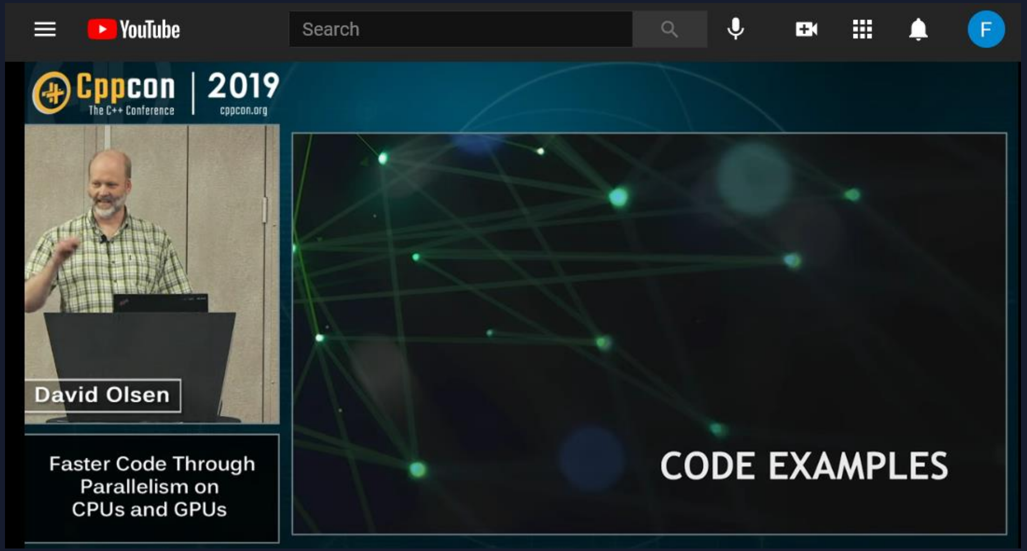
XILINX®

超越CPU及GPU性能的Vitis加速应用C++内核开发实例

李昀
软件与 AI 加速技术市场部
July 2021



XILINX

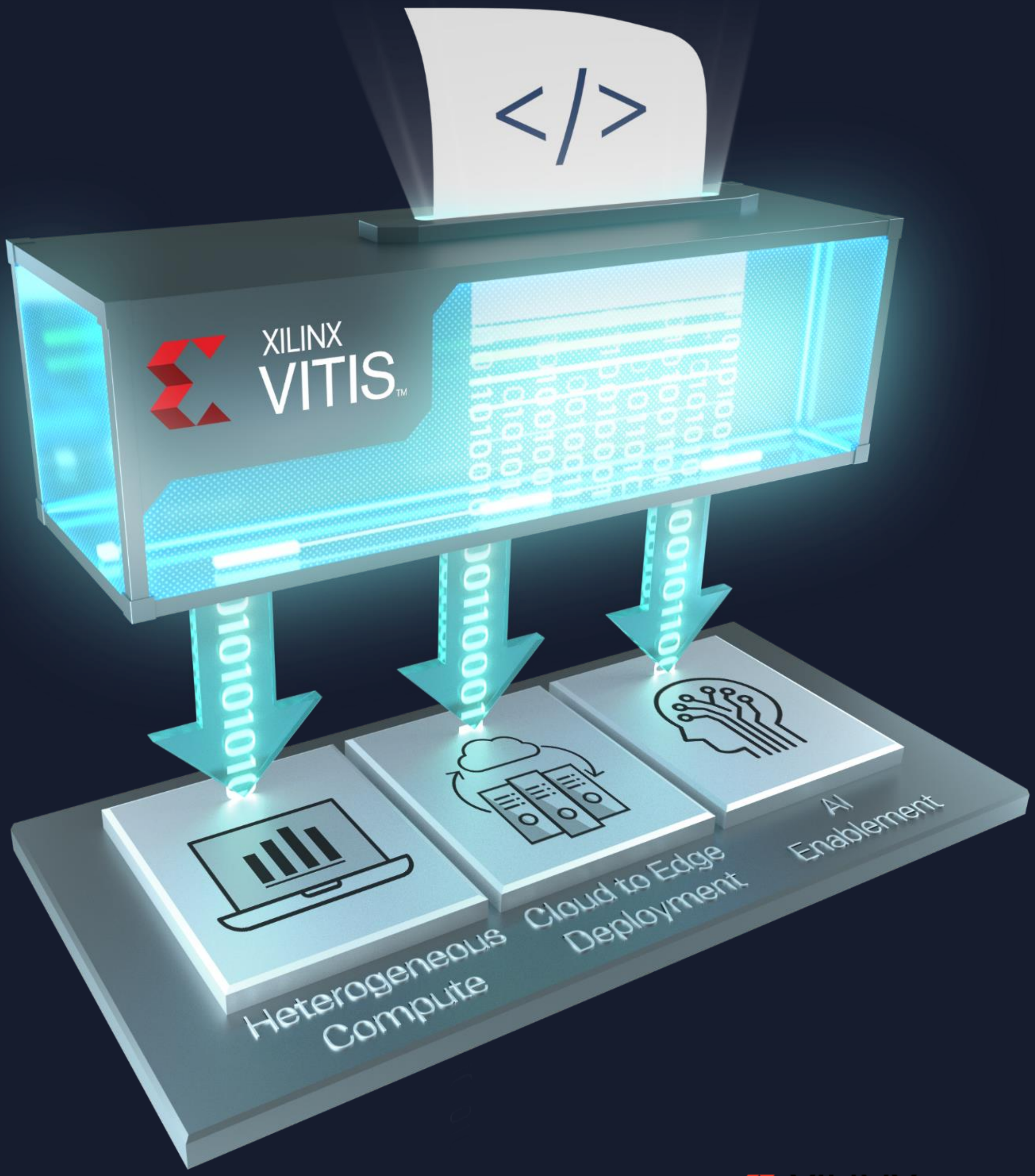
VITIS™

Unified Software Platform

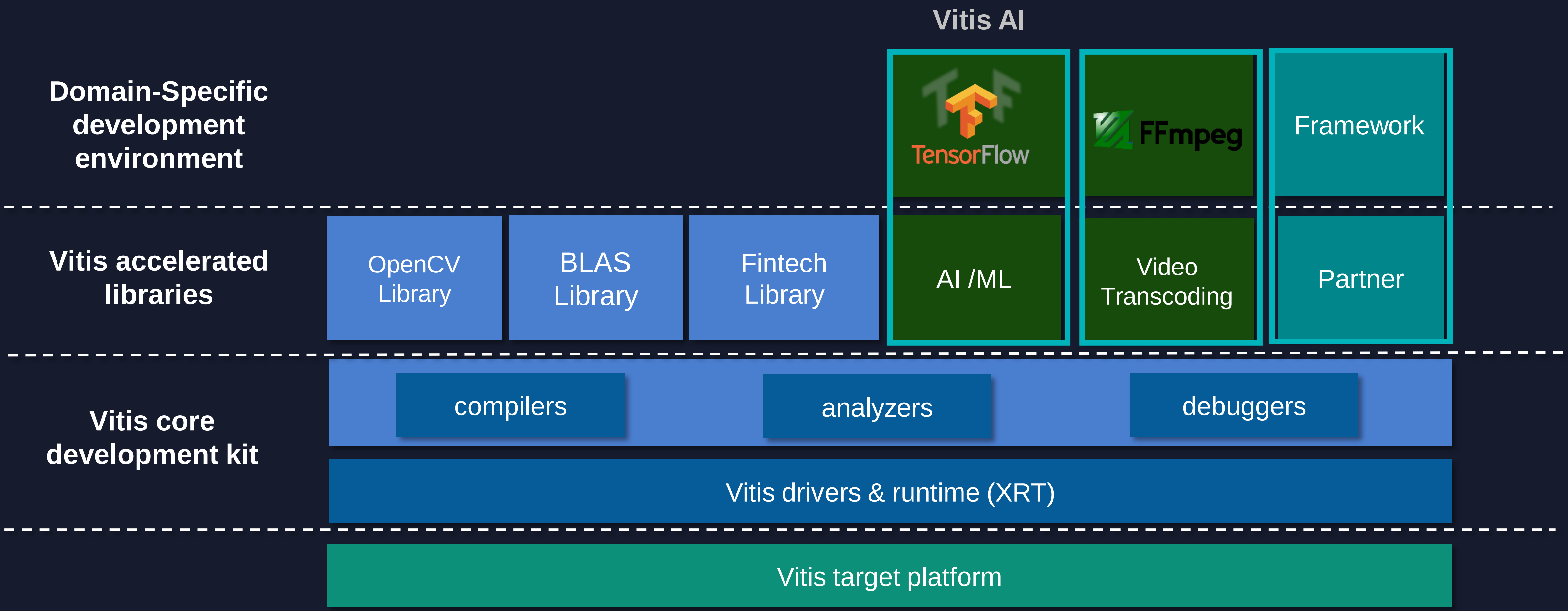
Software & AI

Adaptive Computing

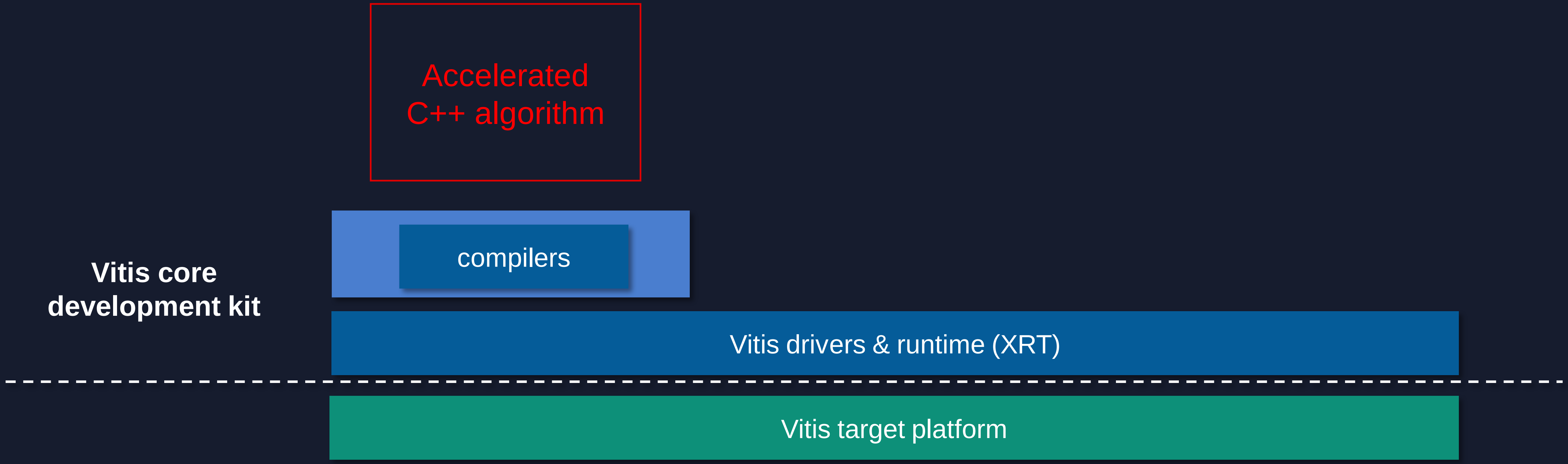
Edge to Cloud



Vitis: Unified Software Platform

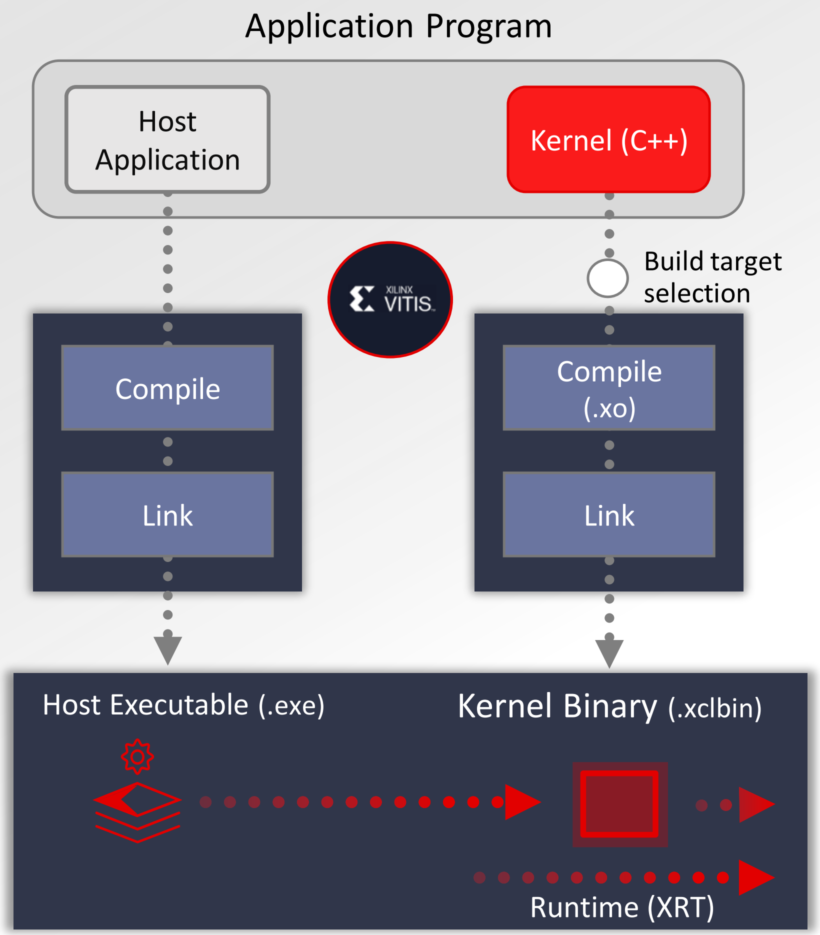


Vitis: Unified Software Platform


Accelerated C++ algorithm
compilers
Vitis core development kit
Vitis drivers & runtime (XRT)
Vitis target platform

Developing Accelerators

- Accelerators placed into the FPGA as “kernels”
- Kernels can be developed using different methods
  - High-level synthesis with C, C++, and OpenCL
  - Model Composer, MATLAB, and Simulink
  - RTL
- Vitis links the kernels into reconfigurable binaries
- Emulation support
  - System-level verification and quick debug



C++ Kernel Build

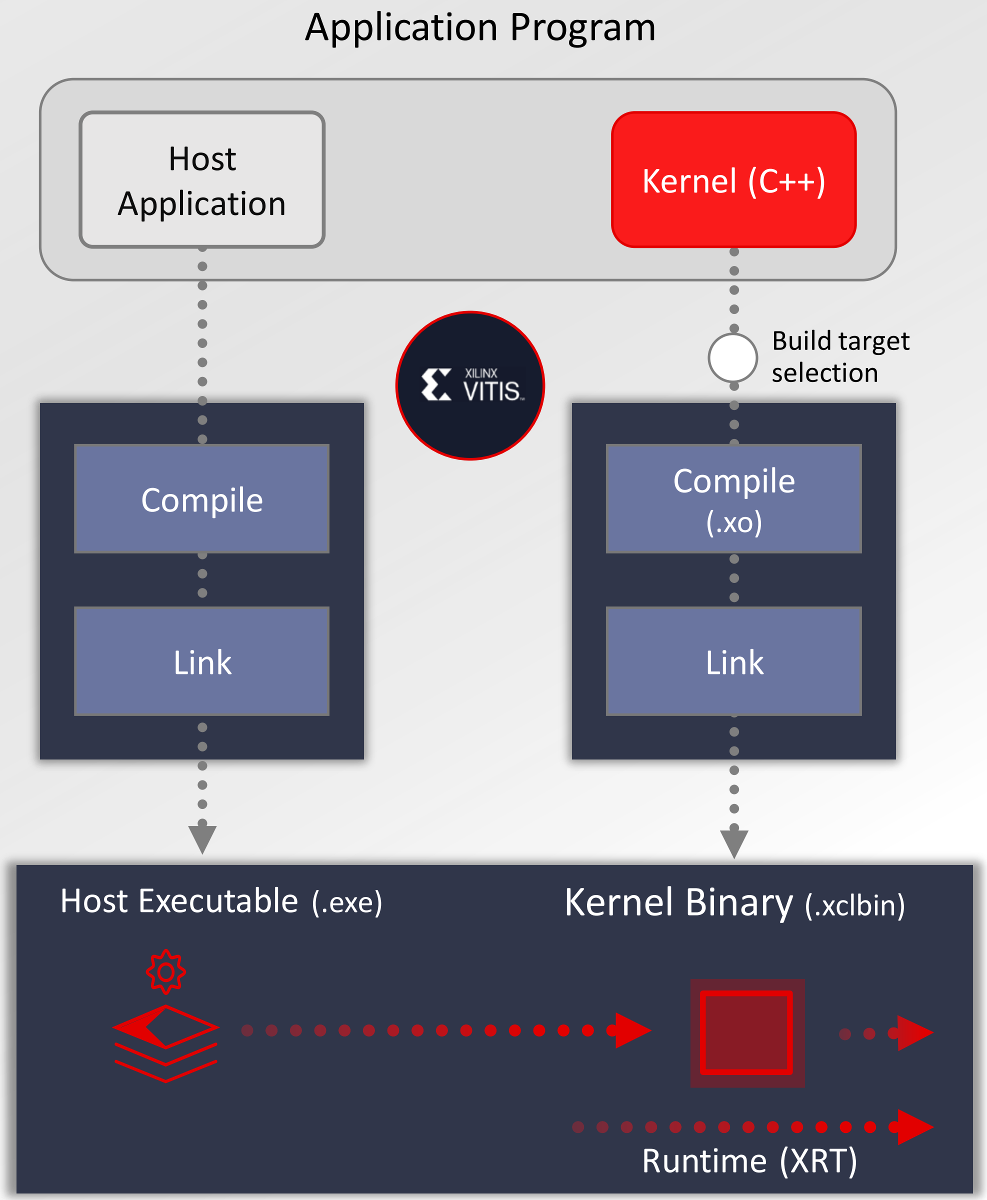
Application Build Process

✓ v++ compiles host code with APIs

✓ v++ compiles kernels into .xo

✓ v++ links kernels to the platform

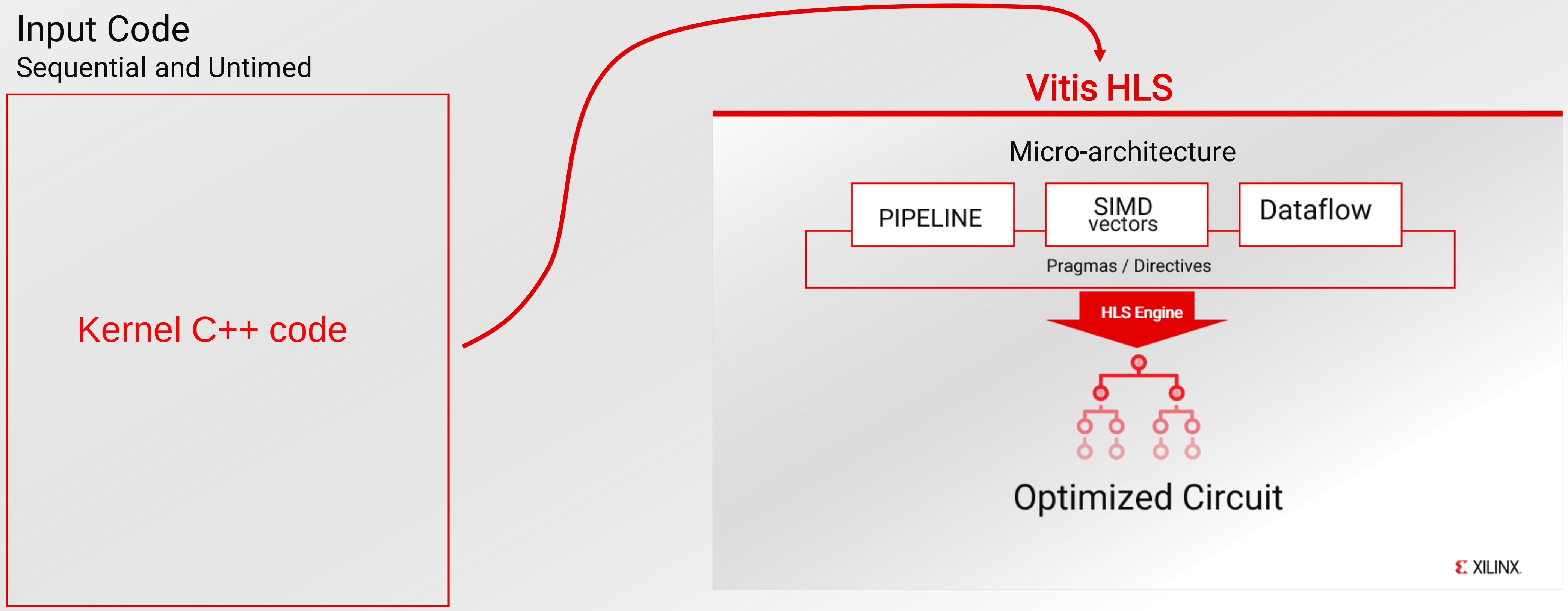
✓ Final .xclbin binary loads into the device

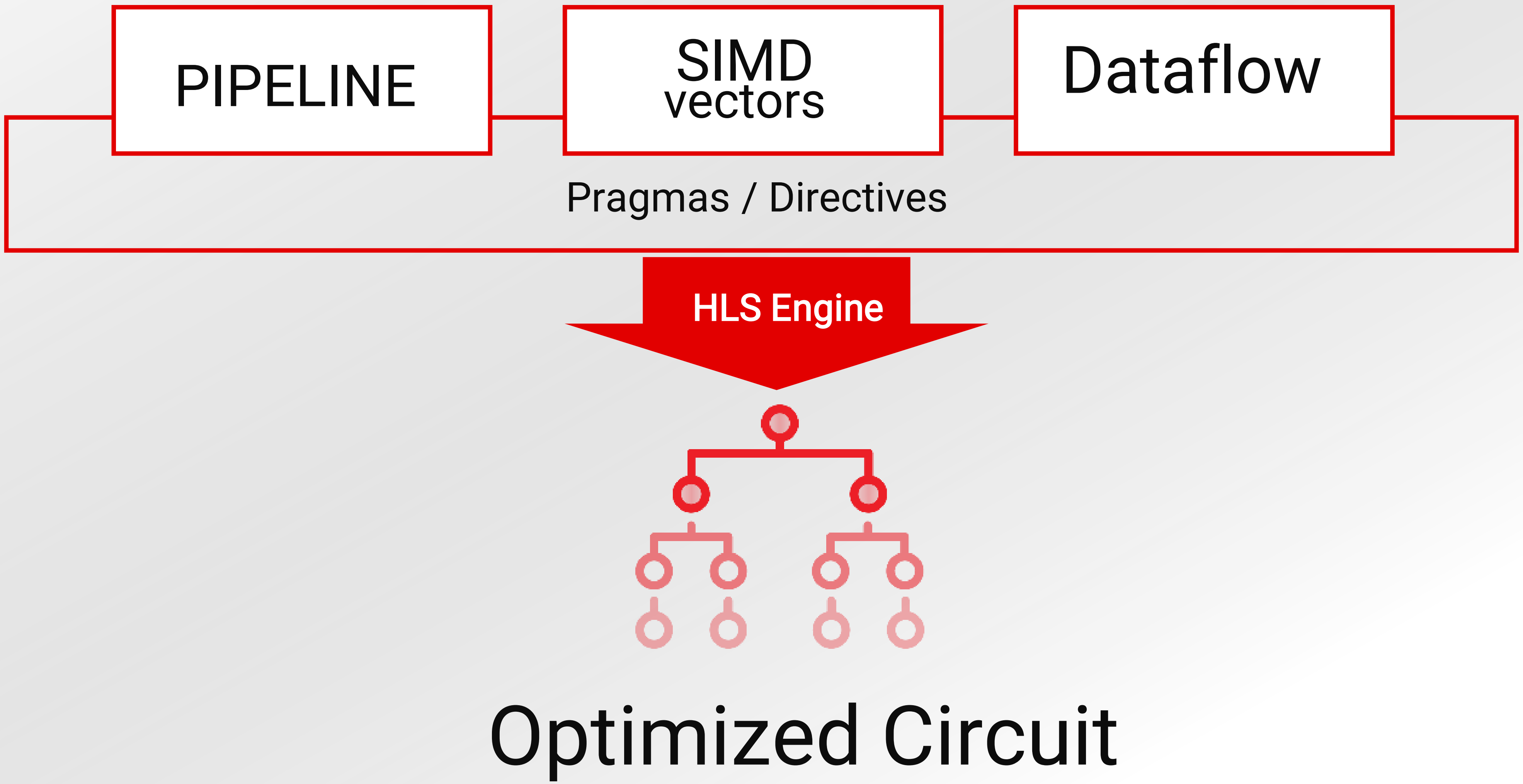


Compiler Directives

Vitis HLS: A Parallel Hardware Compiler

C++ code compiler for highly optimized implementation onto logic fabric




PIPELINE
SIMD vectors
Dataflow
Pragmas / Directives
HLS Engine
Optimized Circuit

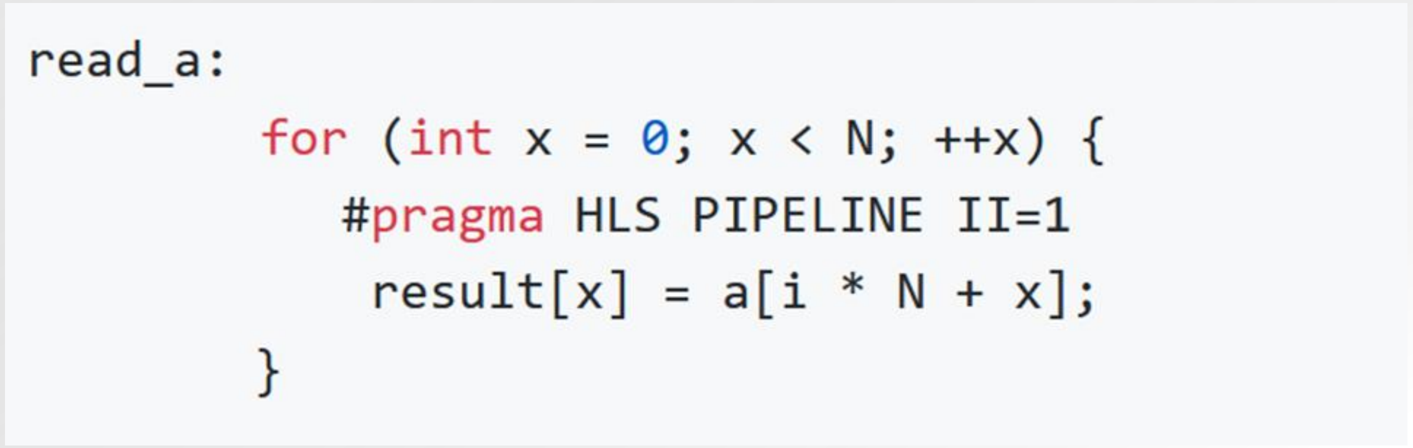
PIPELINE

Reading new inputs before a loop finishes processing current input…

- Tied to the concept of “initiation interval” or II
  - e.g., an initiation interval of 1 means a loop processes an input at every clock cycle
- The tool automatically pipelines the most inner loops
- C functions might be pipelined too but could unroll all loops in function body hence leading to a prohibitive amount of resource used

PIPELINE

Reading new inputs before a loop finishes processing current input…



```
read_a:
        for (int x = 0; x < N; ++x) {
          #pragma HLS PIPELINE II=1
           result[x] = a[i * N + x];
        }
```

https://github.com/Xilinx/Vitis_Accel_Examples/

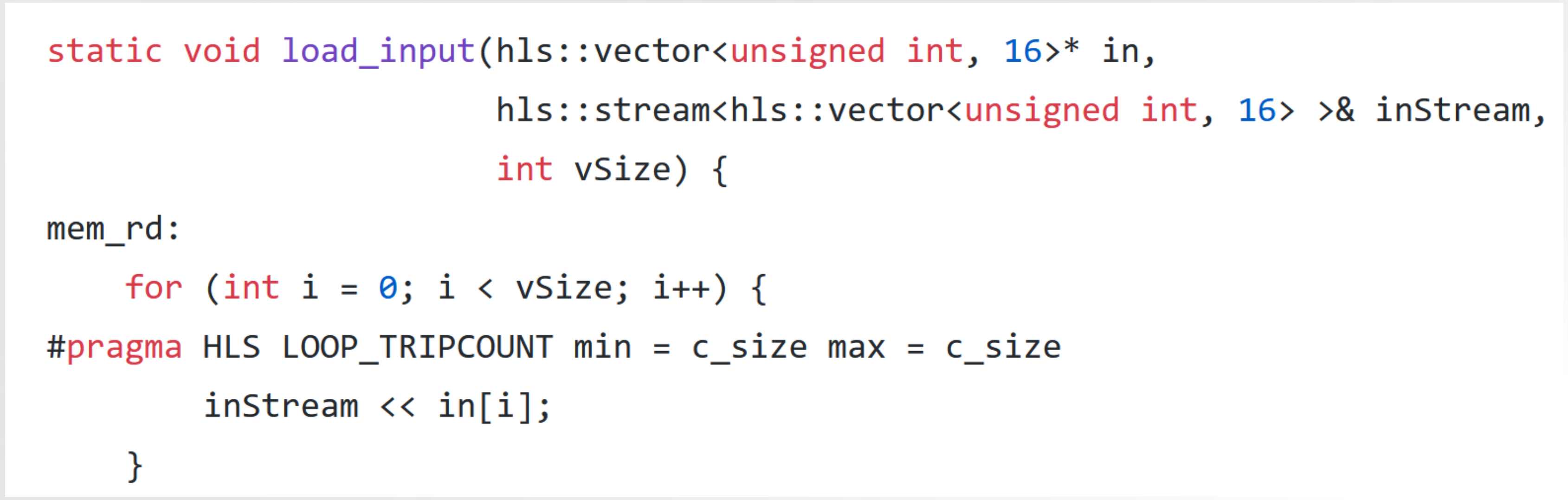
SIMD vectors

Single-Instruction-Multiple-Data and Vectors for parallelism

- Unrolling a loop to call a sub-function multiple times
- Vectors leverage the GCC __attribute__ (vector_size())

SIMD
vectors

Single-Instruction-Multiple-Data and Vectors for parallelism



```
static void load_input(hls::vector<unsigned int, 16>* in,
                       hls::stream<hls::vector<unsigned int, 16> >& inStream,
                       int vSize) {
mem_rd:
    for (int i = 0; i < vSize; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
        inStream << in[i];
    }
```

https://github.com/Xilinx/Vitis_Accel_Examples/

XILINX.

Dataflow

Separating sub-functions as individual processes and creating expanded memory channels…

- Significantly reduces latency and hardware resources for tasks that are otherwise serial
- Duplicated memory channels ensure efficient processing
  - Channels can be FIFO too…

Dataflow

Separating sub-functions as individual processes and creating expanded memory channels…

```
#pragma HLS dataflow
load →         read_input(in, inStream, size);
compute →      compute_add(inStream, outStream, inc, size);
store →        write_result(out, outStream, size);
```

https://github.com/Xilinx/Vitis_Accel_Examples/

PIPELINE

SIMD
vectors

Dataflow

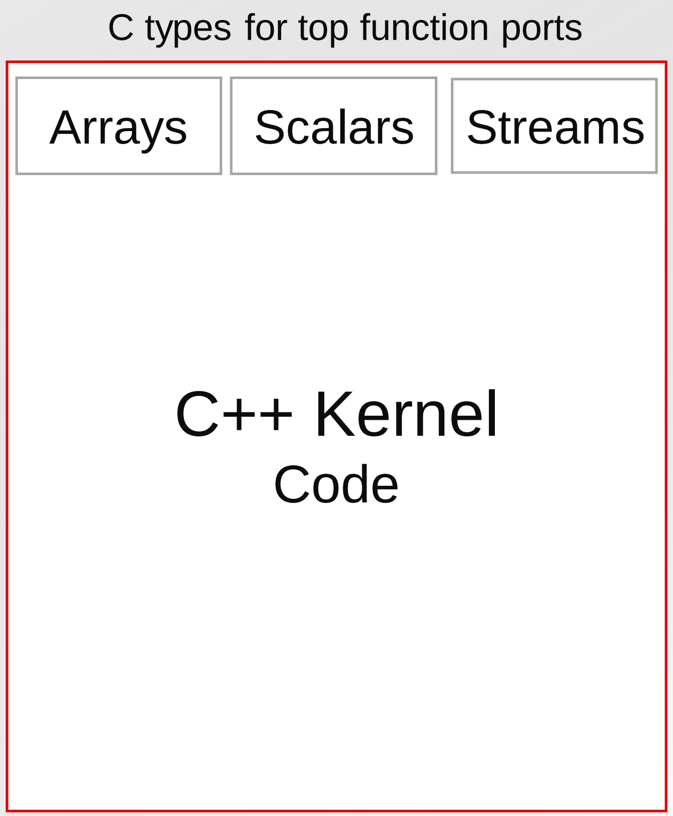
Pragmas / Directives

Other pragmas support the main optimization pillars…

- Array partitioning and reshaping
  - Help ensure the accesses are not limiting the II
- Directives BIND_OP, BIND_STORAGE help customize resources…
- …

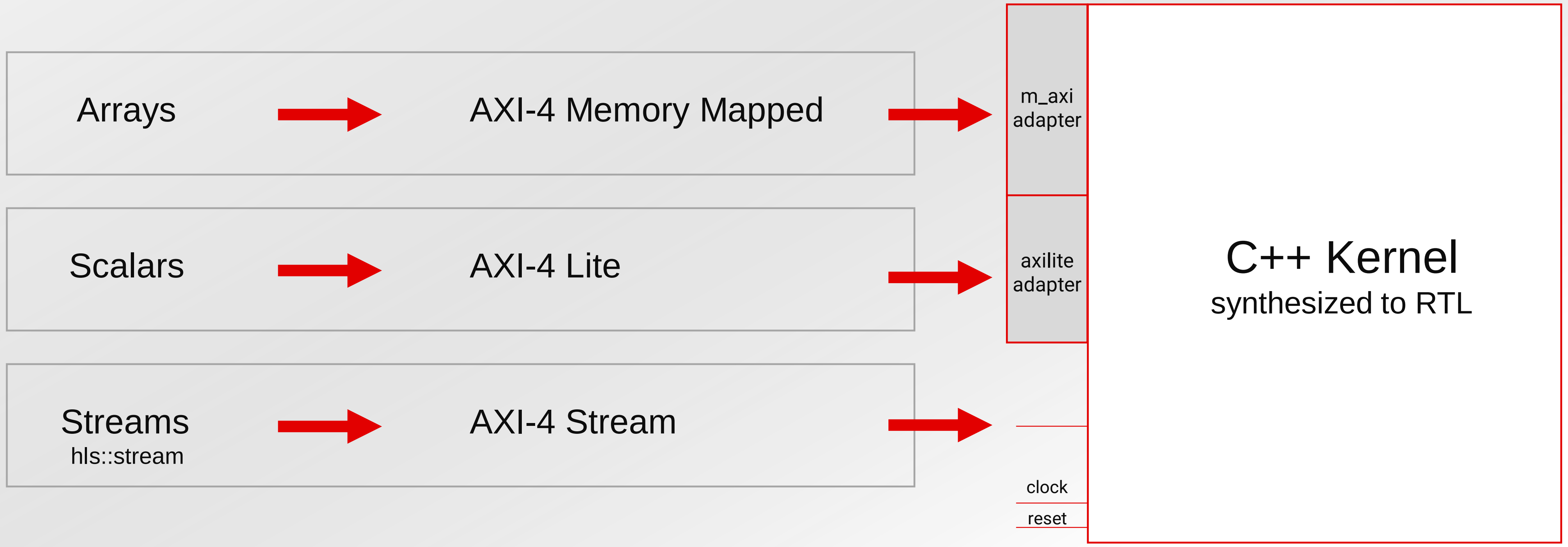
Ports and Interfaces

C++ Kernel Interfaces in Vitis



C++ Kernel Interfaces in Vitis

- C++ datatypes and default hardware implementation…



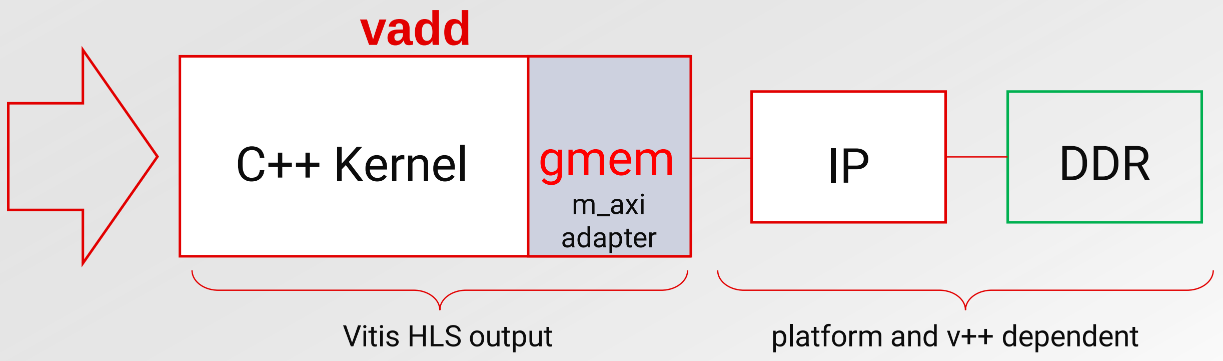
> The INTERFACE pragma specifies the physical connection for C++ function arguments…

XILINX

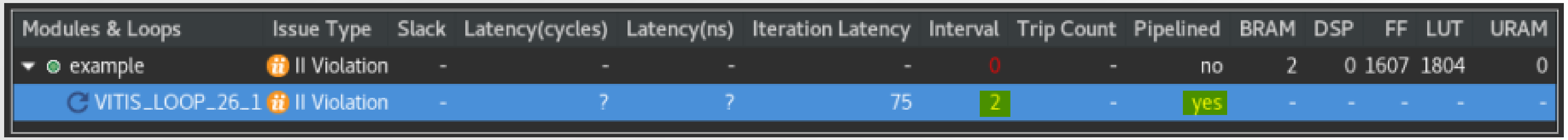
Interface Optimization for Pointers

Step1: Apply the INTERFACE pragmas

```
void vadd(const unsigned int *in1,
          const unsigned int *in2,
          unsigned int *out,
          int size)
{
#pragma HLS INTERFACE m_axi bundle=gmem port=in1
#pragma HLS INTERFACE m_axi bundle=gmem port=in2
#pragma HLS INTERFACE m_axi bundle=gmem port=out
  for(int i=0; i<size; i++)
    out[i] = in1[i] + in2[i]; }
```

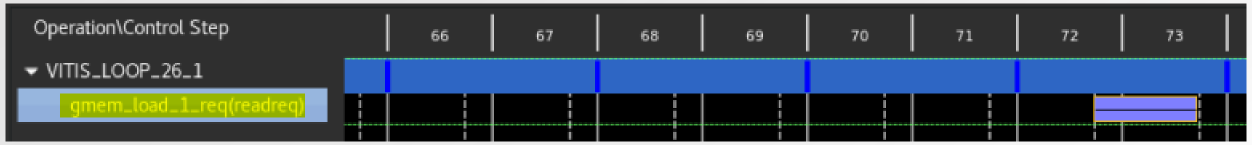


- C synthesis results:



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
example	II Violation	-	-	-	-	0	-	no	2	0	1607	1804	0
VITIS_LOOP_26_1	II Violation	-	?	?	75	2	-	yes	-	-	-	-	-

synthesis report



schedule viewer

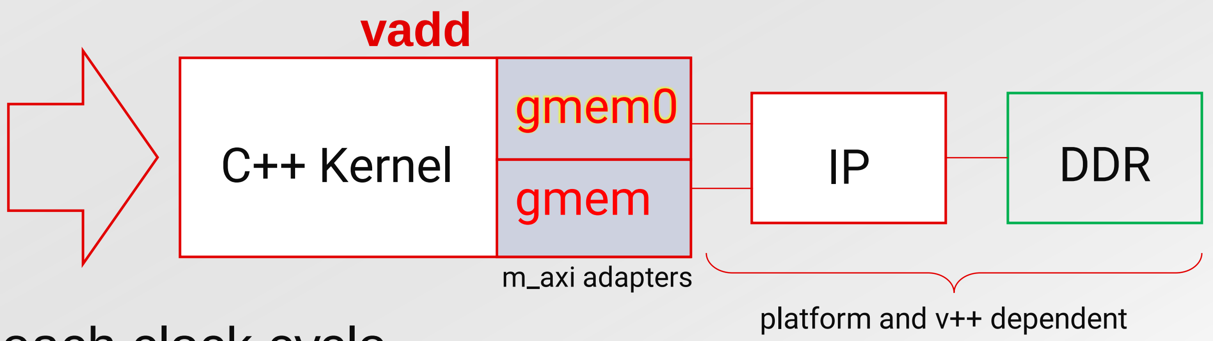
Throughput limited by I/Os
(gmem_load in schedule viewer,
we need more wires!)

XILINX

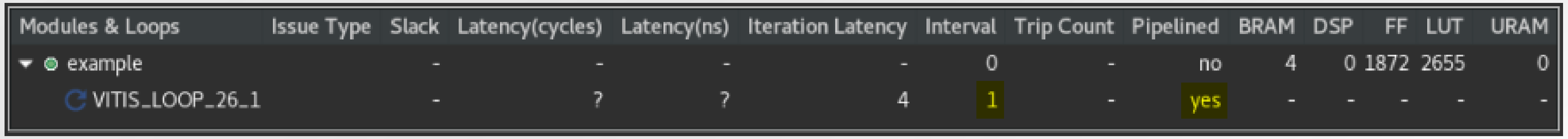
Interface Optimization for Pointers *(continued)*

Step 2: Add a new adapter…

```
...
#pragma HLS INTERFACE m_axi bundle=gmem0 port=in1
#pragma HLS INTERFACE m_axi bundle=gmem  port=in2
#pragma HLS INTERFACE m_axi bundle=gmem  port=out
...
```

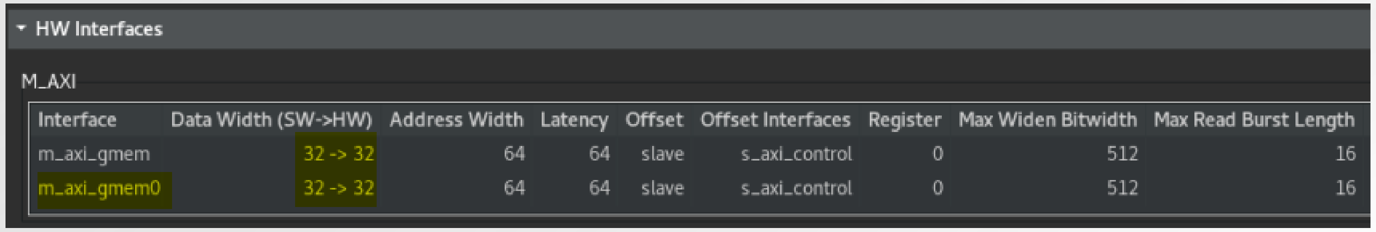


- Now II is 1, data can be written at each clock cycle



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
example		-	-	-	-	0	-	no	4	0	1872	2655	0
VITIS_LOOP_26_1		-	?	?	4	1	-	yes	-	-	-	-	-

- The physical interface interface is 32-bit when the platform can use 512-bit buses



HW Interfaces

M_AXI

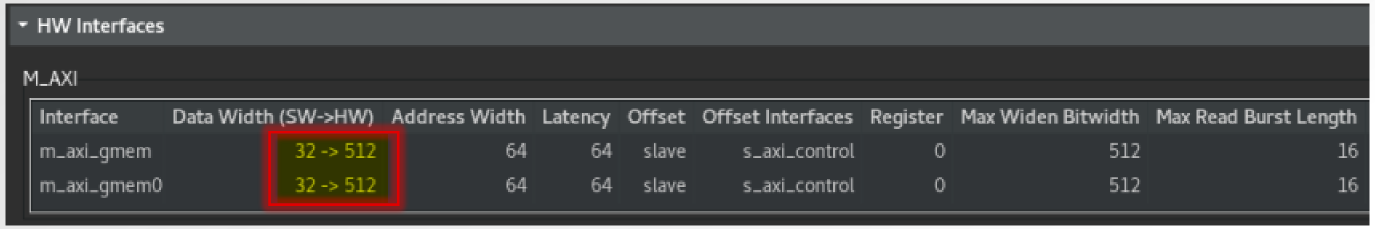
Interface	Data Width (SW->HW)	Address Width	Latency	Offset	Offset Interfaces	Register	Max Widen Bitwidth	Max Read Burst Length
m_axi_gmem	32 -> 32	64	64	slave	s_axi_control	0	512	16
m_axi_gmem0	32 -> 32	64	64	slave	s_axi_control	0	512	16

Interface Optimization for m_axi *(continued)*

Step 3: Provide a hint to the compiler to align data on 512-bit boundaries…

```
void vadd(const unsigned int *in1,
          const unsigned int *in2,
          unsigned int *out,
          int size)
{
#pragma HLS INTERFACE m_axi bundle=gmem0  port=in1
#pragma HLS INTERFACE m_axi bundle=gmem   port=in2
#pragma HLS INTERFACE m_axi bundle=gmem   port=out
  for(int i=0; i<(size/16)*16; i++)
    out[i] = in1[i] + in2[i]; }
```

: the simplest is to pass a fixed sized array… It will also need to be a multiple of 512-bit.



HW Interfaces

M_AXI

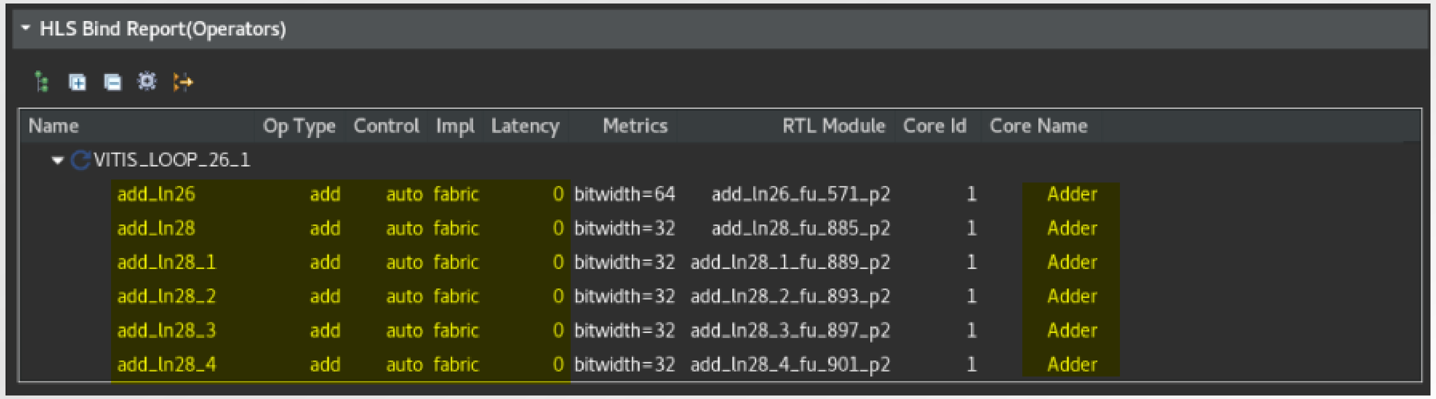
Interface	Data Width (SW->HW)	Address Width	Latency	Offset	Offset Interfaces	Register	Max Widen Bitwidth	Max Read Burst Length
m_axi_gmem	32 -> 512	64	64	slave	s_axi_control	0	512	16
m_axi_gmem0	32 -> 512	64	64	slave	s_axi_control	0	512	16

The bit width is now set to 512-bit…

Interface Optimization for m_axi *(continued)*

Step 4: Unroll by a factor of 16…

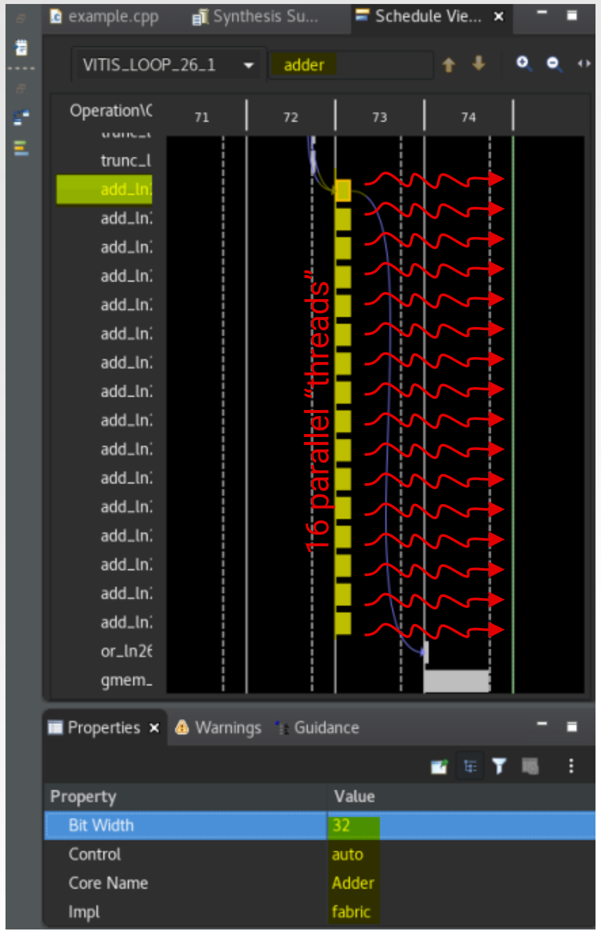
```
void vadd(const unsigned int *in1,
          const unsigned int *in2,
          unsigned int *out,
          int size) {
#pragma HLS INTERFACE m_axi bundle=gmem0 port=in1
#pragma HLS INTERFACE m_axi bundle=gmem  port=in2
#pragma HLS INTERFACE m_axi bundle=gmem  port=out
  for(int i=0; i<(size/16)*16; i++) {
    #pragma HLS UNROLL factor=16
    out[i] = in1[i] + in2[i]; }
}
```



HLS Bind Report(Operators)

Name	Op Type	Control	Impl	Latency	Metrics	RTL Module	Core Id	Core Name
VITIS_LOOP_26_1								
add_ln26	add	auto	fabric	0	bitwidth=64	add_ln26_fu_571_p2	1	Adder
add_ln28	add	auto	fabric	0	bitwidth=32	add_ln28_fu_885_p2	1	Adder
add_ln28_1	add	auto	fabric	0	bitwidth=32	add_ln28_1_fu_889_p2	1	Adder
add_ln28_2	add	auto	fabric	0	bitwidth=32	add_ln28_2_fu_893_p2	1	Adder
add_ln28_3	add	auto	fabric	0	bitwidth=32	add_ln28_3_fu_897_p2	1	Adder
add_ln28_4	add	auto	fabric	0	bitwidth=32	add_ln28_4_fu_901_p2	1	Adder

Bind report



Property	Value
Bit Width	32
Control	auto
Core Name	Adder
Impl	fabric

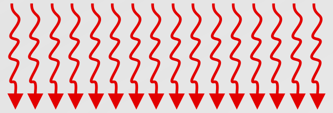
Scheduler view (filtering on "adder")

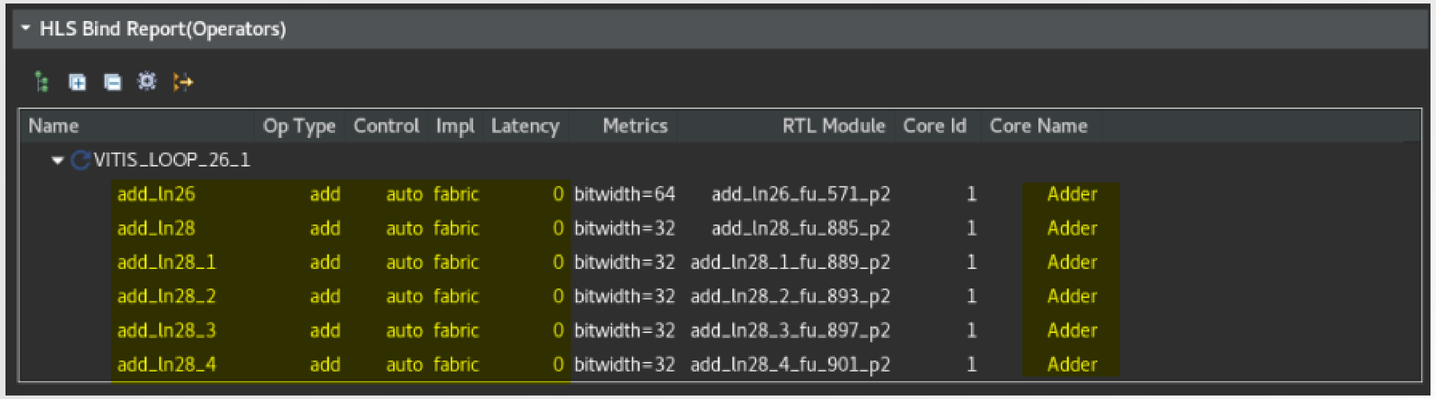
Interface Optimization for m_axi *(continued)*

Step 4: Unroll by a factor of 16…

```
void vadd(const unsigned int *in1,
          const unsigned int *in2,
          unsigned int *out,
          int size) {
#pragma HLS INTERFACE m_axi bundle=gmem0 port=in1
#pragma HLS INTERFACE m_axi bundle=gmem  port=in2
#pragma HLS INTERFACE m_axi bundle=gmem  port=out
  for(int i=0; i<(size/16)*16; i++) {
    #pragma HLS UNROLL factor=16
    out[i] = in1[i] + in2[i]; }
}
```

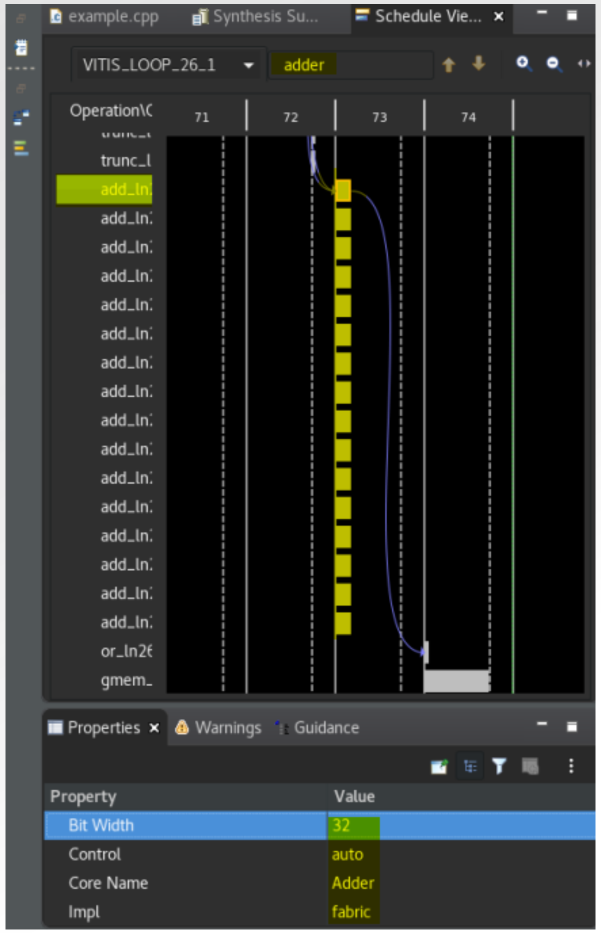
16 parallel “threads”





Name	Op Type	Control	Impl	Latency	Metrics	RTL Module	Core Id	Core Name
VITIS_LOOP_26_1								
add_ln26	add	auto	fabric	0	bitwidth=64	add_ln26_fu_571_p2	1	Adder
add_ln28	add	auto	fabric	0	bitwidth=32	add_ln28_fu_885_p2	1	Adder
add_ln28_1	add	auto	fabric	0	bitwidth=32	add_ln28_1_fu_889_p2	1	Adder
add_ln28_2	add	auto	fabric	0	bitwidth=32	add_ln28_2_fu_893_p2	1	Adder
add_ln28_3	add	auto	fabric	0	bitwidth=32	add_ln28_3_fu_897_p2	1	Adder
add_ln28_4	add	auto	fabric	0	bitwidth=32	add_ln28_4_fu_901_p2	1	Adder

Bind report



Property	Value
Bit Width	32
Control	auto
Core Name	Adder
Impl	fabric

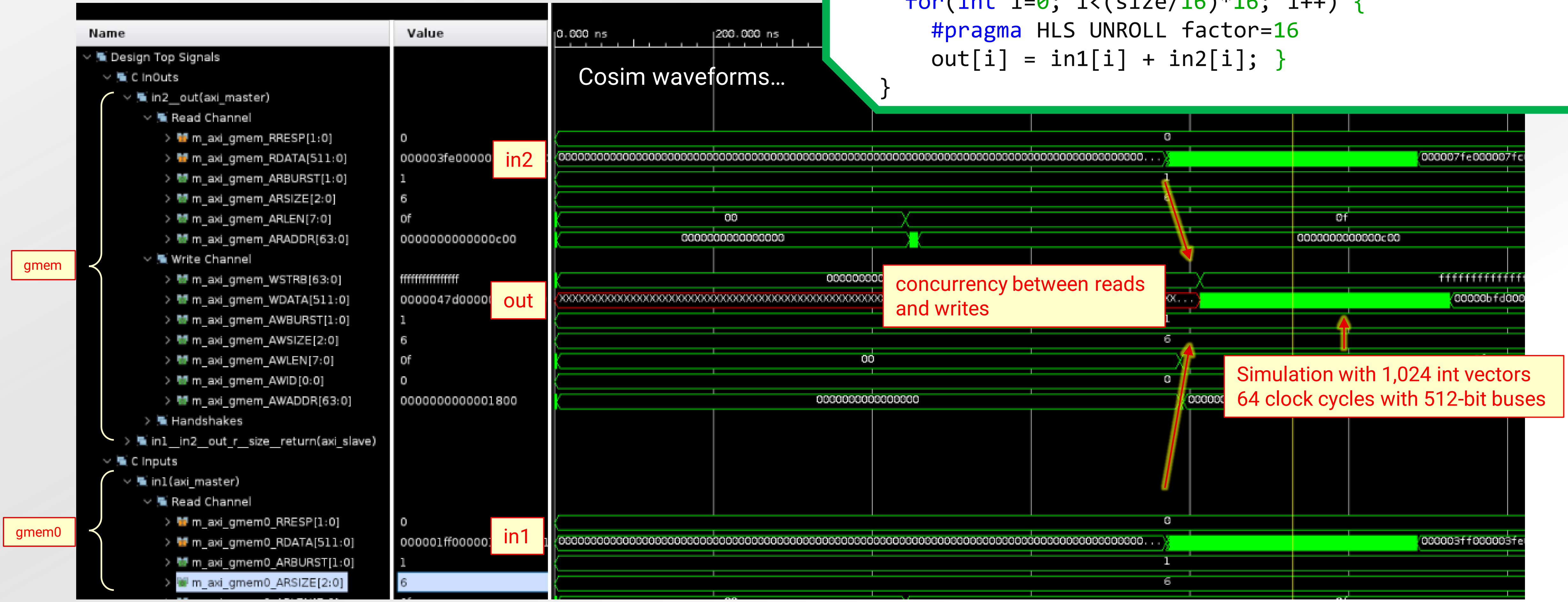
Scheduler view (filtering on “adder”)

Interface:

- Consider duplication adapters
- Use 512-bit alignment to improve throughput

“vector add” with 512-bit wide interfaces

```
void vadd(const unsigned int *in1,
          const unsigned int *in2,
          unsigned int *out, int size) {
#pragma HLS INTERFACE m_axi bundle=gmem0 port=in1
#pragma HLS INTERFACE m_axi bundle=gmem  port=in2
#pragma HLS INTERFACE m_axi bundle=gmem  port=out
  for(int i=0; i<(size/16)*16; i++) {
    #pragma HLS UNROLL factor=16
    out[i] = in1[i] + in2[i]; }
}
```



```
Name                                          Value
Design Top Signals
  C InOuts
    in2__out(axi_master)
      Read Channel
        m_axi_gmem_RRESP[1:0]                 0
        m_axi_gmem_RDATA[511:0]               000003fe00000
        m_axi_gmem_ARBURST[1:0]               1
        m_axi_gmem_ARSIZE[2:0]                6
        m_axi_gmem_ARLEN[7:0]                 0f
        m_axi_gmem_ARADDR[63:0]               0000000000000c00
      Write Channel
        m_axi_gmem_WSTRB[63:0]                ffffffffffffffff
        m_axi_gmem_WDATA[511:0]               0000047d00000
        m_axi_gmem_AWBURST[1:0]               1
        m_axi_gmem_AWSIZE[2:0]                6
        m_axi_gmem_AWLEN[7:0]                 0f
        m_axi_gmem_AWID[0:0]                  0
        m_axi_gmem_AWADDR[63:0]               0000000000001800
      Handshakes
    in1__in2__out_r__size__return(axi_slave)
  C Inputs
    in1(axi_master)
      Read Channel
        m_axi_gmem0_RRESP[1:0]                0
        m_axi_gmem0_RDATA[511:0]              000001ff00000
        m_axi_gmem0_ARBURST[1:0]              1
        m_axi_gmem0_ARSIZE[2:0]               6
```

concurrency between reads and writes

Simulation with 1,024 int vectors
64 clock cycles with 512-bit buses

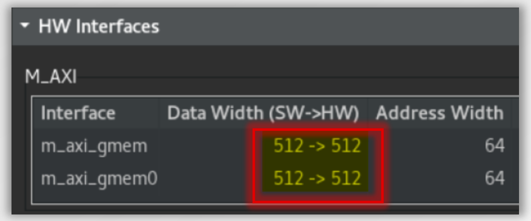
XILINX.

Interface Optimization for m_axi with Vector Types

With vector data types…

```
typedef unsigned int foo __attribute__((vector_size(64)));

void vadd(const foo *in1,
          const foo *in2,
          foo *out,
          int size) {
#pragma HLS INTERFACE m_axi bundle=gmem0 port=in1
#pragma HLS INTERFACE m_axi bundle=gmem  port=in2
#pragma HLS INTERFACE m_axi bundle=gmem  port=out
  for(int i=0; i< size; i++)
    out[i] = in1[i] + in2[i];
}
```

- Simpler coding style
- Explicit widening
- Relies on vector types

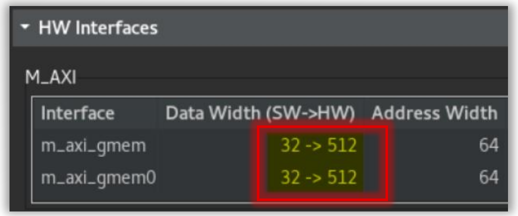


Interface	Data Width (SW->HW)	Address Width
m_axi_gmem	512 -> 512	64
m_axi_gmem0	512 -> 512	64

… and without

```
void vadd(const unsigned int *in1,
          const unsigned int *in2,
          unsigned int *out,
          int size) {
#pragma HLS INTERFACE m_axi bundle=gmem0 port=in1
#pragma HLS INTERFACE m_axi bundle=gmem  port=in2
#pragma HLS INTERFACE m_axi bundle=gmem  port=out
  for(int i=0; i<(size/16)*16; i++) {
    #pragma HLS UNROLL factor=16
    out[i] = in1[i] + in2[i]; }
}
```

- Preserves function signature
- Needs “unroll” pragma
- Relies on the “widen” option



Interface	Data Width (SW->HW)	Address Width
m_axi_gmem	32 -> 512	64
m_axi_gmem0	32 -> 512	64

XILINX

Traveler Salesman Problem

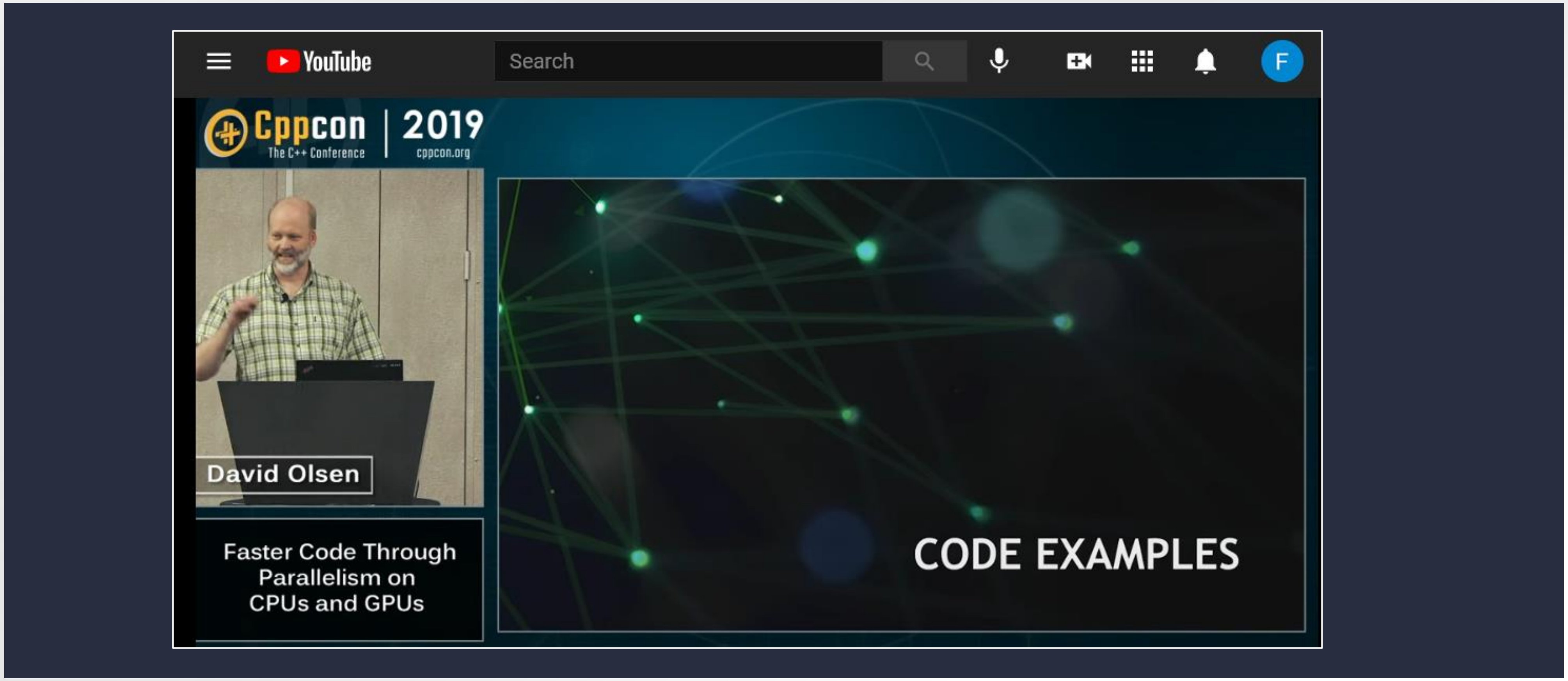
Travelling Salesman Problem (TSP)

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- The algorithm increases superpolynomially with the number of cities
- The most direct solution is to try all permutations to see which one is cheapest
  - Runtime for this approach lies within a polynomial factor of O (n!)

TSP – Benchmarks

- Conference data (Cppcon 2019)
  - Faster Code Through Parallelism on CPUs and GPUs
  - URL: https://www.youtube.com/watch?v=cbbKEAWf1ow : TSP algorithm for 13 cities


YouTube
Search
F
Cppcon The C++ Conference
2019 cppcon.org
David Olsen
Faster Code Through Parallelism on CPUs and GPUs
CODE EXAMPLES

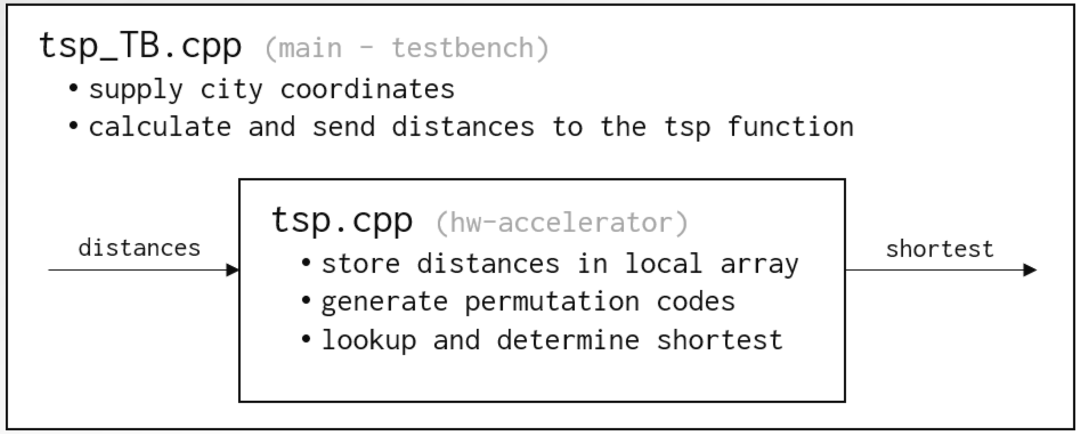
TSP – Benchmarks

- Conference data (Cppcon 2019)

Coding Style	Notes	Speedup (reference: 22min40s)
Sequential code reference	custom compiler PGI (22min40s) GCC 6.2 (27min41s)	1x 0.82x
C++ threads (machine with 40 physical cores)	with PGI with GCC	43.7x 30.6x
OpenMP (with pragma)	same as sequential code and GCC	32.1x
OpenACC (manual reduction)	manual reduction: X30.5 GPU (1.25 seconds)	30.5x 1073x
CUDA	GPU (1.1 seconds)	1248x
Kokkos	OpenMP backend Cuda backend Cuda backend + patch (compute intensive)	33.4x 384x 1241x
C++17	CPU target GPU target (1 second)	33.7x 1355x
C++ HLS	sequential with PIPELINE	(next slides)

Overall Approach – FPGA Implementation

- The distances are sent from the host
  - Loaded in global memory and accessed in the kernel via the m_axi adapter
- Critical for acceleration…
  - Implement an efficient permutation algorithm
  - Run lookups with on-chip memories


tsp_TB.cpp (main - testbench)
• supply city coordinates
• calculate and send distances to the tsp function
distances
tsp.cpp (hw-accelerator)
• store distances in local array
• generate permutation codes
• lookup and determine shortest
shortest

Efficient Permutation – Factoradics!

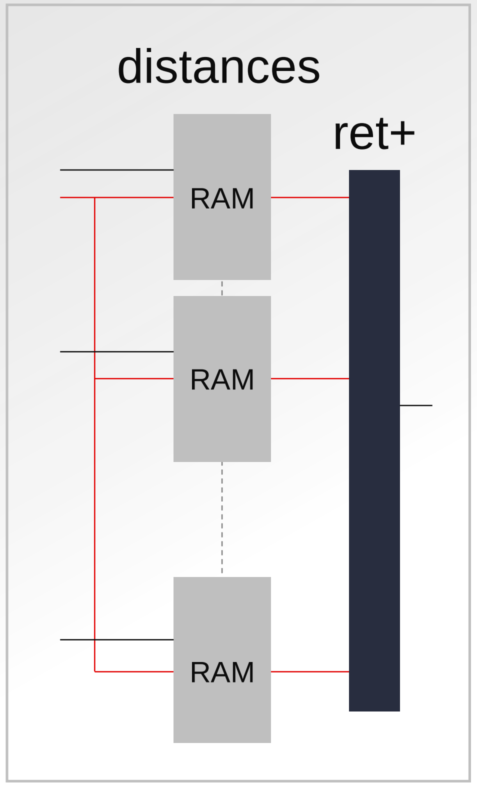
```
auto compute(const unsigned long int i_, const uint16_t distances[N][N])
{
  #pragma HLS INLINE

  unsigned long int i = i_;
  int perm[N] = {0};

  for (int k = 0; k < N; ++k) {
    perm[k] = i / factorial(N - 1 - k);
    i = i % factorial(N - 1 - k);
  }

  for (char k = N - 1; k > 0; --k)
    for (char j = k - 1; j >= 0; --j)
      perm[k] += (perm[j] <= perm[k]);

  cout << "getDistance: "<< getDistance(perm,distances) << endl;
  return getDistance(perm,distances);
}
```

1. Represent the index in its factorial base (first loop)
2. Create a permutation array with the factorial representation (second loop)

On-Chip Memory Lookups

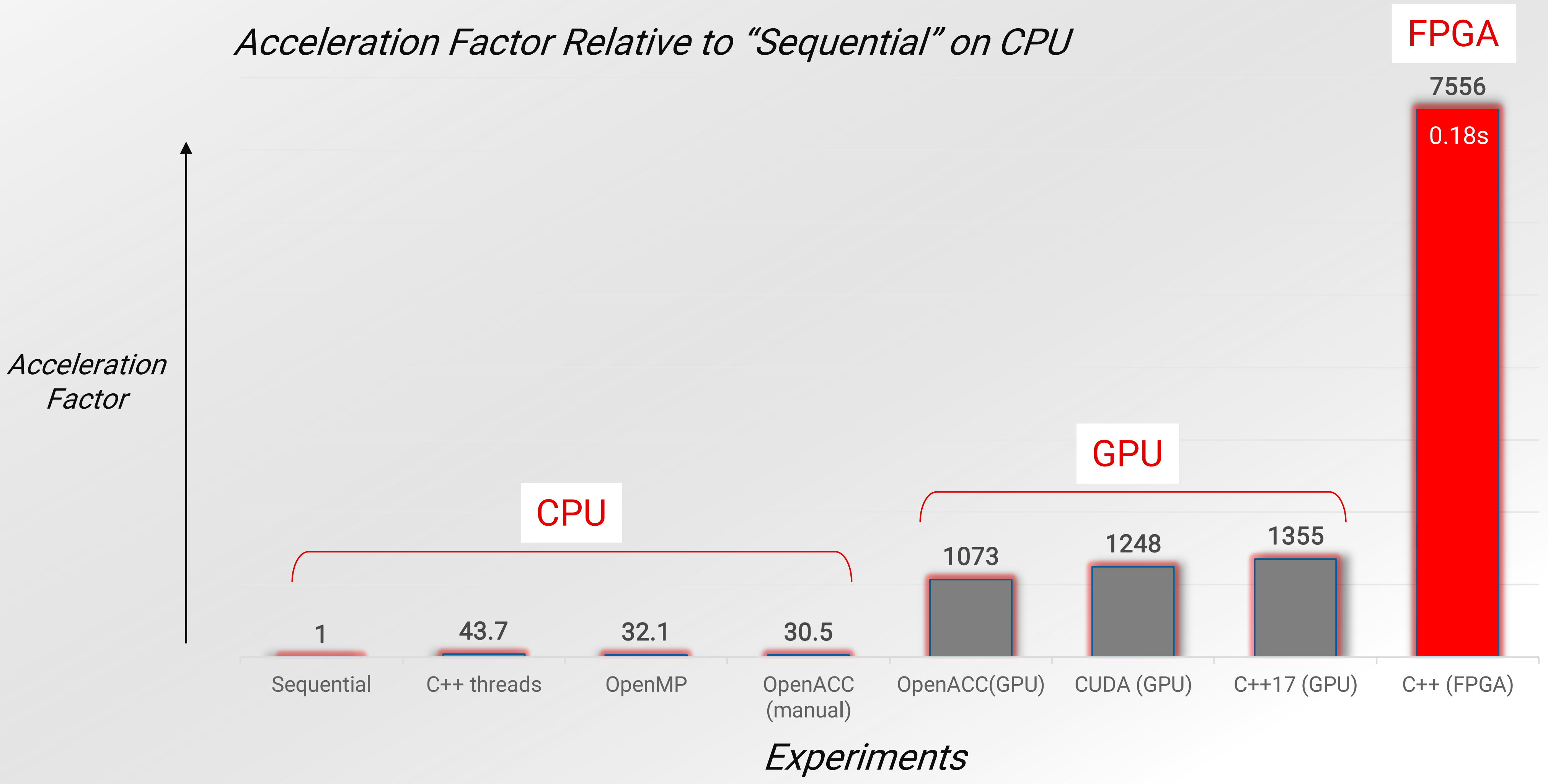
```
template<typename T>
unsigned int getDistance(const T perm[N], const uint16_t distances[N][N])
{
  unsigned int ret = 0;
  for(int i = 0; i < N-1; ++i)
    ret += distances[perm[i]][perm[i+1]];
  return ret;
}
```

- Lookups with on-chip memories
  - Enough ports all necessary reads at each clock cycle
- Distances calculated at each clock cycle



RAM 1WnR

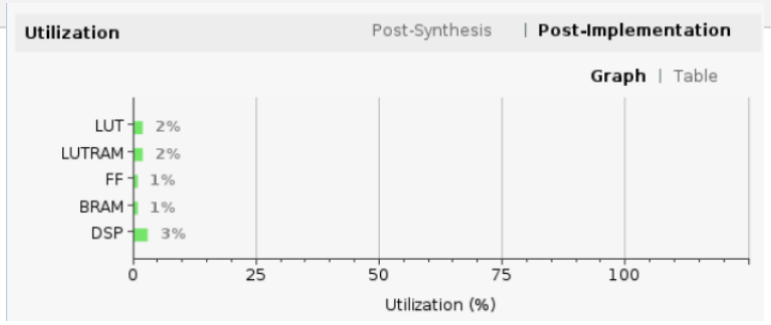
Results – TSP with 13 Cities



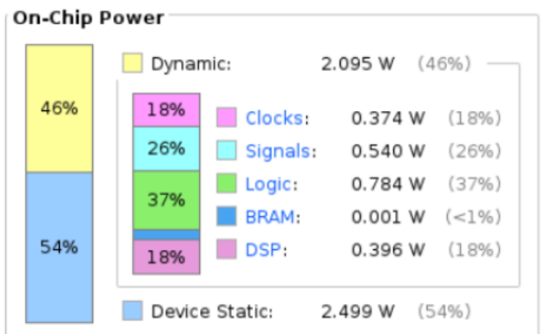
Results – TSP with 13 Cities

- 7,500x speedup
  - 2% LUTs(*)
  - 2.1 W (*) of dynamic power @300MHz

0.18s



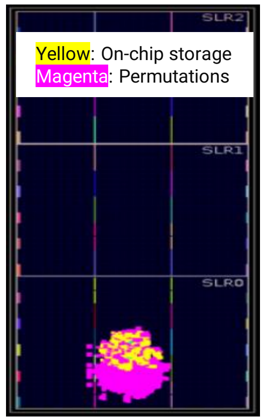
Utilization



On-Chip Power

Dynamic:	2.095 W	(46%)
Clocks:	0.374 W	(18%)
Signals:	0.540 W	(26%)
Logic:	0.784 W	(37%)
BRAM:	0.001 W	(<1%)
DSP:	0.396 W	(18%)
Device Static:	2.499 W	(54%)

Power



Floorplan (VU9P)

(*): Based on UltraScale+ VU9P

Summary and Wrap-up

Summary

- Vitis enables C++ applications
- Directives parallelize the code implementation
- Compute intensive algorithms mapped effectively onto FPGAs
  - … thanks to on-chip RAM, micro-arch restructuring and efficient data types

Resources

- Take a test drive! Try Vitis in the cloud or get an acceleration card!
- Refer to the Vitis getting started examples here (including C++ kernels):
  - https://github.com/Xilinx/Vitis_Accel_Examples
- Point to the Vitis In-Depth Tutorials repo:
  - https://xilinx.github.io/Vitis-Tutorials/master/docs/index.html
- Check out the Xilinx Developer Site!
  - Find tutorials, onboarding, application examples, and documentation to get started
  - https://developer.xilinx.com
- Download Vitis from Xilinx.com today!
  - https://www.xilinx.com/support/download.html

XILINX.

XILINX®

Thank you!