elliptic curve method with Xilinx
 - external channel communication with Intel

My Biases

- Currently more focus on Intel tools due to our hardware setup
- Xilinx SDAccel has an extensive GUI that I mostly ignore here
 - makefile + command line flow to quickly switch targets



- Overview FPGAs and Goals
- OpenCL Overview

- **Example 1: Vector Scale**
 - compilation
 - reports
 - performance analysis
- **Vector Scale Variations**
 - automatic unrolling
- **Example 2: SAXPY**
 - blockwise design pattern
- Outer Loop Pipelining
- Streaming Kernels

Overview: FPGAs and Goals

- **Field-programmable Gate Array**

- **Gates**

- fundamental building blocks are logic gates

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

Configuration memory:
 2^N bits
 Typical input sizes N:
 5-8

- in all current FPGAs: LUTs (Lookup Tables)
- truth table stored in SRAM

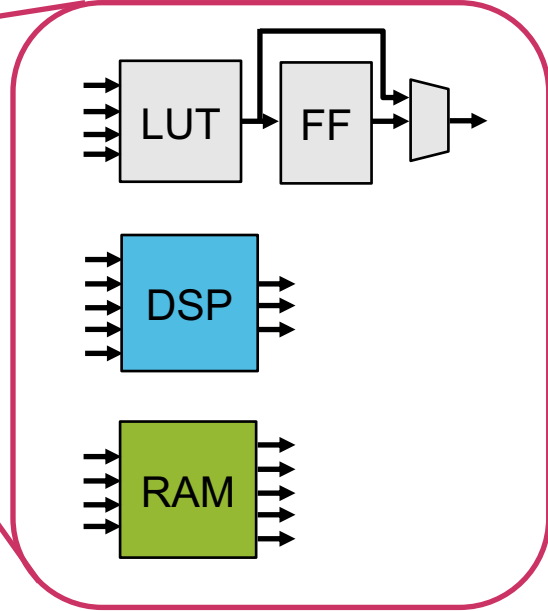
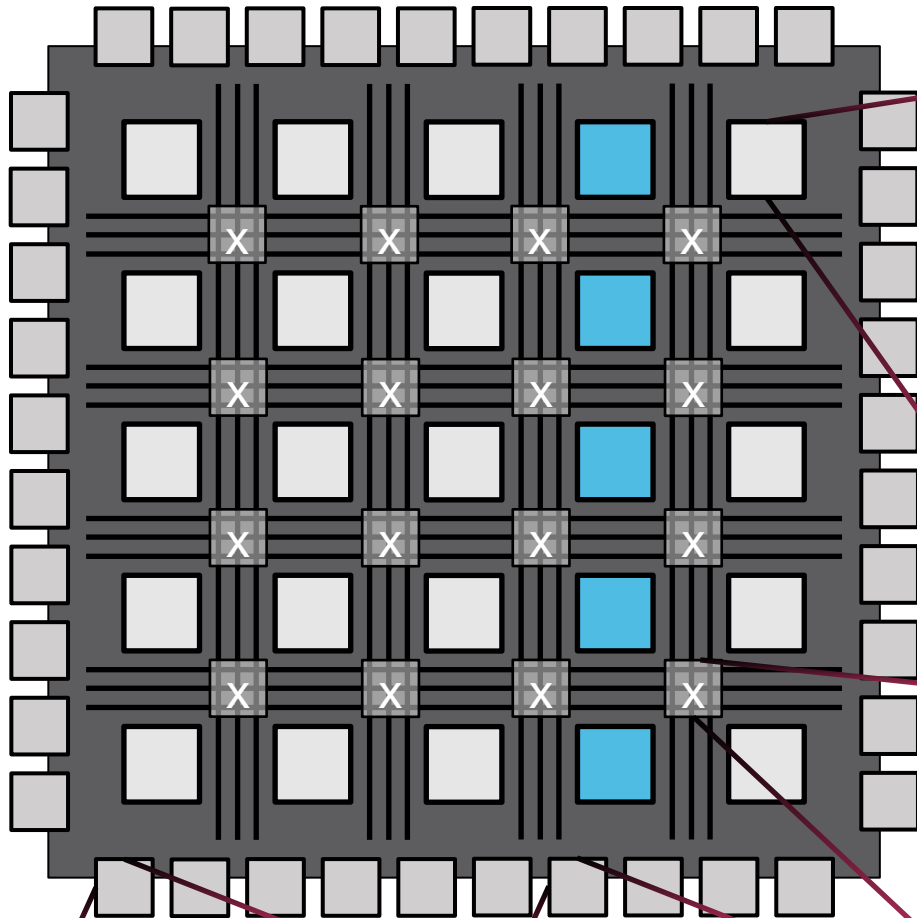
- **Array**

- many gates (LUTs) in a regular 2D structure

- **Field-programmable**

- configuration can be changed “in the field”, many times
- in practice: currently takes up to few 100 ms
- faster alternatives possible

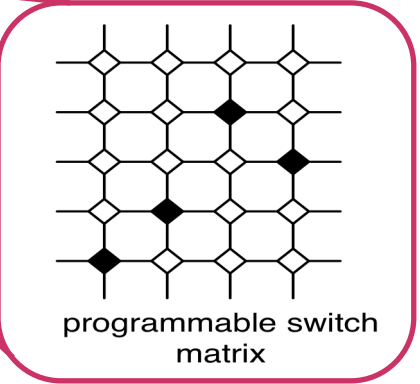
FPGA – Basic Structures



millions

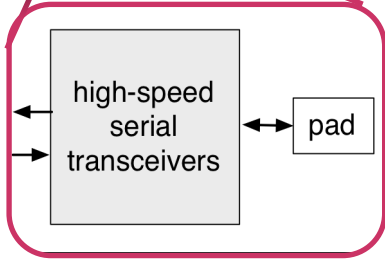
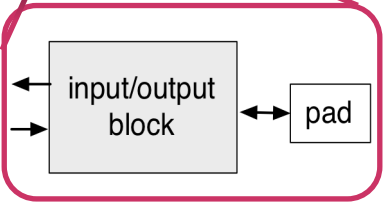
thousands

thousands



configuration bits define functionality

no sequence of instructions



FPGA – Configuration and Application Domains

- configuration
 - all FPGAs components are programmable (logic cell, DSP, IO-block functions, routing, ...)
 - configuration data (bitstream) is stored in SRAM cells
 - bitstream loaded from non-volatile memory at boot time
 - some devices can be re-configured at runtime
- application domains
 - glue logic
 - rapid prototyping, emulation
 - embedded systems
 - configurable system-on-chip
 - ASIC replacement
 - reconfigurable computing
 - computing without CPUs
 - combine processor-like programmability with ASIC-like performance
 - recent hot topic: CNNs with customized precision

FPGA Technology Today

Example: Intel Stratix 10 GX2800 FPGA

- > 900000 configurable logic blocks
 - up to 4 Boolean functions of 8 inputs
- 5760 hardened arithmetic units (DSP)
 - fixed point and IEEE 754 SP floating-point
- 11721 independent SRAM blocks
 - width/depth/ports highly configurable
- integrated DDR4 memory controllers
- up to 96 serial transceivers, up to 28.3 Gbps
- typically about 300-600MHz
- power consumption 50-225W

100 TERRA-OPS

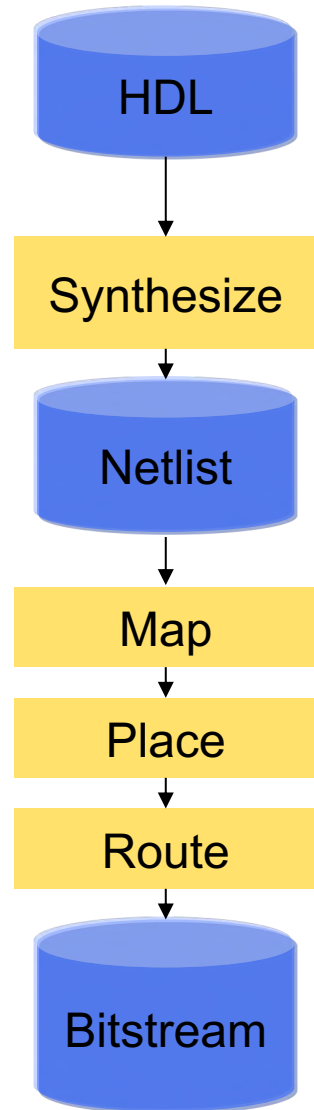
10 single-precision TFLOPS

20 TB/s internal SRAM bandwidth
(full duplex)

300 TB/s communication
bandwidth (full duplex)

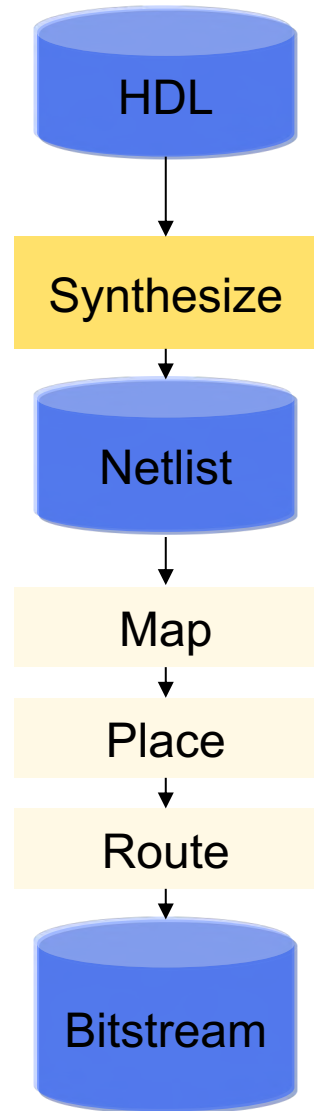
up to 80 GFLOPS/W

Classical FPGA Development

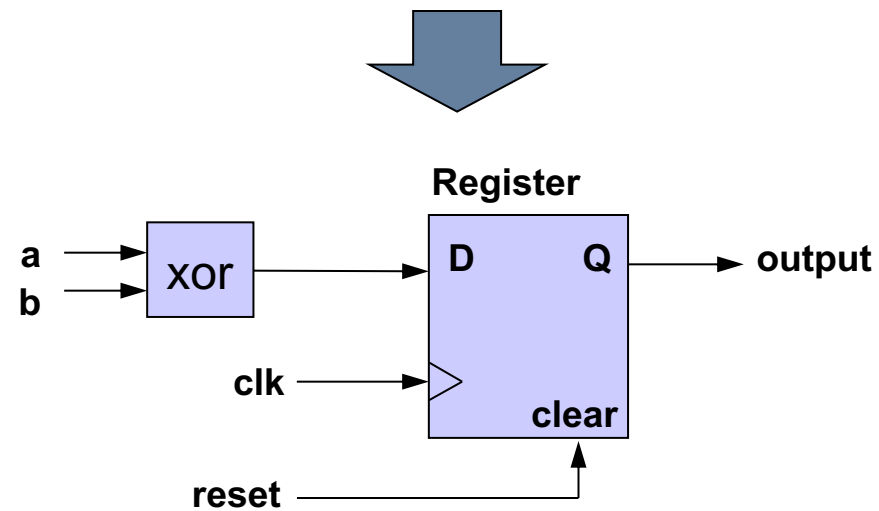


- Hardware design is traditionally done by modeling the system in a hardware description language (e.g. VHDL or Verilog)
- An FPGA synthesis tool (compiler) generates an netlist of basic logic elements,
 - which is then translated (mapped) to components available on the FPGA,
 - which are placed on the chip,
 - and the connecting signals are routed through the interconnection network.
- The resulting configuration data (bitstream) for programing the FPGA is created

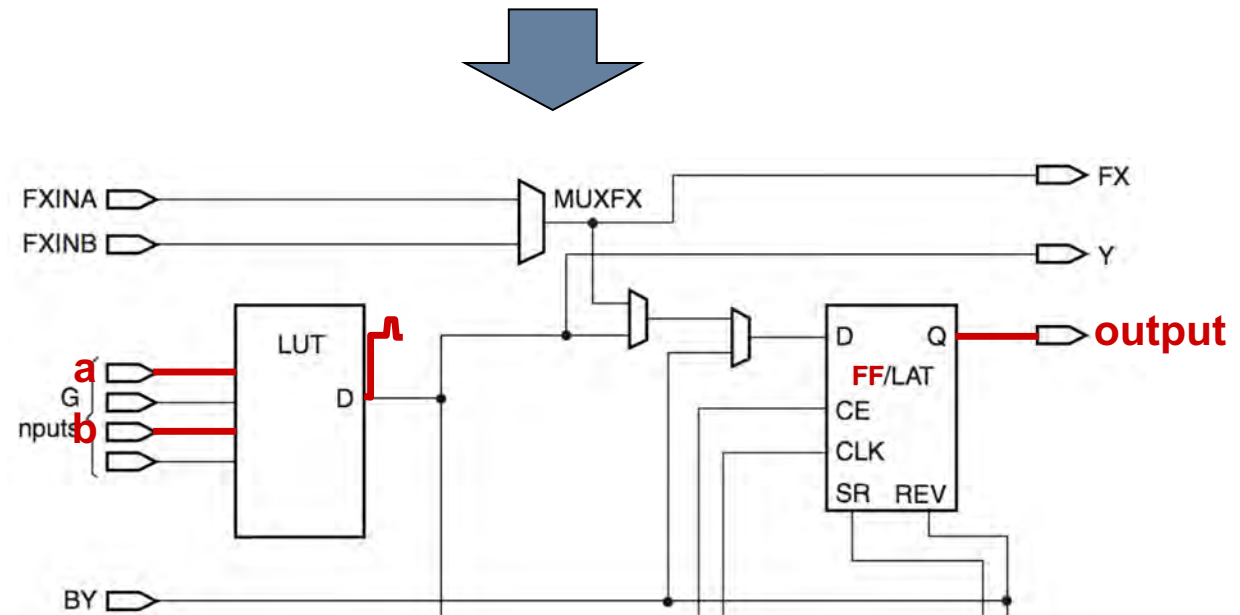
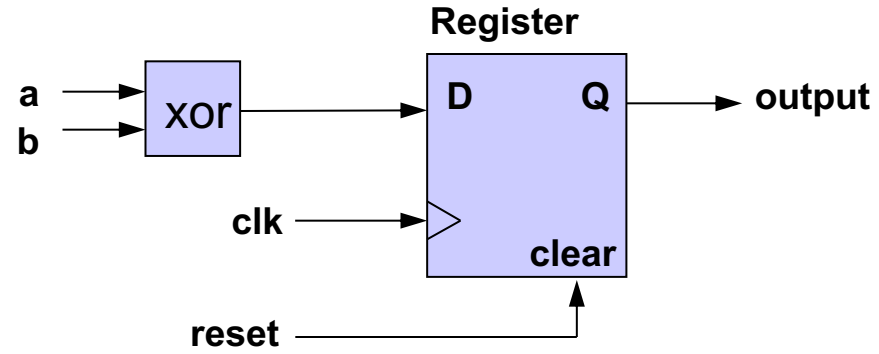
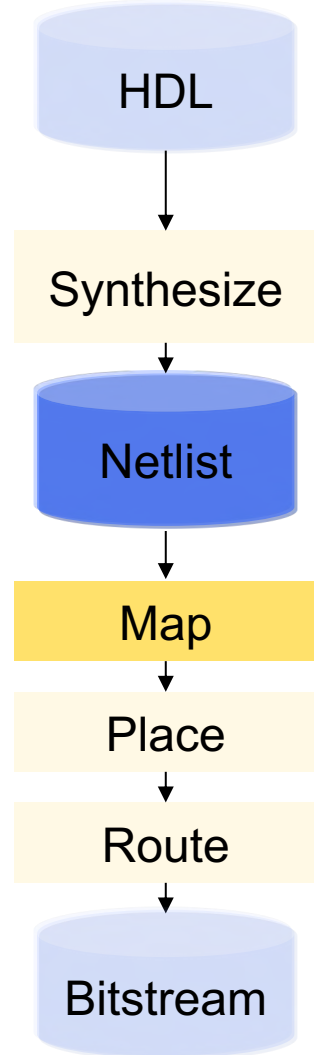
HDL Synthesis

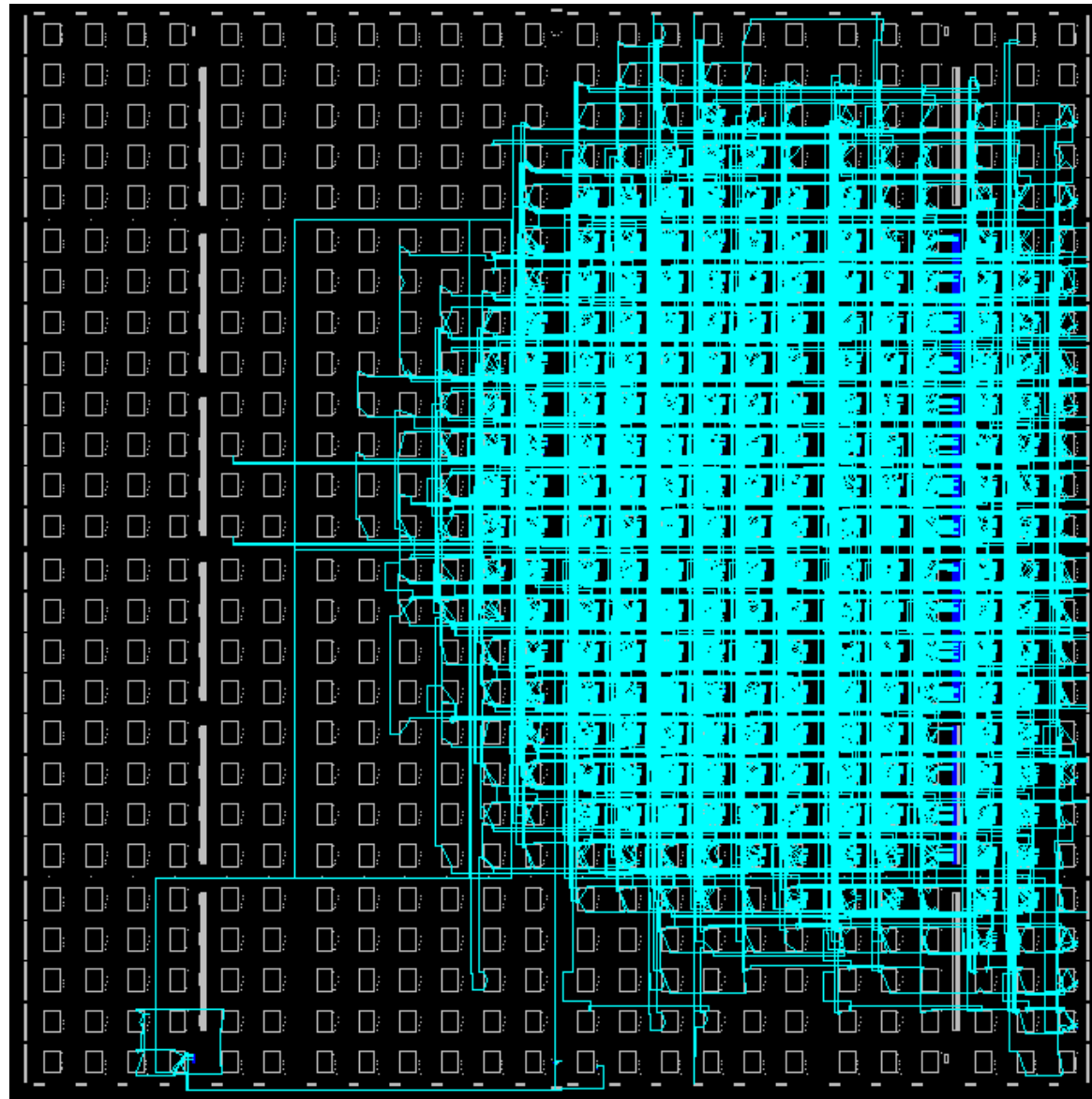
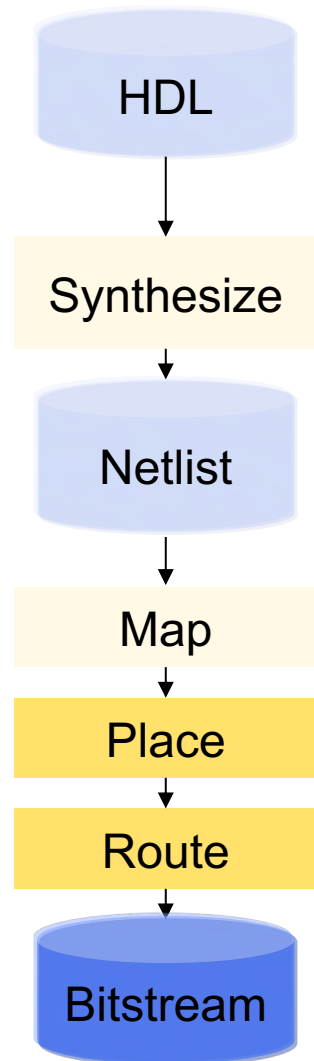


```
process(clk, reset)
begin
  if reset = '1' then
    output <= '0';
  elsif clk'event AND clk = '1' then
    output <= a XOR b;
  end if;
end process;
```

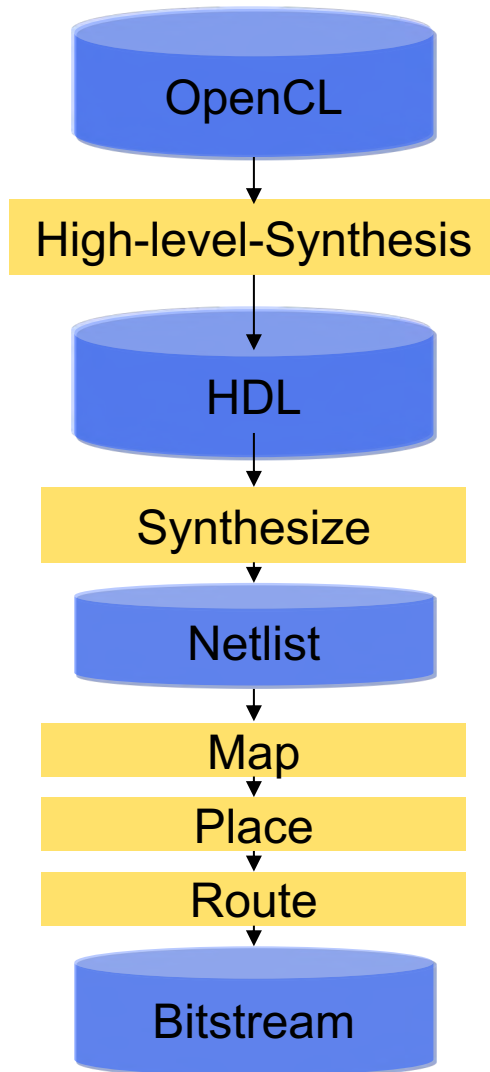


Technology Mapping





Modern FPGA Development



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

Execution on CPU vs on FPGA

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

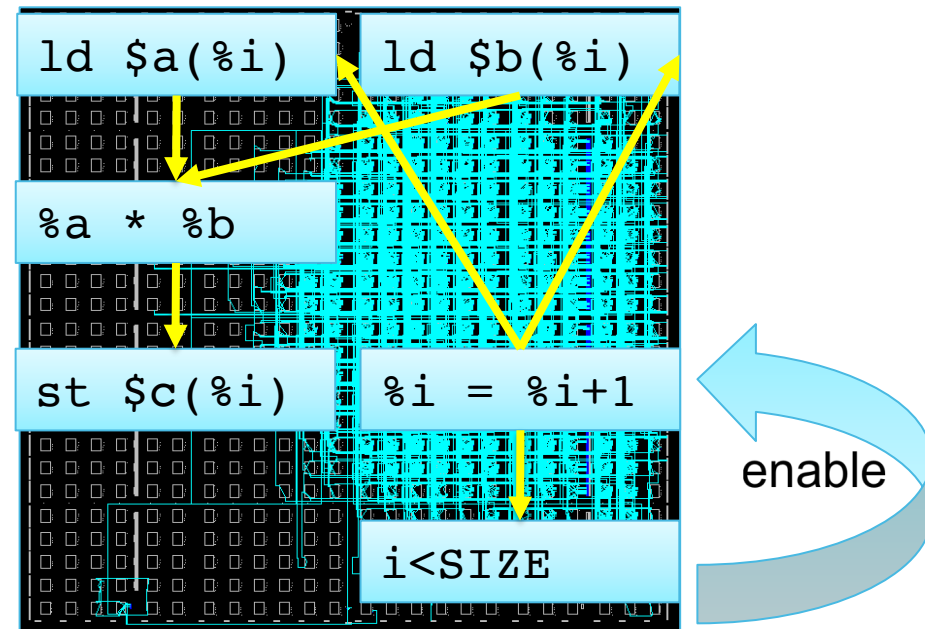
Execution on CPU

- Series of instructions

```
loop:  
    ld %a $a(%i)  
    ld %b $b(%i)  
    %c = %a * %b  
    st $c(%i) %c  
    %i = %i + 1  
    branch i<SIZE: loop
```

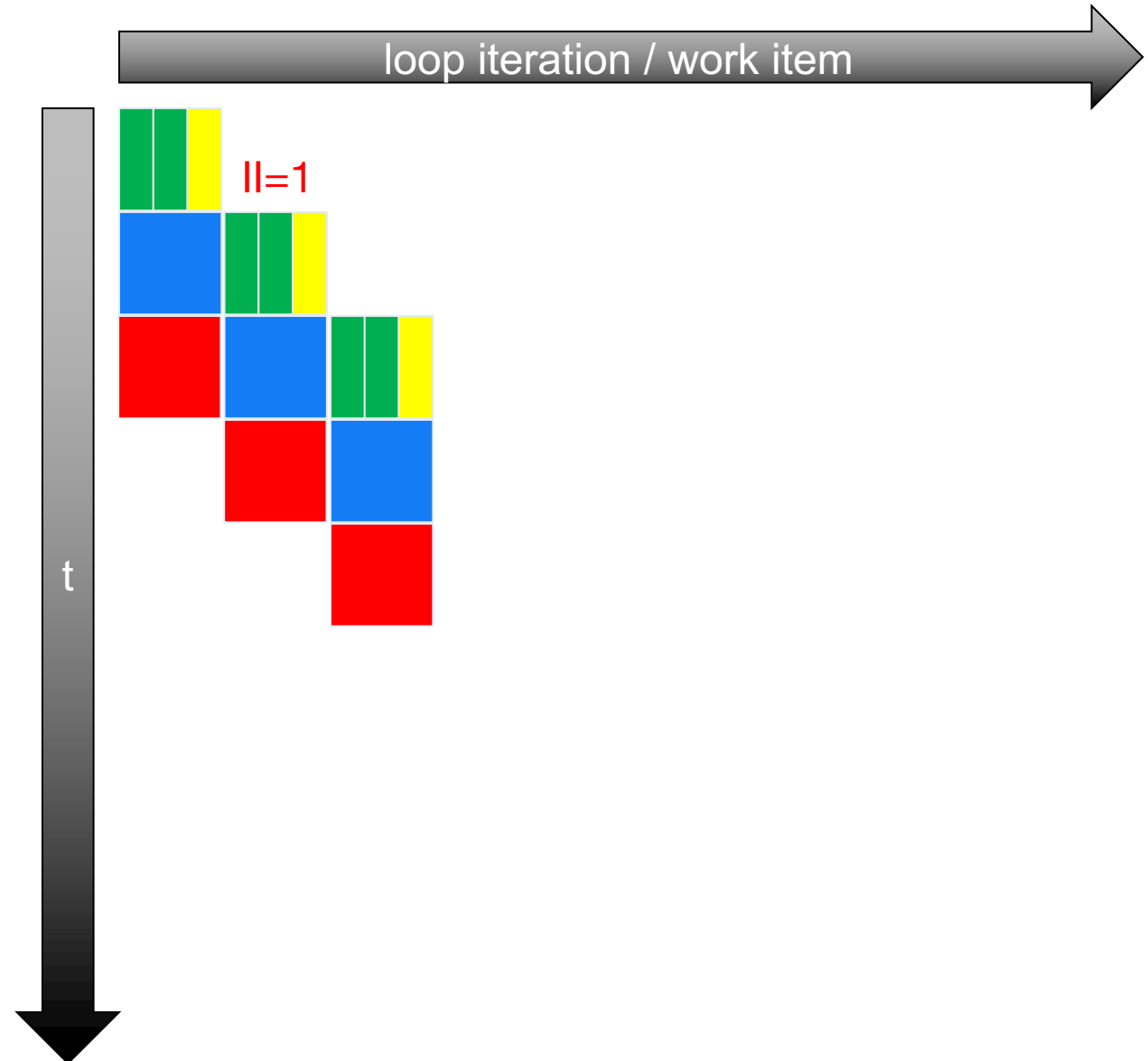
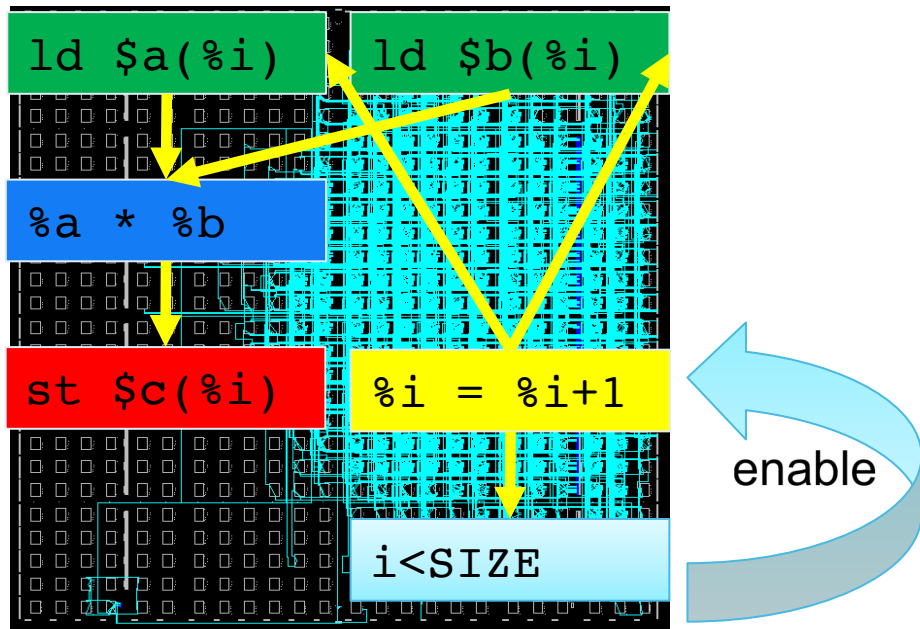
Execution on FPGA

- Spatial data path + control



Pipelining

- Use functional units every cycle
- Initiation interval II
 - describes pipeline fill rate

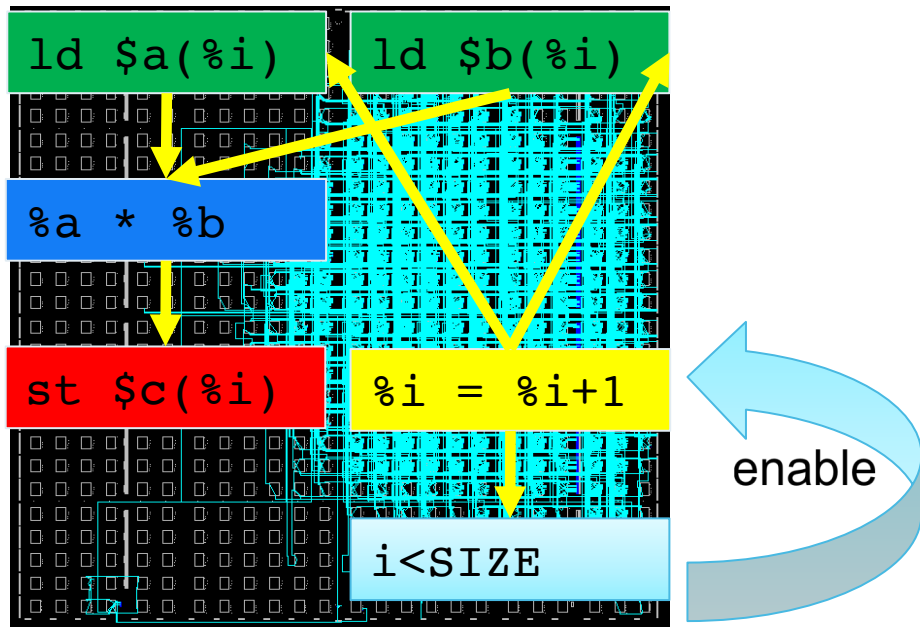


High Level Design Goals

use (expensive) arithmetic units (almost) every cycle



have scaling designs up to resource or bandwidth limits

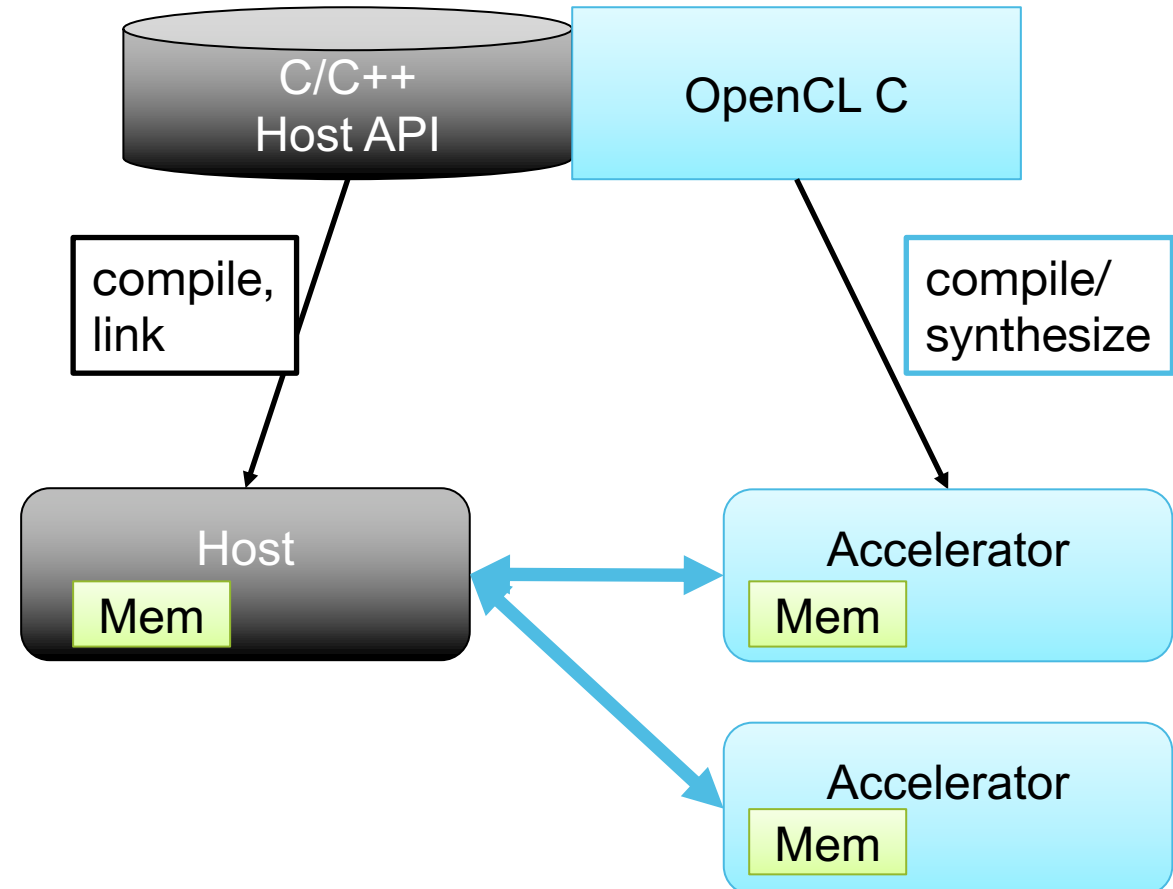


- This loop may use
 - 2 memory blocks for inputs
 - 1 DSP for multiplication
 - 1 memory block for output
 - 280 logic cells for counter and control
- Could create 3907 instances of this block
 - critical resource: 11721 memory blocks / 3
- or 3906 different blocks of this size
- or ...

OpenCL Overview

OpenCL Standard and Platform Model

- Host API and kernel language
- OpenCL platform model
- FPGA platforms
 - OpenCL 1.0 standard + selected features



<https://www.khronos.org/registry/OpenCL/>

- Detect a platform (= runtime library, driver here)
- Detect devices
- Allocate devices (= create context)

- Create and build program (on FPGA platforms = load and configure bitstreams)
- Create kernel objects
- Create command queues

- Allocate device memory
- Transfer data
- Setup kernel arguments
- Call kernels
- Synchronize

- Specify accelerator functionality in C syntax
- Special language features
 - function qualifier (`__kernel`)
 - vector data types and operations
 - address space qualifiers
- NDRangeKernel concept
 - express data parallel execution of work items and work groups
 - `get_global_id`
 - `get_local_id`
 - supported in FPGA platforms, but often not the most efficient method

Used Intel OpenCL Platform

- Intel FPGA SDK for OpenCL 18.1.1
- <https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/openccl/support.html>
 - Release Notes
 - Getting Started Guide
 - Programming Guide
 - Best Practices Guide
 - ...
 - Download the version specific PDFs!
- Target board: Bittware 520N
- Target FPGA: Intel Stratix 10 GX 2800
 - 933120 ALMs
 - 11721 M20k memory blocks (20kb each)
 - 5760 DSP blocks, 1x 32 bit IEEE 754 SP floating-point or 2x 18x19 multipliers



Used Xilinx OpenCL Platform

- Xilinx SDx 2018.3 SDAccel
- https://www.xilinx.com/html_docs/xilinx2018_3/sdaccel_doc/index.html
- <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html#documentation>
 - Release Notes, Installation, and Licensing Guide
 - Environment User Guide
 - SDAccel Environment Programmers Guide
 - SDAccel Environment Profiling and Optimization Guide
 - SDx Pragma Reference Guide
 - ...
 - Download the version specific PDFs!
- Target board: Alpha Data ADM-PCIE-8k5
- Target FPGA: Xilinx Kintex Ultrascale KU115-
 - 663360 CLB LUTs
 - 2160 BRAM blocks, 36kb each
 - 5520 DSP slices, 27x18 multipliers



Note on Xilinx Tool Scope

- **SDx combines GUI tool and command line compiler for**
 - **SoCs (Zynq) and discrete target platforms (PCIe)**
 - **SoCs**
 - enables shared memory and CPU-FPGA interactions beyond OpenCL platform model
 - uses SDSoC license
 - **discrete platforms**
 - use BSP following OpenCL platform model
 - use SDAccel license
 - **OpenCL and C/C++ kernel specification**
 - **OpenCL**
 - attributes can be used to guide high-level synthesis step
 - **C/C++**
 - HLS pragmas are used to guide high-level synthesis step (more available)
 - fixed kernel interface for discrete target platforms
- **Scope in this talk: discrete target platforms with OpenCL**

- Overview FPGAs and Goals
- OpenCL Overview

- Example 1: Vector Scale
 - compilation
 - reports
 - performance analysis
- Vector Scale Variations
 - automatic unrolling
- Example 2: SAXPY
 - blockwise design pattern
- Outer Loop Pipelining
- Streaming Kernels

Example 1: vector scale

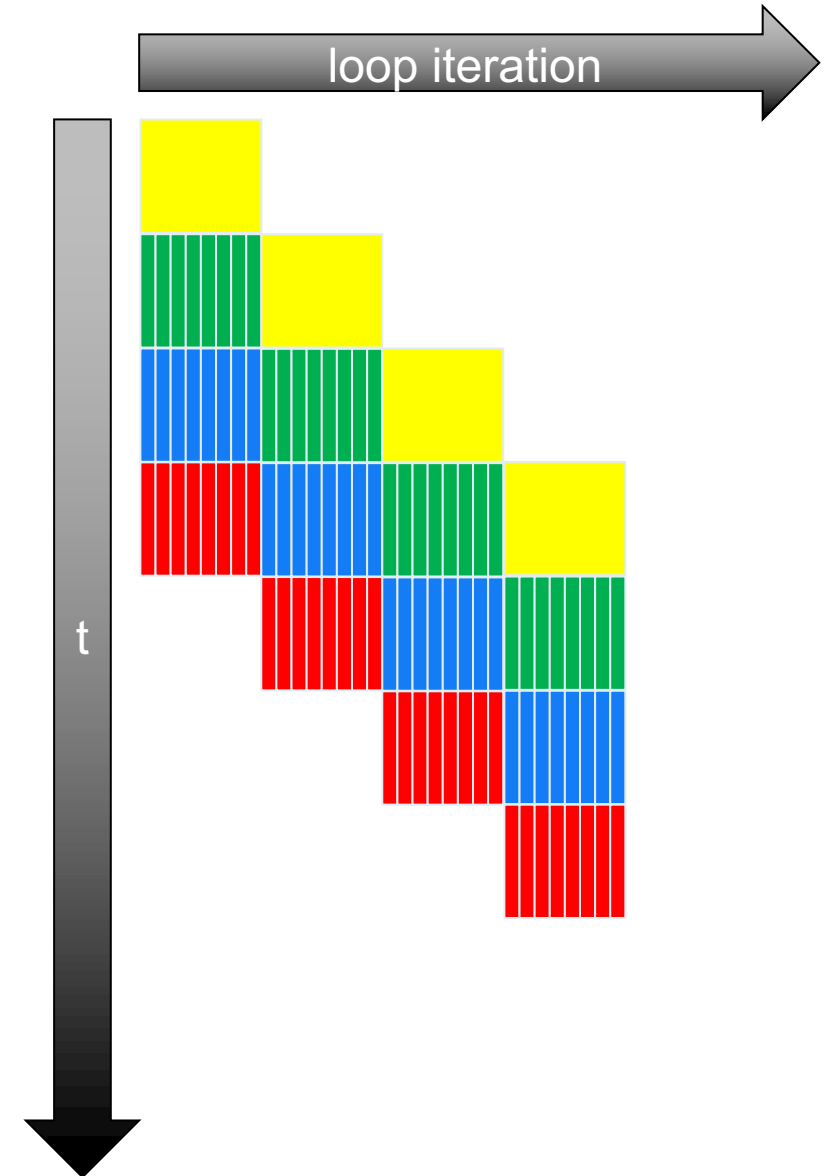
Vector Scale Single-Work Item Kernel

- Examples and essential reports of both tools available at <https://github.com/kenter/OpenCL-FPGA-examples>

```
__kernel
void vscale(
__global float16 *restrict x,
__global float16 *restrict y,
const float a,
const int size16)
{
    vscale:
    for(int i=0; i<size16; i++){
        y[i] = x[i]*a;
    }
}
```

Pipelining: Expectation

```
__kernel
void vscale(
__global float8 *restrict x,
__global float8 *restrict y,
const float a,
const int size8)
{
    vscale:
    for(int i=0; i<size8; i++){
        y[i] = x[i]*a;
    }
}
```



Compiling with Intel FPGA SDK for OpenCL

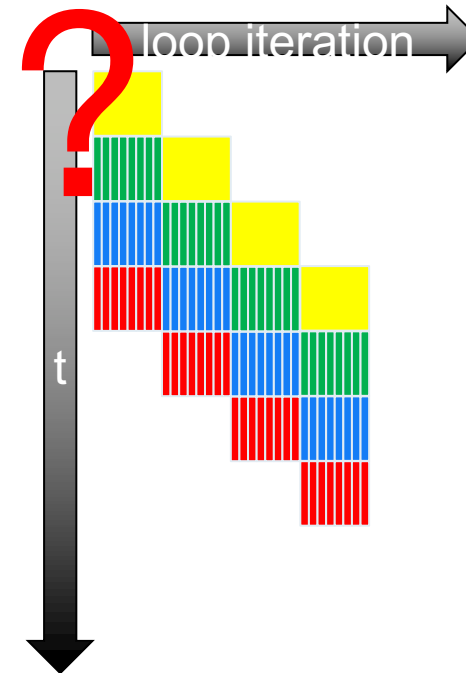
- `aoc -rtl -report -v -board=p520_max_sg2801 -fp-relaxed -fpc device/vscale1_vec.cl`

```
[kenter@fe-1 examples]$ make reportIntel-vscale1_vec
aoc -rtl -report -v -board=p520_max_sg2801 -fp-relaxed -fpc device/vscale1_vec.cl
aoc: Environment checks are completed successfully.
aoc: Cached files in /var/tmp/aocl/ may be used to reduce compilation time
aoc: Selected target board p520_max_sg2801
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Linking Object files....
aoc: Optimizing and doing static analysis of code...
aoc: Linking with IP library ...
Checking if memory usage is larger than 100%
```

```
!=====  
! The report below may be inaccurate. A more comprehensive  
! resource usage report can be found at vscale1_vec/reports/report.html  
!=====
```

; Estimated Resource Usage Summary	
; Resource	+ Usage
; Logic utilization	; 69%
; ALUTs	; 36%
; Dedicated logic registers	; 36%
; Memory blocks	; 32%
; DSP blocks	; 29%

```
aoc: First stage compilation completed successfully.
```



Intel Report (1) Summary

- reports/report.html
- Summary
 - 1 Single work-item kernel
 - high resource includes BSP

Summary

Info

Project Name	vscale1_vec
Target Family, Device, Board	Stratix 10, 1SG280LU3F50E1VGS1, nalla_pcie:p520_max_sg280l
AOC Version	18.1.1 Build 263
Quartus Version	18.1.1 Build 263 Pro
Command	aoc -rtl -report -v -board=p520_max_sg280l -fp-relaxed -fpc device/vscale1_vec.cl
Reports Generated At	Fri Mar 22 13:52:36 2019

Quartus Fit Summary

Run Quartus compile to populate this section. See details for more information.

Kernel Summary

Kernel Name	Kernel Type	Autorun	Workgroup Size	# Compute Units	HyperFlex Control Optimization
vscale	Single work-item	No	1,1,1	1	On

Estimated Resource Usage

Kernel Name	ALUTs	FFs	RAMs	DSPs	MLABs
vscale	3846	9482	46	16	8
Global Interconnect	7490	15614	52	0	0
Board Interface	480580	961160	2766	1292	0
Total	491918 (36%)	986327 (36%)	2866 (32%)	1308 (29%)	8 (69%)
Available	1385660	2771320	8955	4468	0

vscale1_vec.cl

```
1 #include "macros.h"
2
3 __kernel
4 void vscale(
5   __global float16 *restrict x,
6   __global float16 *restrict y,
7   const float a,
8   const int size)
9 {
10     vscale:
11     //__attribute__((xcl_pipeline_loop(1)))
12     for(int i=0; i<size; i++){
13         y[i] = x[i]*a;
14     }
15 }
16
```

Intel Report (2) Loop Analysis

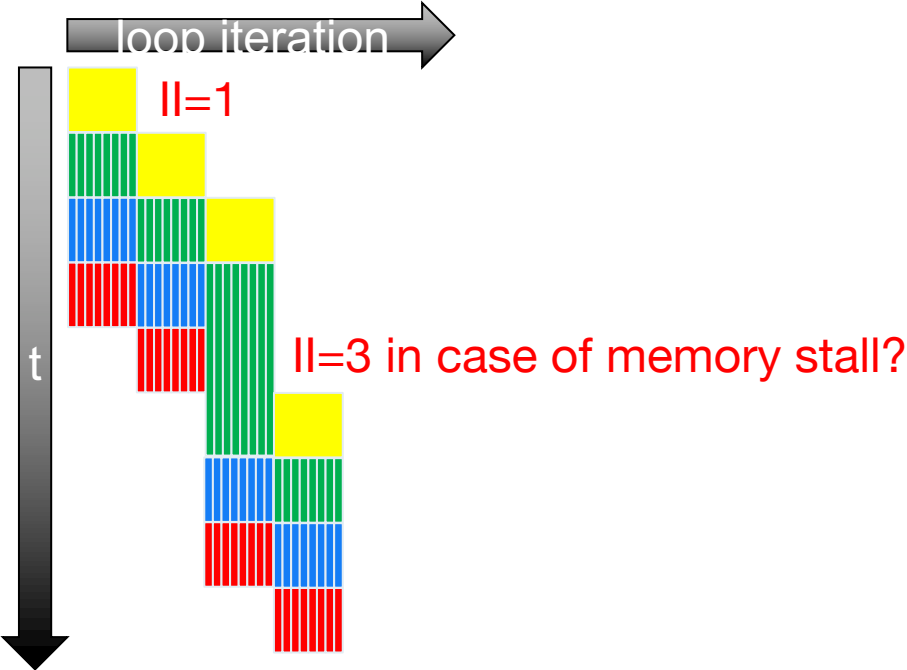
- Loop analysis

Loops analysis		<input checked="" type="checkbox"/> Show fully unrolled loops		
	Pipelined	II	Bottleneck	Details
Kernel: vscale (vscale1_vec.cl:4)				Single work-item ex...
vscale.B2 (vscale1_vec.cl:12)	Yes	~1	n/a	II is an approximation.

Details

vscale.B2:

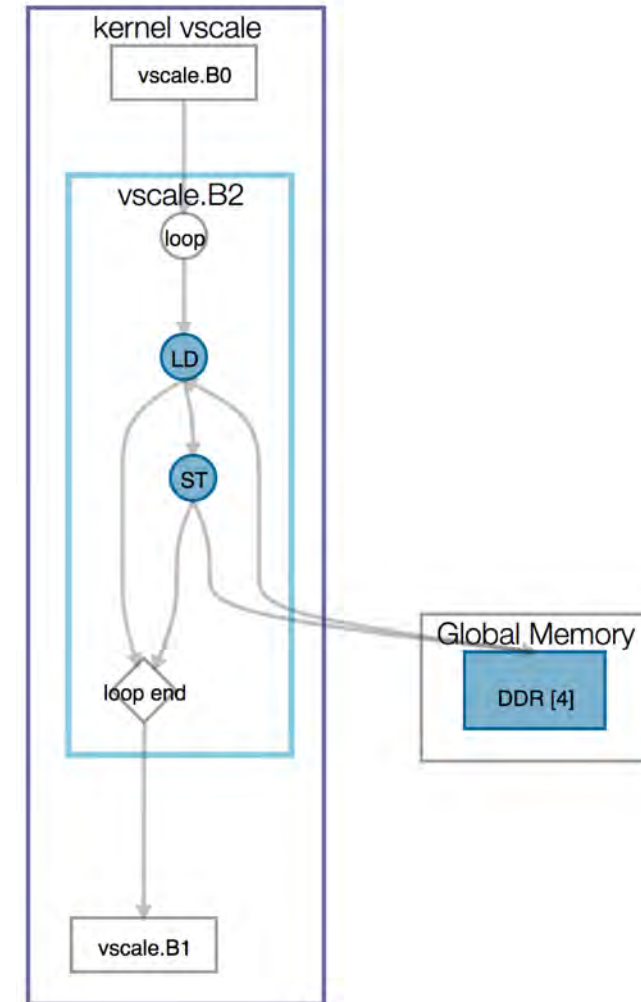
- Loop orchestration compiler optimization is enabled.
- II is an approximation due to the following storable instructions:
 - Load Operation (vscale1_vec.cl: 13)
 - Store Operation (vscale1_vec.cl: 13)



Intel Report (3) System viewer

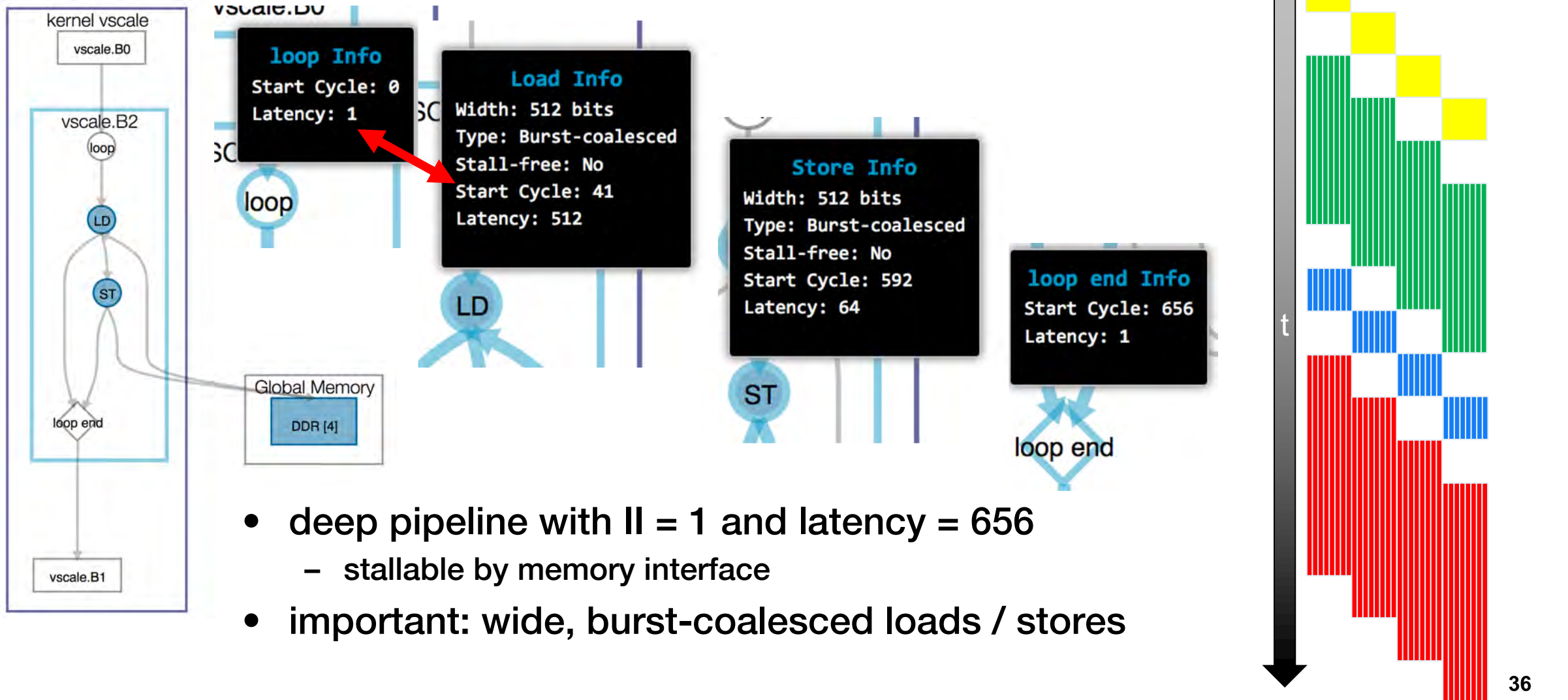
- System viewer
 - selecting the loop denoted as vsacle.B2

Details	
vscale.B2:	
Latency	656
II	1
Subloops	No
Pipelined	Yes
Fmax Bottlenecks	No
Loop Info	



Pipeline Following System Viewer

- Tooltip or details pane reveals more details on individual nodes



Intel Report (4) Area Analysis

Area analysis of system
(area utilization values are estimated)

Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	MLABs	Details
▶ Static Partition	480580 (35%)	961160 (35%)	2766 (31%)	1292 (29%)	0 (0%)	
▼ Kernel System	11338 (1%)	25167 (1%)	100 (1%)	16 (0%)	8 (0%)	
Global interconnect	7490	15614	52	0	0	For 1 global load a...
System description ROM	2	71	2	0	0	Contains informati...
▼ vscale	3846 (0%)	9482 (0%)	46 (1%)	16 (0%)	8 (0%)	1 compute unit.
Function overhead	1463	1467	0	0	6	Kernel dispatch lo...
Private Variable: - 'i' (vscale1_vec.cl:12)	32	130	0	0	0	Register, 1 reg, 32 width, 1 reg, 33 width
▶ vscale.B0	191 (0%)	144 (0%)	0 (0%)	0 (0%)	1 (0%)	
▼ vscale.B2	2160 (0%)	7741 (0%)	46 (1%)	16 (0%)	1 (0%)	
Cluster logic	418	722	16	0	1	Logic required to e...
▶ State	34	697	1	0	0	Live values and co...
▶ Feedback	65	41	0	0	0	Loop-carried depe...
▼ Computation	1643	6281	29	16	0	
▶ vscale1_vec.cl:12	105	0	0	0	0	
▶ vscale1_vec.cl:13	1538	6281	29	16	0	

```
vscale1_vec.cl
1 #include "macros.h"
2
3 __kernel
4 void vscale(
5 __global float16 *restrict x,
6 __global float16 *restrict y,
7 const float a,
8 const int size)
9 {
10     vscale:
11     //__attribute__((xcl_pipeline_loop(1)))
12     for(int i=0; i<size; i++){
13         y[i] = x[i]*a;
14     }
15 }
16
```

- 16 DSPs for 16 float multiplications

Design Review (1)

use (expensive) arithmetic units (almost) every cycle

have scaling designs up to resource or bandwidth limits

- Initiation Interval $II = 1$
- Latency $L = 656$
- Iterations N
- Time in Cycles $C = N \times II + L$

N	C	Efficiency
10	666	1.5%
100	756	13.2%
1000	1656	60.4%
10000	10656	93.8%
100000	100656	99.3%

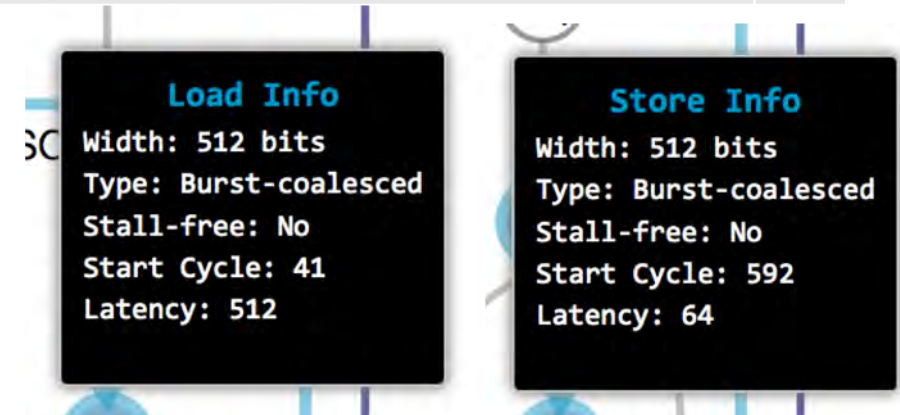
Design Review (2)

use (expensive) arithmetic units (almost) every cycle



have scaling designs up to resource or bandwidth limits

- Read and write 16 floats (32 bit) per cycle
 - 2 x 512 bit = 2 x 64 byte per cycle
- Peak bandwidth of board
 - 4 x (64+8) bit x 2400 MHz (physical interface)
 - 4 x 512 bit x 300 MHz (OpenCL interface)
 - can unroll 2x more or have 2 compute units
- Kernel can run at > 300 MHz (350-400 MHz for this type of simple kernel)
 - 2x unrolled version mildly bandwidth limited
- Main problem: low arithmetic intensity
 - only 16 of 5760 DSPs used – 0.28% utilization – 0.55% with another 2x unrolling



Compiling with Xilinx SDx (SDAccel)

- `xocc -g -R 2 -s --platform=alpha-data_adm-pcie-8k5_dynamic_5_0 --memory_port_data_width all:512 -c device/vscale1_vec.cl -o vscale1_vec.xo`

```
[kenter@fe-1 examples]$ make reportXilinx-vscale1_vec
```

```
make: aocl: Command not found
```

```
make: aocl: Command not found
```

```
xocc -g -R 2 -s --platform=alpha-data_adm-pcie-8k5_dynamic_5_0 --memory_port_data_width all:512 -c device/vscale1_vec.cl -o vscale1_vec.xo
```

```
***** xocc v2018.3 (64-bit)
```

```
**** SW Build 2405991 on Thu Dec 6 23:36:41 MST 2018
```

```
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
```

```
Attempting to get a license: ap_opencl
```

```
INFO: [XOCC 60-1306] Additional information associated with this xocc compile can be found at:
```

```
Reports: /upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/_x/reports/vscale1_vec
```

```
Log files: /upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/_x/logs/vscale1_vec
```

```
INFO: [XOCC 60-585] Compiling for hardware target
```

```
Running SDx Rule Check Server on port:36586
```

```
INFO: [XOCC 60-895] Target platform: /opt/Xilinx/SDx/2018.3/platforms/alpha-data_adm-pcie-8k5_dynamic_5_0/alpha-data_adm-pcie-8k5_dynamic_5_0.xpfm
```

```
INFO: [XOCC 60-423] Target device: alpha-data_adm-pcie-8k5_dynamic_5_0
```

```
INFO: [XOCC 60-242] Creating kernel: 'vscale'
```

```
====>The following messages were generated while performing high-level synthesis for kernel: vscale Log file: /upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/_x/vscale1_vec/vscale/vivado_hls.log :
```

```
INFO: [XOCC 204-601] Pipelining loop 'vscale'.
```

```
INFO: [XOCC 204-601] Pipelining result : Target II = 1, Final II = 1, Depth = 10.
```

```
INFO: [XOCC 60-501] Finished kernel compilation
```

```
INFO: [XOCC 60-244] Generating system estimate report...
```

```
INFO: [XOCC 60-1092] Generated system estimate report: /upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/_x/reports/vscale1_vec/system_estimate_vscale1_vec.txt
```

```
INFO: [XOCC 60-791] Total elapsed time: 0h 0m 48s
```


Xilinx Report (1) Vivado HLS Log

- Vivado HLS log

```
38 INFO: [HLS 214-115] Burst read of variable length and width 512 has been inferred on 'gmem' (/upb/scratch/
departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/device/vscale1_vec.cl:12:5)
39 INFO: [HLS 214-115] Burst write of variable length and width 512 has been inferred on 'gmem' (/upb/scratch/
departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/device/vscale1_vec.cl:12:5)
```

- Similar 512 bit burst loads / stores

```
53 INFO: [HLS 200-10] -----
54 INFO: [SCHED 204-11] Starting scheduling ...
55 INFO: [SCHED 204-61] Pipelining loop 'vscale'.
56 INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 10.
```

- $II = 1$
- Depth = 10 vs. Latency = 656 in Intel Design
 - different terminology, different treatment of off-chip memory latency
 - latency is still there (will see in next example) – estimate of loop efficiency harder

Xilinx Report (2) System Estimate

- System estimate

```
35 Area Information
36 Compute Unit  Kernel Name  Module Name  FF    LUT    DSP    BRAM
37 -----      -
```

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
vscale_1	<u>vscale</u>	<u>vscale</u>	6213	4806	48	30

```
39 -----
```

- 3 DSPs (+ some logic) per MUL

- need to combine 27x18 multipliers
- Vivado HLS provides some control over balance between DSPs and logic
- SDx with OpenCL inputs not directly
- short multiplications can be done with single DSP

```
xocc -g -R 2 -s --platform=alpha-data_adm-pcie-8k5_dynamic_5_0 --memory_port_data_width all:256
-c device/vscale5_short.cl -o vscale5_short.xo
```

```
35 Area Information
36 Compute Unit  Kernel Name  Module Name  FF    LUT    DSP    BRAM
37 -----      -
```

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
vscale_1	<u>vscale</u>	<u>vscale</u>	1982	2273	16	16

```
39 -----
```

Vector Scale Summary

- 2 very similar designs
- Found pipelining in reports
- Found 512 bit wide burst-coalesced loads / stores in reports
- Found 16 parallel floating point MULs indirectly in resource estimate

use (expensive) arithmetic units (almost) every cycle



have scaling designs up to resource or **bandwidth** limits



- It's much easier to reach bandwidth limits than compute resource limits

Vector Scale Variations

Vector Scale with Unrolling

```
__kernel
void vscale(
__global float *restrict x,
__global float *restrict y,
const float a,
const int size)
{
    __attribute__((openc1_unroll_hint(16)))
    for(int i=0; i<size; i++){
        y[i] = x[i]*a;
    }
}
```

More typical alternative for
Intel compiler
#pragma unroll 16

- <https://github.com/kenter/OpenCL-FPGA-examples> -> vscale2_u.cl
 - report files in reportIntel and reportXilinx
 - What has changed in contrast to vscale1_vec (throughput, resources, ...)?

Intel Report: Area Analysis

old

Area analysis of system
(area utilization values are estimated)
Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs
▶ Static Partition	480580 (35%)	961160 (35%)	2766 (31%)
▼ Kernel System	11338 (1%)	25167 (1%)	100 (1%)
Global interconnect	7490	15614	52
System description ROM	2	71	2
▼ vscale	3846 (0%)	9482 (0%)	46 (1%)
Function overhead	1463	1467	0
Private Variable: - 'i' (vscale1_vec.cl:12)	32	130	0
▶ vscale.B0	191 (0%)	144 (0%)	0 (0%)
▼ vscale.B2	2160 (0%)	7741 (0%)	46 (1%)
Cluster logic	418	722	16
▶ State	34	697	1
▶ Feedback	65	41	0
▼ Computation	1643	6281	29
▶ vscale1_vec.cl:12	105	0	0
▶ vscale1_vec.cl:13	1538	6281	29

Area analysis of system
(area utilization values are estimated) ▲ Collapse All ▼
Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	MLABs	Details
▶ Static Partition	480580 (35%)	961160 (35%)	2766 (31%)	1292 (29%)	0 (0%)	
▼ Kernel System	12134 (1%)	25559 (1%)	105 (1%)	16 (0%)	12 (0%)	
Global interconnect	7490	15614	52	0	0	For 1 global load a...
System description ROM	2	71	2	0	0	Contains informati...
▼ vscale	4642 (0%)	9874 (0%)	51 (1%)	16 (0%)	12 (0%)	1 compute unit.
Function overhead	1463	1467	0	0	6	Kernel dispatch lo...
Private Variable: - 'i' (vscale2_u.cl:12)	24	64	0	0	0	Register, 1 reg, 32 width
▶ vscale.B0	59 (0%)	75 (0%)	0 (0%)	0 (0%)	1 (0%)	
▼ vscale.B1	3096 (0%)	8268 (0%)	51 (1%)	16 (0%)	5 (0%)	
Cluster logic	497	865	20	0	1	Logic required to e...
▶ State	92	972	2	0	4	Live values and co...
▶ Feedback	48	41	0	0	0	Loop-carried depe...
▼ Computation	2459	6390	29	16	0	
▶ No Source Line	256	64	0	0	0	
▶ vscale2_u.cl:12	664	44	0	0	0	
▶ vscale2_u.cl:13	1539	6282	29	16	0	


new

- Same functionality, increased resources, predication for loop epilogue

- Unroll hint ignored

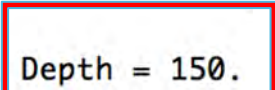
- Xilinx compiler doesn't generate automatic epilogues
- no explicit message
- area report reveals it

```
35 Area Information
36 Compute Unit  Kernel Name  Module Name  FF    LUT  DSP  BRAM
37 -----
38 vscale_1      vscale      vscale      3551  7082  3    32
39 -----
```



- and pipelining result?

```
INFO: [XOCC 204-61] Pipelining loop 'Loop 1'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 150. was 10 before!
```



- memory interface width doesn't fit

```
xocc -g -R 2 -s --platform=alpha-data_adm-pcie-8k5_dynamic_5_0 --memory_port_data_width all:32 -c device/vscale2_u.cl -o vscale
```

```
INFO: [XOCC 204-61] Pipelining loop 'Loop 1'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 10.
```

- **When code pattern doesn't fit**
 - Attributes and pragmas ignored by compilers
- **When code pattern fits**
 - Attributes and pragmas often not needed
 - `__attribute__((xcl_pipeline_loop(1)))`
 - `#pragma ii <desired_initiation_interval>`
- **Xilinx compiler doesn't generate automatic epilogues**

Vector Scale with simpler Unrolling

```
__kernel
void vscale(
__global float *restrict x,
__global float *restrict y,
const float a,
const int size)
{
    // attention, functionality only
    // identical if size is multiple of 16
    const int size16 = size / 16;
    __attribute__((opencl_unroll_hint(16)))
    for(int i=0; i<size16*16; i++){
        y[i] = x[i]*a;
    }
}
```

35	Area Information						
36	Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
37	-----	-----	-----	-----	-----	-----	-----
38	vscale_1	<u>vscale</u>	<u>vscale</u>	6454	4992	48	30
39	-----	-----	-----	-----	-----	-----	-----

Unrolled Vector Scale with Epilogue

```
__kernel
void vscale(
__global float *restrict x,
__global float *restrict y,
const float a,
const int size)
{
    const int size16 = size / 16;
    __attribute__((opencl_unroll_hint(16)))
    for(int i=0; i<size16*16; i++){
        y[i] = x[i]*a;
    }
    const int rest = size - size16;
    for(int i=size16*16; i<size16*16+rest; i++){
        y[i] = x[i]*a;
    }
}
```

```
INFO: [XOCC 204-61] Pipelining loop 'Loop 1'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 10.
INFO: [XOCC 204-61] Pipelining loop 'Loop 2'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 150.
```

Notes:

- logic resource overhead
- simplify iteration expressions
- try predication
- **tradeoff between portability and performance!**

35	Area Information						
36	Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
37	-----	-----	-----	-----	-----	-----	-----
38	vscale_1	<u>vscale</u>	<u>vscale</u>	8196	10177	48	32
39	-----	-----	-----	-----	-----	-----	-----

- Overview FPGAs and Goals
- OpenCL Overview

- **Example 1: Vector Scale**
 - compilation
 - reports
 - performance analysis
- **Vector Scale Variations**
 - automatic unrolling
- **Example 2: SAXPY**
 - blockwise design pattern
- Outer Loop Pipelining
- Streaming Kernels

Example 2: SAXPY

- Level 1 BLAS routine (single precision **a** times **x** plus **y**)

```
__kernel
void SAXPY(
__global const float *restrict x,
__global float *restrict y,
const int a,
const int size)
{
    for (int i=0; i<size; i++)
        y[i] = a*x[i] + y[i];
}
```

Differences to previous example

- Uses y as input and output
- 2 loads + 1 store

- <https://github.com/kenter/OpenCL-FPGA-examples> -> SAXPY1.cl
 - report files in reportIntel and reportXilinx
 - How does pipelining work out here?

```
INFO: [XOCC 204-61] Pipelining loop 'Loop 1'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 153, Depth = 161.
```

- Xilinx compiler generates at most 2 concurrent bursts
- More global memory access will compete for 'gmem' port of memory controller

```
WARNING: [SCHED 204-68] The II Violation in module 'SAXPY': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1)  
  between bus access on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/  
  device/SAXPY.cl:11) and bus request on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
  gitlab/2019-date-tutorial/examples/device/SAXPY.cl:11).  
WARNING: [SCHED 204-68] The II Violation in module 'SAXPY': Unable to enforce a carried dependence constraint (II = 2, distance = 1, offset = 1)  
  between bus access on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/  
  device/SAXPY.cl:11) and bus request on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
  gitlab/2019-date-tutorial/examples/device/SAXPY.cl:11).  
WARNING: [SCHED 204-68] The II Violation in module 'SAXPY': Unable to enforce a carried dependence constraint (II = 3, distance = 1, offset = 1)  
  between bus access on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/  
  device/SAXPY.cl:11) and bus request on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
  gitlab/2019-date-tutorial/examples/device/SAXPY.cl:11).  
WARNING: [SCHED 204-68] The II Violation in module 'SAXPY': Unable to enforce a carried dependence constraint (II = 4, distance = 1, offset = 1)  
  between bus access on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/  
  device/SAXPY.cl:11) and bus request on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
  gitlab/2019-date-tutorial/examples/device/SAXPY.cl:11).  
WARNING: [SCHED 204-68] The II Violation in module 'SAXPY': Unable to enforce a carried dependence constraint (II = 130, distance = 1, offset = 1)  
  between bus access on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/  
  device/SAXPY.cl:11) and bus request on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
  gitlab/2019-date-tutorial/examples/device/SAXPY.cl:11).  
WARNING: [SCHED 204-68] The II Violation in module 'SAXPY': Unable to enforce a carried dependence constraint (II = 145, distance = 1, offset = 1)  
  between bus access on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/gitlab/2019-date-tutorial/examples/  
  device/SAXPY.cl:11) and bus request on port 'gmem' (/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
  gitlab/2019-date-tutorial/examples/device/SAXPY.cl:11).
```

Design Pattern: Blockwise Read-Modify-Write

- Data without address space qualifier goes to __local memory (on chip BRAM)

```
#define BLOCK_SIZE 1024
__kernel
void SAXPY(
    __global float *restrict x,
    __global float *restrict y,
    const int a, const int size)
{
    for (int i=0; i<size;
        i+=BLOCK_SIZE)
    {
        ...
    }
}
```

```
{
    float local_x[BLOCK_SIZE];
    float local_y[BLOCK_SIZE];
    __attribute__((opencl_unroll_hint(16)))
    for(int j=0; j<BLOCK_SIZE; j++){
        local_x[j] = x[i+j];
    }
    __attribute__((opencl_unroll_hint(16)))
    for(int j=0; j<BLOCK_SIZE; j++){
        local_y[j] = y[i+j];
    }
    __attribute__((opencl_unroll_hint(16)))
    for (int j=0; j<BLOCK_SIZE; j++){
        y[j] = a*local_x[j] + local_y[j];
    }
}
```


Xilinx Reports and Performance Model

- Xilinx Pipelining

```
INFO: [XOCC 204-61] Pipelining loop 'Loop 1.1'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 3.  
INFO: [XOCC 204-61] Pipelining loop 'Loop 1.2'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 3.  
INFO: [XOCC 204-61] Pipelining loop 'Loop 1.3'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 21.
```

- 3 pipelined loops inside sequential outer loop

- Per outer loop iteration

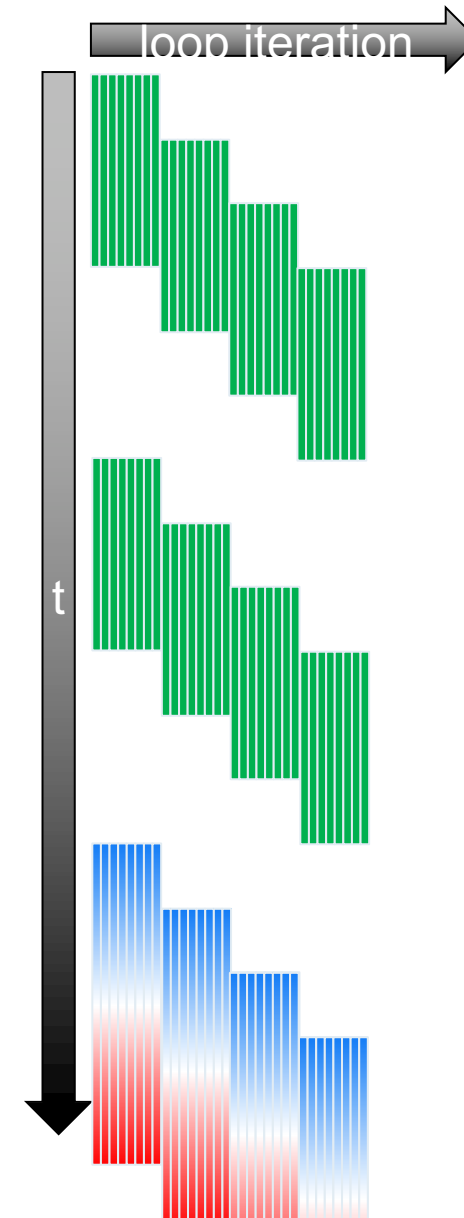
- Time in Cycles $C =$

- $N \times II(\text{Loop 1.1}) + L(\text{Loop 1.1}) +$
 - $N \times II(\text{Loop 1.2}) + L(\text{Loop 1.2}) +$
 - $N \times II(\text{Loop 1.3}) + L(\text{Loop 1.3})$

$$= 1024 + 3 + 1024 + 3 + 1024 + 21$$

- Asymptotically $N \times 3$

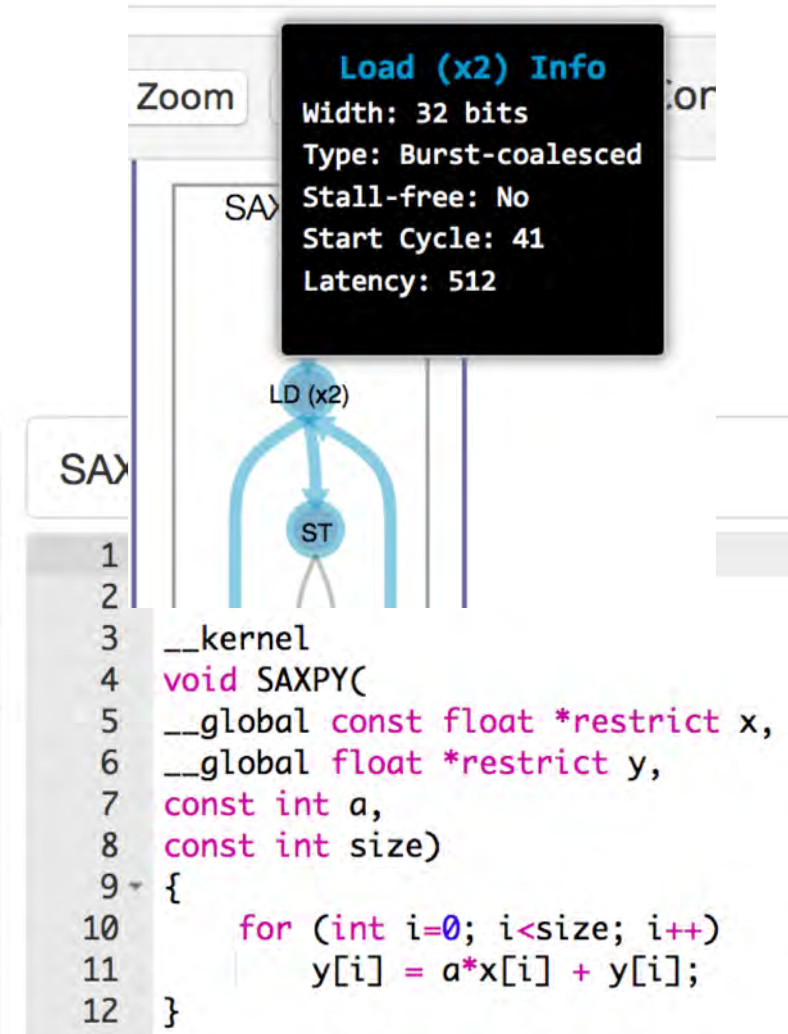
- Still much better than II 151 and no bursts



Intel SAXPY: the Good...

- No fixed 'ports' on global memory
- Can sustain multiple burst transfers concurrently
 - see later case study on efficiency
- Original SAXYP implementation efficiently pipelined

Loops analysis		<input checked="" type="checkbox"/> Show fully unrolled loops		
	Pipelined	II	Bottleneck	Details
Kernel: SAXPY (SAXPY.cl:4)				Single work-item execution
SAXPY.B2 (SAXPY.cl:10)	Yes	~1	n/a	II is an approximation.



Intel SAXPY Blockwise: the (not so) Bad...

- In this example: blockwise design for portability
- General reasons for blockwise designs
 - data reuse within block
 - reordering / indirect / irregular data access

The screenshot displays the Intel VTune Loops analysis tool interface. The top window, titled "Loops analysis", shows a table of kernel regions. The "SAXPY.B2" region is highlighted in yellow. Below this, the "Details" window provides specific information about the "SAXPY.B2" region, including a list of operations and their dependencies.

	Pipelined	II	Bottleneck	Details
Kernel: SAXPY (SAXPY_block.cl:6)				Single work-item ...
SAXPY.B2 (SAXPY_block.cl:12)	Yes	>=1	n/a	Serial exe; Memo...
16X Partially unrolled SAXPY.B4 (SAXPY_block.cl:16)	Yes	~1	n/a	It is an approxima...
16X Partially unrolled SAXPY.B6 (SAXPY_block.cl:20)	Yes	~1	n/a	It is an approxima...
16X Partially unrolled SAXPY.B8 (SAXPY_block.cl:24)	Yes	~1	n/a	It is an approxima...

```
SAXPY_block.cl
12 - for (int i=0; i<size; i+=BLOCK_SIZE){
13     float local_x[BLOCK_SIZE];
14     float local_y[BLOCK_SIZE];
15     __attribute__((opencl_unroll_hint(16)))
16 -     for (int j=0; j<BLOCK_SIZE; j++){
17         local_x[j] = x[i+j];
18     }
19     __attribute__((opencl_unroll_hint(16)))
20 -     for (int j=0; j<BLOCK_SIZE; j++){
21         local_y[j] = y[i+j];
22     }
23     __attribute__((opencl_unroll_hint(16)))
24 -     for (int j=0; j<BLOCK_SIZE; j++){
25         y[j] = a*local_x[j] + local_y[j];
26     }
27 }
28 }
```

SAXPY.B2:

- Iteration executed serially across SAXPY.B6, SAXPY.B8. Only a single loop iteration will execute inside this region due to memory dependency:
 - From: Load Operation (SAXPY_block.cl: 21)
 - To: Store Operation (SAXPY_block.cl: 25)

Intel Serial Execution (1)

- Technically the outer loop is pipelined, check [aocl-best-practices-guide](#) for details

Serial Execution

	N	400												
	i	0	0	0	0	...	0	1	...	6	...	7	...	8
	j	0	1	2	3	...	399	0	...	0	...	0	...	0
Clock Cycles	1	i = 0												
	2							i = 1						
	3													
	...													
	6													
	7													
	8	i = 0									i = 6			i = 7
	9		i = 0											
	10			i = 0										
	11				i = 0									
	12													
	...													
	407						...	i = 0						
	408													
	409													
410														
411								i = 1					i = 9...	
412														

Intel Serial Execution (2)

- Technically the outer loop is pipelined, check [aocl-best-practices-guide](#) for details

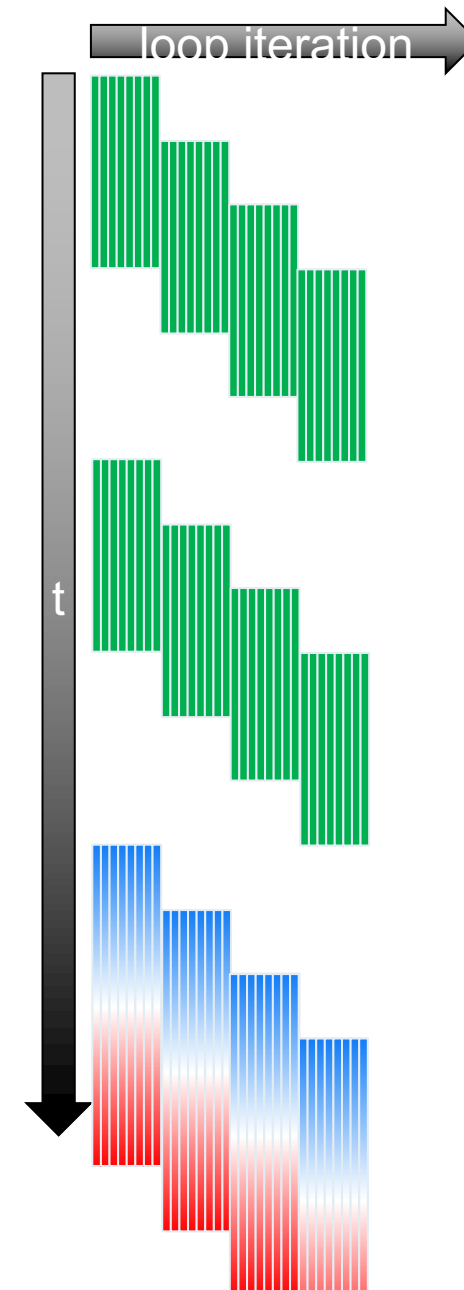
Consider the following code example:

```
kernel void serially_execute (global int * restrict A,
                             global int * restrict B,
                             global int * restrict result,
                             unsigned N) {
    int sum = 0;
    for (unsigned i = 0; i < N; i++) {
        int res;
        for (int j = 0; j < N; j++) {
            sum += A[i*N+j];
        }
        sum += B[i];
    }
    *result = sum;
}
```

In the example, the dependence in the outer loop resulted in the serial execution of the inner loop. The main difference in performance is the steady state II of outer loop = II of inner loop * (trip count of inner loop - 1) + latency. In this example, II of inner loop is 1 with latency of 4 and II of outer loop is 1 with latency of 7. If N is large, such as 400, when compared to latency, then serial execution has little impact from the outer loop II.

Intel SAXPY Blockwise: the (not so) Bad... Performance

- 3 pipelined loops inside serial execution outer loop
- Per outer loop iteration
 - Time in Cycles $C =$
 - $N \times II(\text{Loop 1.1}) + L(\text{Loop 1.1}) +$
 - $N \times II(\text{Loop 1.2}) + L(\text{Loop 1.2}) +$
 - $N \times II(\text{Loop 1.3}) + L(\text{Loop 1.3})$
 - $= 1024 + 244 + 1024 + 244 + 1024 + 79$
 - Asymptotically $N \times 3$
- Asymptotically same throughput as Xilinx design



Intel SAXPY Blockwise: the (slightly) Ugly

- Additional memory resources are allocated for outer loop pipelining

SAXPY_block.cl:13 (local_x):	
Requested size	4096 bytes
Implemented size	16384 bytes
Total replication	4
Number of banks	1 (banked on bit 4294967295)
Bank depth	64 words

- minor overhead in this case
- can be modified
 - `#pragma max_concurrency 1`

- Requested size 4096 bytes, implemented size 16384 bytes, replicated 4 times total, stall-free, 1 read and 1 write.
- 4 independent copies of this memory were created to enable simultaneous execution of 4 loop iterations defined at ([SAXPY_block.cl: 12](#))
- You can reduce the number of copies of this memory by limiting the concurrency of its loop; see the OpenCL Programming Guide for details.
- Private memory implemented in on-chip block RAM.

Lessons from SAXPY

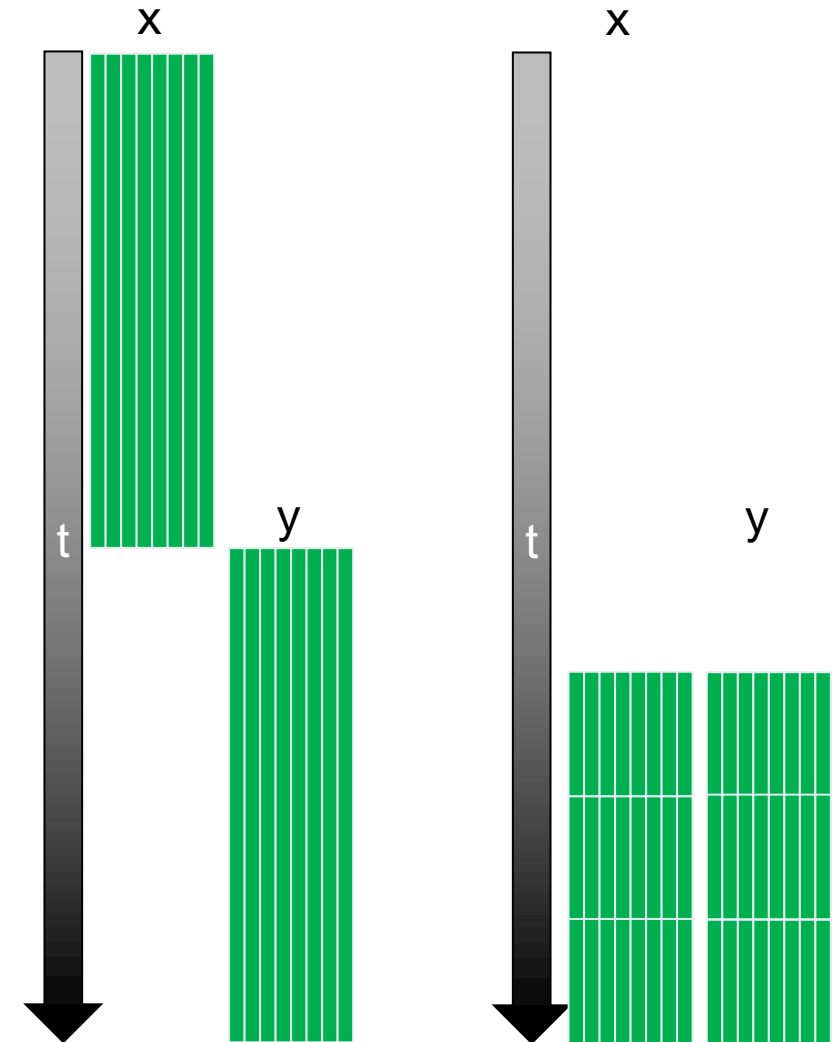
- **Xilinx designs suffer from competition on 'gmem' ports**
 - next slide: brief look at Intel LSUs
- **Blockwise designs can involve overheads like 3 x**
 - will introduce streaming kernels as broadly applicable pattern to overcome this
 - sometimes the solution is simpler
- **Intel compiler replicates local memories for outer loop pipelining**
 - will look at example without 'serial execution'

- **LSU: Load Store Unit**
 - initiate burst transfer to local buffer
 - feed kernel with data from local buffer
- **Linear buffer or cache**
 - automatic decision, mostly works well

Cached

Burst-coalesced LSUs might sometimes include a cache. A cache is created when the memory access pattern is data-dependent or appears to be repetitive. The cache cannot be shared with other loads even if the loads want the same data. The cache is flushed on kernel start and consumes more hardware resources than an equivalent LSU without a cache. The cache can be disabled by simplifying the access pattern or marking the pointer as **volatile**.

```
kernel void cached (global int * restrict in,  
                   global int * restrict out) {  
    int i = get_global_id(0);  
    int idx = out[i];  
    int cached_value = in[idx]; // Burst-coalesced cached LSU  
    out[i] = cached_value;  
}
```



LSU - DDR memory

kernel - LSU

Lessons from SAXPY

- **Xilinx designs suffer from competition on ‘gmem’ ports**
 - next slide: brief look at Intel LSUs ✓
- **Blockwise designs can involve overheads like 3 x**
 - will introduce streaming kernels as broadly applicable pattern to overcome this
 - sometimes the solution is simpler
- **Intel compiler replicates local memories for outer loop pipelining**
 - will look at example without ‘serial execution’

Outer Loop Pipelining

Resolving Serial Execution (1)

- Review reason for serial execution

	Pipelined	II	Bottleneck	Details
Kernel: SAXPY (SAXPY_block.cl:6)				Single work-item ...
SAXPY.B2 (SAXPY_block.cl:12)	Yes	>=1	n/a	Serial exe: Memo...
16X Partially unrolled SAXPY.B4 (SAXPY_block.cl:16)	Yes	~1	n/a	II is an approxima...
16X Partially unrolled SAXPY.B6 (SAXPY_block.cl:20)	Yes	~1	n/a	II is an approxima...
16X Partially unrolled SAXPY.B8 (SAXPY_block.cl:24)	Yes	~1	n/a	II is an approxima...

```
14 float local_y[BLOCK_SIZE];
15 __attribute__((opencl_unroll_hint(16)))
16 for (int j=0; j<BLOCK_SIZE; j++){
17     local_x[j] = x[i+j];
18 }
19 __attribute__((opencl_unroll_hint(16)))
20 for (int j=0; j<BLOCK_SIZE; j++){
21     local_y[j] = y[i+j];
22 }
23 __attribute__((opencl_unroll_hint(16)))
24 for (int j=0; j<BLOCK_SIZE; j++){
25     y[i+j] = a*local_x[j] + local_y[j];
26 }
27 }
28 }
29 }
```

SAXPY.B2:

- Iteration executed serially across SAXPY.B6, SAXPY.B8. Only a single loop iteration will execute inside this region due to memory dependency:
 - From: Load Operation (SAXPY_block.cl: 21)
 - To: Store Operation (SAXPY_block.cl: 25)
- See [Best Practices Guide : Nested Loops](#) for more information

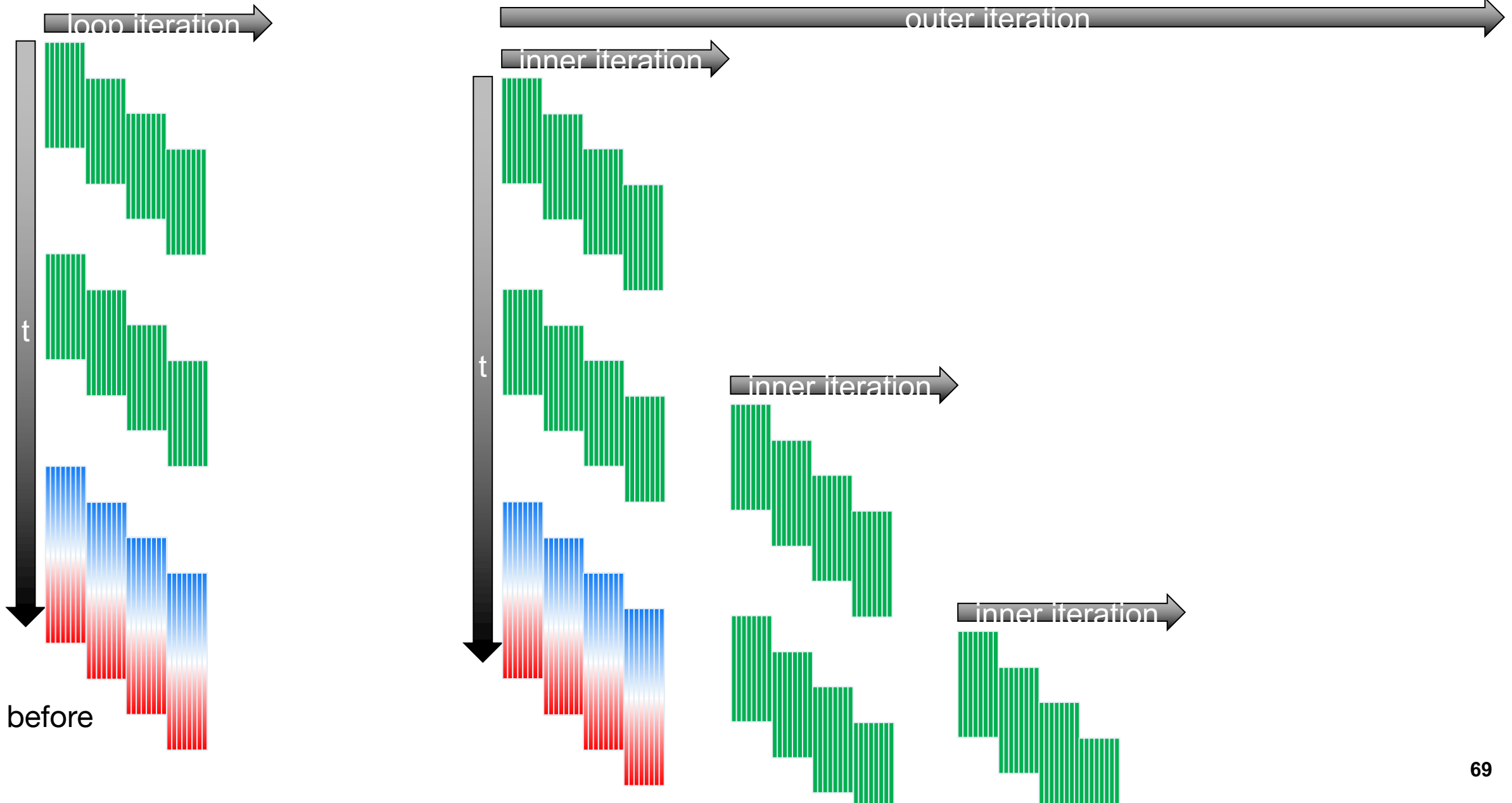
Resolving Serial Execution (2)

- Tell compiler that blocks are independent
 - #pragma ivdep

	Pipelined	II	Bottleneck	Details
Kernel: SAXPY (SAXPY_ivdep.cl:6)				Single work-item ...
SAXPY.B2 (SAXPY_ivdep.cl:13)	Yes	>=1	n/a	
16X Partially unrolled SAXPY.B4 (SAXPY_ivdep.cl:17)	Yes	~1	n/a	II is an approxima...
16X Partially unrolled SAXPY.B6 (SAXPY_ivdep.cl:21)	Yes	~1	n/a	II is an approxima...
16X Partially unrolled SAXPY.B8 (SAXPY_ivdep.cl:25)	Yes	~1	n/a	II is an approxima...

```
10 const int size)
11 {
12     #pragma ivdep
13     for (int i=0; i<size; i+=BLOCK_SIZE){
14         float local_x[BLOCK_SIZE];
15         float local_y[BLOCK_SIZE];
16         __attribute__((opencl_unroll_hint(16)))
17         for (int j=0; j<BLOCK_SIZE; j++){
18             local_x[j] = x[i+j];
19         }
20         __attribute__((opencl_unroll_hint(16)))
21         for (int j=0; j<BLOCK_SIZE; j++){
22             local_y[j] = y[i+j];
```

Execution flow with Outer Loop Pipelining



Outer Loop Pipelining Performance

- Asymptotically all functional units filled in every cycle
- Pipeline takes long to fill
 - recap from earlier example

N	C	Efficiency
10	666	1.5%
100	756	13.2%
1000	1656	60.4%
10000	10656	93.8%
100000	100656	99.3%

- now similar efficiency considerations apply to inner and outer loops
- e.g. N inner = N outer = 1000
 - efficiency = $0.604 * 0.604 = 0.365$ -> 36.5%
- in practice, latency of outer loop is much higher!

Intel Outer Loop Pipelining Summary

- **Very powerful tool**
 - this example: constant and identical trip counts of inner loops
 - successfully tested: different trip counts of inner loops based on runtime arguments
 - works also for deeper nesting levels
- **Memory replication can be very costly**
 - resource balance: ~2 block RAMs for 1 DSP
 - replication can easily lead to 3-5 x more block RAM usage

Xilinx Counterpart

- Can request pipelining in one outer loop (or function)
- `__attribute__((xcl_dataflow))`
- Generally: less flexible than Intel counterpart
- In this example: doesn't overcome 'gmem' conflict

```
ERROR: [XOCC 203-711] Bundled bus interface gmem failed dataflow checking: it cannot read data in multiple processes.  
ERROR: [XOCC 203-711] Bundled bus interface gmem has read operations in function: 'SAXPY_proc.1.1' and 'SAXPY_proc.1.1.2'.
```

```
m failed dataflow checking: it cannot read data in multiple processes.  
m has read operations in function: 'SAXPY_proc.1.1' and 'SAXPY_proc.1.1.1'.
```


Streaming Kernels

Task-level Parallelism

use (expensive) arithmetic units (almost) every cycle

have scaling designs up to resource or bandwidth limits

- **Scaling option:** add more different tasks
- **Advantage:** may lead to better balanced resource mix
- **Key goals**
 - execute tasks concurrently
 - forward reused data on chip from one task to the next
 - FPGA architecture: wires, FIFO buffers
- **OpenCL 2.0 feature:** pipe

OpenCL FPGA Tool Adaptions of Pipes

- OpenCL 2.0 pipe
 - dynamic allocation from host code
 - CPUs and GPUs don't have kernel-to-kernel wires, use shared memory
 - default: non blocking (polling)
- Intel FPGA adaptation
 - introduce name channel
 - `#pragma OPENCL EXTENSION cl_intel_channels : enable`
 - require static instantiation in .cl file
 - allow `__attribute__((depth(N)))`
 - default: blocking
 - less efficient, more standard conform pipes available
- Xilinx adaptation
 - require static instantiation in .cl file
 - require `__attribute__((xcl_reqd_pipe_depth(N)))` N in [16, 32, 64, ...32768]
 - add blocking mode (and recommend using it)

Header File for Portable FPGA Pipes

- Use blocking semantics by default

```
2  #pragma OPENCL EXTENSION cl_intel_channels : enable
3
4  #if defined(__xilinx__)
5      #define PIPE pipe
6      #define PIPE_READ(name, val) read_pipe_block(name, &val)
7      #define PIPE_WRITE(name, val) write_pipe_block(name, &val)
8      #define LABEL(x) x:
9  #elif defined(INTELFPGA_CL)
10     #define PIPE channel
11     #define PIPE_READ(name, val) val = read_channel_intel(name)
12     #define PIPE_WRITE(name, val) write_channel_intel(name, val)
13     #define LABEL(x)
14 #endif
```

Pipes in SAXPY Streaming Kernel

```
#include "macros.h"

PIPE float p_y
__attribute__((xcl_reqd_pipe_depth
(32)));

__kernel
void readY(
__global float16 *restrict y,
const int size16
)
{
    for (int i=0; i<size16; i++){
        float16 y_in = y[i];
        PIPE_WRITE(p_y, y_in);
    }
}
```

```
__kernel
void SAXPY(
__global const float16 *restrict x,
__global float16 *restrict y,
const int a,
const int size16
)
{
    for (int i=0; i<size16; i++){
        float16 y_in;
        PIPE_READ(p_y, y_in);
        y[i] = a*x[i] + y_in;
    }
}
```


SAXPY Streaming Result

- Xilinx (and Intel) design with 2 overlapping kernels with II = 1 loops

```
===>The following messages were generated while performing high-level synthesis for
kernel: SAXPY Log file: /upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/
gitlab/2019-date-tutorial/examples/_x/SAXPY_streaming16/SAXPY/vivado_hls.log :
INFO: [XOCC 204-61] Option 'relax_ii_for_timing' is enabled, will increase II to pre
serve clock frequency constraints.
INFO: [XOCC 204-61] Pipelining loop 'XCL_WG_DIM_Z_XCL_WG_DIM_Y_XCL_WG_DIM_X.1'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 21.
```

```
===>The following messages were generated while performing high-level synthesis for
kernel: readY Log file: /upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/
gitlab/2019-date-tutorial/examples/_x/SAXPY_streaming16/readY/vivado_hls.log :
INFO: [XOCC 204-61] Option 'relax_ii_for_timing' is enabled, will increase II to pre
serve clock frequency constraints.
INFO: [XOCC 204-61] Pipelining loop 'XCL_WG_DIM_Z_XCL_WG_DIM_Y_XCL_WG_DIM_X.1'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 3.
INFO: [XOCC 60-594] Finished kernel compilation
```

Streaming Kernel Summary

- Pipes for task-level parallelism
- Decoupling with pipes can also resolve other pipelining obstacles or kernel stalls
 - here: global memory interface restrictions for Xilinx
- Note on resources
 - visible resource utilization low
 - but pipes need wires – can prohibit successful routing
 - rule of thumb 512 bit pipes (like memory interface) are fine
 - much wider pipes cause problems
- Note on host code
 - OpenCL command queues are sequential by default
 - use multiple command queues for concurrent kernel execution
 - Xilinx only alternative: out-of-order command queue

Conclusion Part 1

- Overview FPGAs and Goals
- OpenCL Overview

- **Example 1: Vector Scale**
 - compilation
 - reports
 - performance analysis
- **Vector Scale Variations**
 - automatic unrolling
- **Example 2: SAXPY**
 - blockwise design pattern
- Outer Loop Pipelining
- Streaming Kernels

Concept Summary

- **Covered concepts**
 - Pipelining
 - Unrolling / Vectorization
 - Local Memory
 - Blockwise operations
 - Outer loop pipelining
 - Streaming

- **Other important concepts**
 - Local memory layout
 - Loop coalescing
 - Reductions
 - Shift Registers
 - Latency hiding

General Remarks

- Intel and Xilinx OpenCL compilers have mostly improved over the last 3+ years
 - Intel removed support for variadic macros
- New FPGA architectures always pose challenges
 - Xilinx introduction of super logic regions (SLRs) – seems well resolved now
 - Xilinx introduction of UltraRAM – unknown status to me
 - Intel Stratix 10 HyperFlex
 - higher clock frequencies partially realized already
 - tools have introduced much higher latencies
 - blocking channels discouraged
 - next challenge for both: high bandwidth memory (HMB)
 - 32 x 512 bit memory interfaces?

Part 2

**Vendor Matrix Multiplications
Complex Design Examples**

Simple, yet Efficient Matrix Multiplication Designs with OpenCL

Vendor Example Resources

Intel FPGA

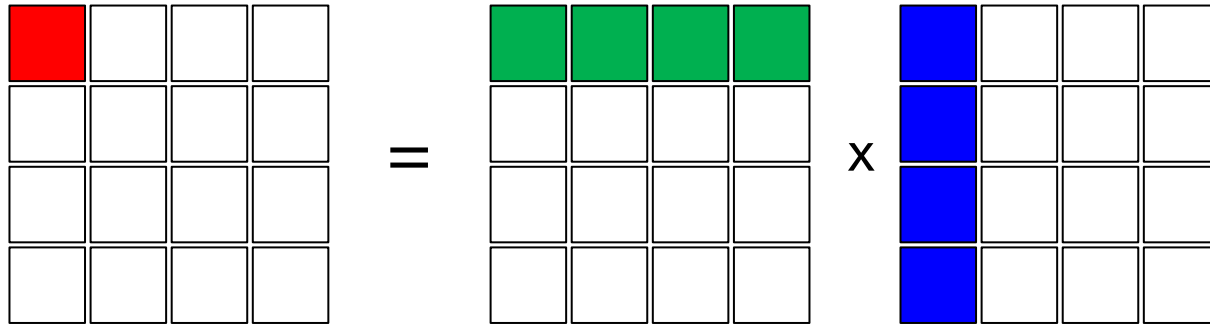
- <https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html#design-examples>
 - examples driven by application scenario, pragmatic combination of concepts
 - each example optimized for peak performance on one target device

Xilinx

- https://github.com/Xilinx/SDAccel_Examples
 - focus on presenting one or few concepts in working example
 - most examples (getting started group) not optimized to fully utilize device

Matrix Multiplication

- $C = A \times B$



- Overall data used: $3 \times N^2$
- Computations (MAC) per element: N
- Overall computations: N^3
- Peak arithmetic intensity: N

Tutorial Examples for Matrix Multiplication

Intel FPGA

- <https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html>
- Matrix Multiplication with ND range kernel
 - 64x64 tiles, up to 16x64 MAC operations per cycle

Xilinx

- https://github.com/Xilinx/SDAccel_Examples/tree/master/getting_started/kernel_opt/systolic_array_ocl
- Matrix Multiplication with systolic array
 - integer operations

Tutorial copies

- <https://github.com/kenter/OpenCL-FPGA-examples>
 - matrix_mult.cl
 - mmult.cl

Intel FPGA matrix_mul

- NDRange kernel

```
105  __kernel
106  __attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
107  __attribute__((num_simd_work_items(SIMD_WORK_ITEMS)))
108  void matrixMult( // Input and output matrices
```

- Read code inside kernel from perspective of one work item
- IDs are used to determine element positions

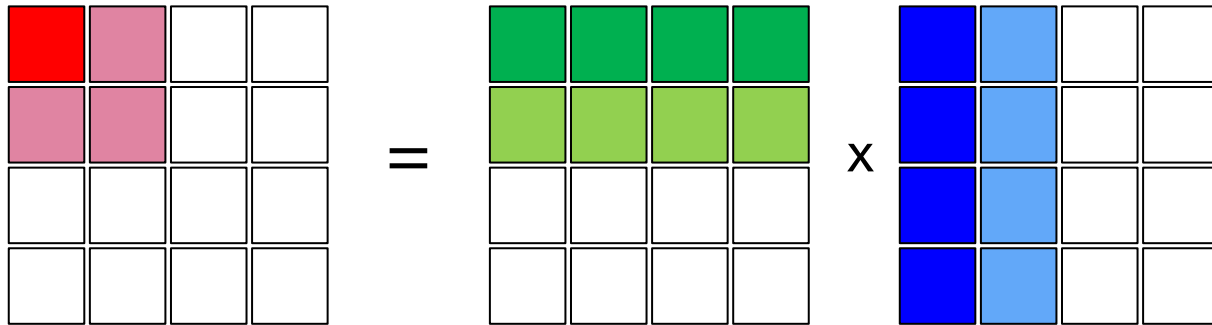
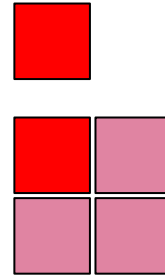
```
// Block index
int block_x = get_group_id(0);
int block_y = get_group_id(1);

// Compute loop bounds
int a_start = A_width * BLOCK_SIZE * block_y;
int a_end   = a_start + A_width - 1;
int b_start = BLOCK_SIZE * block_x;

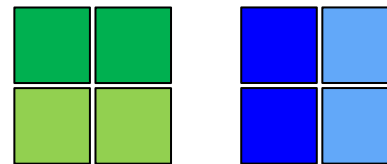
// Local ID index (offset within a block)
int local_x = get_local_id(0);
int local_y = get_local_id(1);
```

Tiling in Work Groups and Items

- Work item computes result element
- Work group computes result tile



- Process inputs per tile



- <https://github.com/kenter/OpenCL-FPGA-examples> -> matrix_mult.cl
 - additional reports in reportIntel
 - Throughput of this design?

Allocation of Local Memory

- Input Tiles

```
// Local storage for a block of input matrices A and B
__local float A_local[BLOCK_SIZE][BLOCK_SIZE];
__local float B_local[BLOCK_SIZE][BLOCK_SIZE];
```

- Where's the output tile?

```
float running_sum = 0.0f;
```

- Only output value work item
- Input tiles are shared in group (__local)
- Output elements are work-item private (still __local memory space)

```
// Store result in matrix C
C[get_global_id(1) * get_global_size(0) + get_global_id(0)] =
    running_sum;
```

Two loops

- Loop over input tiles

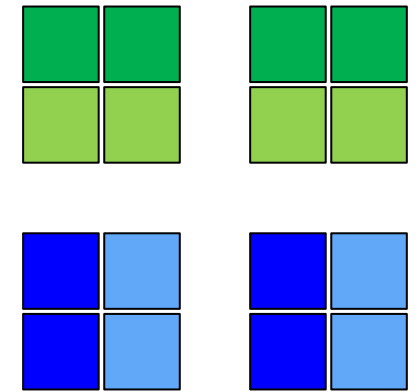
```
for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b  
    += (BLOCK_SIZE * B_width))  
{
```

- Note: need to synchronize between work items after loading tiles

- Loop over tile vectors

```
#pragma unroll  
for (int k = 0; k < BLOCK_SIZE; ++k)  
{  
    running_sum += A_local[local_y][k] * B_local[local_x][k];  
}
```

- Fully unrolled: 64 MACs per cycle

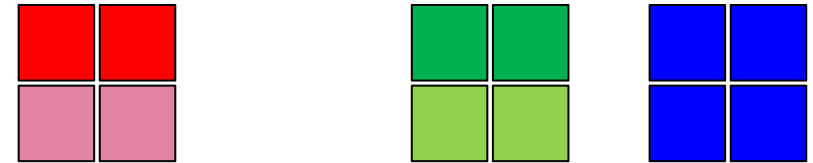


SIMD Work Items

- NDRange kernel feature SIMD Work Items (max 16)

```
105  __kernel
106  __attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
107  __attribute__((num_simd_work_items(SIMD_WORK_ITEMS)))
108  void matrixMult( // Input and output matrices
```

- Process multiple work items per cycle
- Need more elements of B concurrently



Resource Usage of Banking

- Local buffer size increased by 16 banks

matrix_mult_v1.cl:116 (A_local)	0	0	52	87	//	multipl value
matrix_mult_v1.cl:117 (B_local)	0	0	832	88	//	output compute in a ! BLOCK_S multipl SIMD_WO 89 // See tl

Details

matrix_mult_v1.cl:117 (B_local):

Requested size	16384 bytes
Implemented size	49152 bytes
Total replication	3
Number of banks	16 (banked on bits 8, 9, 10, 11)
Bank depth	4 words

Intel MM Design Evaluation

- 64 (unrolled inner loop) x 16 (SIMD work items) MAC operations per cycle
- Balanced resource usage (good fit for Arria 10 GX1150)
 - ~1024 DSPs, ~1350 BRAMs (832 for local B tile)
- Performance considerations, per pair of input tiles
 - calculate 64x64 work items, 16 in parallel -> $64 \times 64 / 16 = 64 \times 4 = 256$ cycles per pair of input tiles
 - need to load 2 tiles à 64x64 floats (eventually store 1 tile à 64x64 floats)
 - 2 x 64x64 x 32 bit = 2 x 128kb per tile
 - have 256 cycles: 2 x 512 bit per cycle – perfect match for memory interface
- Scaling considerations (Stratix 10)
 - higher compute to bandwidth ratio needs larger tiles
 - scaling problems for banked B tiles and registers for work item state (running_sum and more)

- **Covered concepts**
 - **Pipelining (different here: NDRange)**
 - **Unrolling / Vectorization**
 - **Local Memory**
 - **Blockwise operations**
 - **Outer loop pipelining (different here: work groups)**
 - **Streaming**

- **Other important concepts**
 - **Local memory layout**
 - **Loop coalescing**
 - **Reductions**
 - **Shift Registers**
 - **Latency hiding**

Xilinx mmult

- Educational example on technique systolic array
- <https://github.com/kenter/OpenCL-FPGA-examples> -> mmult.cl
 - additional reports in reportXilinx
 - How many pipelined loops?
- Blockwise processing
 - 2 read blocks, 1 compute block, 1 write back block

```
INFO: [XOCC 204-61] Pipelining loop 'readA'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 74.
INFO: [XOCC 204-61] Pipelining loop 'readB'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 74.
INFO: [XOCC 204-61] Pipelining loop 'systolic1'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 8.
INFO: [XOCC 204-61] Pipelining loop 'writeC'.
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 73.
INFO: [XOCC 60-594] Finished kernel compilation
```

Xilinx mmult.cl Snippets (1)

- Array partitioning for parallel access
 - in first dimension
 - in second dimension
 - in all dimensions

```
int localA[MAX_SIZE][MAX_SIZE] __attribute__((xcl_array_partition(complete, 1)));  
int localB[MAX_SIZE][MAX_SIZE] __attribute__((xcl_array_partition(complete, 2)));  
int localC[MAX_SIZE][MAX_SIZE] __attribute__((xcl_array_partition(complete, 0)));
```

Xilinx mmult.cl Snippets (2)

- Outer loop pipelined

```
__attribute__((xcl_pipeline_loop(1)))  
__attribute__((xcl_loop_tripcount(c_size, c_size)))  
systolic1: for(int k = 0; k < a_col; k++) {  
    __attribute__((xcl_loop_tripcount(c_size, c_size)))  
    systolic2: for(int i = 0; i < MAX_SIZE; i++) {  
        __attribute__((xcl_loop_tripcount(c_size, c_size)))  
        systolic3: for(int j = 0; j < MAX_SIZE; j++) {
```

- what about the two loops inside?
 - again, code pattern determines further transformations

```
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'systolic1'  
(/upb/scratch/departments/pc2/groups/pc2-mitarbeiter/kenter/  
gitlab/2019-date-tutorial/examples/device/mmult.cl:151) in function  
'mmult' for pipelining.
```


Xilinx mmult.cl Snippets (3)

- 2D loop unrolling
 - simple form of systolic array

```
__attribute__((xcl_pipeline_loop(1)))
__attribute__((xcl_loop_tripcount(c_size, c_size)))
systolic1: for(int k = 0; k < a_col; k++) {
    __attribute__((xcl_loop_tripcount(c_size, c_size)))
    systolic2: for(int i = 0; i < MAX_SIZE; i++) {
        __attribute__((xcl_loop_tripcount(c_size, c_size)))
        systolic3: for(int j = 0; j < MAX_SIZE; j++) {
```

```
//      B_0      B_1      B_2      B_3
//      |        |        |        |
//      v        v        v        v
//      _____
//      |  |      |  |      |  |      |  |
// A0_->|C00| ---- |C01| ---- |C02| ---- |C03|
//      |  |      |  |      |  |      |  |
//      _____
//      |  |      |  |      |  |      |  |
// A1_->|C10| ---- |C11| ---- |C12| ---- |C13|
//      |  |      |  |      |  |      |  |
//      _____
//      |  |      |  |      |  |      |  |
// A2_->|C20| ---- |C21| ---- |C22| ---- |C23|
//      |  |      |  |      |  |      |  |
//      _____
//      |  |      |  |      |  |      |  |
// A3_->|C30| ---- |C31| ---- |C32| ---- |C33|
//      |  |      |  |      |  |      |  |
```

Xilinx mmult.cl Snippets (4)

- code inside loop
 - PEs get data directly from input array

```
// Get previous sum
int last = (k==0) ? 0 : localC[i][j];

// Update current sum
// Handle boundary conditions
int a_val = (i < a_row && k < a_col)? localA[i][k] : 0;
int b_val = (k < b_row && j < b_col)? localB[k][j] : 0;
int result = last + a_val*b_val;

// Write back results
localC[i][j] = result;
```

accumulation in 1 cycle required

```
//      B_0      B_1      B_2      B_3
//      |      |      |      |
//      v      v      v      v
//      _____
//      |  |      |  |      |  |      |  |
// A0_->|C00| ---- |C01| ---- |C02| ---- |C03|
//      |  |      |  |      |  |      |  |
//      _____
//      |  |      |  |      |  |      |  |
// A1_->|C10| ---- |C11| ---- |C12| ---- |C13|
//      |  |      |  |      |  |      |  |
//      _____
//      |  |      |  |      |  |      |  |
// A2_->|C20| ---- |C21| ---- |C22| ---- |C23|
//      |  |      |  |      |  |      |  |
//      _____
//      |  |      |  |      |  |      |  |
// A3_->|C30| ---- |C31| ---- |C32| ---- |C33|
//      |  |      |  |      |  |      |  |
```

Xilinx mmult.cl Results + Limitations

- Given example with $12 \times 12 = 144$ parallel operations

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
mmult_1	<u>mmult</u>	<u>mmult</u>	52635	23720	441	30

- Single cycle accumulation not possible for floating point

```
INFO: [XOCC 204-61] Pipelining loop 'readA'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 74.  
INFO: [XOCC 204-61] Pipelining loop 'readB'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 74.  
INFO: [XOCC 204-61] Pipelining loop 'systolic1'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 7, Depth = 23.  
INFO: [XOCC 204-61] Pipelining loop 'writeC'.  
INFO: [XOCC 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 73.
```

- Need to accumulate into several registers (~latency) and later sum up

- Covered concepts
 - **Pipelining**
 - **Unrolling / Vectorization (different here: systolic array, unrolling in 2 dimensions)**
 - **Local Memory**
 - **Blockwise operations**
 - **Outer loop pipelining**
 - **Streaming**

- Other important concepts
 - **Local memory layout**
 - **Loop coalescing**
 - **Reductions**
 - **Shift Registers**
 - **Latency hiding**

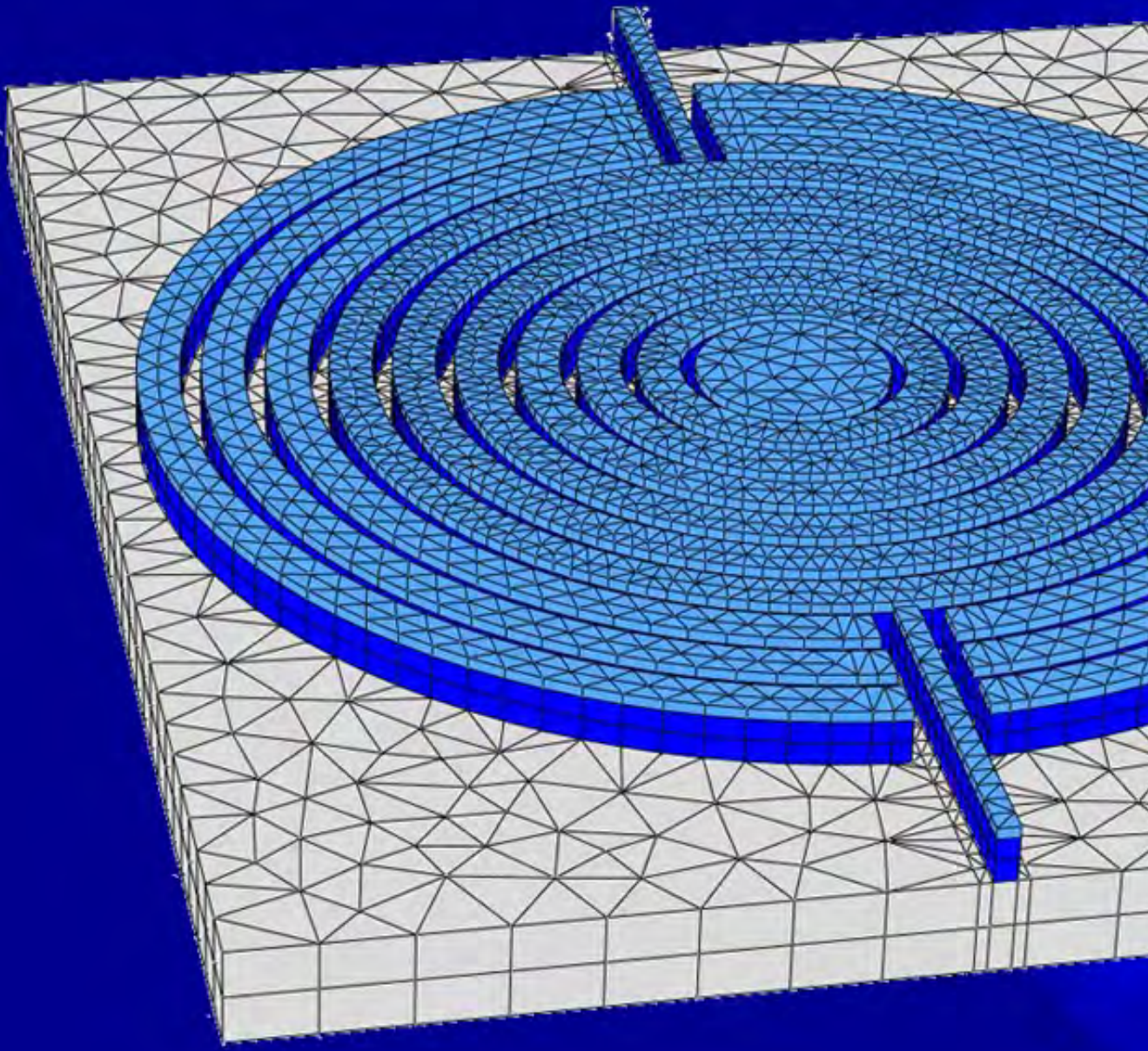
Success Stories

Complex Design Examples

Electrical Engineering: Nanophotonics Simulations

- FDTD stencil solver for Maxwell equations
 - regular 2D grid
 - acceleration with FPGAs
 - generalization of OpenCL design for Xilinx and Intel FPGA compilers
- Kenter et. al: **Flexible FPGA design for FDTD using OpenCL**. *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*. Sep. 2017.

Electrical Engineering: Nanophotonics Simulations (2)



- **Discontinuous Galerkin solver for Maxwell equations**
 - regular operations on unstructured grids
 - acceleration mit FPGAs
 - generalization in domain specific language (DSL) and compiler
- Kenter et. al: **OpenCL-based FPGA design to accelerate the nodal Discontinuous Galerkin method for unstructured meshes.** *Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2018.

Thank you!
Questions?