FPGA Development Flow with Intel® oneAPI

oneAPI – 가속 컴퓨팅을 개발하기 위한 스마트한 방식

2022. 04. 15.

MOASYS

oneAPI Smart Development Series (2021)

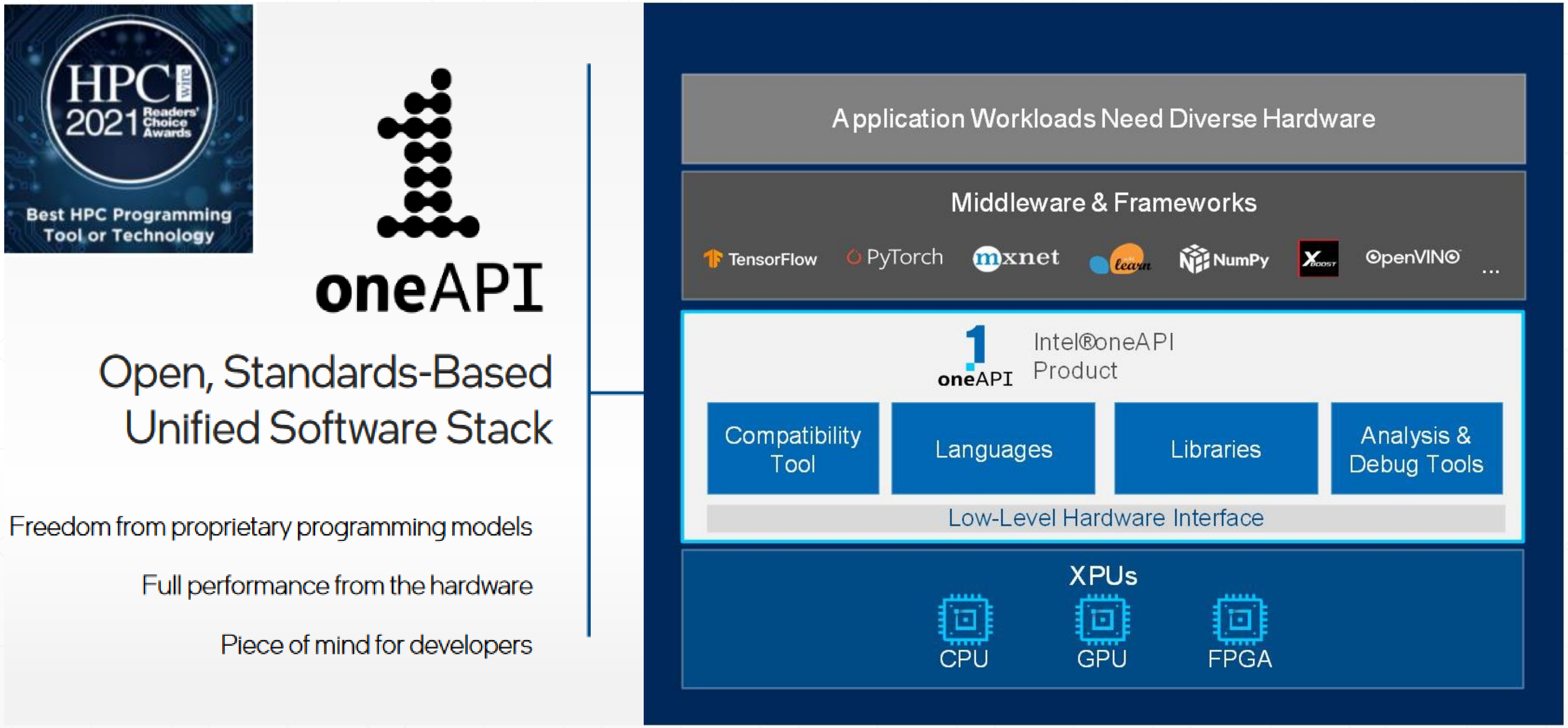
1. Introduction to Intel oneAPI for HPC and AI-DL
   - https://www.allshowtv.com/detail.html?idx=474
2. Benchmarking the Performance of oneAPI on Heterogeneous Computing Platforms
   - https://www.allshowtv.com/detail.html?idx=660
3. Optimization and GPU Offloading Workflow with Intel oneAPI
   - https://www.allshowtv.com/detail.html?idx=826
4. Leveraging Intel® oneDNN for AI Workload
   - https://www.allshowtv.com/detail.html?idx=909

oneAPI Smart Development Series (2022)

1. FPGA Development Flow with Intel® oneAPI Base Toolkit
   - https://www.allshowtv.com/detail.html?idx=995
2. OpenMP Offload with Intel® oneAPI HPC Toolkit
   - TBA
3. Essential DPC++ Optimization Techniques for Accelerators
   - TBA
4. Introduction to Intel® oneAPI Rendering Toolkit
   - TBA

Contents

Overview of oneAPI for Heterogenous Computing


HPC wire 2021 Readers' Choice Awards
Best HPC Programming Tool or Technology
oneAPI
Open, Standards-Based Unified Software Stack
Freedom from proprietary programming models
Full performance from the hardware
Piece of mind for developers
Application Workloads Need Diverse Hardware
Middleware & Frameworks
TensorFlow
PyTorch
mxnet
learn
NumPy
XGBoost
OpenVINO
...
oneAPI
Intel® oneAPI Product
Compatibility Tool
Languages
Libraries
Analysis & Debug Tools
Low-Level Hardware Interface
XPUs
CPU
GPU
FPGA

- Support diverse accelerator devices (XPU) such as CPU, GPU and FPGA
- Continuously evolving specifications for high performance computing and machine learning

oneAPI Industry Initiative Progress

2019 | 2020 | 2021 Q1 | Q2 | Q3 | Q4

Industry Initiative

- 2019: **Industry Initiative Announced** — 0.5 oneAPI Spec
- 2020: **oneAPI 1.0 Spec.**
- 2021 Q1: oneAPI AI Tech Advisory Board setup
- 2021 Q1: **oneAPI 1.1 Provisional Spec.** Level Zero, oneDNN Graph, Advanced Ray Tracing, oneVPL
- 2021 Q2: **oneAPI Prov Spec.**
- 2021 Q3: **oneAPI Prov Spec.**
- 2021 Q4: **oneAPI 1.1 Spec.**

Industry Adoption

- CodePlay delivers oneAPI support for Nvidia GPUs, + oneMKL support
- Fujitsu & Riken: oneDNN open source on ARM for Fugaku supercomputer
- Univ. Heidelberg delivers DPC++ AMD GPU support
- NERSC, ALCF, Codeplay to enhance LLVM SYCL GPU compiler for Nvidia GPUs
- Huawei extends DPC++ support for Ascent AI chipset
- Argonne, OLCF to provide SYCL support for AMD GPUs
- Bittware FPGA deploys oneAPI training
- Sbercloud launches with oneAPI for AI development

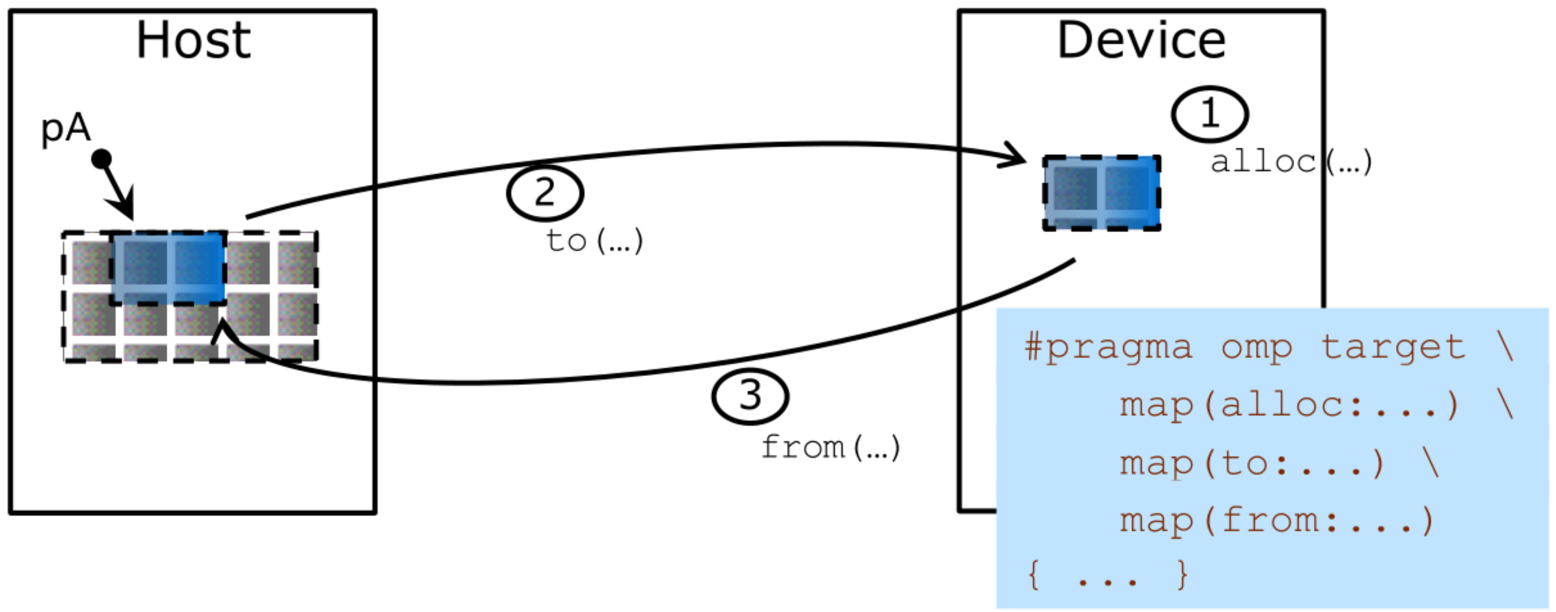
Aurora Blade
Building Block for the ExaScale Supercomputer
oneAPI
Argonne
ENERGY
Hewlett Packard Enterprise
intel

> 2 Exaflops (2022)

oneAPI Developer Summits + Geo-targeted sessions
IWOCL ■ Int'l SuperComputing ■ Intel Innovation ■ SuperComputing

- **Cross-Vendor implementations:**
  - ARM CPU: Fujitsu & Riken (Japan)
  - NVIDIA GPU: CodePlay (USA), NERSC (USA), Argonne National Lab (USA)
  - AMD GPU: Heidelberg Computing Center (Germany), Argonne National Lab (USA), Oak Ridge National Lab(USA)

Preview: OpenMP Offload Capabilities in oneAPI HPC Toolkit

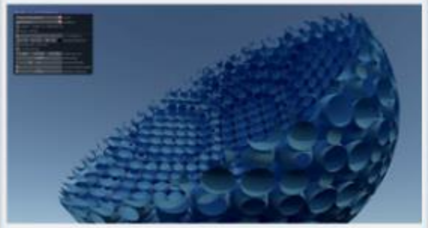
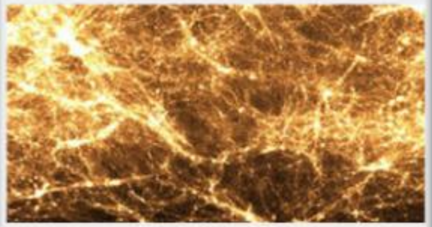


```
#pragma omp target \
    map(alloc:...) \
    map(to:...) \
    map(from:...)
{ ... }
```

- OpenMP: Portable, performance and productive parallel programming interface
- New compiler technology based on LLVM (DPCPP, ICX and IFX)
  - Built-in support for X^{e} GPUs
  - Mixing host and GPU parallelism for maximum efficiency
  - Support for Unified Share Memory (USM)
  - Support for offloading oneMKL routines to GPUs

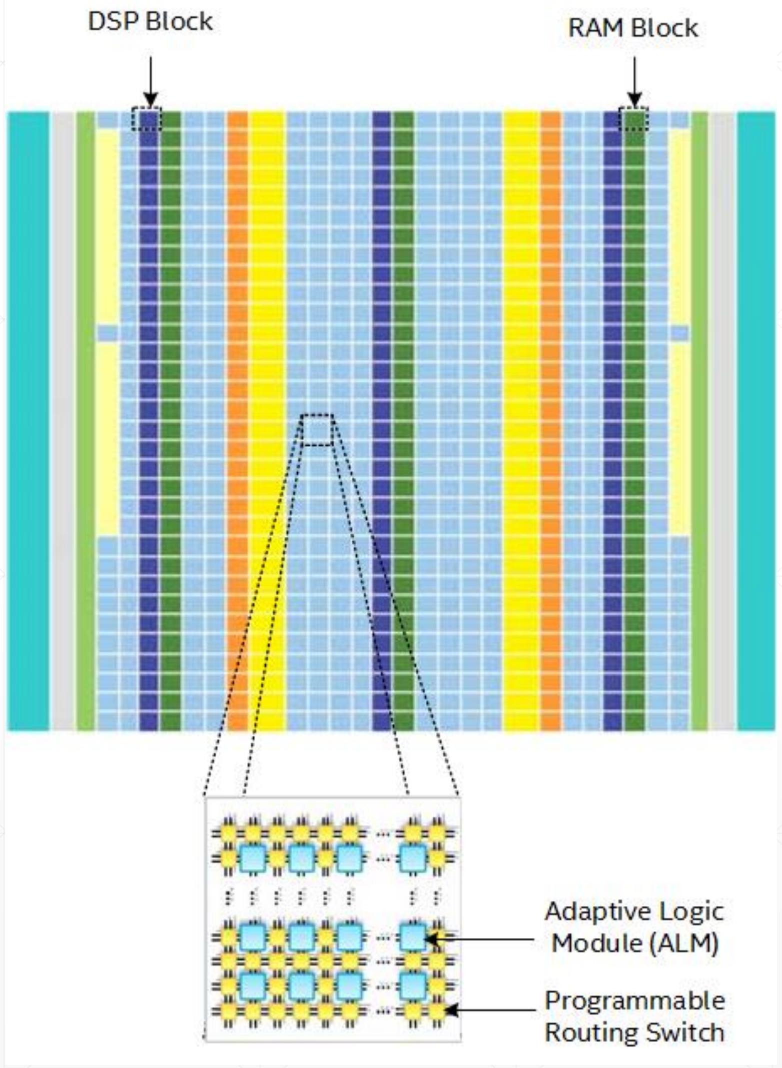
Preview: Introduction to oneAPI Rendering Toolkit

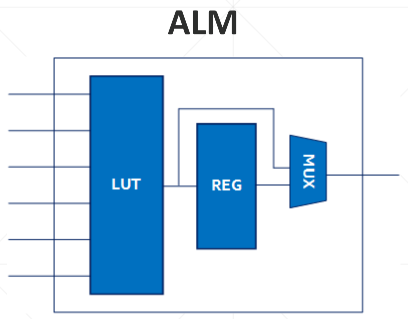
Intel® oneAPI Rendering Toolkit Libraries

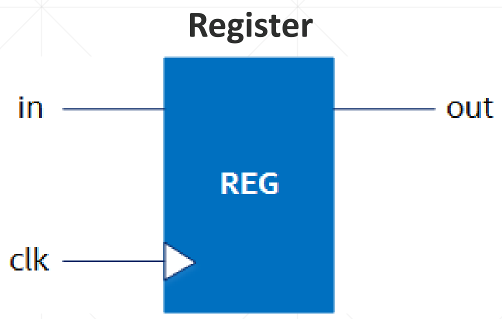
Intel® Embree	Intel® OSPRay	Intel® Open Image Denoise	Intel® Open Volume Kernel Library	Intel® OpenSWR
High-Performance, Feature-Rich Ray Tracing & Photorealistic Rendering	Scalable, Portable, Distributed Rendering API	AI-Accelerated Denoiser for Superior Visual Quality	Render & Simulate 3D Spatial Data Processing	High-Performance, Scalable, OpenGL*-Compatible Rasterizer
				

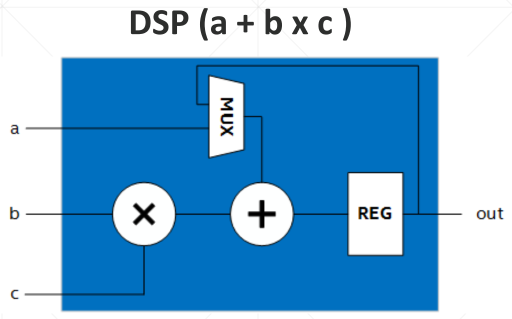
- Intel leadership in ray tracing for high performance graphics and compute
- Run at scale, from laptop to workstation and enterprise HPC cluster
  - Hybrid rendering on Intel CPUs and X^e GPUs
  - Visualize huge dataset interactively
  - Persistent memory support for reducing restart times and improving I/O performance
  - Combined volume and geometric rendering
  - High-fidelity visualization including shadows, ambient occlusion, global illumination, motion blur

FPGA Architecture Overview



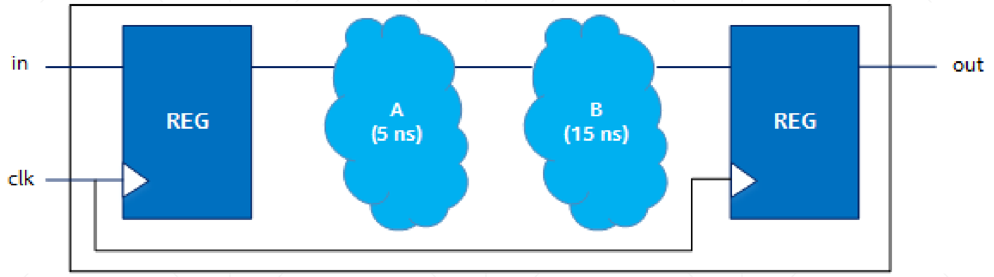




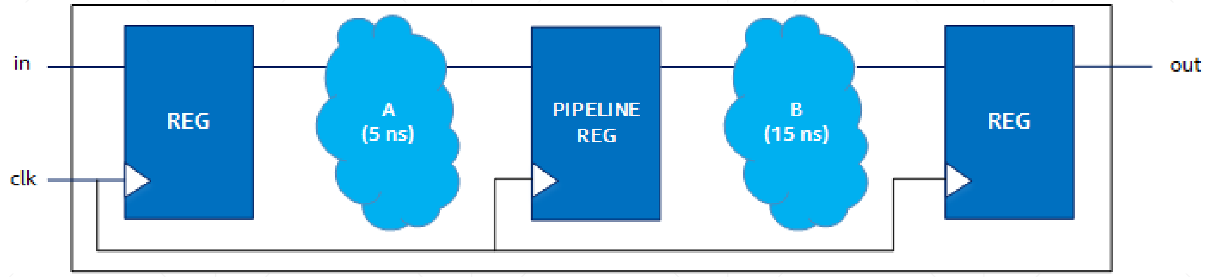


- **FPGA (Field-Programmable Gate Array)**
  - Reconfigurable semiconductor integrated circuit (IC).
  - Adaptive logic module (ALM):
    - Basic building block of FPGA
    - Look-up table (LUT) and output register to build a logic circuit
  - Register:
    - Basic storage element of FPGA: input, output and clock signal (clk)
    - Output is synchronized *every* clock cycle
  - Digital processing block (DSP):
    - Support for common fixed-point and floating-point arithmetic

FPGA Design Concept: Basics



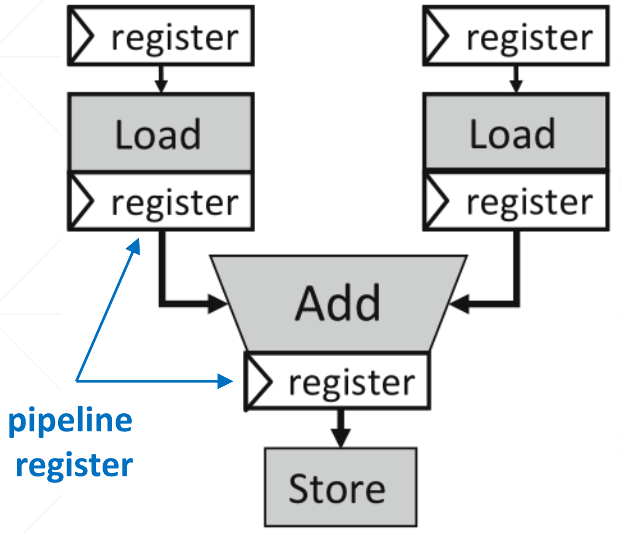
Delay: 20 ns / Latency: 2 Clks / Fmax: 50 Mhz



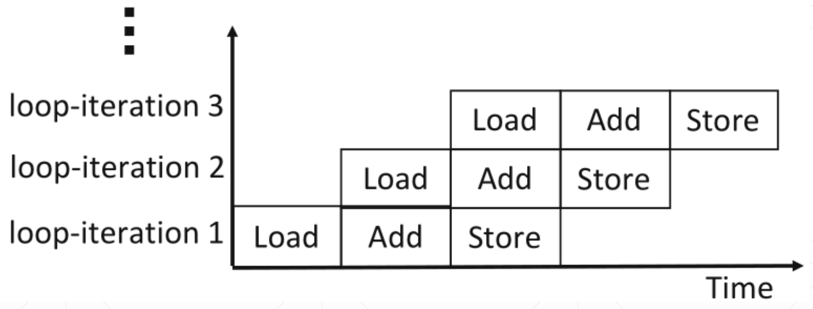
Delay: 15 ns / Latency: 3 Clks / Fmax: 66.6 Mhz

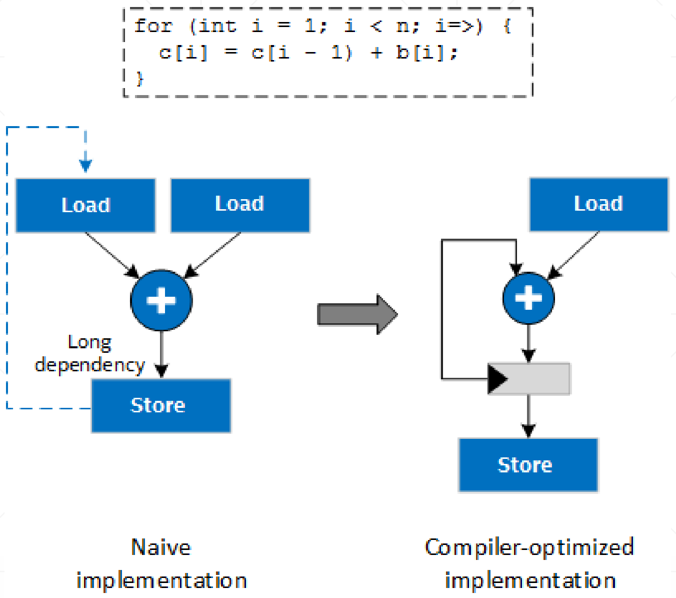
- **Critical Path**
  - The path between any two consecutive registers with the highest delay
- **Maximum frequency (f_{max})**
  - The maximum rate of output registry update, defined as 1/crital_path_delay
- **Latency**
  - How many clock cycles to complete operations in a digital circuit
- **Pipeline**
  - Adding registers to the critical path, which decreases the amount of logic between each register
  - Increasement in f_{max} but also in latency
- **Datapath**
  - A chain of registers and combinational logic in a digital circuit that performs computations.
  - Input Reg -> A -> Pipeline Reg -> B -> Output Reg

FPGA Design Concept: Pipeline Parallelism with Single-Work Item Kernel



```
q.submit([&](handler& h) {
    h.single_task([=](){
        for (int i = 0; i < kSize; ++i) {
            r[i] = a[i] + b[i]
        }
    });
});
```

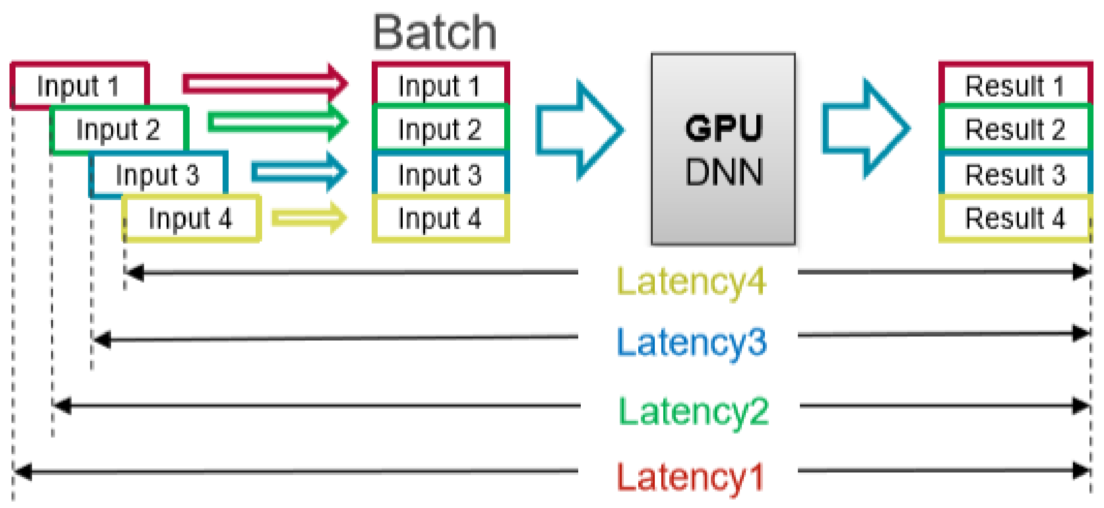


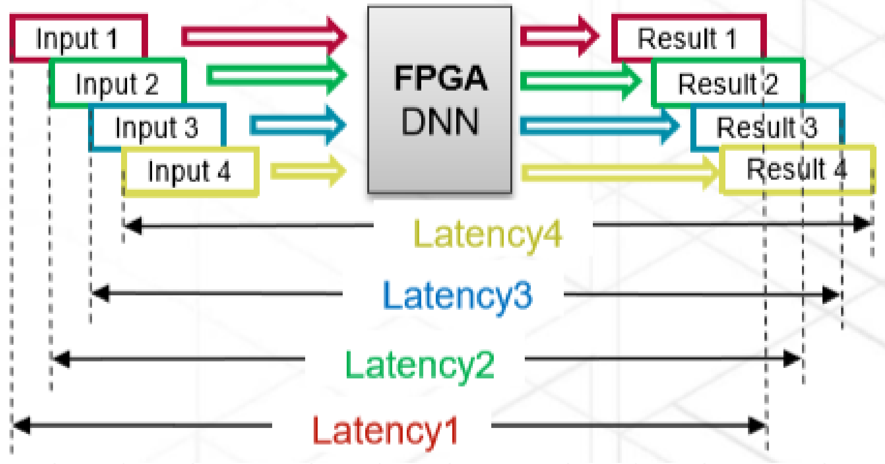


```
for (int i = 1; i < n; i=>) {
  c[i] = c[i - 1) + b[i];
}
```

- Advantages of using single-work item kernel over ND-range kernel
  - Loop iteration as basic unit of execution, identical to standard C/C++ code
  - Automatic creation of pipeline register by DPCPP compiler
  - Automatic dependencies resolution by DPCPP compiler
    - c[i - 1] stored in *pipeline register* and feedback loop is automatically created
- Initiation Interval (II):
  - The number of clock cycles between the launch of successive loop iterations.
  - For ideal pipeline with high throughput: II = 1

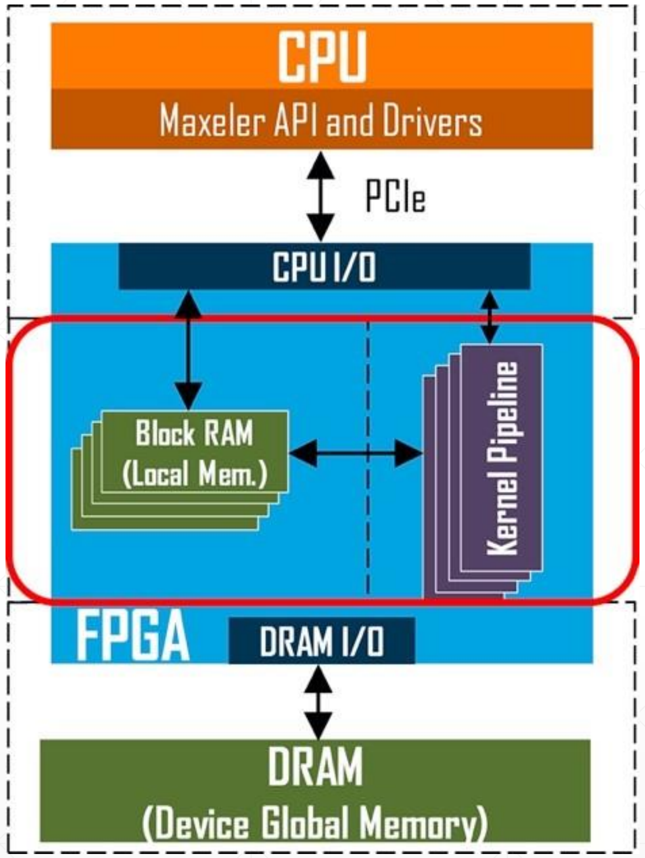
FPGA vs. GPU: AI Inference for Autonomous Driving System

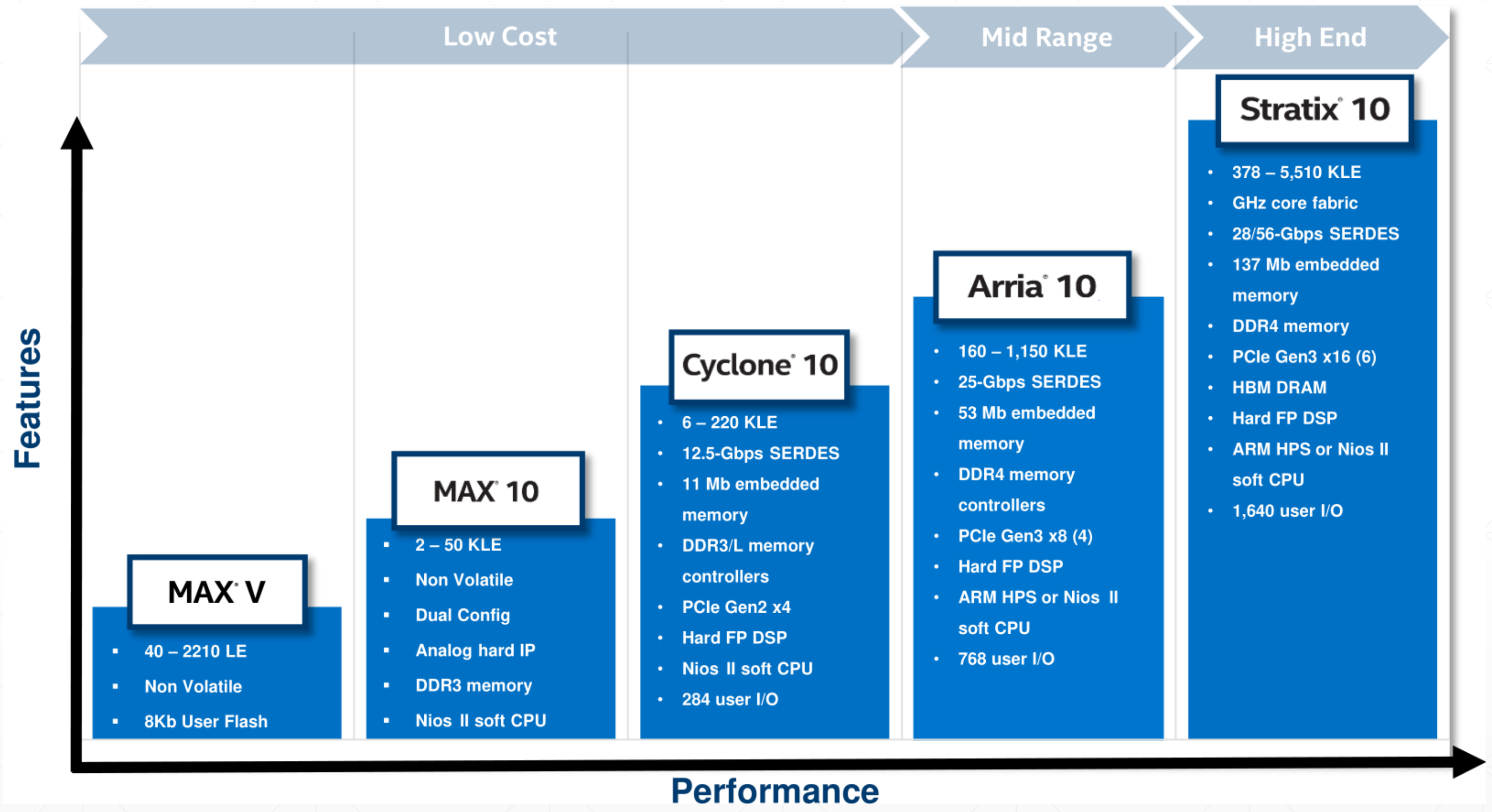




- **GPU disadvantages:**
  - Batch inference requires addition synchronization
  - Large batch size -> higher throughput and latency
  - Small batch size -> lower throughput and latency
- **FPGA advantages:**
  - Batch-less inference through pipeline parallelism (first in first out)
  - Consistent and predictable latency and high throughput
- Train on GPU and inference on FPGA for autonomous driving application

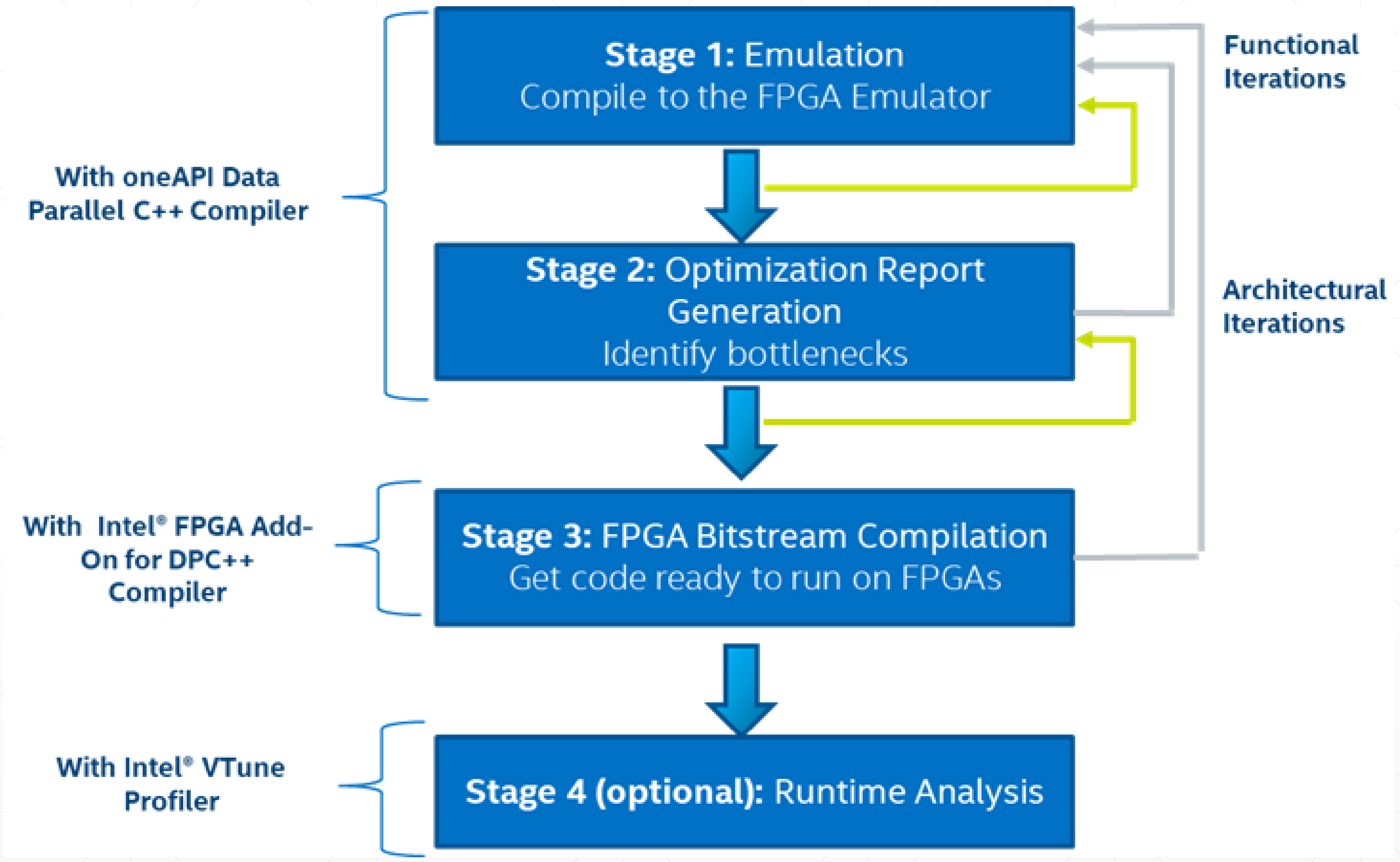
Intel FPGA Accelerators





- CPU offloads complex computation tasks to FPGA connected via PCI express
- Targeted workload: streaming analytics, fintech, genomics, artificial intelligence

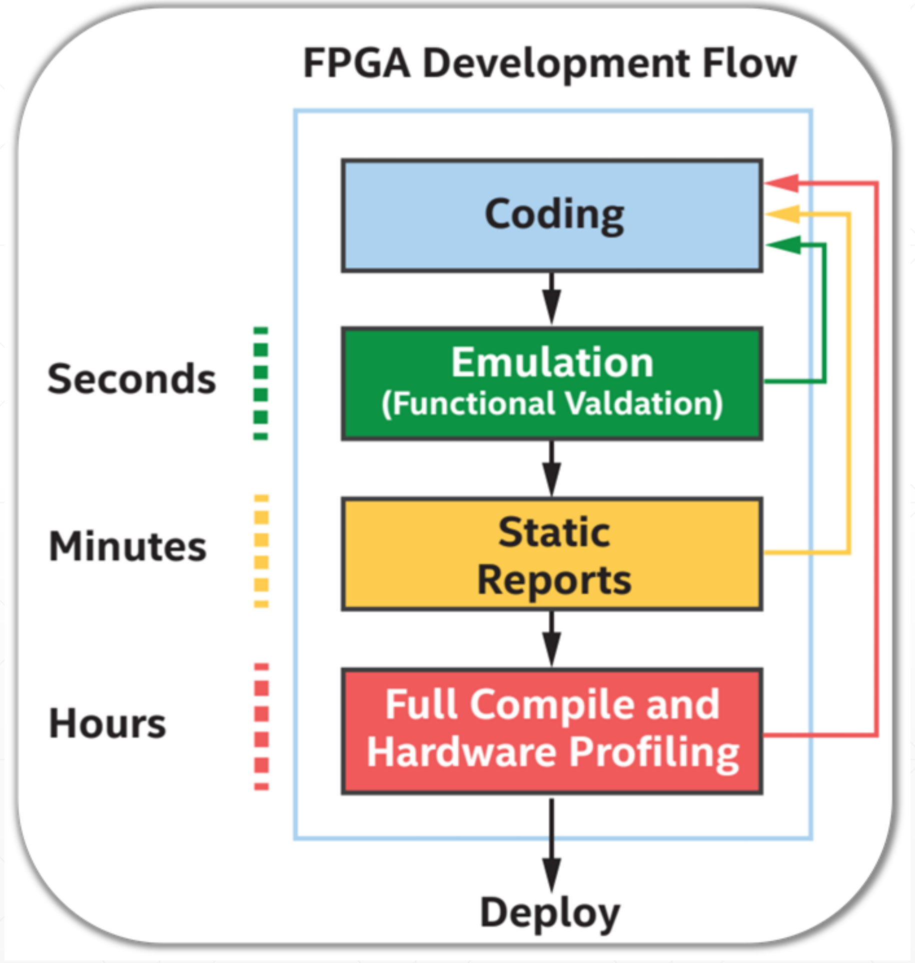
Why is FPGA Compilation Different?


With oneAPI Data Parallel C++ Compiler
Stage 1: Emulation
Compile to the FPGA Emulator
Functional Iterations
Stage 2: Optimization Report Generation
Identify bottlenecks
Architectural Iterations
With Intel® FPGA Add-On for DPC++ Compiler
Stage 3: FPGA Bitstream Compilation
Get code ready to run on FPGAs
With Intel® VTune Profiler
Stage 4 (optional): Runtime Analysis

- Key difference of development flow between FPGA and CPU (or GPU)
  - Only ahead-of-time compilation is support due to *very* time consuming FPGA bitstream compilation

Types of DPC++ FPGA Compilation

- FPGA Emulation
  - Generation of FPGA emulator image
  - Fastest method to verify the correctness of the code on CPU
  - Timing on emulator does not corresponds to FPGA hardware
- FPGA Static Report
  - Generation of FPGA early image (not executable)
  - Visualization of structure created by FPGA
  - Performance and bottleneck
  - Estimation of resource utilization
- FPGA Hardware Compile and Profiling
  - Generation of FPGA hardware image (bitstream)
  - Require Intel® FPGA Add-On for oneAPI Base Toolkit
  - Target Intel® Aria 10 GX, Stratix 10 SX or any custom board


FPGA Development Flow
Coding
Seconds
Emulation (Functional Valdation)
Minutes
Static Reports
Hours
Full Compile and Hardware Profiling
Deploy

FPGA Compilation Flags

```
# FPGA emulator image
dpcpp -fintelfpga -DFPGA_EMULATOR fpga_compile.cpp -o fpga_compile.fpga_emu

# FPGA early image (with optimization report): default board
dpcpp -fintelfpga -Xshardware -fsycl-link=early fpga_compile.cpp -o fpga_compile_report.a

# FPGA early image (with optimization report): explicit board
dpcpp -fintelfpga -Xshardware -fsycl-link=early -Xsboard=intel_s10sx_pac:pac_s10 fpga_compile.cpp -o fpga_compile_report.a

# FPGA hardware image: default board
dpcpp -fintelfpga -Xshardware fpga_compile.cpp -o fpga_compile.fpga

# FPGA hardware image: explicit board
dpcpp -fintelfpga -Xshardware -Xsboard=intel_s10sx_pac:pac_s10 fpga_compile.cpp -o fpga_compile.fpga
```

Flag	Explanation
`-fintelfpga`	Performs ahead-of-time (offline) compilation for FPGA
`-DFPGA_EMULATOR`	Preprocessor for device selection
`-Xshardware`	Instructs the compiler to target FPGA hardware (Default: FPGA emulator)
`-fsycl-link=early`	Instructs the compiler to stop after creating the FPGA early image (and associated optimization report)
`-Xsboard=<bsp:variant>`	Specifies the FPGA board variant (Default: Target Intel® Aria 10 GX)
`-Xsfast-compile`	Allows faster compile time but at a cost of reduced performance of the compiled FPGA hardware image
`-reuse-exe=<exe_name>`	Instruct the compiler to attempt to reuse the existing FPGA device image for fast compilation

FPGA Device Selector

```
#include <CL/sycl.hpp>

// FPGA device selectors are defined in this utility header, along with
// all FPGA extensions such as pipes and fpga_reg
#include <sycl/ext/intel/fpga_extensions.hpp>

int main() {
// Select either:
//  - the FPGA emulator device (CPU emulation of the FPGA)
//  - the FPGA device (a real FPGA, can be used for simulation too)
#if defined(FPGA_EMULATOR)
    ext::intel::fpga_emulator_selector device_selector;
#else
    ext::intel::fpga_selector device_selector;
#endif

queue q(device_selector);
...

// Print out the device information.
std::cout << "Running on device: "
          << q.get_device().get_info<info::device::name>() << "\n";
...
}
```

- FPGA emulator and FPGA are distintive devices
- Only a head of time compilation supported, making the *default_selector* less useful

FPGA Sample Code: VecAdd

```
std::vector<int> vec_a(kSize), vec_b(kSize), vec_r(kSize);

for (int i = 0; i < kSize; i++) {
    vec_a[i] = rand();
    vec_b[i] = rand();
}

// Create buffers to share data between host and device.
// The runtime will copy the necessary data to the FPGA device memory when the kernel is launched.
buffer buf_a(vec_a);
buffer buf_b(vec_b);
buffer buf_r(vec_r);

// Submit a command group to the device queue.
q.submit([&](handler& h) {
    // The SYCL runtime uses the accessors to infer data dependencies.
    // A "read" accessor must wait for data to be copied to the device before the kernel can start.
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor r(buf_r, h, write_only, no_init);

    // The task's for loop is executed in pipeline parallel on the FPGA
    h.single_task<VectorAdd>([=]() [[intel::kernel_args_restrict]] {
        for (int i = 0; i < kSize; ++i) {
            r[i] = a[i] + b[i];
        }
    });
});
```

FPGA Fast Compilation: Reuse FPGA Hardware Image

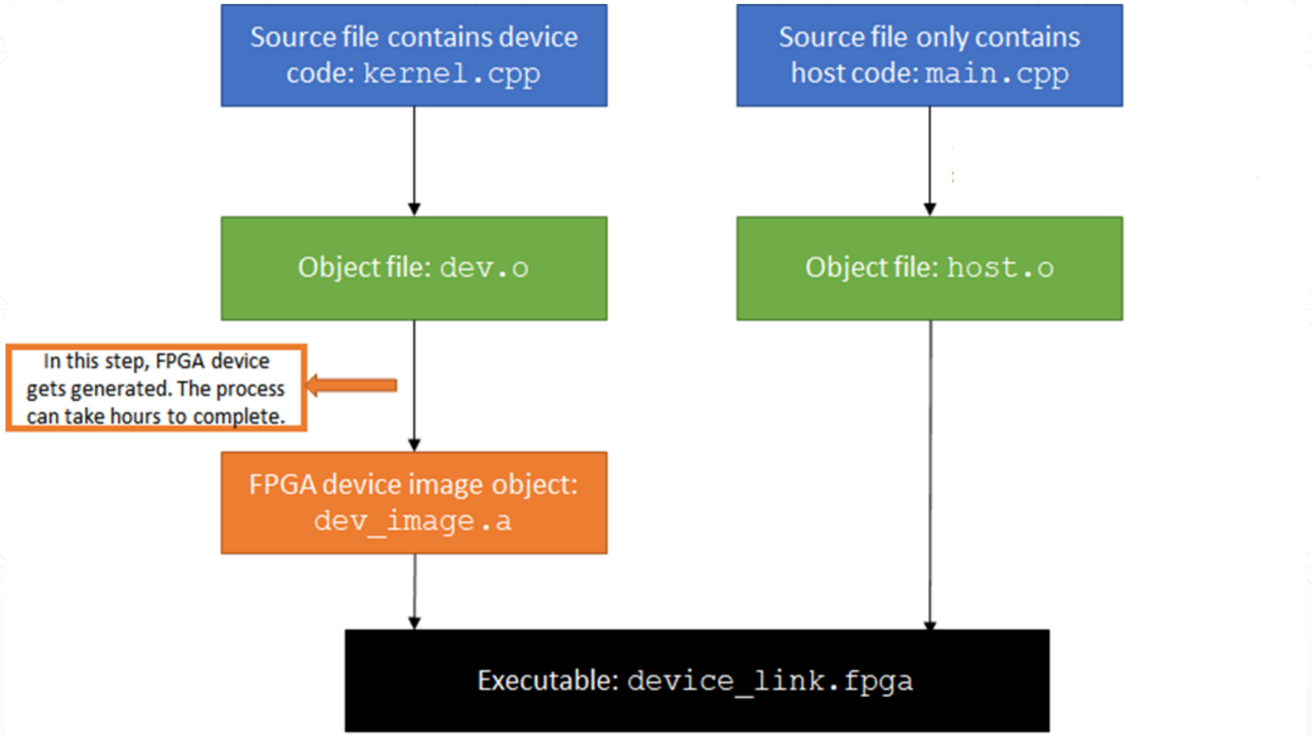
```
# Initial compilation
dpcpp -fintelfpga -Xshardware <files.cpp> -o out.fpga

# Subsequent recompilation
dpcpp <files.cpp> -o out.fpga -reuse-exe=out.fpga -Xshardware -fintelfpga
```

- AOT compilation takes several hours to generate FPGA hardware image
  - `-reuse-exe=<exe_name>` to instruct the compiler to reuse the existing FPGA device image.
  - Separating the host and device code into separate files, i.e device link method

- Reuse FPGA image
  - If no change in device code is detected by compiler, only host code is recompiled (few minutes)
  - If changes in device is detected by compiler, FPGA device is fully recompiled (few hours)

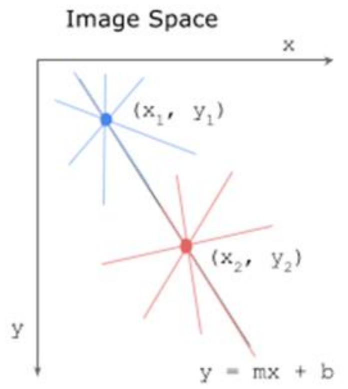
- Caveats:
  - Strong coupling between host and devices code within single source file
  - Very limited in practice, probably only suitable for simple design
  - Recompilation triggering when compiler falsely detect change in device code

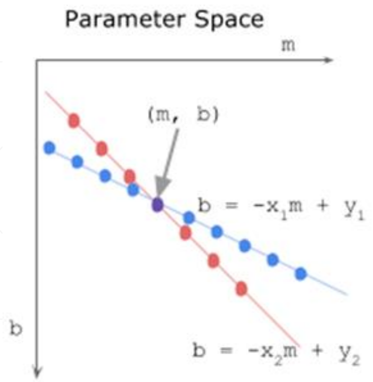
FPGA Fast Compilation: Device Link Method

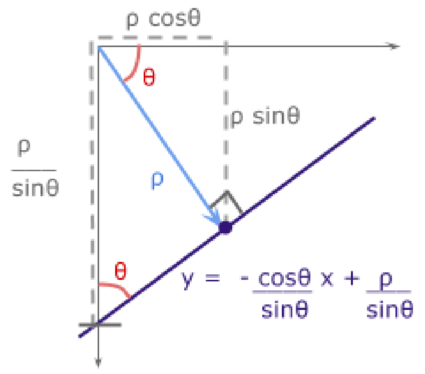

Source file contains device code: kernel.cpp
Source file only contains host code: main.cpp
Object file: dev.o
Object file: host.o
In this step, FPGA device gets generated. The process can take hours to complete.
FPGA device image object: dev_image.a
Executable: device_link.fpga

```
# Compile device code
dpcpp -fintelfpga -fsycl-link=image kernel.cpp -o dev_image.a -Xshardware

# Compile host code (redo after host code changed)
dpcpp -fintelfpga host.cpp -c -o host.o

# Create device link (redo after host code changed)
dpcpp -fintelfpga host.o dev_image.a -o fast_recompile.fpga
```

Hough Transformation for Boundary Detection



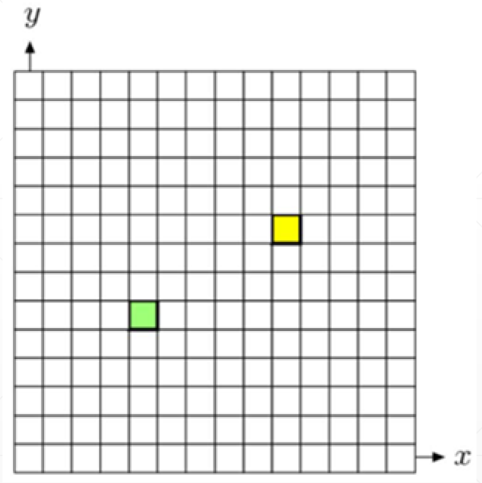


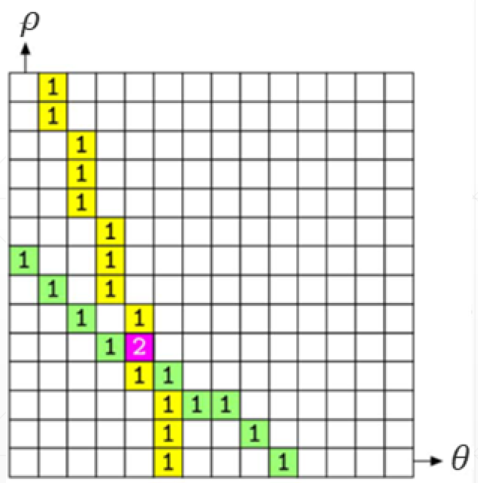

$\rho \cos\theta$
$\rho \sin\theta$
$\frac{\rho}{\sin\theta}$
θ
ρ
θ
$y = -\frac{\cos\theta}{\sin\theta} x + \frac{\rho}{\sin\theta}$

Input (edge detected)

Output (line detected)

```
for all x
    for all y
         if edge point is at (x,y)
             for all thetas:
                 rho = x * cos(theta) + y * sin(theta)
                 accumulator(rho, theta) += 1
             end
         end
     end
end
```

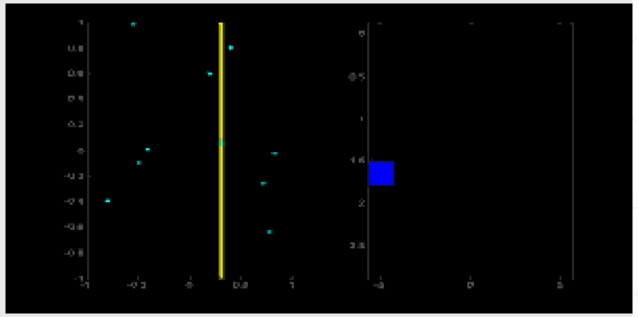




- Consider a set of (x_i, y_i) corresponds to detected edges of image object:
  - $y_i = mx_i + b$ (point/image space) $\leftrightarrow$ $b = -mx_i + y_i$ (line/parameter space)
  - Detection of vertical lines: parameterized representation (ρ, θ)

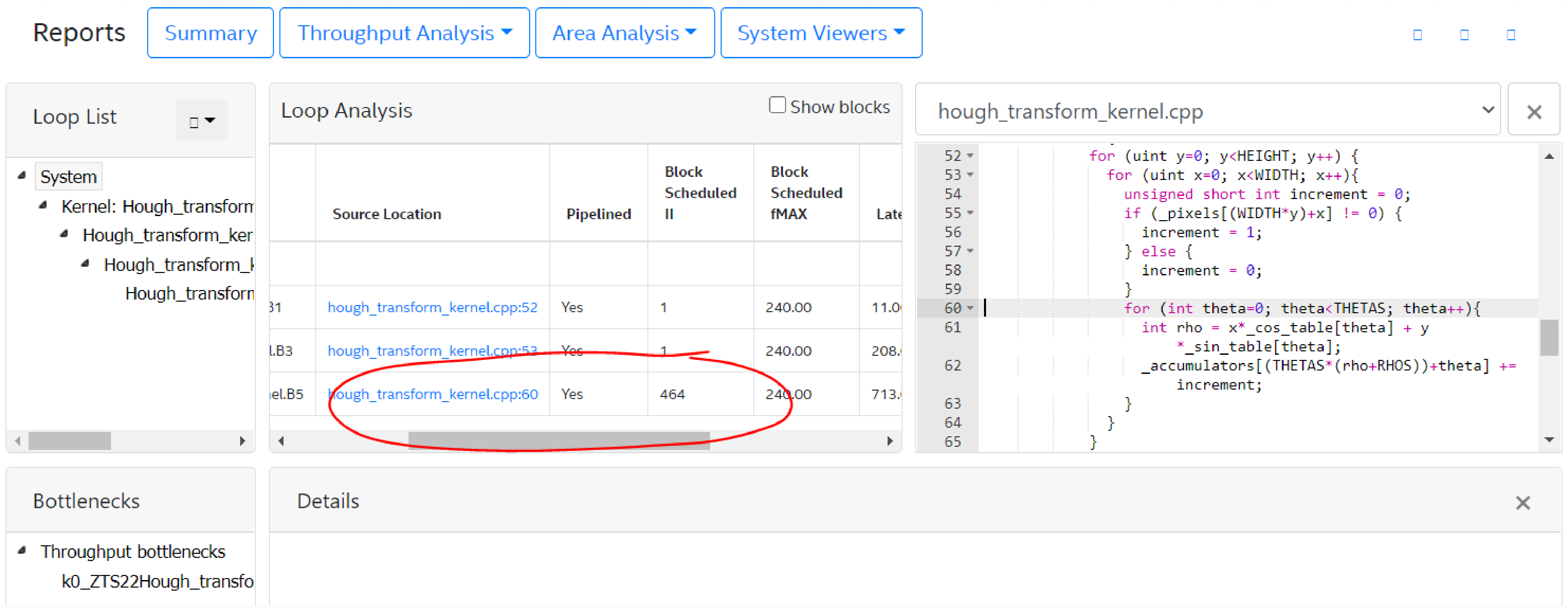
FPGA Performance Optimization: Hough Transformation

```
queue_event = device_queue.submit([&](sycl::handler &cgh) {
//Create accessors
    auto _pixels       = pixels_buf.get_access<sycl::access::mode::read>(cgh);
    auto _sin_table    = sin_table_buf.get_access<sycl::access::mode::read>(cgh);
    auto _cos_table    = cos_table_buf.get_access<sycl::access::mode::read>(cgh);
    auto _accumulators = accumulators_buf.get_access<sycl::access::mode::read_write>(cgh);

    //Call the kernel
    cgh.single_task<class Hough_transform_kernel>([=]() {
        for (uint y=0; y<HEIGHT; y++) {
            for (uint x=0; x<WIDTH; x++){
                unsigned short int increment = 0;
                if (_pixels[(WIDTH*y)+x] != 0) {
                    increment = 1;
                } else {
                    increment = 0;
                }

                for (int theta=0; theta<THETAS; theta++){
                    int rho = x*_cos_table[theta] + y*_sin_table[theta];
                    _accumulators[(THETAS*(rho+RHOS))+theta] += increment;
                }
            }
        }
    });
});
```



- _accumulators matrix is updated by scanning over possible value of thetas

FPGA Performance Optimization: Early Image Report

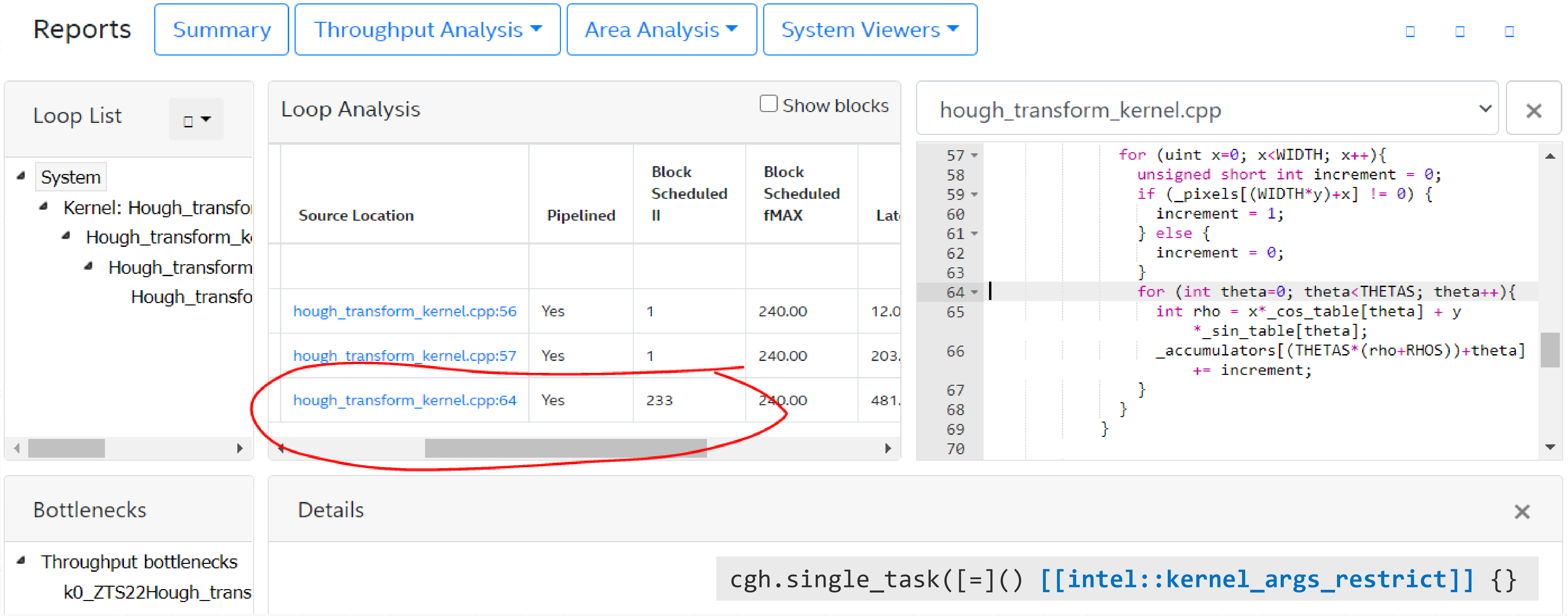


	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Late
31	hough_transform_kernel.cpp:52	Yes	1	240.00	11.0
l.B3	hough_transform_kernel.cpp:53	Yes	1	240.00	208.
el.B5	hough_transform_kernel.cpp:60	Yes	464	240.00	713.

```
for (uint y=0; y<HEIGHT; y++) {
  for (uint x=0; x<WIDTH; x++){
    unsigned short int increment = 0;
    if (_pixels[(WIDTH*y)+x] != 0) {
      increment = 1;
    } else {
      increment = 0;
    }
    for (int theta=0; theta<THETAS; theta++){
      int rho = x*_cos_table[theta] + y
          *_sin_table[theta];
      _accumulators[(THETAS*(rho+RHOS))+theta] +=
          increment;
    }
  }
}
```

- Generate early image (not executable) with architecture optimization report:
  - `dpcpp -fintelfpga -Xshardware -fsycl-link=early main.cpp hough_transform.cpp -o fpga_compile_report.a`
- HTML report: *fpga_compile_report.prj/reports/report.html*

FPGA Performance Optimization: Avoid Aliasing of Kernel Arguments



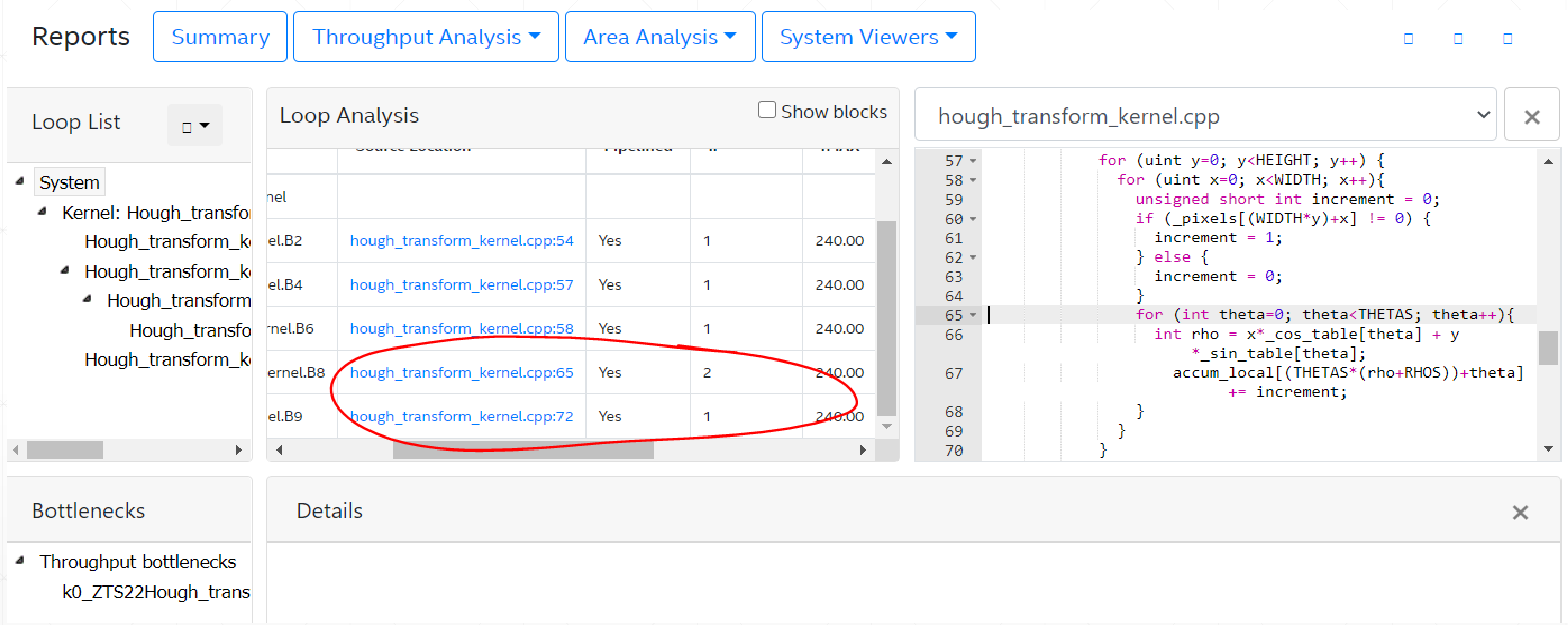
Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Lat
hough_transform_kernel.cpp:56	Yes	1	240.00	12.0
hough_transform_kernel.cpp:57	Yes	1	240.00	203.
hough_transform_kernel.cpp:64	Yes	233	240.00	481.

```
    for (uint x=0; x<WIDTH; x++){
      unsigned short int increment = 0;
      if (_pixels[(WIDTH*y)+x] != 0) {
        increment = 1;
      } else {
        increment = 0;
      }
      for (int theta=0; theta<THETAS; theta++){
        int rho = x*_cos_table[theta] + y
            *_sin_table[theta];
        _accumulators[(THETAS*(rho+RHOS))+theta]
            += increment;
      }
    }
  }
}
```

```
cgh.single_task([=]() [[intel::kernel_args_restrict]] {}
```

- Pointer aliasing occurs when the same memory location is accessed using different names
- kernel_args_restrict for more aggressive compiler optimizations and improved FPGA performance

FPGA Performance Optimization: Local Memory



- System
  - Kernel: Hough_transfo
    - Hough_transform_k
    - Hough_transform_k
      - Hough_transform
        - Hough_transfo
    - Hough_transform_k

nel				
el.B2	hough_transform_kernel.cpp:54	Yes	1	240.00
el.B4	hough_transform_kernel.cpp:57	Yes	1	240.00
rnel.B6	hough_transform_kernel.cpp:58	Yes	1	240.00
ernel.B8	hough_transform_kernel.cpp:65	Yes	2	240.00
el.B9	hough_transform_kernel.cpp:72	Yes	1	240.00

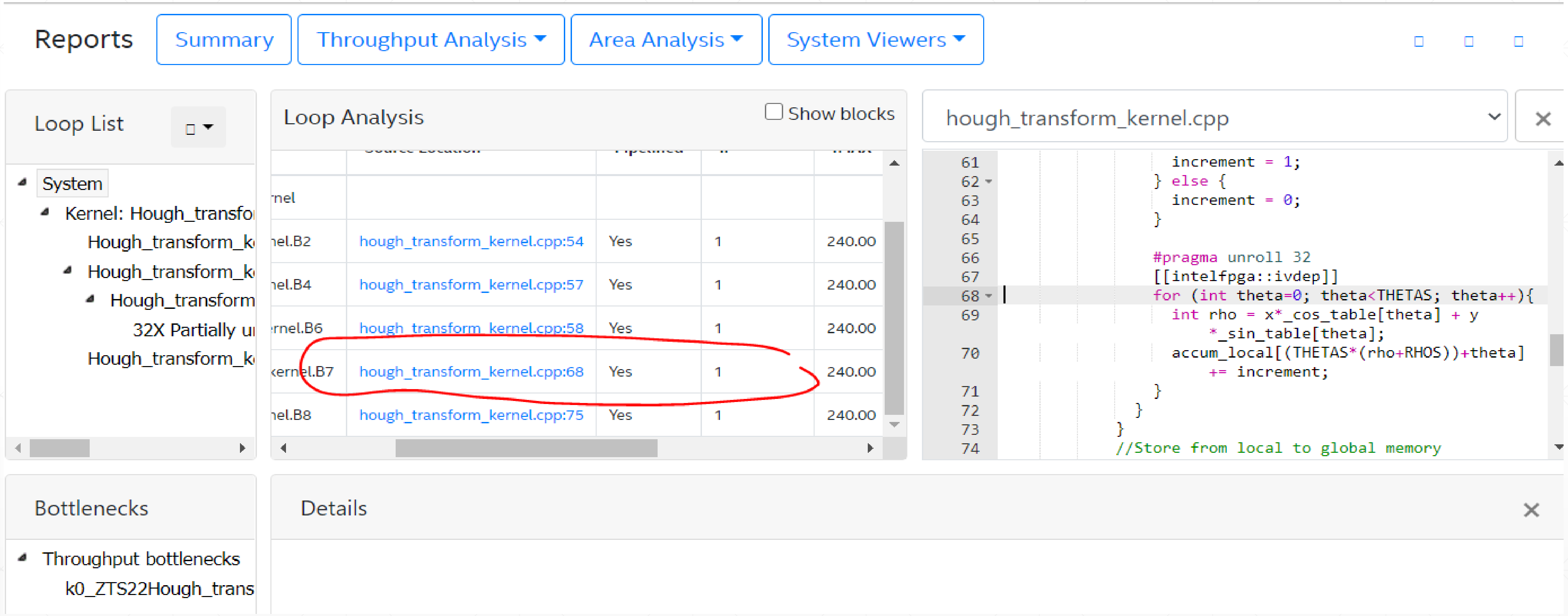
hough_transform_kernel.cpp

```
57          for (uint y=0; y<HEIGHT; y++) {
58            for (uint x=0; x<WIDTH; x++){
59              unsigned short int increment = 0;
60              if (_pixels[(WIDTH*y)+x] != 0) {
61                increment = 1;
62              } else {
63                increment = 0;
64              }
65              for (int theta=0; theta<THETAS; theta++){
66                int rho = x*_cos_table[theta] + y
                    *_sin_table[theta];
67                  accum_local[(THETAS*(rho+RHOS))+theta]
                        += increment;
68              }
69            }
70          }
```

- Throughput bottlenecks
  - k0_ZTS22Hough_trans

- Slow data retrieval from global memory, increasing initiation interval (II)
- Local memory is referred to as on-chip memory created from FPGA’s RAM blocks

FPGA Performance Optimization: Loop Unroll and IVDEP


Reports | Summary | Throughput Analysis | Area Analysis | System Viewers

Loop List

- System
  - Kernel: Hough_transfo
    - Hough_transform_k
    - Hough_transform_k
      - Hough_transform
        - 32X Partially u
    - Hough_transform_k

Loop Analysis ☐ Show blocks

	Source Location	Pipelined	II	Fmax
rnel				
nel.B2	hough_transform_kernel.cpp:54	Yes	1	240.00
nel.B4	hough_transform_kernel.cpp:57	Yes	1	240.00
rnel.B6	hough_transform_kernel.cpp:58	Yes	1	240.00
kernel.B7	hough_transform_kernel.cpp:68	Yes	1	240.00
nel.B8	hough_transform_kernel.cpp:75	Yes	1	240.00

hough_transform_kernel.cpp

```
61              increment = 1;
62            } else {
63              increment = 0;
64            }
65
66            #pragma unroll 32
67            [[intelfpga::ivdep]]
68            for (int theta=0; theta<THETAS; theta++){
69              int rho = x*_cos_table[theta] + y
                    *_sin_table[theta];
70              accum_local[(THETAS*(rho+RHOS))+theta]
                    += increment;
71            }
72          }
73        }
74        //Store from local to global memory
```

Bottlenecks

- Throughput bottlenecks
  - k0_ZTS22Hough_trans

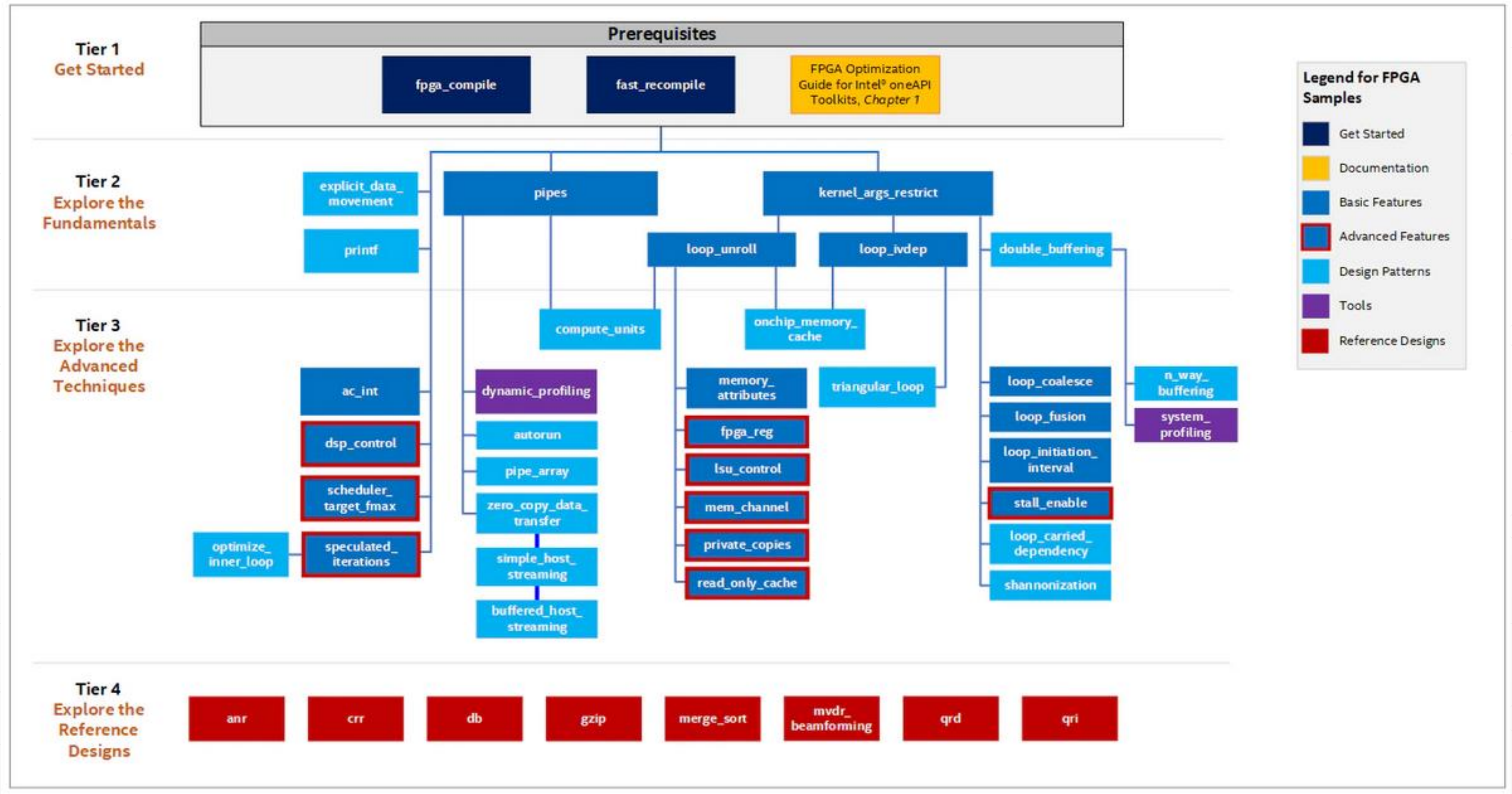
Details

- Improved parallelism by duplicating the compute logic with loop unrolling
- Ignore loop carries dependency based on iteration distance across multiple loops

FPGA Performance Optimization: Hough Transformation Summary

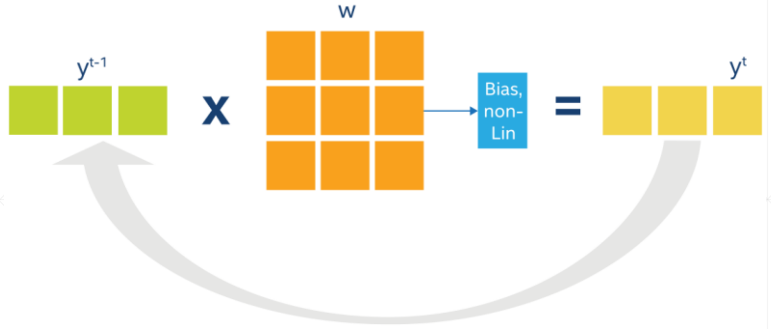
```
cgh.single_task([=]() [[intel::kernel_args_restrict]] {
    //Load from global to local memory
    short accum_local[RHOS*2*THETAS];
    for (int i = 0; i < RHOS*2*THETAS; i++) {
        accum_local[i] = 0;
    }
    for (uint y=0; y<HEIGHT; y++) {
        for (uint x=0; x<WIDTH; x++){
        unsigned short int increment = 0;
            if (_pixels[(WIDTH*y)+x] != 0) {
                increment = 1;
            } else {
                increment = 0;
            }

            #pragma unroll 32
            [[intel::ivdep]]
            for (int theta=0; theta<THETAS; theta++){
                int rho = x*_cos_table[theta] + y*_sin_table[theta];
                accum_local[(THETAS*(rho+RHOS))+theta] += increment;
            }
        }
    }
    //Store from local to global memory
    for (int i = 0; i < RHOS*2*THETAS; i++) {
        _accumulators[i] = accum_local[i];
    }
});
```

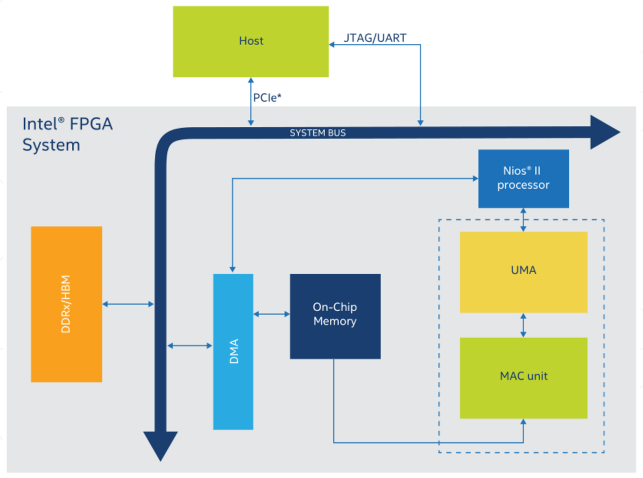
Explore DPC++ Through Intel® FPGA Code Samples

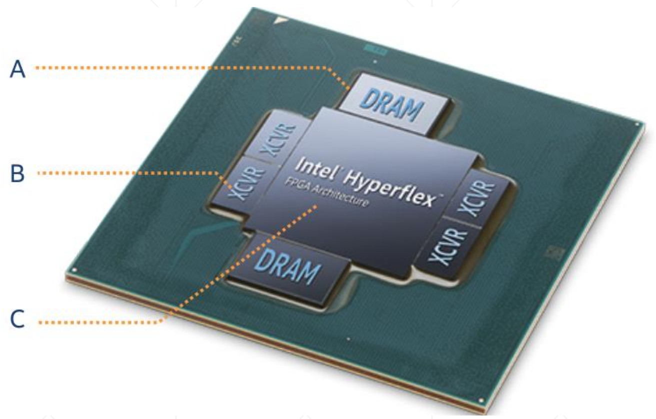


https://www.intel.com/content/www/us/en/developer/articles/code-sample/explore-dpcpp-through-intel-fpga-code-samples.html

Case Study: Accelerating Memory Bound AI Inference Workloads

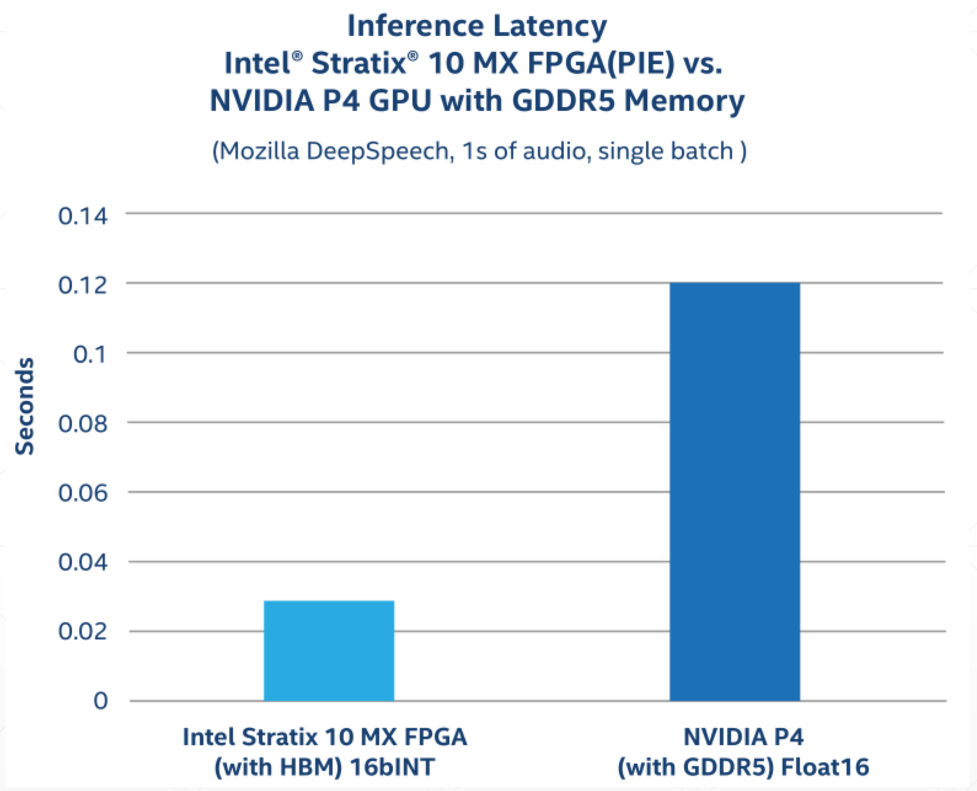


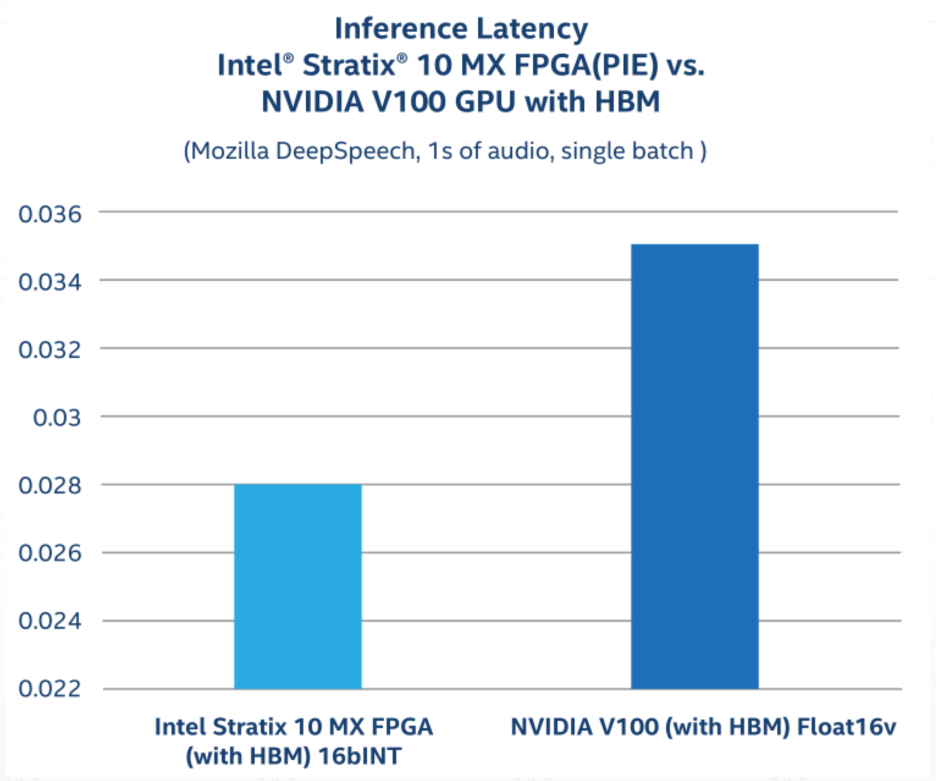




- Recurrent Neural Network (RNN):
  - Reflect the influence of past data on current data
  - Finance, genome mapping, speech AI, automatic speech recognition (ASR), etc
  - Repeated application of weight matrix each time step, i.e low latency required for real time application
  - Suitable for FPGA accelerator thanks to pipeline parallelism
- Intel Stratix 10 MX FPGA:
  - Single package of a state-of-the-art Intel® FPGA with Samsung High Bandwidth Memory 2 (HBM2)
  - 10x bandwidth with highest performance per Watt in comparison to SDRAM
  - Programmable high performance AI inference accelerator for Intel FPGAs

Case Study: Accelerating Memory Bound AI Inference Workloads





- Mozilla DeepSpeech algorithm was accelerated using Intel FPGA
  - Stratix 10MX has ~ 4x lower latency than NVIDIA P4
  - Stratix 10MX has ~ 20% lower latency than NVIDIA V100 (HBM)

Conclusion

- FPGA offers advantages over traditional accelerators such as CPU and GPU
  - Highly efficiency via pipeline parallelism
  - Data dependency across parallel work automatically resolved by DPCPP compiler
- FPGA development workflow with oneAPI
  - Emulation -> optimization report -> bitstream compilation
  - Single-work item kernel instead ND-range kernel
  - Optimization techniques
- FPGA case study
  - Stratix 10 MX offer lower latency than GPUs for RNN workload
- **Contact**: sales@moasys.com
  - GPU, FPGA code migration
  - Optimization and parallelization consultant
  - Customized HPC education