SimBSP

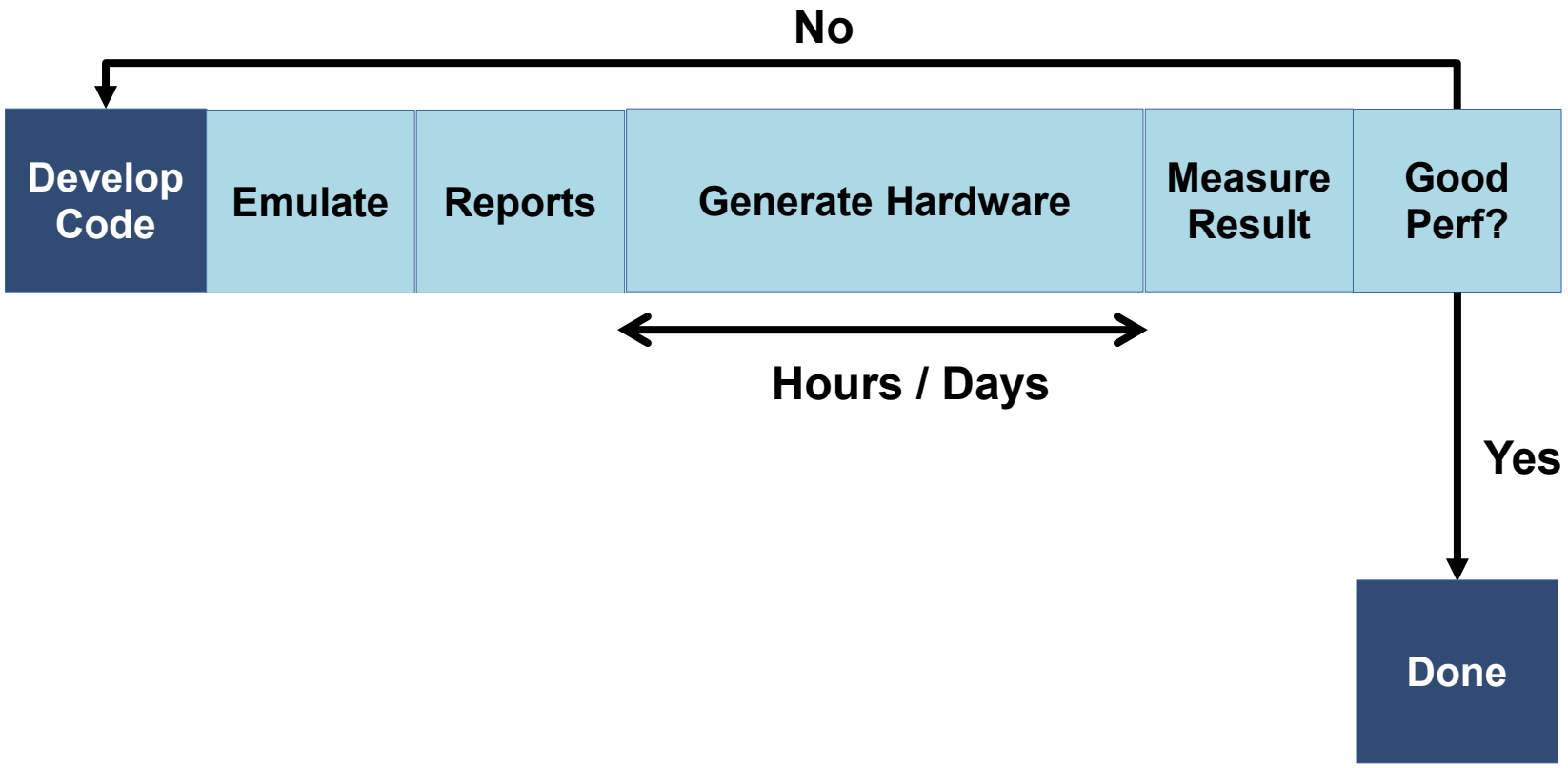
Enabling RTL Simulation for Intel FPGA OpenCL Kernels

Ahmed Sanaullah, Chen Yang, Daniel Crawley and Martin C. Herbordt

Department of Electrical and Computer Engineering, Boston University

BOSTON UNIVERSITY

The Intel OpenCL Toolflow



Code Development Challenges

- There are a number of challenges associated with code development, such as:
  - Reducing development time
  - Implementing efficient pipelines
    - Even codes which took a long time to develop are not guaranteed to be efficient
  - Reducing lines of code needed to express designs
  - Maintaining designs with relative ease across toolflow/API/SDK updates
  - Expertise required and learning curves

Project Overview

Advancing OpenCL for FPGAs

(without modifying the compiler or other proprietary design tools)

An Empirically Guided Optimization Framework

FFT In-Depth Case Study
HEART18

Optimization Characterization
FPT18

Hacking the Toolflow to get …

Rapid turnaround
HPEC18

RTL Simulation
H2RC18

Augmentations

Problems to be addressed –

1. Performance-programmability gap: Optimizing OpenCL code is hard
2. No RTL simulation capabilities in the standard OpenCL toolflow
3. Hard to integrate OpenCL generated pipelines into existing structures

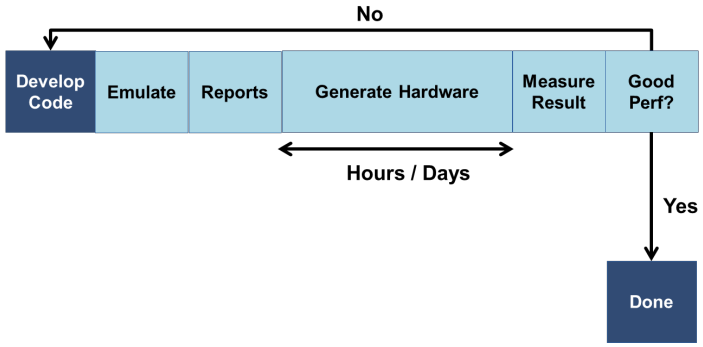
Solution

- Cannot rewrite a new compiler to address these shortcomings
- Hence, we augment the existing Intel toolflow

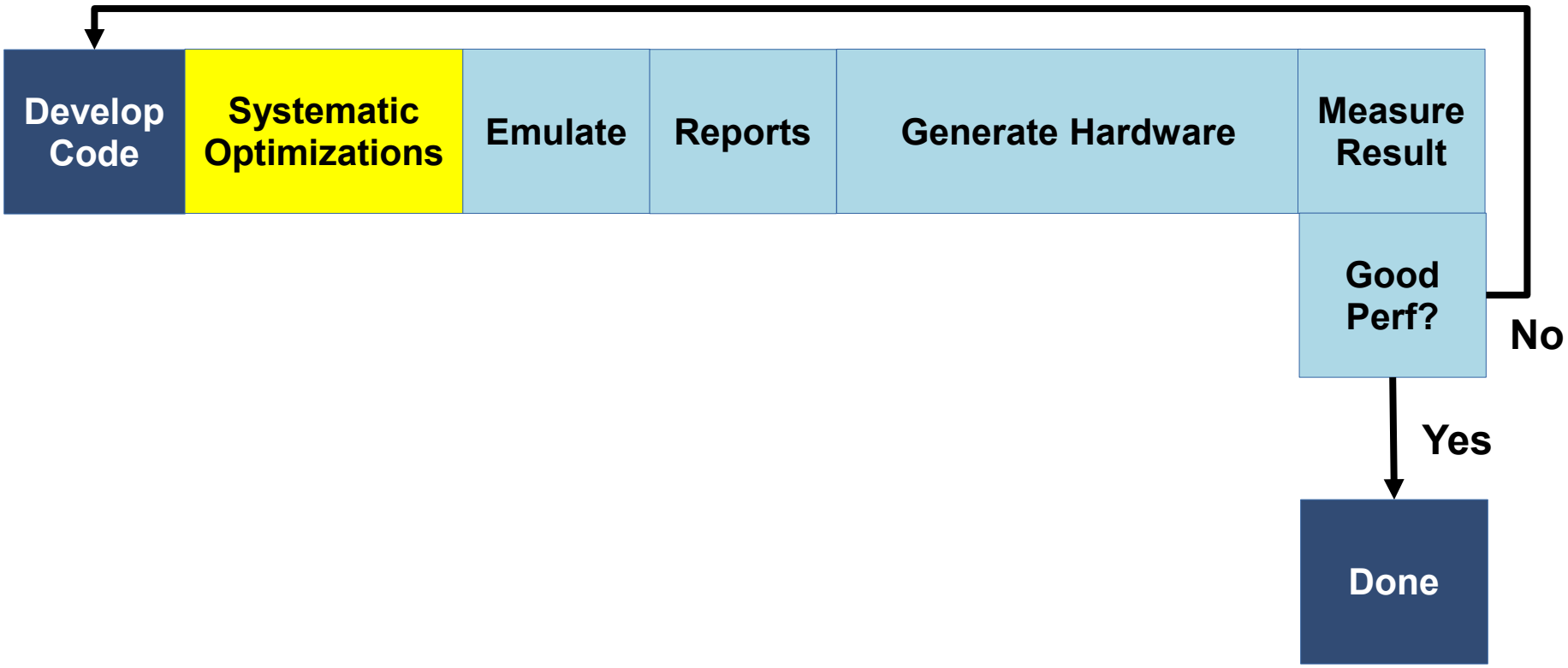
Benefits of Our Work

- Systematic optimizations that are easy to apply, and can even be automated
- Rapid turnaround necessary to be able to evaluate optimizations
- Integration of OpenCL pipelines into existing structures

Augmentation 1



FPT18, HEART18



Apply optimization methods to parallel computing dwarfs

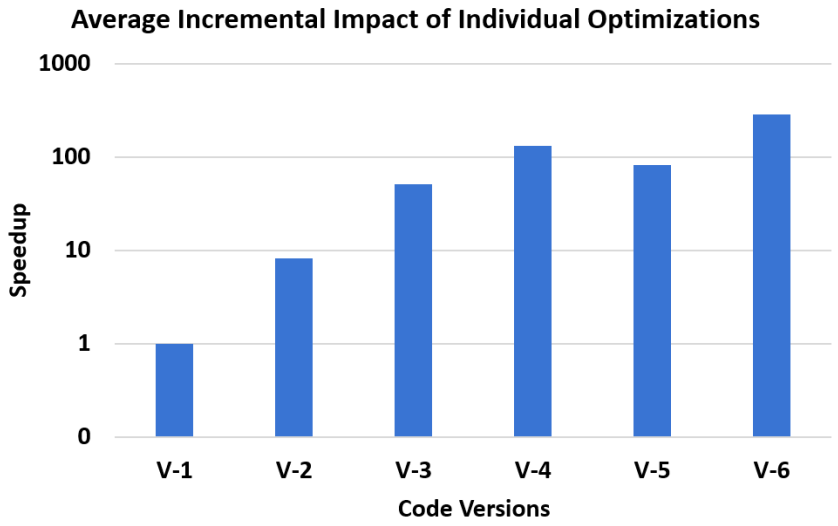
A. Sanaullah, R. Patel, and M. Herbordt, "*An Empirically Guided Optimization Framework for FPGA OpenCL*," in Proc. IEEE Conf. on Field Programmable Technology, 2018.

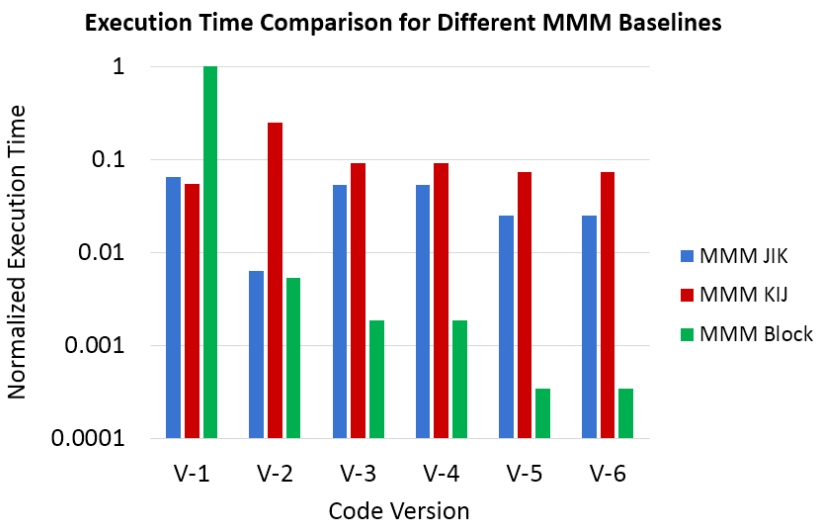
Benchmarks	Dwarf	Problem Size
NW	Dynamic Programming	16K x16K Integer Table
FFT	Spectral Methods	64 point Radix-2 1D FFT, 8192 Vectors
Range Limited	N-Body	180 particles per cell, 15% pass rate
PME	Structured Grids	1,000,000 Particles, 32^3 grid, 3D Tri-Cubic Interpolation
MMM	Dense Linear Algebra	1K x 1K Matrix, Single Precision
SpMV	Sparse Linear Algebra	1K x 1K Matrix, Single Precision, 5%-Sparsity, NZ=51122
CRC	Combinational Logic	100MB CRC32

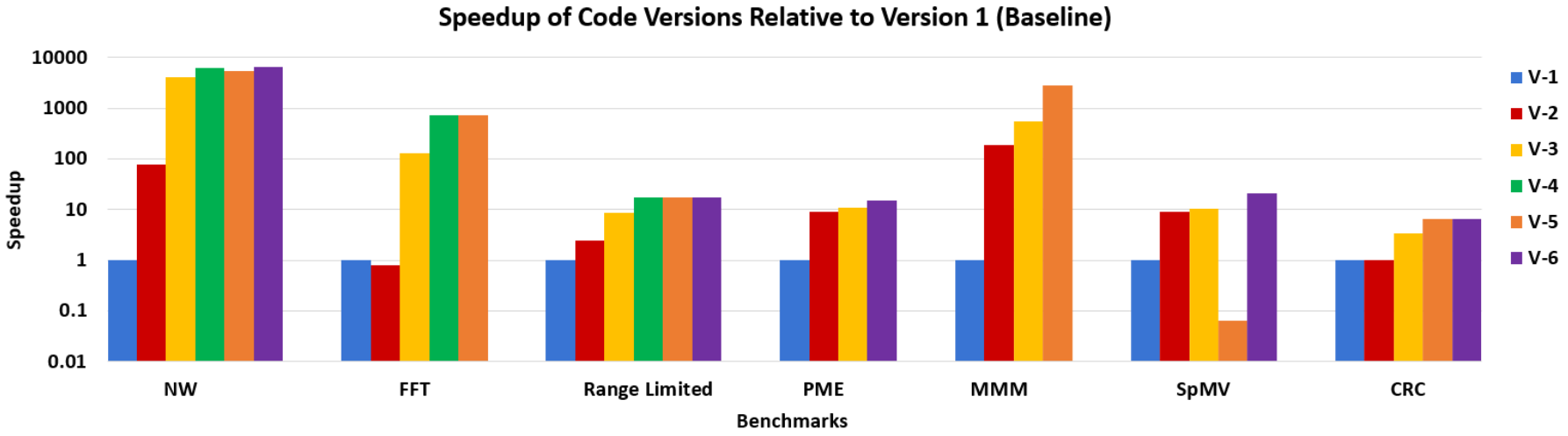
Ver.	Optimizations
0	(**GPU code** for porting to FPGA OpenCL)
1	**Single thread code with cache optimization**
2	Implement **task parallel computations in separate kernels** and connect them using channels
	Fully unroll all loops w/ #*pragma unroll*
	Minimize variable declaration outside compute loops – use temps where possible
	Use **constants** for problem sizes and data values – instead of relying on off-chip memory access
	Coalesce memory operations
3	Implement the entire computation within a **single kernel** and avoid using channels
4	**Reduce array sizes** to infer pipeline registers as registers, instead of BRAMs
5	Perform **computations in detail**, using temporary variables to store intermediate results
6	Use **predication** instead of conditional branch statements when defining forks in the data path

Benchmarks	V-1	V-2	V-3	V-4	V-5	V-6
NW	✓	✓	✓	✓	✓	✓
FFT	✓	✓	✓	✓	✓	
Range Limited	✓	✓	✓	✓	✓	✓
PME	✓	✓	✓			✓
MMM	✓	✓	✓		✓	
SpMV	✓	✓	✓		✓	✓
CRC	✓	✓	✓		✓	✓

Characterization of Optimizations

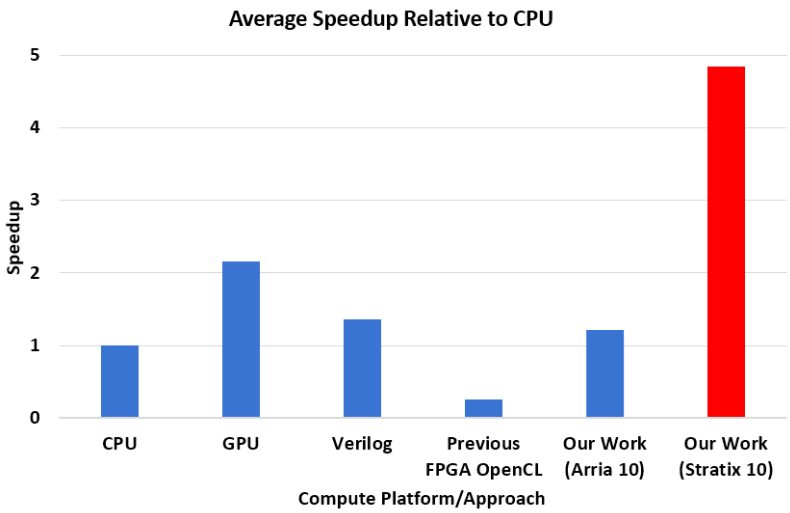


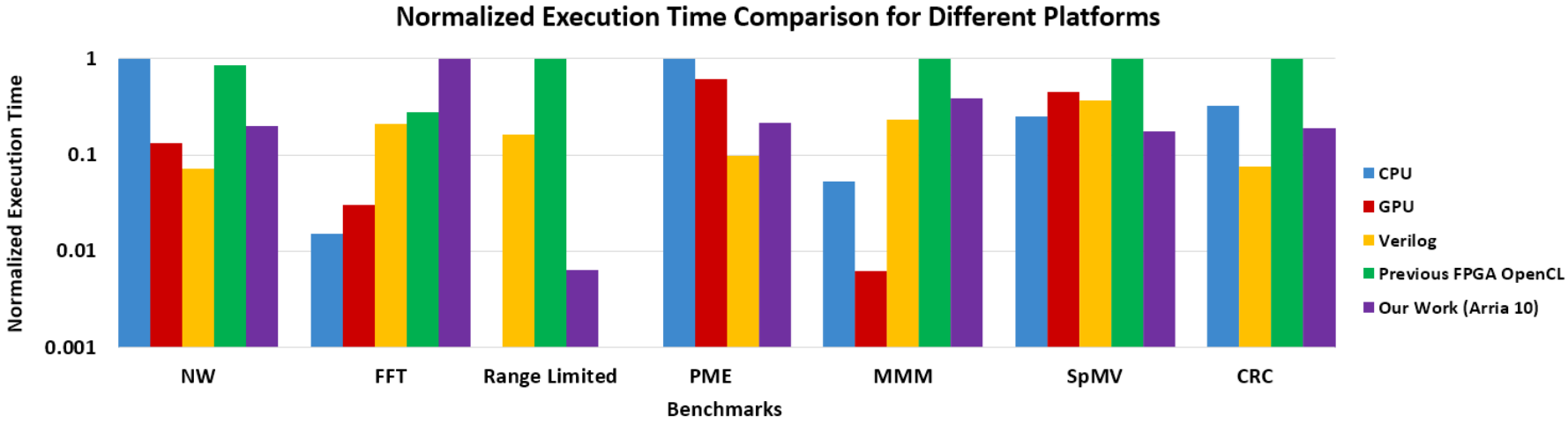




Comparison with Different Platforms

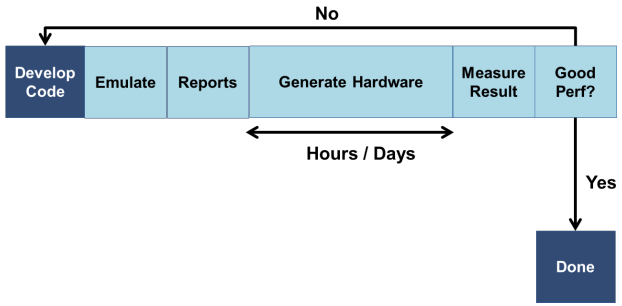
- Outperform existing CPU by **1.2x** on average
- Outperform previous FPGA OpenCL by **5x** on average
- Within **12%** of average hand tuned HDL performance
- **We estimate a 4x increase in performance of our OpenCL designs using Intel Stratix 10**





Augmentation 2

FPT18, HEART18
HPEC18, HEART18



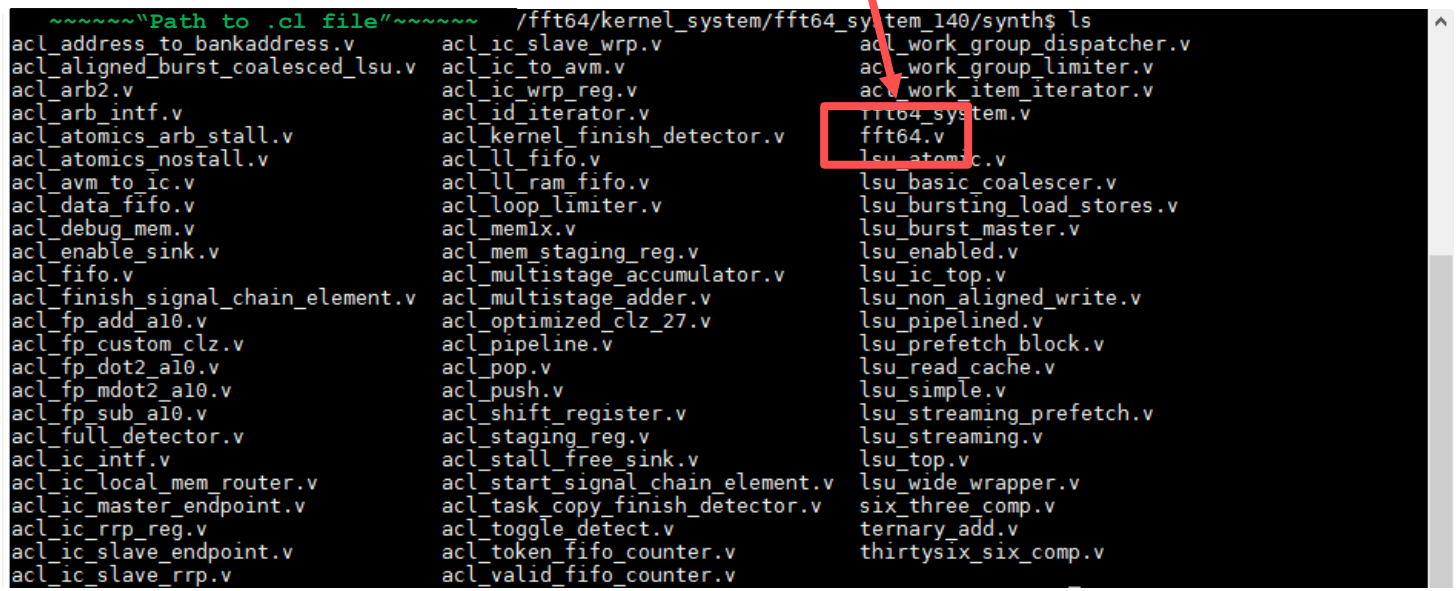
Develop Code
Systematic Optimizations
Emulate
Reports
Gen HDL
OpenCL-HDL
Isolate Pipelines
Re-interface
RTL Simulation
Good Perf?
Yes
No
Integrate?
Yes
No
Done
Generate Hardware for custom system
Done
Generate Hardware for Full OpenCL

Usage of Compute Pipeline HDL

- Analytics for optimizing kernel code
  - RTL simulation
    - Determine latency
    - Verify functional correctness
    - Remember to include all the files from the source file directory
  - Compilation
    - Post-synthesis resource usage
      - More reliable than post compilation log file of normal toolflow
    - Post place&route resource usage and frequency
      - Create custom wrapper and fit design to board
      - Significantly smaller fitting time due to no BSP
      - Faster design iterations based on feedback
- Custom deployment
  - Integration of compute pipelines into existing HDL codes

Finding Our Source File

- Compilation Breakpoint
  - Perform full compilation with the –v flag
  - Terminate once successful source file generation is displayed (can also be automated by modifying tcl files)
- Source Files
  - Located *in [Path to Kernel File]/<kernel_filename>/kernel_subsystem/<kernel_filename>_system_140/synth/*
  - Folder contains the implemented kernel file, <kernel_filename>.v, as well as additional modules needed for compilation **(including custom RTL)**



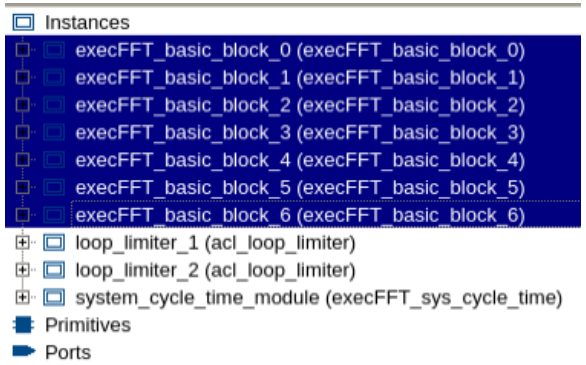
  - **Change .v to .sv before using**

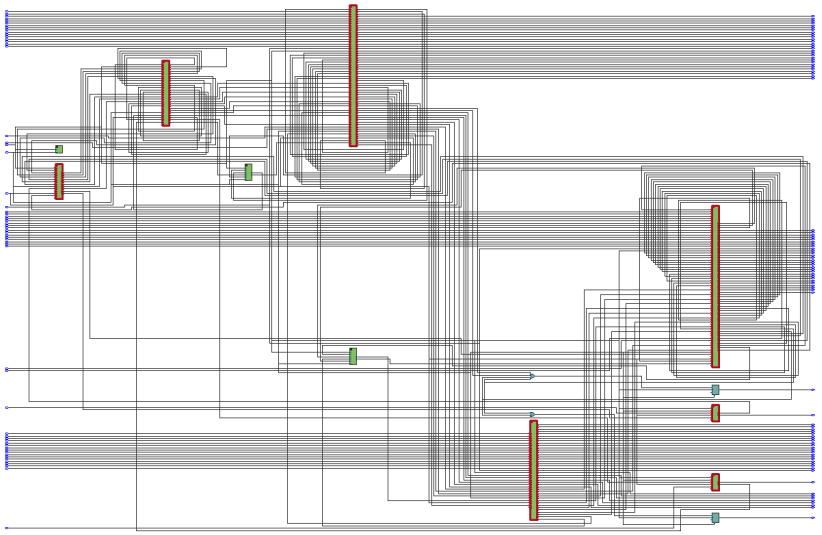
Finding Our Code Within the File

- Basic blocks are modules used to implement the kernel
  - Construct logic using basic behavioral functions and Altera IP blocks
  - A single kernel can generate multiple basic blocks
  - Number and function of these modules depends on kernel implementation

- Observed rules of typical basic block generation
  - Each normal loop generates a basic block module
  - Nested normal loops generate independent modules and connect to their parent loop module
  - Unrolled loops will also generate a separate module.
    - However, consecutive unrolled loops, or any unrolled nested loop within an unrolled loop, will not generate a new module

- All compute pipelines are within the same basic block
  - Since all compute loops are unrolled





Results

- FPGA
  - Intel® Arria® 10AX115H3F34E2SGE3
    - 427,200 ALMs
    - 1506K Logic Elements
    - 1518 DSP Blocks
    - 53Mb On-chip Memory
  - Intel® SDK for OpenCL™ 16.0
  - Intel® FFT IP Core
- CPU
  - 2.7 GHz Intel® Xeon® E5-2680
    - eight core
  - Intel® C++ Compiler
  - Intel® MKL DFTI
- GPU
  - NVIDIA Tesla P100 PCIe 12GB
    - 3584 CUDA Cores
    - 549 GB/s Off-Chip Memory (HBM2)
  - CUDA 8.0
  - cuFFT

TABLE I
LATENCY AND RESOURCE USAGE FOR OPENCL-HDL 1D FFT

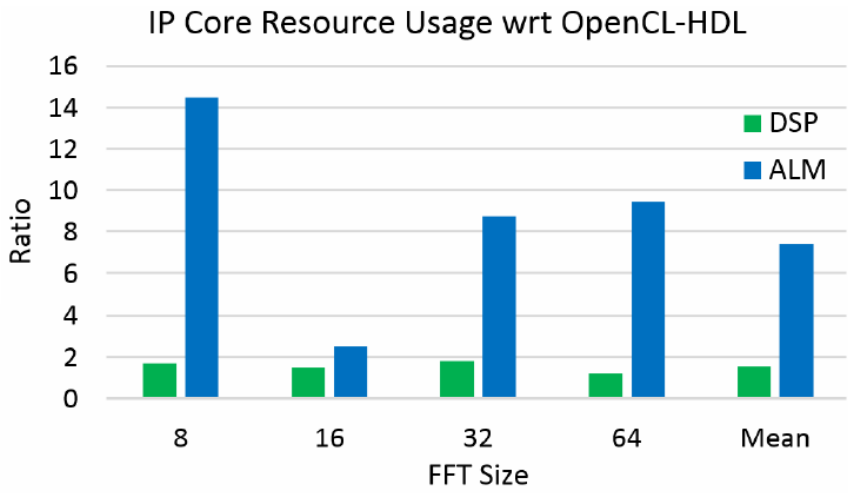
FFT Size	Latency (cycles)	ALM	DSP
8	20	1,849(<1%)	56(4%)
16	37	4,387(1%)	168(11%)
32	41	7,237(2%)	456(30%)
64	53	18,705(4%)	1160(76%)

TABLE II
LATENCY AND RESOURCE USAGE FOR IP-CORE 1D FFT

FFT Size	Latency (cycles)	ALM	DSP
8	16	26,759(6%)	96(6%)
16	32	11,132(3%)	256(17%)
32	64	63,322(15%)	832(55%)
64	128	176,285(41%)	1412(93%)

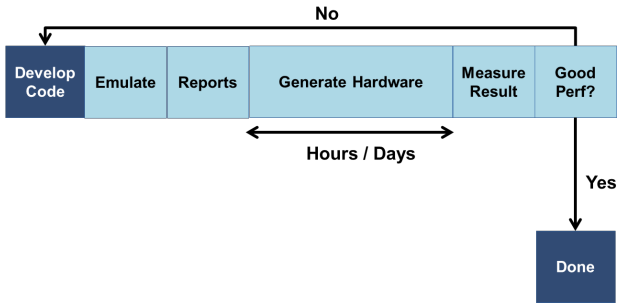
EXECUTION TIME (US) FOR 3D FFT IMPLEMENTATIONS

Design	16^3	32^3	64^3
CPU	22.0	55.0	288.0
GPU	20.7	23.6	43.1
IP Core	1.8	6.8	31.1
OpenCL-HDL	1.8	6.6	25.8



IP core resource usage with respect to OpenCL-HDL. OpenCL-HDL designs consume both fewer ALMs and DSPs.

Augmentation 3



- FPT18, HEART18
- HPEC18, HEART18
- H2RC18

Develop Code
Systematic Optimizations
Emulate
Reports
Gen HDL
Flow Type?
OpenCL-HDL
SimBSP
Isolate Pipelines
No
Done
Generate Hardware for Full OpenCL
Yes
Good Perf?
RTL Simulation
Re-interface
No
Done
Generate Hardware for custom system
Yes
Integrate?
Yes
Good Perf?
RTL Simulation
No

Emulation

- Used to simulate kernel code for functional verification.
- Compiling for emulation allows the compiler to
  - generate CPU equivalent code for FPGA-specific constructs
    - such as channels
  - execute the entire computation in software.
- This is useful for:
  - ensuring that computation and memory accesses have been correctly defined
  - identify run-time faults
    - such as occurrences of deadlocks.

Emulation

- Used to simulate kernel code for functional verification.
- Compiling for emulation allows the compiler to
  - generate CPU equivalent code for FPGA-specific constructs
    - such as channels
  - execute the entire computation in software
  - ensuring that computation and memory accesses have been correctly defined
  - identify run-time faults
    - such as occurrences of deadlocks.

Emulation does not provide any information regarding kernel code mapping to hardware or estimated performance

Reports

- Generated automatically during the initial compilation (C-HDL translation)
- Give the following information
  - **Loop analysis**
    - Used to determine initiation intervals (II) for loops in the kernel and the dependencies causing high IIs.
    - Resolving these dependencies allows loops to operate stall free.
  - **Area analysis**
    - Provides estimates of resource usage and implementation details for data structures.
    - This is particularly useful for determining if the compiler:
      - has correctly inferred the optimal hardware based on access patterns.
      - is resorting to sub-optimal, high-resource “safe” options such as memory replication and barrel shifters.
  - **System viewer**
    - Gives a graphical overview of the kernel computation and memory accesses.
    - Kernel execution is represented as sequential blocks, with each block carrying out a varying number of operations
      - such as memory transactions, channel calls and loop iterations.
    - Details provided include
      - latencies, stalls, types and sizes of Load-Store units created for each memory transaction,
      - the dependencies between blocks.
  - **Kernel memory viewer**
    - Gives a graphical overview of the connectivity of Load-Store units with external memory banks.
    - Can be used to verify that the compiler has correctly inferred off-chip access patterns.

.

Reports

- Generated automatically during the initial compilation (C-HDL translation)
- Gives the following information
  - **Loop analysis**
    - Used to determine initiation intervals (II) for loops in the kernel and the dependencies causing high IIs.
    - Resolving these dependencies allows loops to operate stall free.
  - **Area analysis**
    - Provides estimates of resource usage and implementation details for data structures.
    - This is particularly useful for determining if the compiler:
      - has correctly inferred the optimal hardware based on access patterns.

Kernel codes with no loop dependencies, initialization intervals equal to 1, efficient memories and low latencies can still be sub-optimal

    - Gives a graphical overview of the kernel computation and memory accesses.
    - Kernel execution is represented as sequential blocks, with each block carrying out a varying number of operations
      - such as memory transactions, channel calls and loop iterations.
    - Details provided include
      - latencies, stalls, types and sizes of Load-Store units created for each memory transaction,
      - the dependencies between blocks.
  - **Kernel memory viewer**
    - Gives a graphical overview of the connectivity of Load-Store units with external memory banks.
    - Can be used to verify that the compiler has correctly inferred off-chip access patterns.

.

What is a BSP?

- BSP = the files needed to wrap user specified kernel logic

i) compilation scripts

ii) board.qsys (the Shell)

iii) XML files which are used to tell the compiler, among other things, which scripts to pick up and execute

iv) some HDL files for "top" and "freeze_wrapper" modules.

To modify

i) For the XML file, we link to a main "TCL" compilation script (called "import_compile.tcl" in most BSPs) which ends up calling the rest. These compilation scripts are responsible for all operations after the C-to-HDL translation.

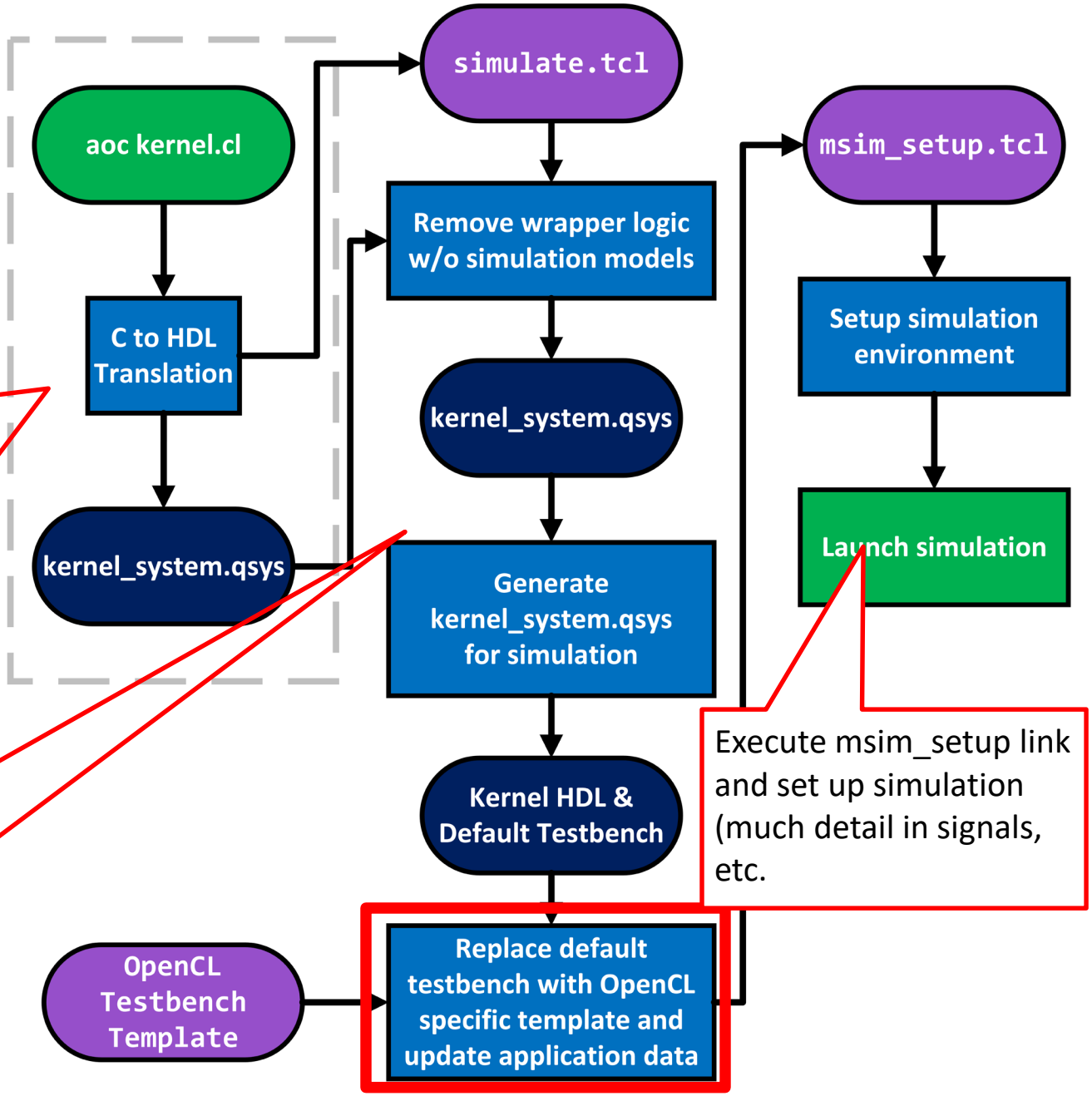
ii) The last script takes the bitstream (.sof file) and packages it into the OpenCL bitstream (.aocx file).

SimBSP

Very lightweight – no “board.qsys”, just

- XML files
- Compilation scripts
- Testbench template

SimBSP Compilation Flow



Done for all compilations.
We replace “freeze wrapper” with the testbench.
In the "board_spec.xml" file, we modify the "synthesize cmd" to link compilation to our custom "simulate.tcl" script.

Input HDL & QSYS files.
OpenCL compiler execs our simulate.tcl script and removes non-simulation models. Compile as usual.
Replace some files with custom ones (testbench)

Execute msim_setup link and set up simulation (much detail in signals, etc.

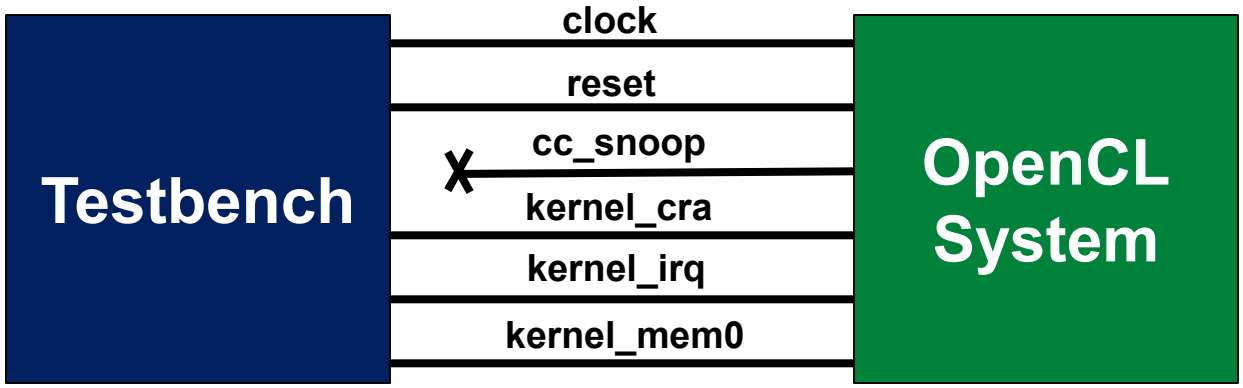
Matrix Multiply Kernel Code

```
#define SIZE 16
__kernel  void mmm(__global float* restrict a, __global float* restrict b, __global
float* restrict c){
   for (int i = 0; i < SIZE; i++){
           for (int j =0; j < SIZE; j++){
               float temp = 0;
               for (int k =0; k < SIZE; k++){
                     temp += (a[i*SIZE+k] * b[k*SIZE+j]);
               }
               c[i*SIZE+j] = temp;
           }
   }
}
```

```
E:\Research\PAPPA\kernel_file>aoc -v mmm.cl
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected default target board a10gx
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: Hardware generation completed successfully.
aoc: Warning: Cannot find a FPGA programming (.sof) file
```

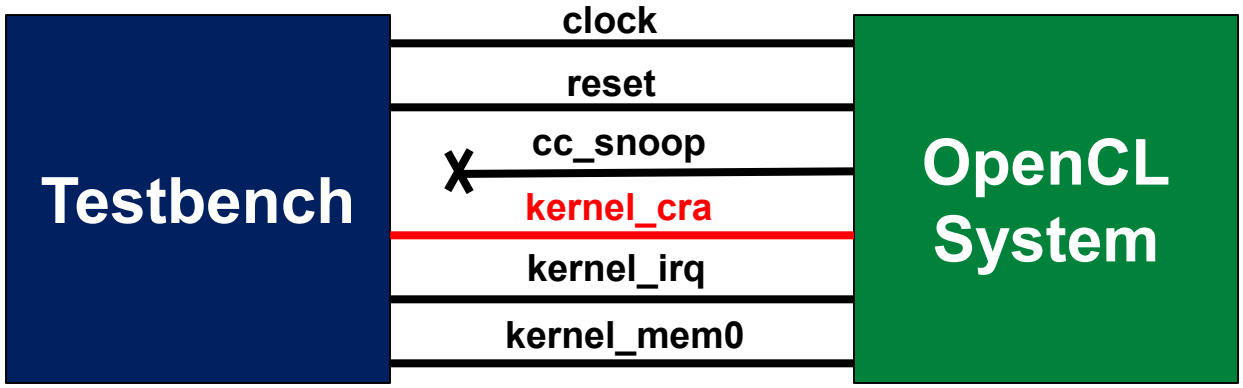
After linking to SimBSP, we compile the kernel as usual. Since the compilation forks after generating HDL, i.e. to set up the sim environment, no .sof file is generated.

Testbench: Interfaces



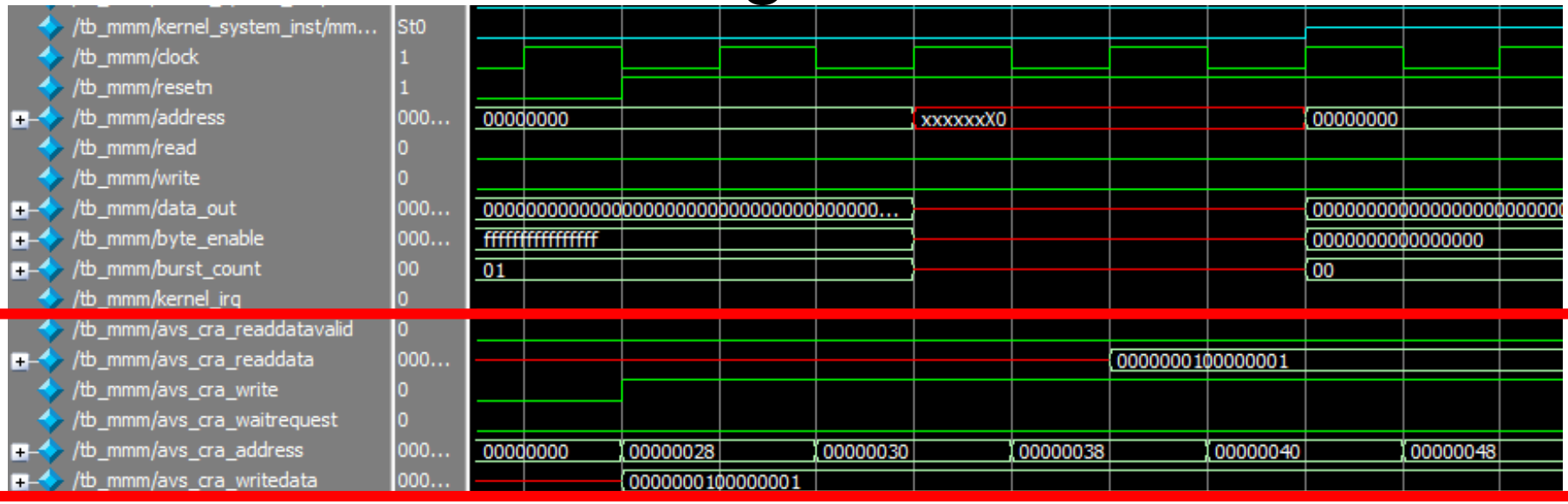
Name	Type	Interface Description
clock_clk	Clock	Kernel clock
clock_reset_n	Reset	Active low kernel reset
cc_snoop	Streaming	Not used
kernel_cra	Memory Mapped	Interface to configuration registers
kernel_irq	Interrupt	Interrupt to host machine
kernel_mem0	Memory Mapped	Interface to global memory

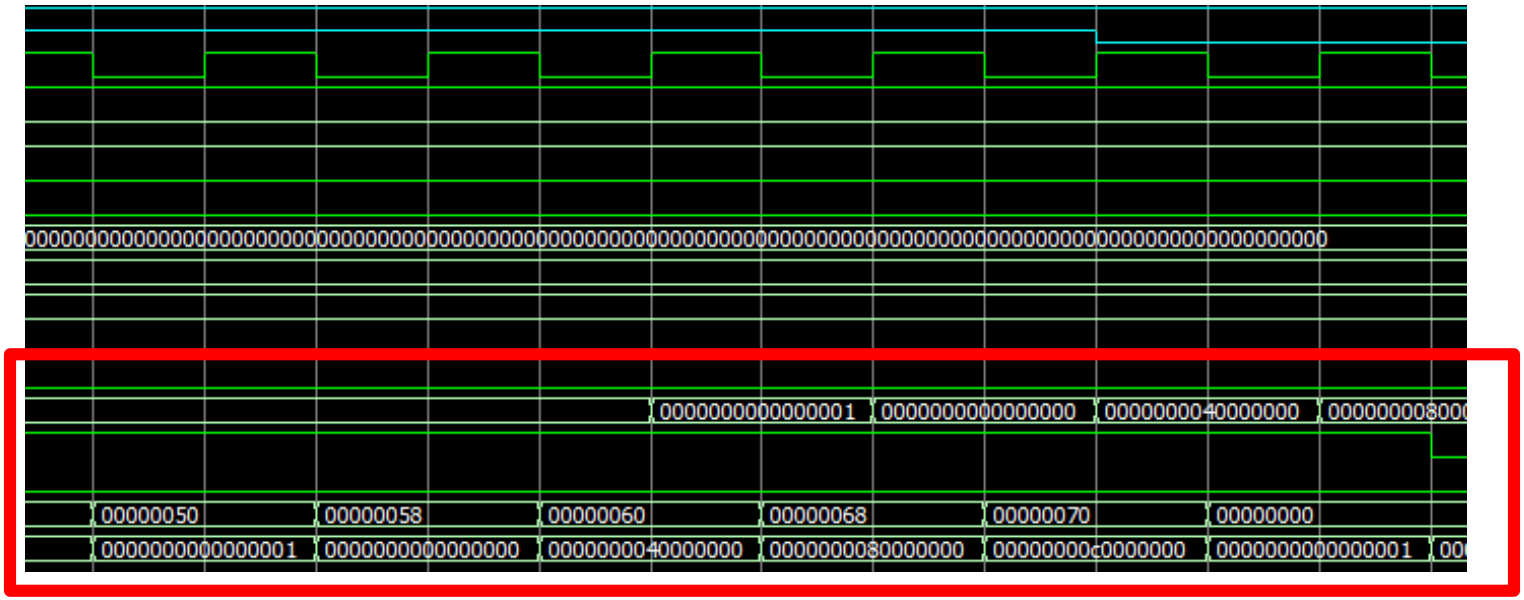
Testbench: Configuration Registers



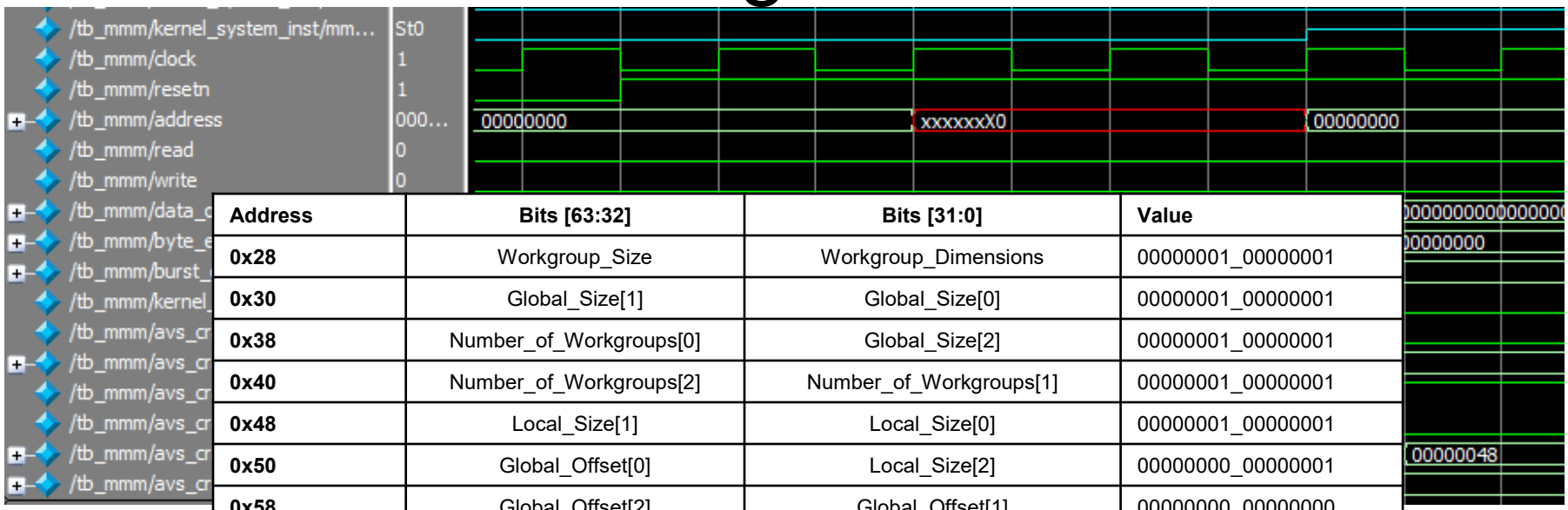
Address	Bits [63:32]	Bits [31:0]
0x0	-	Start (Bit 0)
0x28	Workgroup_Size	Workgroup_Dimensions
0x30	Global_Size[1]	Global_Size[0]
0x38	Number_of_Workgroups[0]	Global_Size[2]
0x40	Number_of_Workgroups[2]	Number_of_Workgroups[1]
0x48	Local_Size[1]	Local_Size[0]
0x50	Global_Offset[0]	Local_Size[2]
0x58	Global_Offset[2]	Global_Offset[1]
0x60 - end	Argument_Pointer[63:32]	Argument_Pointer[31:0]

Waveforms: Configuration

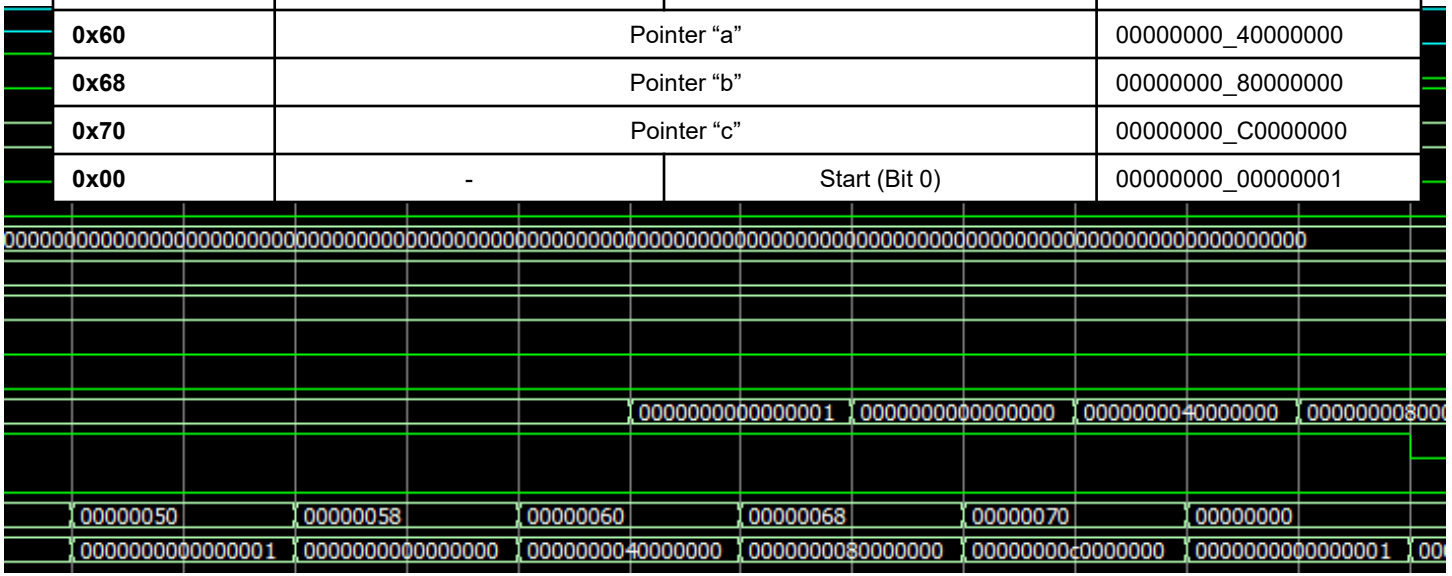




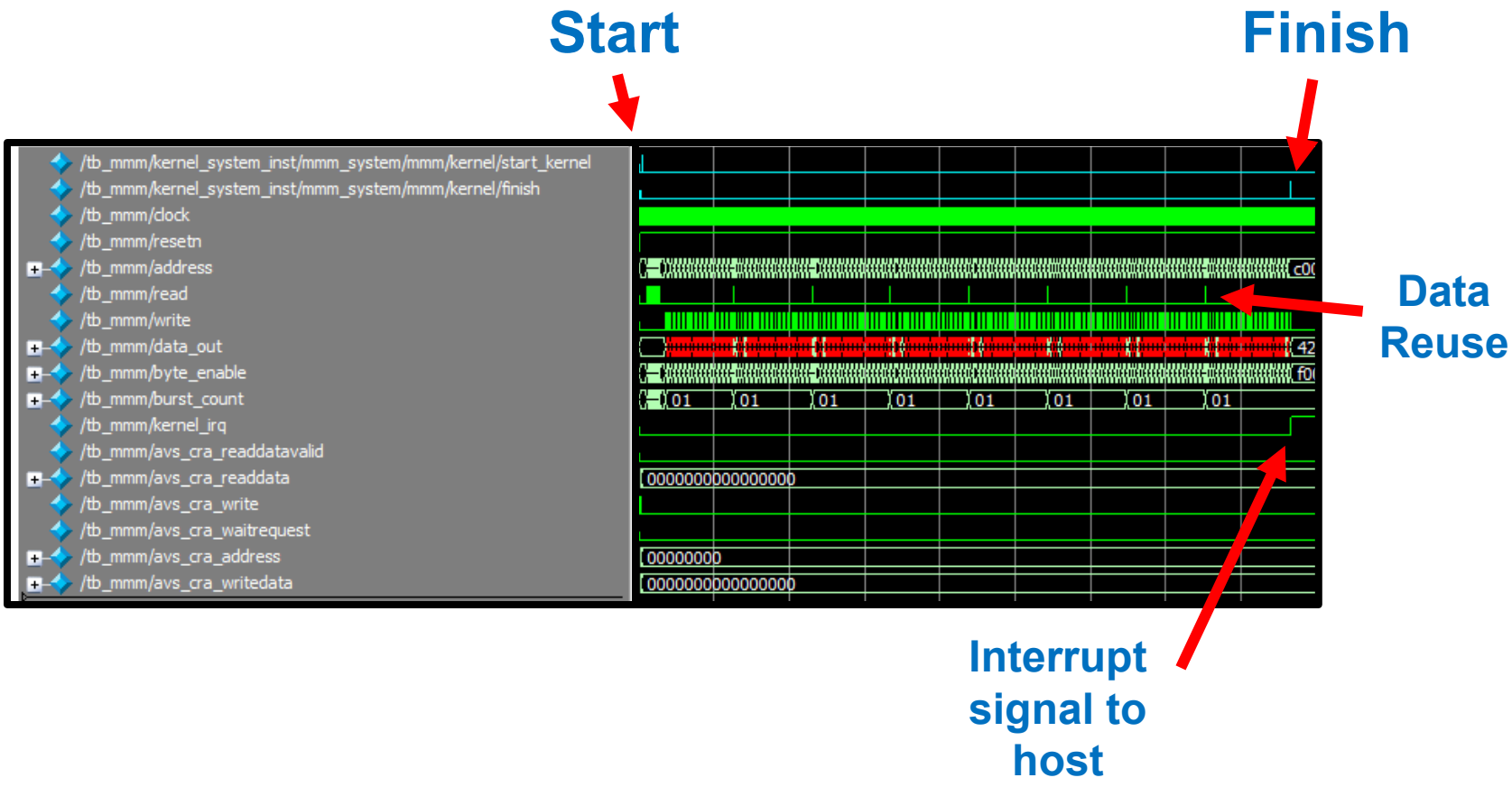
Waveforms: Configuration



Address	Bits [63:32]	Bits [31:0]	Value
0x28	Workgroup_Size	Workgroup_Dimensions	00000001_00000001
0x30	Global_Size[1]	Global_Size[0]	00000001_00000001
0x38	Number_of_Workgroups[0]	Global_Size[2]	00000001_00000001
0x40	Number_of_Workgroups[2]	Number_of_Workgroups[1]	00000001_00000001
0x48	Local_Size[1]	Local_Size[0]	00000001_00000001
0x50	Global_Offset[0]	Local_Size[2]	00000000_00000001
0x58	Global_Offset[2]	Global_Offset[1]	00000000_00000000
0x60	Pointer “a”		00000000_40000000
0x68	Pointer “b”		00000000_80000000
0x70	Pointer “c”		00000000_C0000000
0x00	-	Start (Bit 0)	00000000_00000001



Waveforms: Kernel Execution



THANK YOU

Programming Model?

This is an issue because OpenCL is not sufficiently expressive to allow programmer to explicitly express all Intel FPGA capabilities. Several alternatives:

Choice 1: Program Intel FPGA as if it were a GPU

Rationale: Since OpenCL maps well to GPUs, and the GPU model is a subset of the Intel FPGA model, and GPU code is efficient for many applications, this should be reasonable, if not perfect.

Problem: Gives very poor performance

Choice 2: Program Intel FPGA as if it were an Intel® FPGA

Rationale: Although support is a subset of HDL support, there are Intel® FPGA-specific extensions, including channels and embedded HDL.

Problem: Gives very poor performance

Choice 3: Program Intel FPGA as if were a single threaded CPU

This means: Single work item/group AND single work group

Rationale: Huh?

Rationale: The Intel OpenCL compiler is REALLY GOOD – turn it loose!

Problems (solved here): Long compile times, need systematic optimizations

Optimizing OpenCL Kernels

- V1: Cache Optimized CPU Code
  - CPU architecture resembles FPGAs more than GPUs
- V2: Typical FPGA Optimizations / Recommended Best Practices
  - SWI kernels, Channels, Constants, Loop unrolling, Coalesced loops
- V3: Single Kernel Implementation
  - Remove channels
    - Channels isolate resources and prevent global optimizations
    - Kernels are launched sequentially, potentially resulting in race conditions, deadlocks and high synchronization costs
  - Instead, let Intel OpenCL compiler infer task parallelism within a single kernel
    - Use code fragments that are tied to the same loop iterator but have no data dependencies
    - Intel OpenCL compiler utilizes delay modules instead of blocking channels for synchronization of data paths which is more efficient

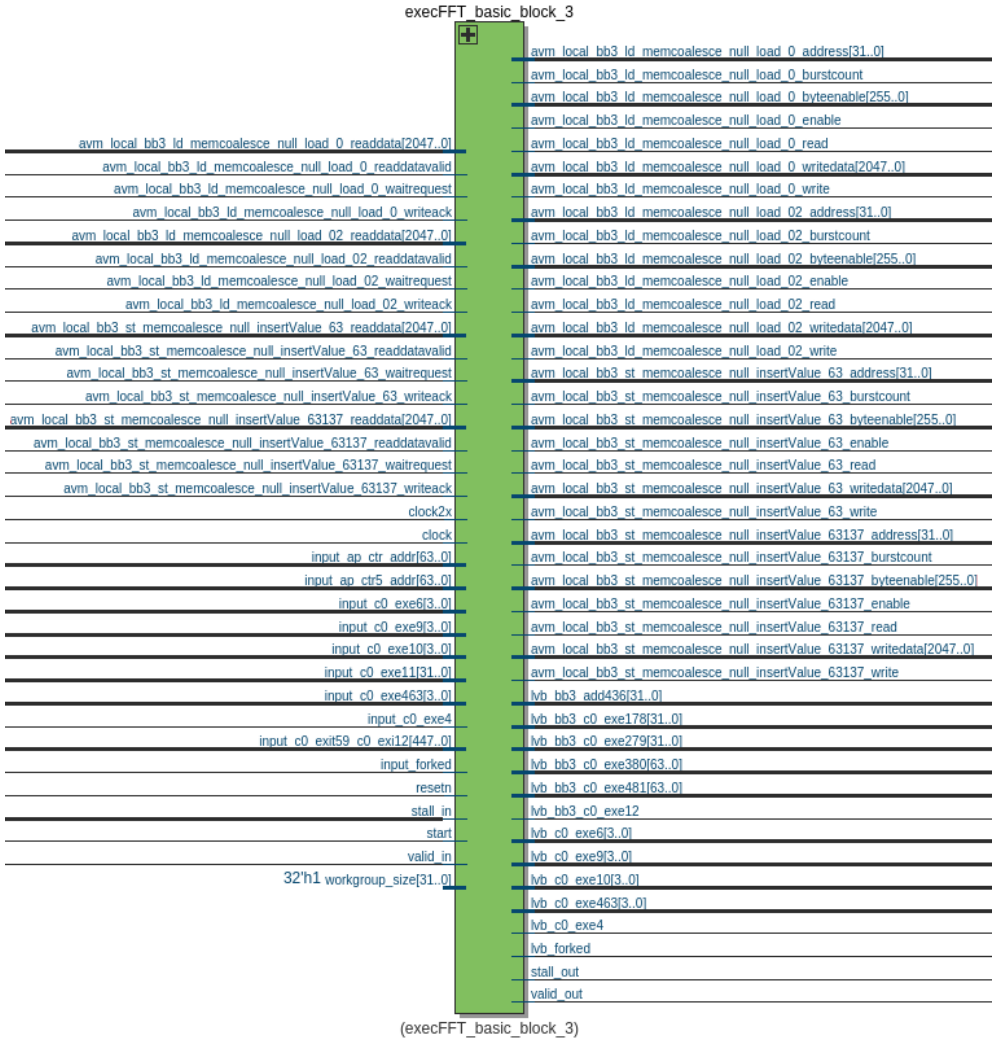
Optimizing OpenCL Kernels, cont.

- V4: Infer Pipeline Registers as Registers
  - Large arrays can be inferred as BRAMs instead of registers
    - Rapidly increases resource usage due to memory replication to meet throughput
  - Avoid using large variable arrays where possible
    - Use scripts to generate and use individual variables
- V5: Explicit computations
  - Provide as much detail as possible regarding the computation
    - Even if it is an apparent suboptimal practice such as
      - Un-coalescing loops to help infer access patterns
      - Small constant arrays instead of single large one
      - Intermediate variables for complex computations
      - Manually unrolling a loop and specifying tree based computations using parenthesis
      - Extra kernel parameters that are pointers to the same memory space
        - One per task
        - manually guarantee no RAW data dependencies between tasks
  - Helps the Intel OpenCL compiler infer the desired architecture with greater accuracy

V6: Avoid conditional statements e.g. if, else

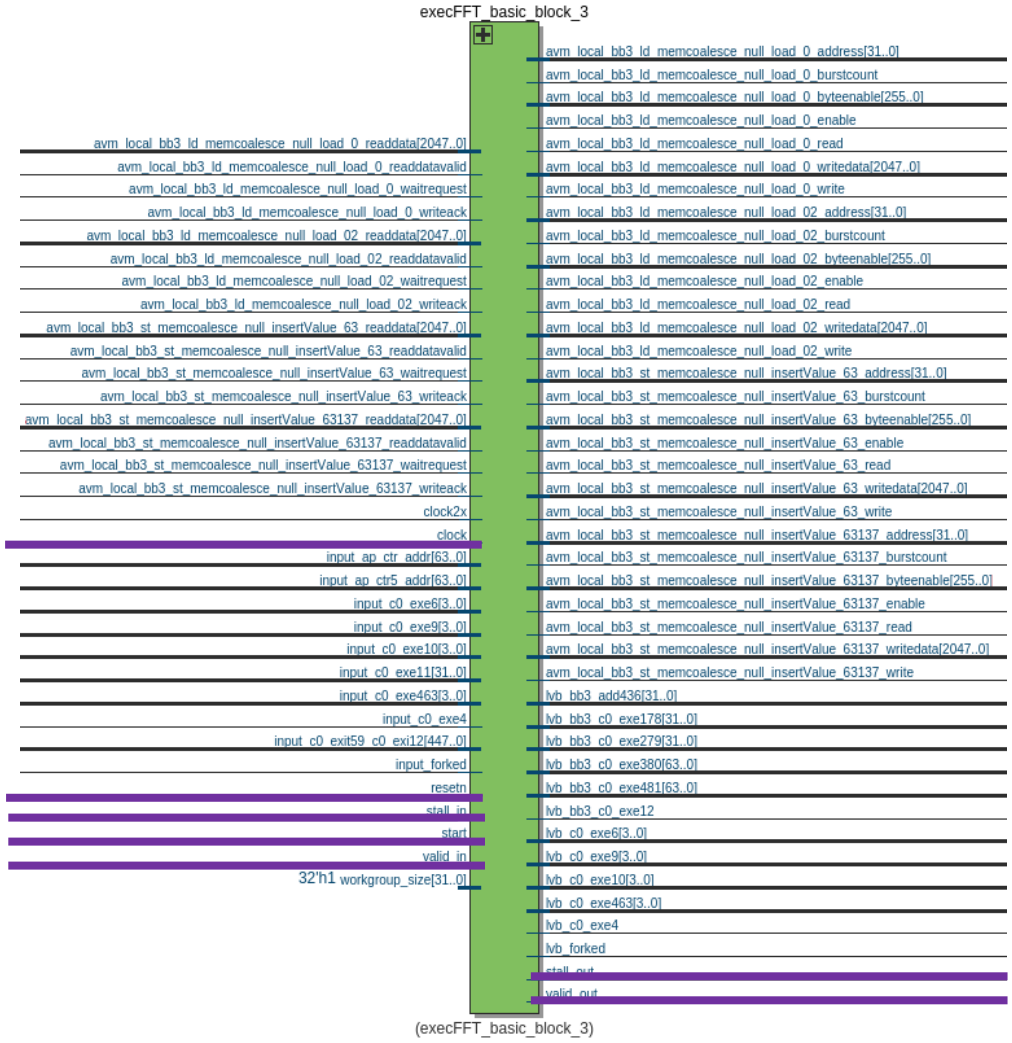
- Use conditional assignments instead

Creating a New Interface



Creating a New Interface

— Control Ports

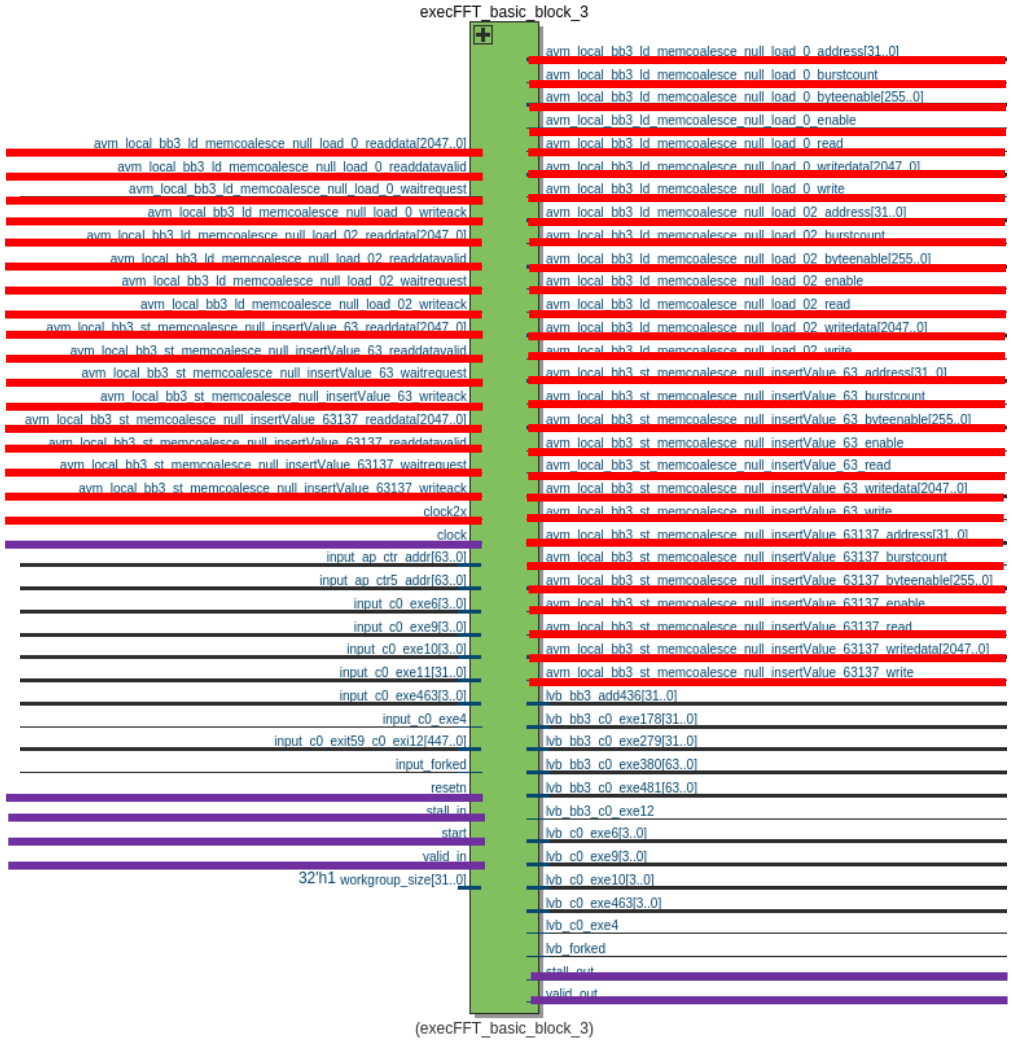


Control Ports

- Consists of clock, resetn, Stall and Valid ports
- Stall ports used by a basic block to stall upstream modules
- Valid ports used to stall downstream basic blocks

Creating a New Interface

Control Ports
Load Store Unit Ports

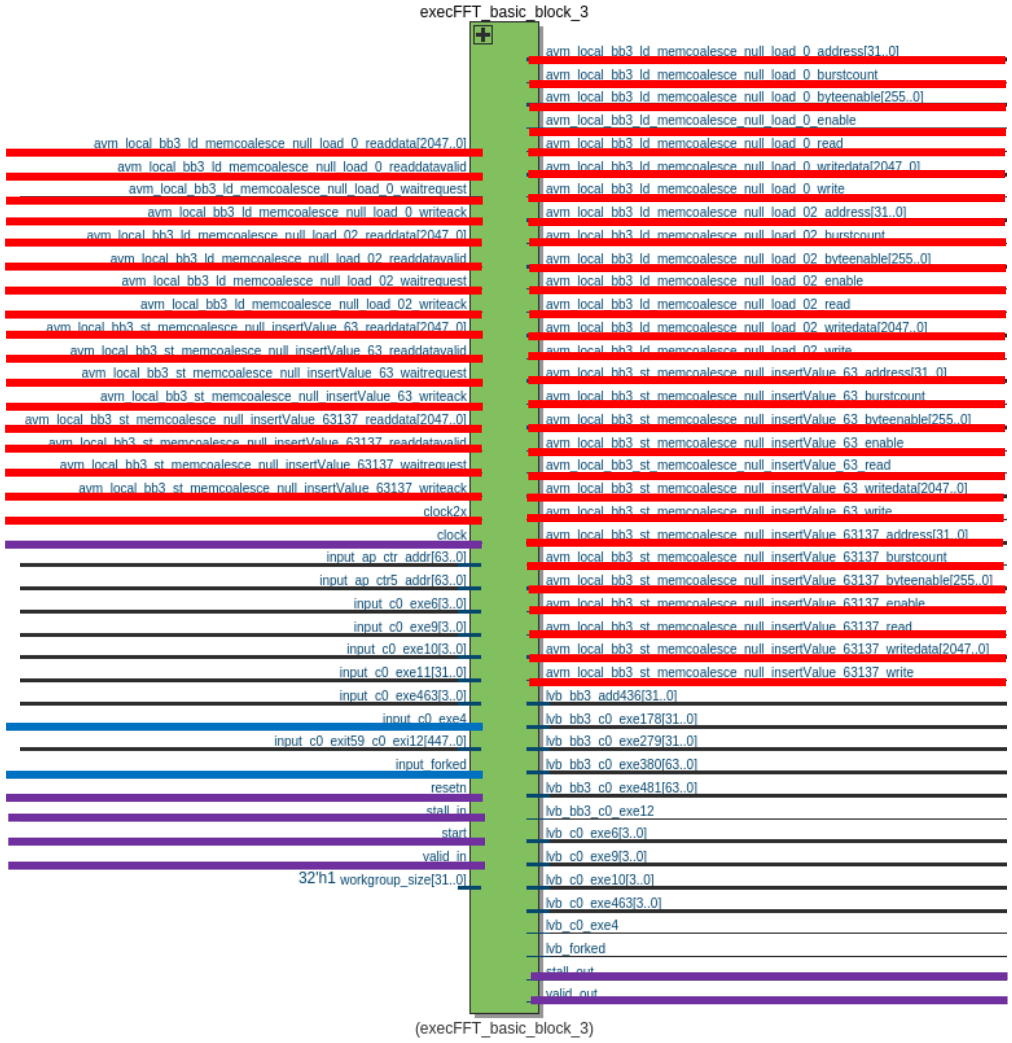


Load Store Unit (LSU) Ports

- Consists of Avalon interfaces to LSU modules
- LSU modules sink and source data to compute pipelines
- Remove LSU modules and interface pipelines with required data-buses

Creating a New Interface

- Control Ports
- Load Store Unit Ports
- Feedback Ports

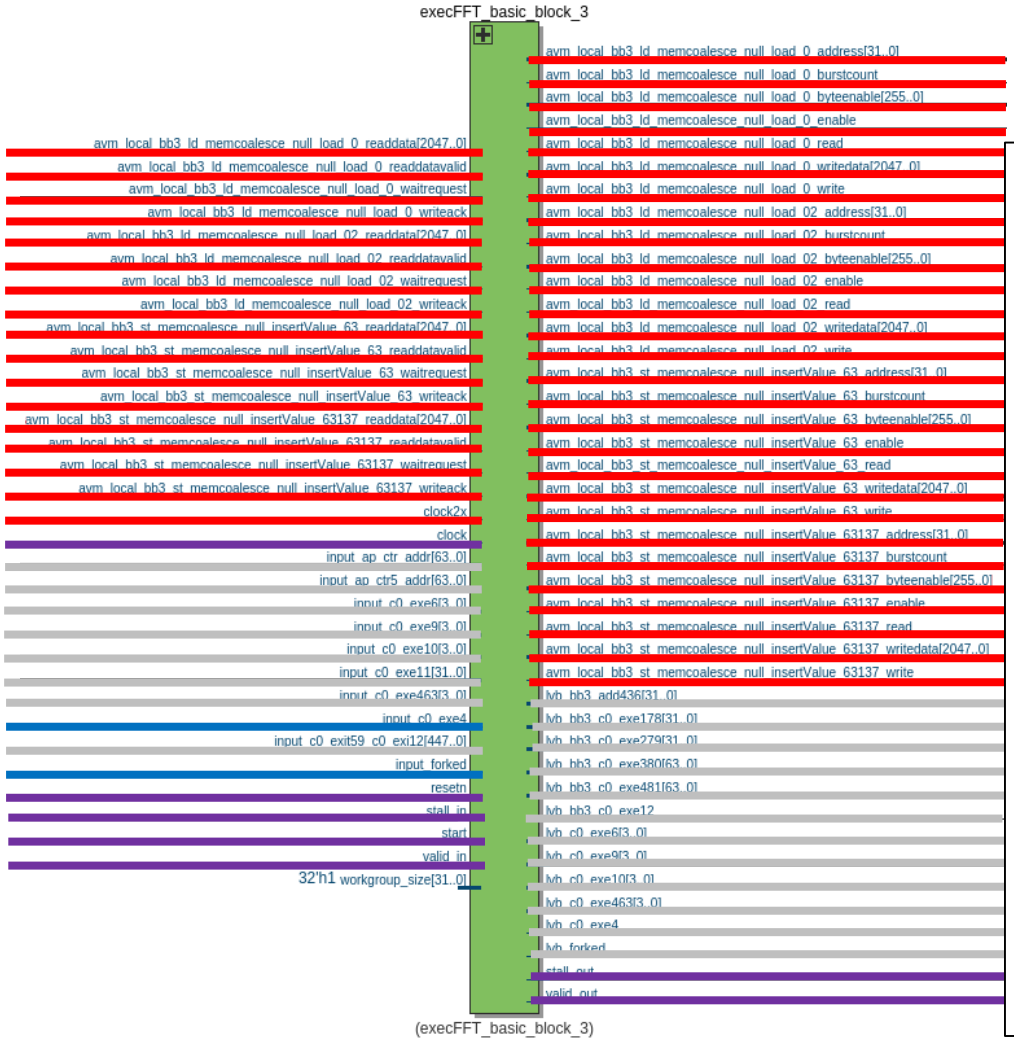


Feedback Ports

- Used to select between initial and steady state values of state registers
- Hardwired – User cannot modify at run time
- Paired output-input ports connect to each other while individual inputs have a constant value

Creating a New Interface

- Control Ports
- Load Store Unit Ports
- Feedback Ports
- Don't Care Ports



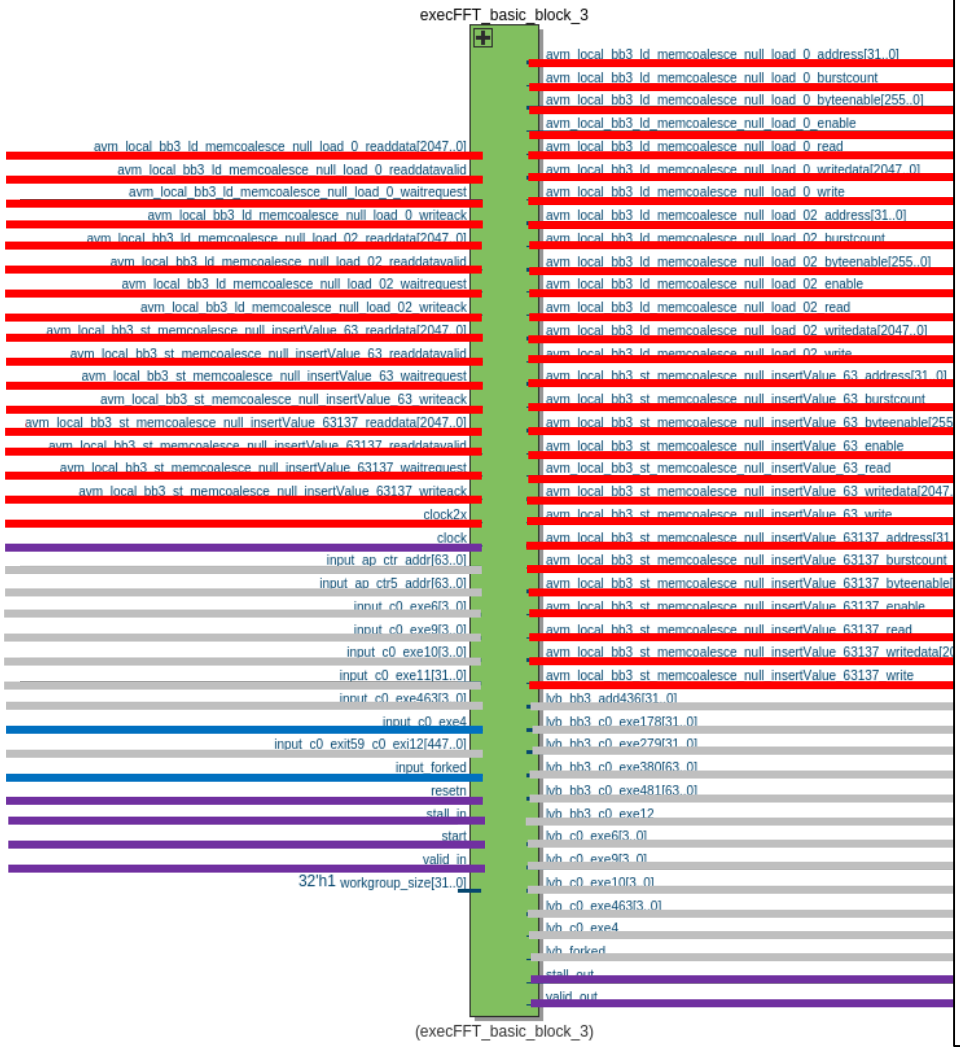
Don't Care Ports

- Typically correspond to a variety of logic such as
  - Parallel control and data paths that do not interact with compute pipelines
  - Logic that interfaces load store unit (LSU) modules e.g. address computation
- Since we remove LSU modules, and since there is no interaction with compute pipelines, we can leave these pins unconnected

Creating a New Interface

Control Ports

Load Store Unit Ports



Direct Ports

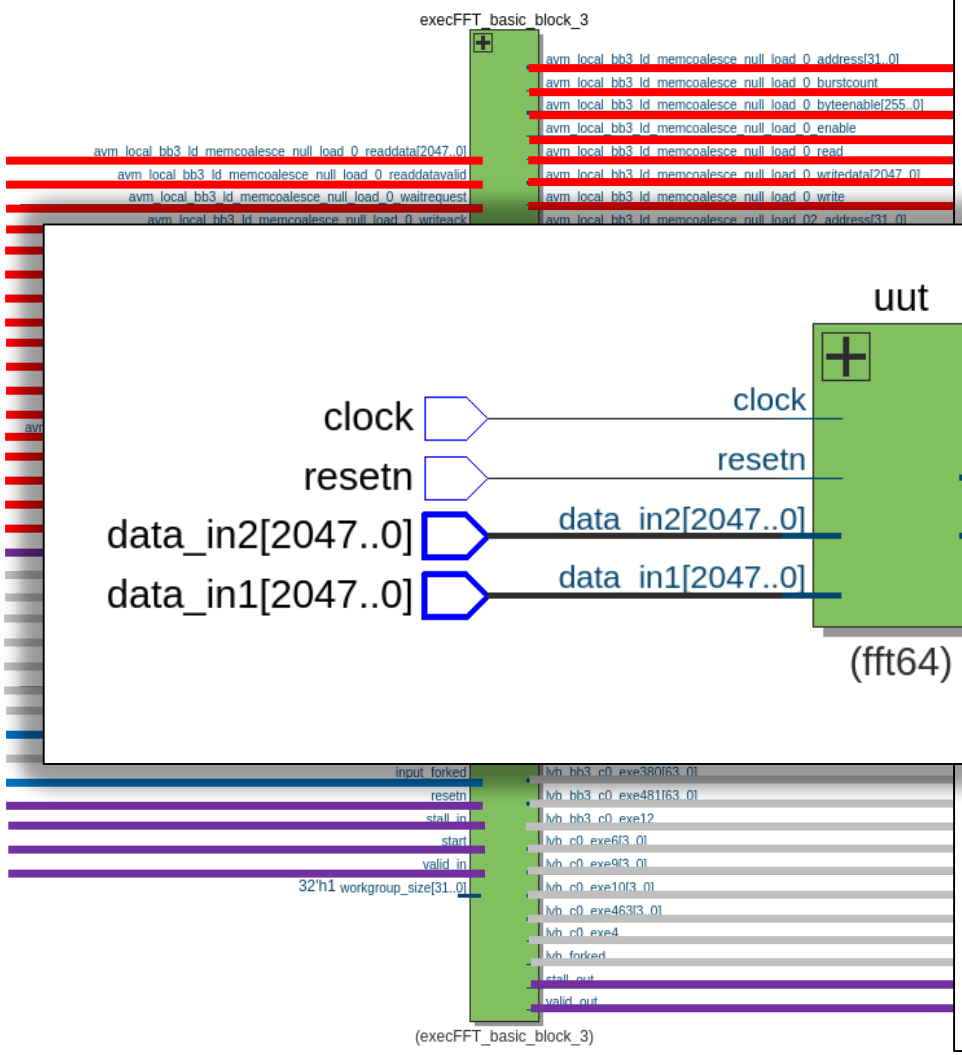
(not shown)

- Direct Kernel Input : if the kernel has a constant input. Appears as a N bit input to the kernel based on variable type
- Direct Data : Data is available as a direct bus. Typically occurs when compiler moves LSU unit out of the basic block due to:
  - outer loop variable not used in index/address computations
  - Problem size is too small
- Initial Values

Creating a New Interface

Control Ports

Load Store Unit Ports



Direct Ports

(not shown)

- Direct Kernel Input : if the kernel has a constant input. Appears as a

index/address computations

- Problem size is too small

- Initial Values