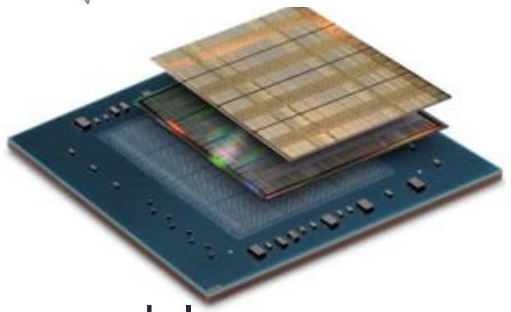
FCUDA-SoC: Platform Integration for Field-Programmable SoC with the CUDA-to-FPGA Compiler

Tan Nguyen[1], Swathi Gurumani[1], Kyle Rupnow[1], Deming Chen[2]

[1] Advanced Digital Sciences Center, Singapore
{tan.nguyen, swathi.g, k.rupnow}@adsc.com.sg
[2] Electrical and Computer Engineering,
University of Illinois at Urbana-Champaign, USA
dchen@illinois.edu

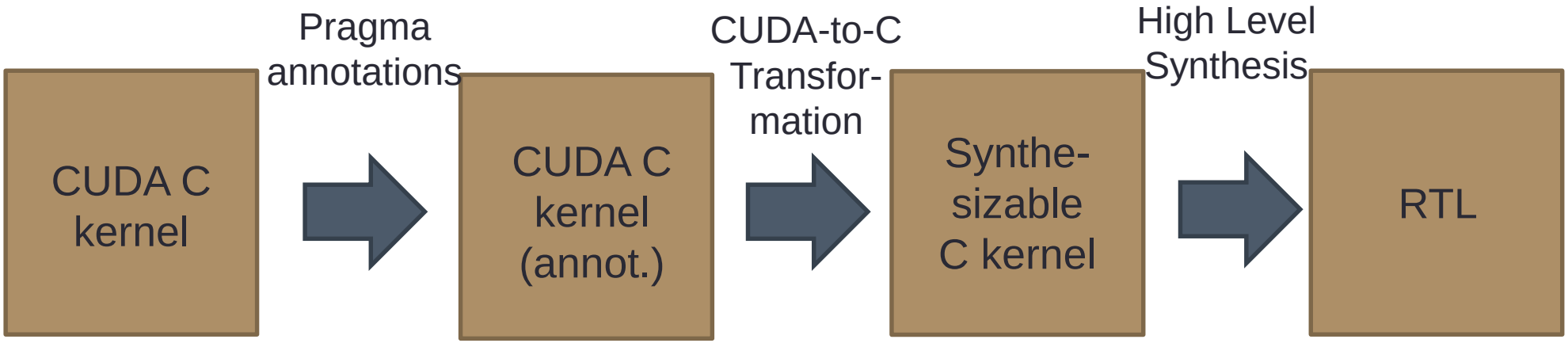
Motivations



- GPUs and FPGAs: accelerators in HPC
  - NVIDIA GPUs (Google, Baidu, Supercomputers, …)
  - FPGAs (Microsoft Bing search, …)
- NVIDIA GPUs
  - Massive parallelism thanks to the CUDA parallel programming model
  - Power hungry
- FPGAs
  - Low energy consumption, high flexibility
  - Low programming abstraction
  - → High Level Synthesis
  - But parallelism extraction may be limited

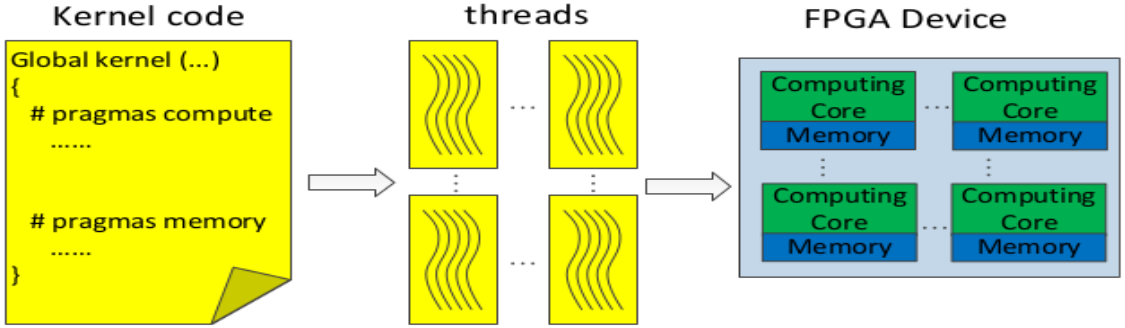
FCUDA History

- FCUDA Introduction
  - CUDA-to-FPGA compiler [ASAP'09], [FCCM'11], [DAC'13]
  - Front-end: CUDA-to-C (Cetus + MCUDA)
  - Back-end: C-to-RTL (High Level Synthesis tool --Vivado HLS)
- Source-to-source transformation
- Does not employ NVIDIA compiler infrastructure (NVCC, NVVM IR, ...)



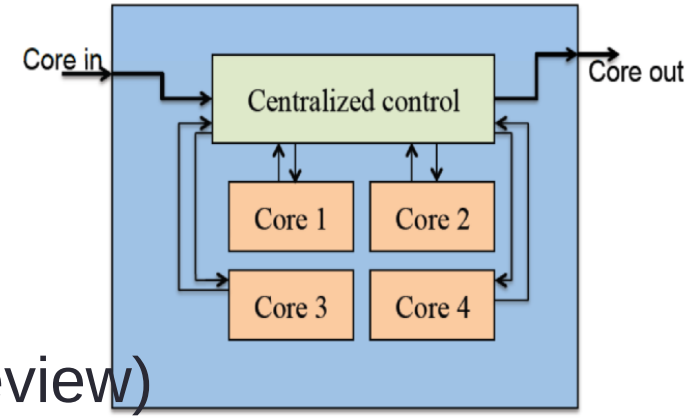
FCUDA History

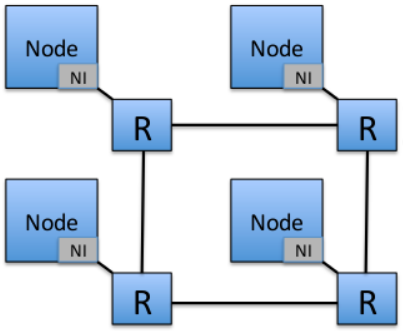
- FCUDA Introduction
  - CUDA-to-FPGA compiler [ASAP'09], [FCCM'11], [DAC'13]
  - Front-end: CUDA-to-C (Cetus + MCUDA)
  - Back-end: C-to-RTL (High Level Synthesis tool --Vivado HLS)
- Source-to-source transformation
- Does not employ NVIDIA compiler infrastructure (NVCC, NVVM IR, ...)
- Why FCUDA?
  - Enable CUDA kernel's execution on FPGA
  - CUDA's SIMD programming model is attractive to FPGA



FCUDA System Backend

- Prior FCUDA works focused on building a “prototype”
  - Single top-level function with centralized control of cores
  - No discussion on a full system integration of cores vs. external memory
- System implementation works:
  - FCUDA NoC: Mesh-based NoC [TVLSI’15]
  - FCUDA HB: Hierarchical AXI Bus (under review)
  - FCUDA SoC (this work)

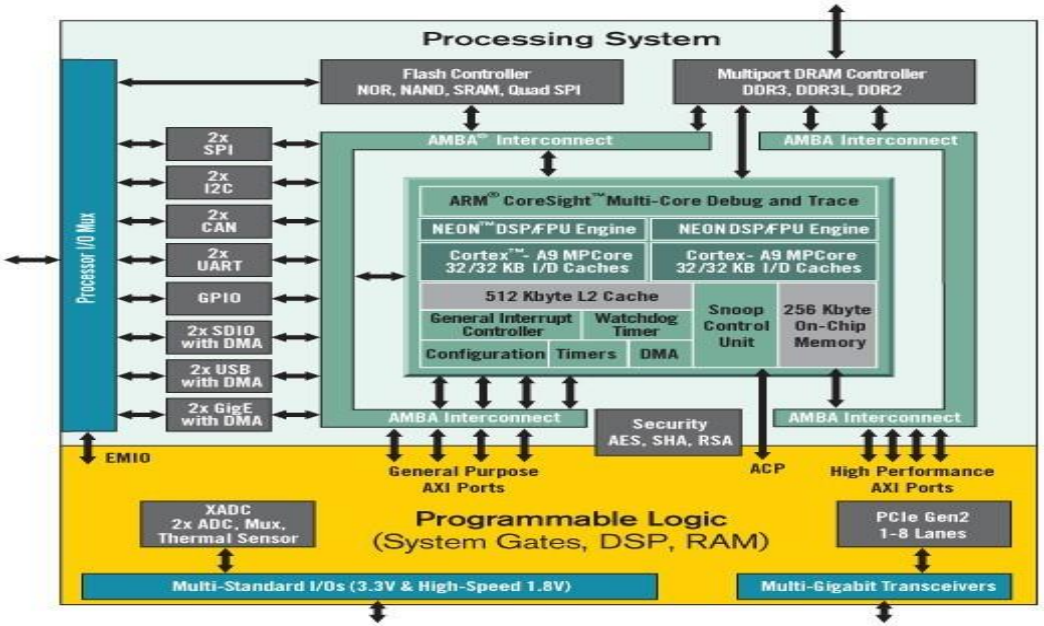




Why SoC FPGA?

- ARM core(s) tightly coupled with FPGA fabric via AXI interconnect
- ARM cores for control, FPGA for accelerator
- Communication supported by Xilinx software (drivers)

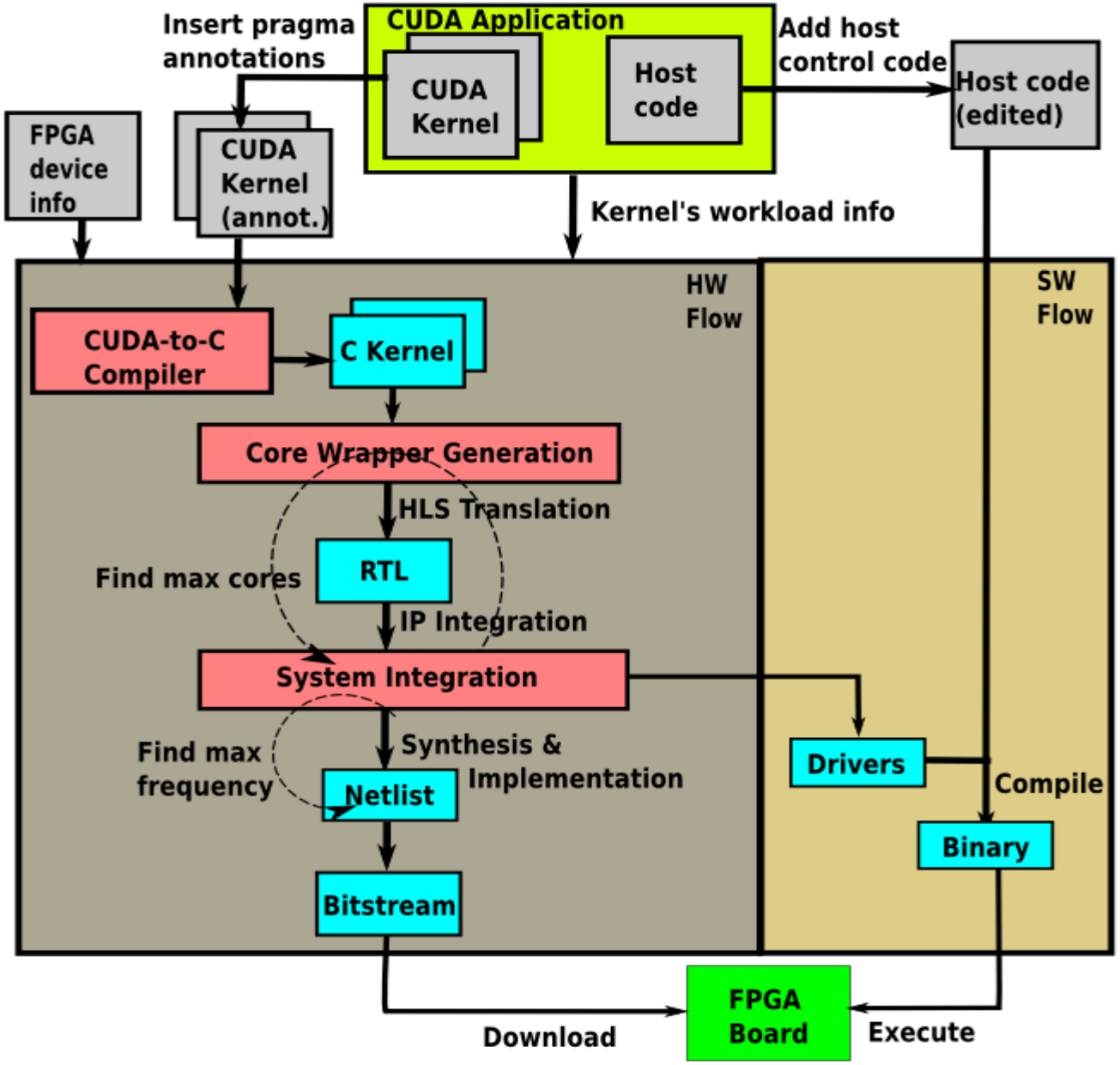
→ a suitable candidate for full system implementation!



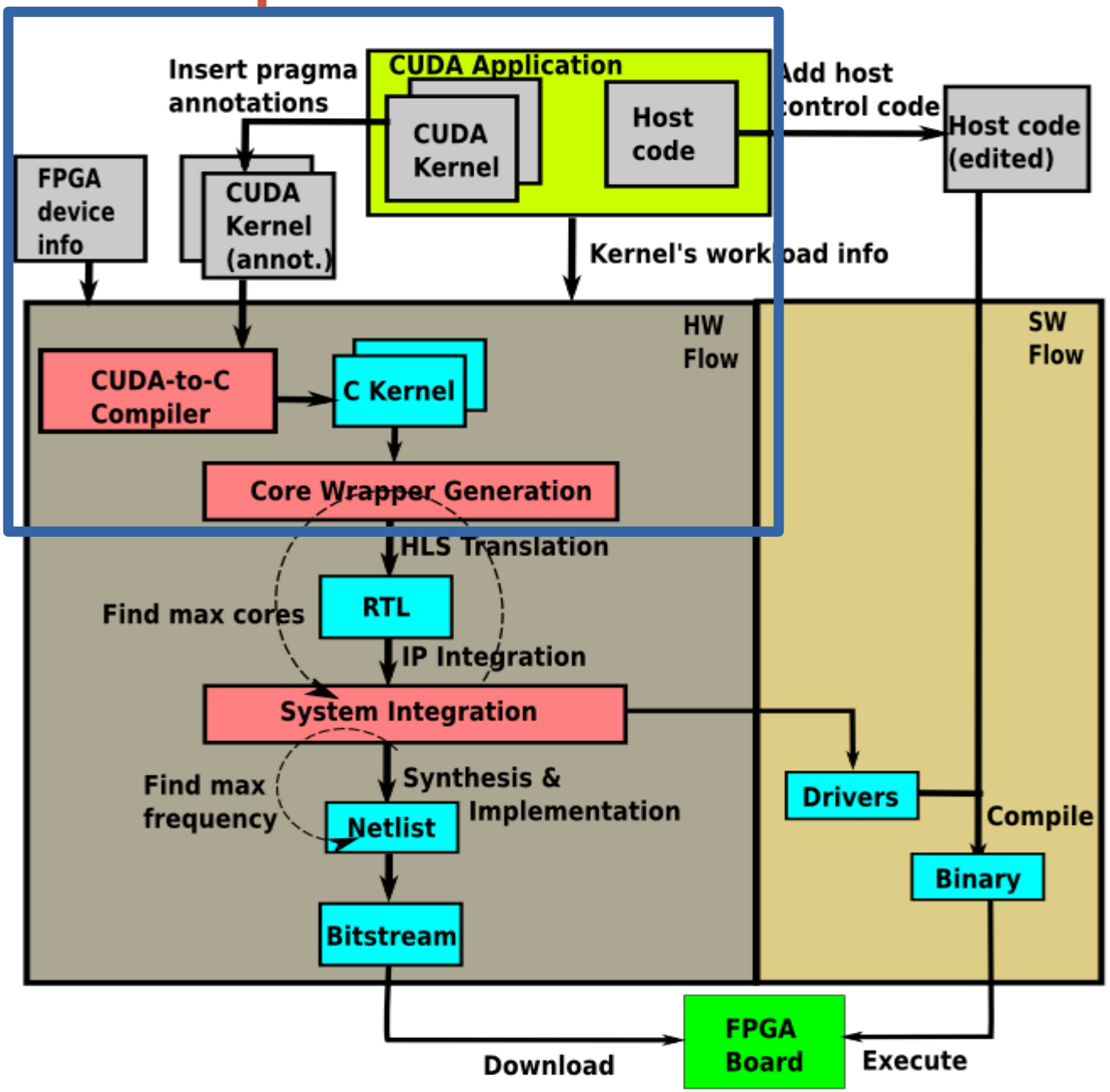
FCUDA Open-source

- Available to download at: http://dchen.ece.illinois.edu/tools.html
- The open-source software includes:
  - FCUDA CUDA-to-C compiler
  - FCUDA Benchmarks: set of benchmarks for testing FCUDA
  - FCUDA SoC tool flow: scripts for the automation of FCUDA system on SoC platform
- Future plan:
  - FCUDA NoC, FCUDA HB, Design Space Exploration framework

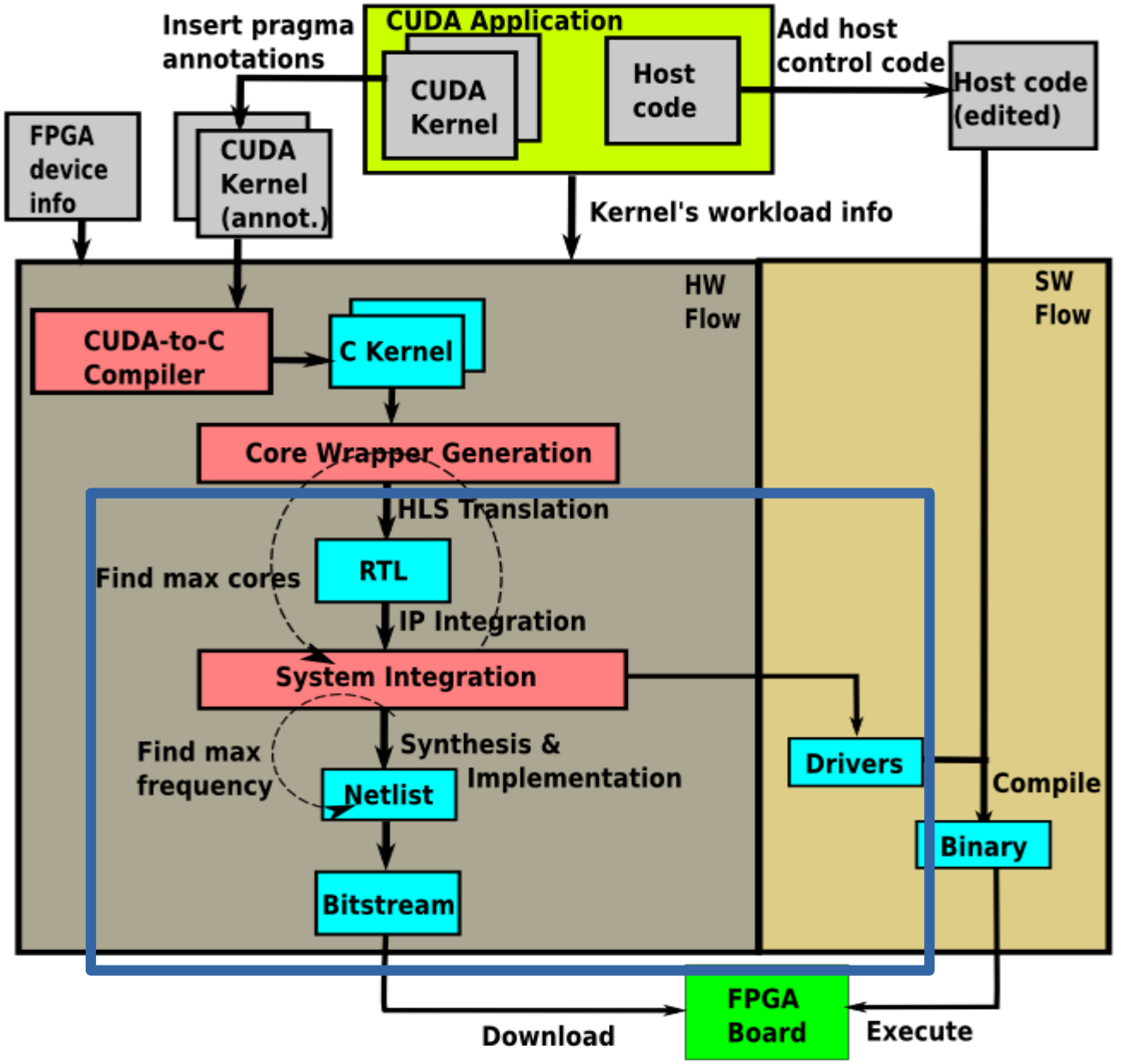
FCUDA SoC – An overview flow



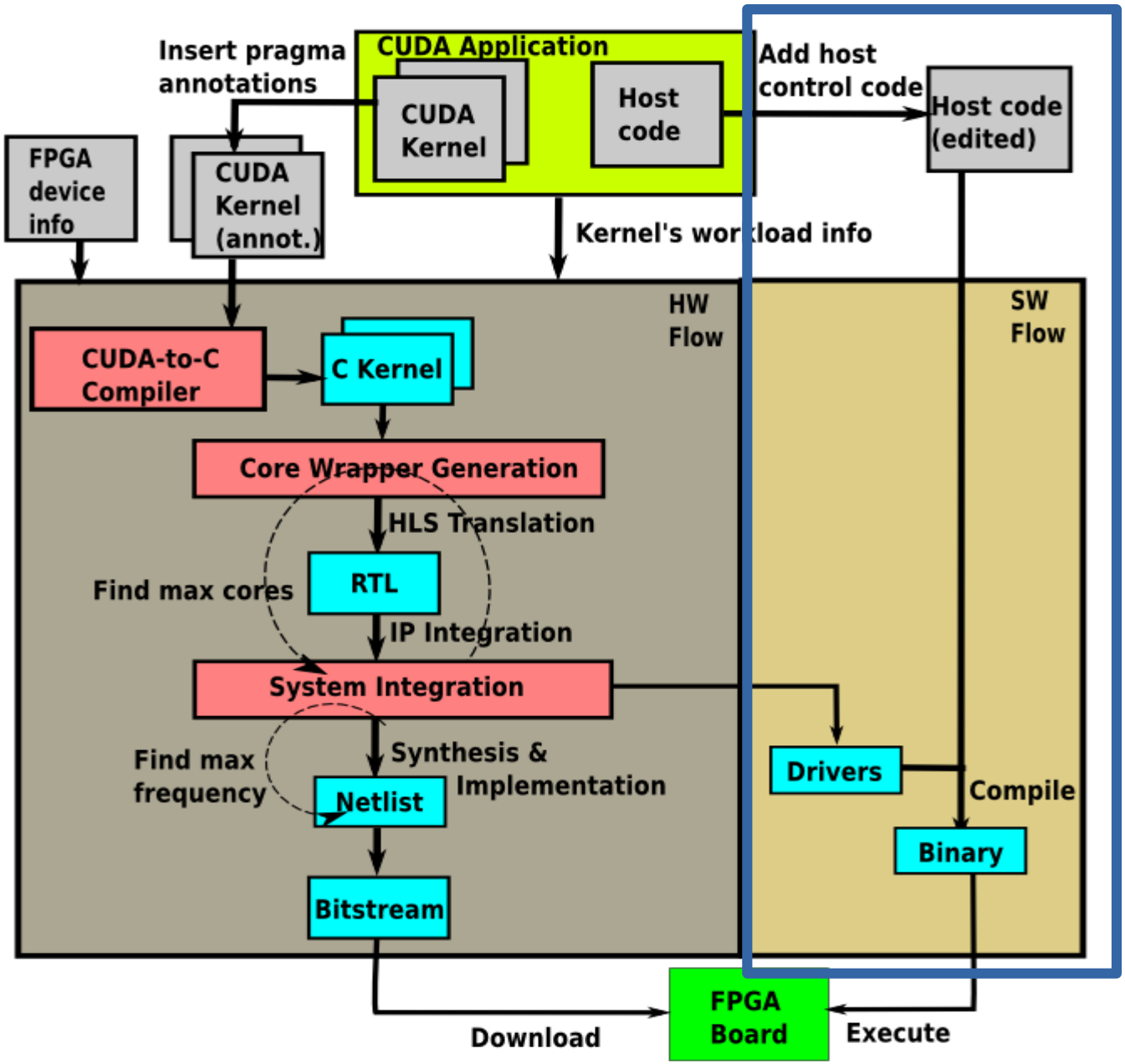
FCUDA SoC – Preproc. flow



FCUDA SoC – HW flow



FCUDA SoC – SW flow



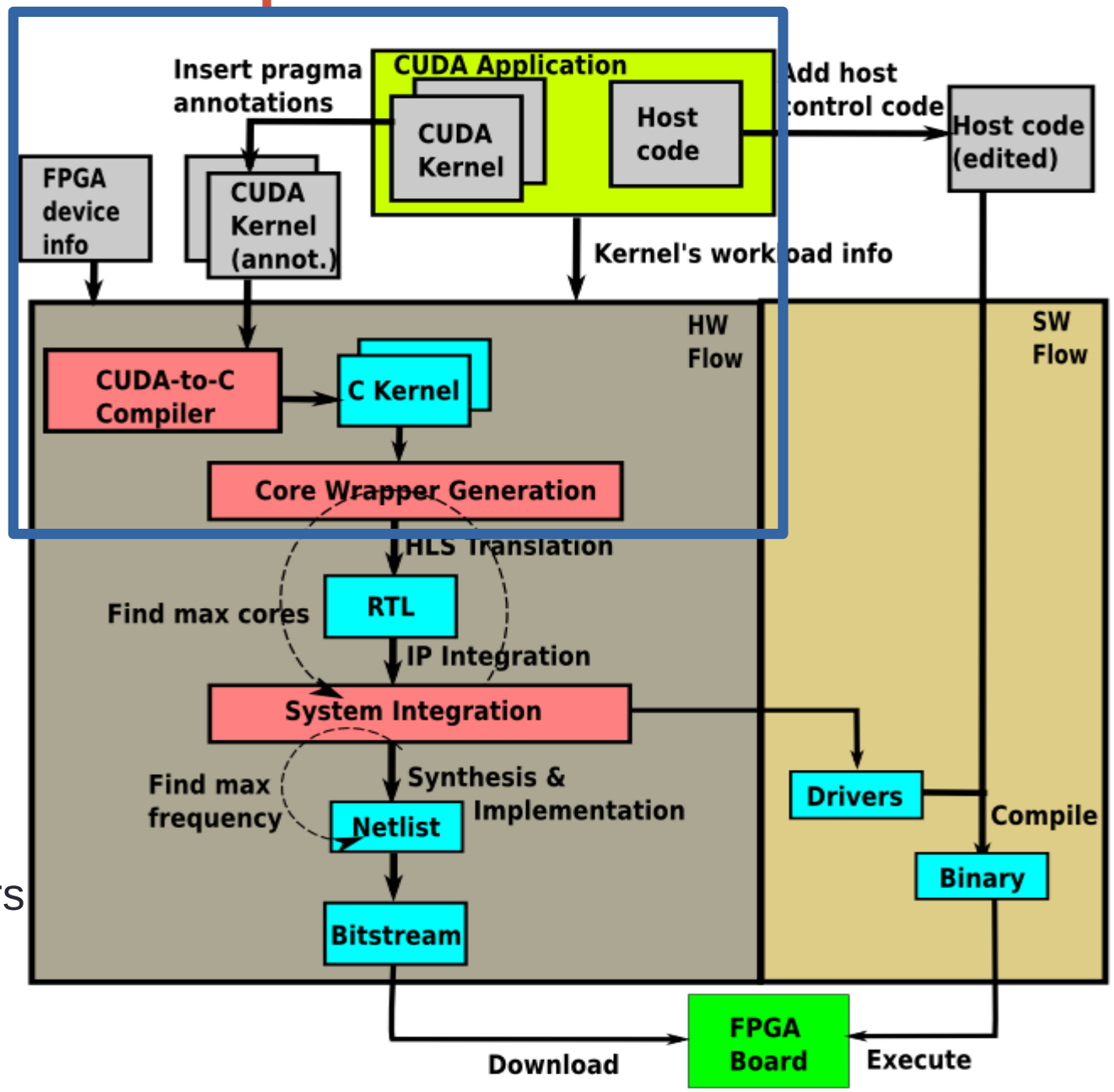
FCUDA SoC – Preproc. flow

1. Separation of CUDA kernels from Host code

2. Pragma annotations
   - FCUDA pragmas
   - Vivado HLS interface pragmas

3. CUDA-to-C Transformation

4. Generation of wrappers



FCUDA SoC – Pragma annotations

	Pragmas	Purpose
CUDA code	#pragma FCUDA TRANSFER	Translate assignment statement to memcpy()
CUDA code	#pragma FCUDA COMPUTE	Insert thread loops at synchronization points
Gen. C code	#pragma HLS INTERFACE ap_bus port=X	Apply ap_bus protocol to pointer X
Gen. C code	#pragma HLS RESOURCE core=AXI4M variable=X	Create AXI4M bus wrapper for pointer X
Gen. C code	#pragma HLS INTERFACE ap_none register port=x	Apply ap_register protocol to scalar x
Gen. C code	#pragma HLS RESOURCE core=AXI4LiteS variable=x	Create AXI4LiteS bus wrapper for scalar x
Gen. C code	#pragma HLS RESOURCE core=AXI4LiteS variable=return	Generate driver files for the HLS IP

FCUDA SoC – An Example

```
#define BLOCK_SIZE 256
__global__ void vecAdd(int *A, int *B, int *C, int x, int y)
{
  __shared__ int local_A[BLOCK_SIZE];
  __shared__ int local_B[BLOCK_SIZE];
  __shared__ int local_C[BLOCK_SIZE];

  //fetch

  #pragma FCUDA TRANSFER name=fetch begin dir=0 size=[BLOCK_SIZE|BLOCK_SIZE] pointer=[A|B]

  local_A[threadIdx.x] = A[threadIdx.x + blockIdx.x * blockDim.x];
  local_B[threadIdx.x] = B[threadIdx.x + blockIdx.x * blockDim.x];

  #pragma FCUDA TRANSFER name=fetch end dir=0 size=[BLOCK_SIZE|BLOCK_SIZE] pointer=[A|B]

  __syncthreads();

  //compute

  #pragma FCUDA COMPUTE name=compute begin

  local_C[threadIdx.x] = x * local_A[threadIdx.x] + y * local_B[threadIdx.x];

  #pragma FCUDA COMPUTE name=compute end

  __syncthreads();

  //write

  #pragma FCUDA TRANSFER name=write begin dir=1 size=[BLOCK_SIZE] pointer=[C]

  C[threadIdx.x + blockIdx.x * blockDim.x] = local_C[threadIdx.x];

  #pragma FCUDA TRANSFER name=write end dir=1 size=[BLOCK_SIZE] pointer=[C]

}
```

$$C = A * x + B * y$$

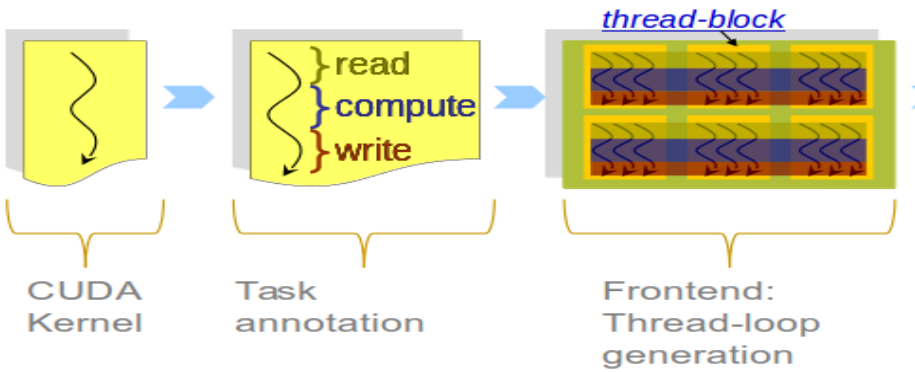
FCUDA SoC – CUDA-to-C transformation

```
void fetch(...)
{
  memcpy();
  memcpy();
}

void compute(...)
{
  for (threadIdx.x = 0; threadIdx.x < blockDim.x; threadIdx.x++) { ... }
}

void write(...)
{
  memcpy();
}

void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim,
                    int A_addr, int B_addr, int C_addr, int num_cores, int core_id)
{
    for (blockIdx.x = core_id; blockIdx.x < gridDim.x; blockIdx.x+=num_cores) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```



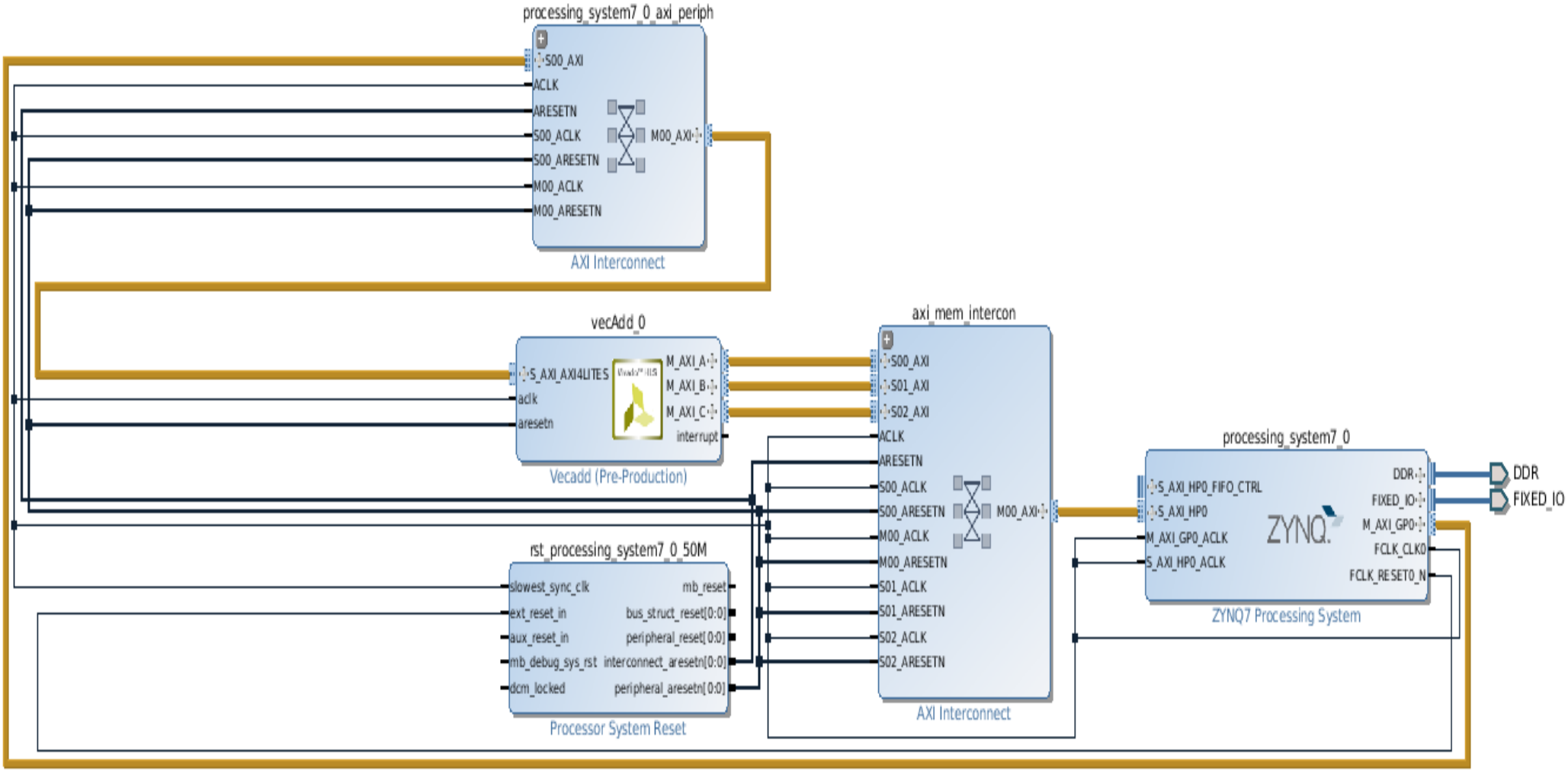
Workload distribution

FCUDA SoC – An Example

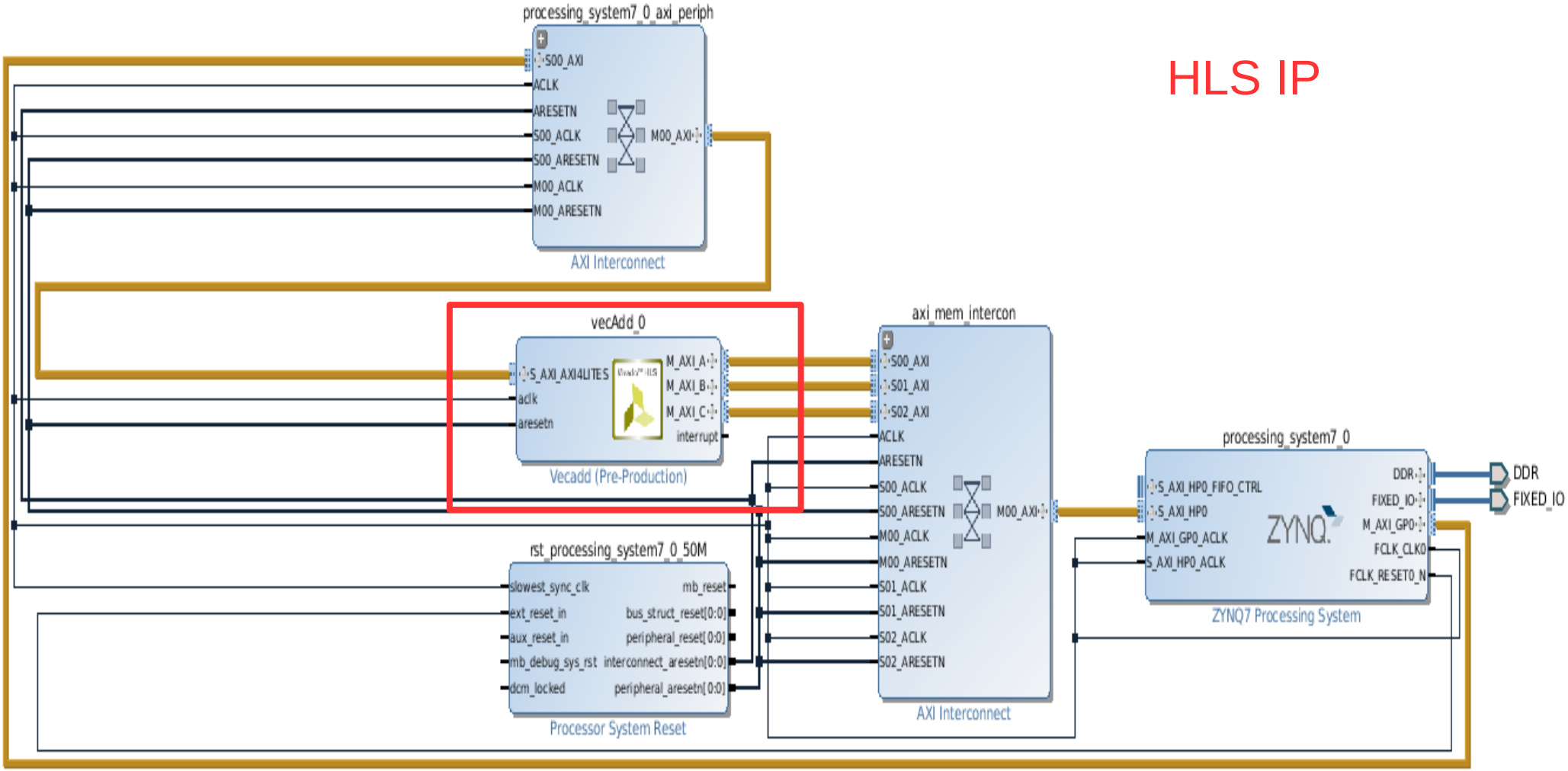
- Communication interface
  - pointers: ap_bus – read/write data to/from memory
  - scalars: register – static during core's execution

```
void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim)
{
    #pragma HLS interface ap_bus port=A
    #pragma HLS resource core=AXI4M variable=A
    #pragma HLS interface ap_none register port=x
    #pragma HLS resource core=AXI4LiteS variable=x
    ...
    #pragma HLS resource core=AXI4LiteS variable=return
    for (blockIdx.x = 0; blockIdx.x < gridDim.x; blockIdx.x++) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```

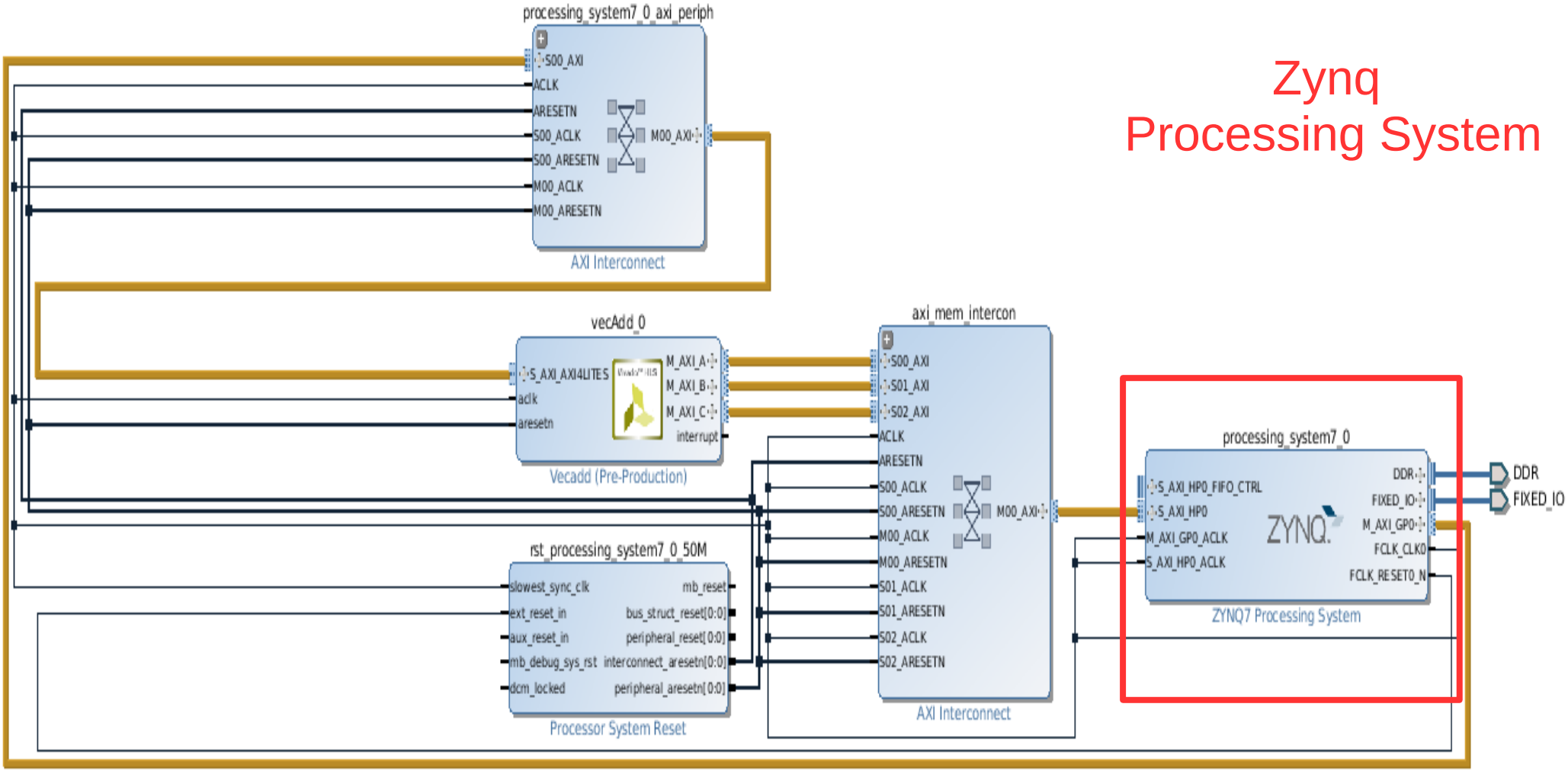
FCUDA SoC – Single core design



FCUDA SoC – Single core design

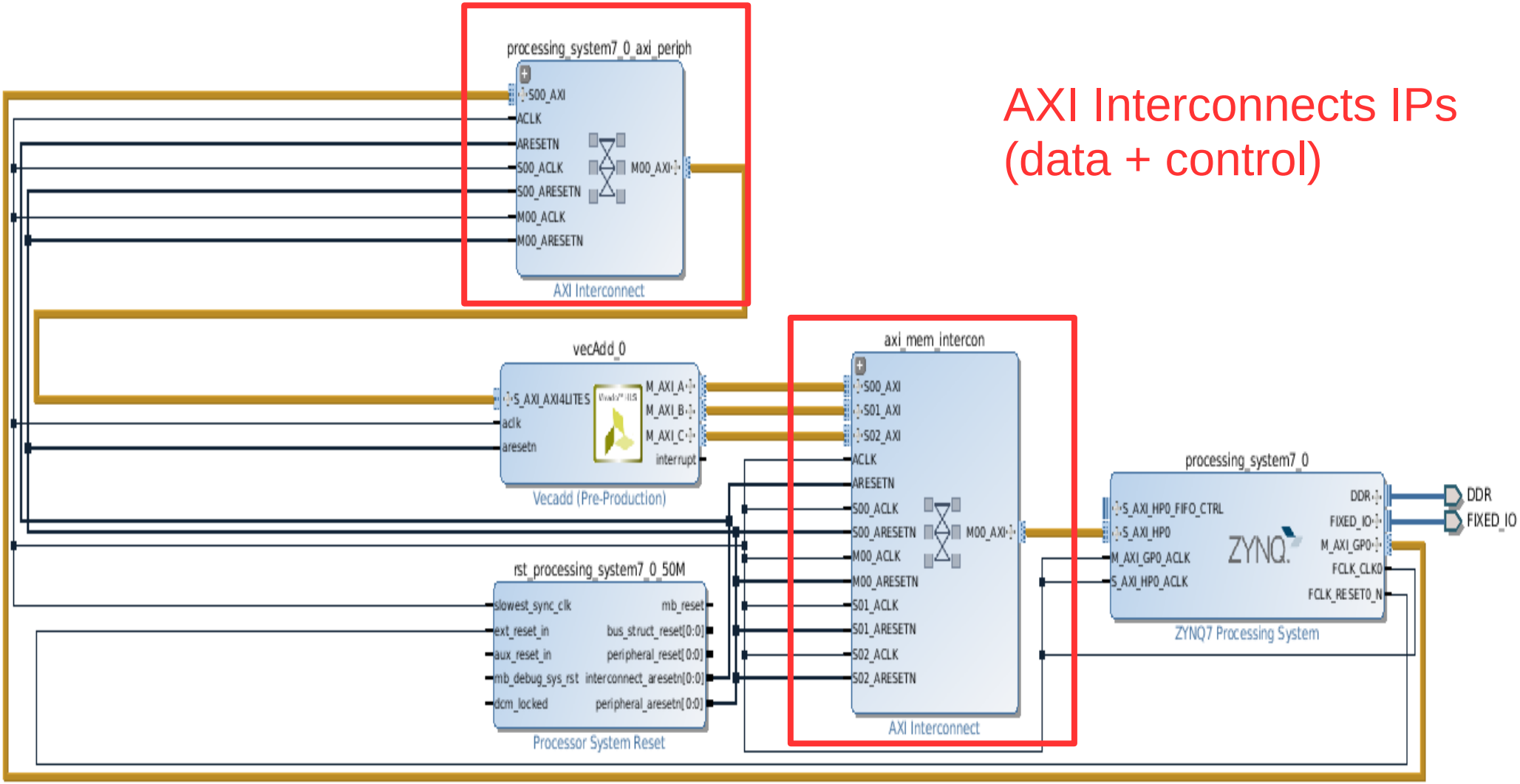


FCUDA SoC – Single core design



Zynq
Processing System

FCUDA SoC – Single core design



AXI Interconnects IPs
(data + control)

FCUDA SoC – Multi-core design

- Fixed core design (original FCUDA design)

Even workloads

```
void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim,
                    int A_addr, int B_addr, int C_addr)
{
    for (blockIdx.x = 0; blockIdx.x < gridDim.x; blockIdx.x+=2) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```

Odd workloads

```
void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim,
                    int A_addr, int B_addr, int C_addr)
{
    for (blockIdx.x = 1; blockIdx.x < gridDim.x; blockIdx.x+=2) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```

FCUDA SoC – Multi-core design

- Fixed core design (original FCUDA design)
  - Each core is a different IP! → different drivers

Even threadblocks

```
void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim,
                    int A_addr, int B_addr, int C_addr)
{
    for (blockIdx.x = 0; blockIdx.x < gridDim.x; blockIdx.x+=2) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```

Odd threadblocks

```
void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim,
                    int A_addr, int B_addr, int C_addr)
{
    for (blockIdx.x = 1; blockIdx.x < gridDim.x; blockIdx.x+=2) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```

FCUDA SoC – Multi-core design

- Flexible core design
  - Core is parameterized

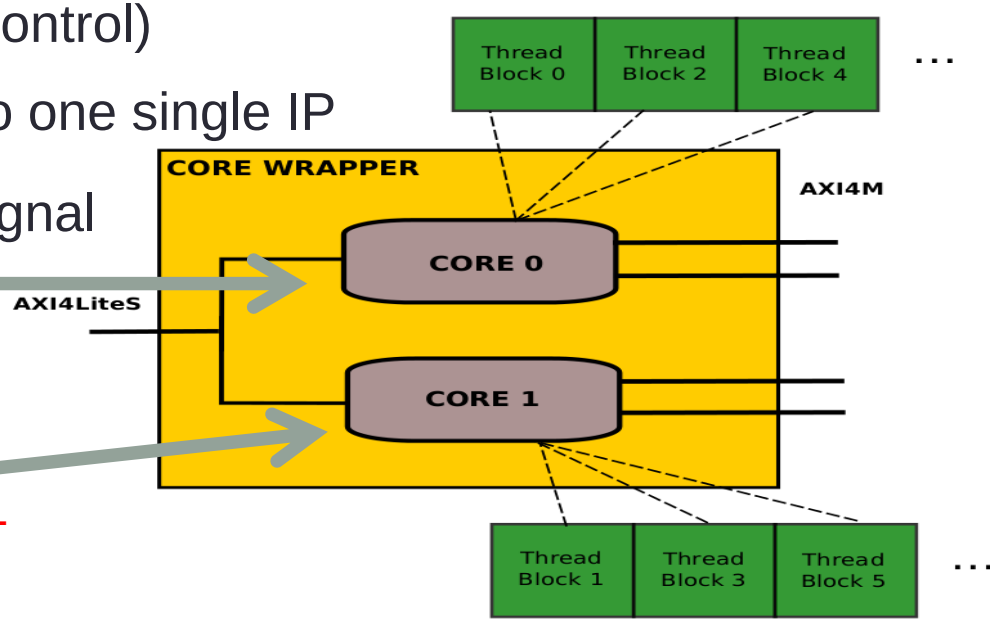
```
void vecAdd(int *A, int *B, int *C, int x, int y, dim3 blockDim, dim3 gridDim,
                  int A_addr, int B_addr, int C_addr, int num_cores, int core_id)
{
    for (blockIdx.x = core_id; blockIdx.x < gridDim.x; blockIdx.x+=num_cores) {
      fetch(...);
      compute(...);
      write(...);
    }
}
```

FCUDA SoC – Multi-core design

- Instantiate cores into the design with different core_id and same num_cores
- Issues:
  - Cores will have separate control! → sequential control of cores
  - The number of AXI Interconnect increases
    - Connection of AXI4M (data)
    - Connection of AXI4LiteS (control)
- Solution: grouping all the cores into one single IP
- All cores share the same control signal

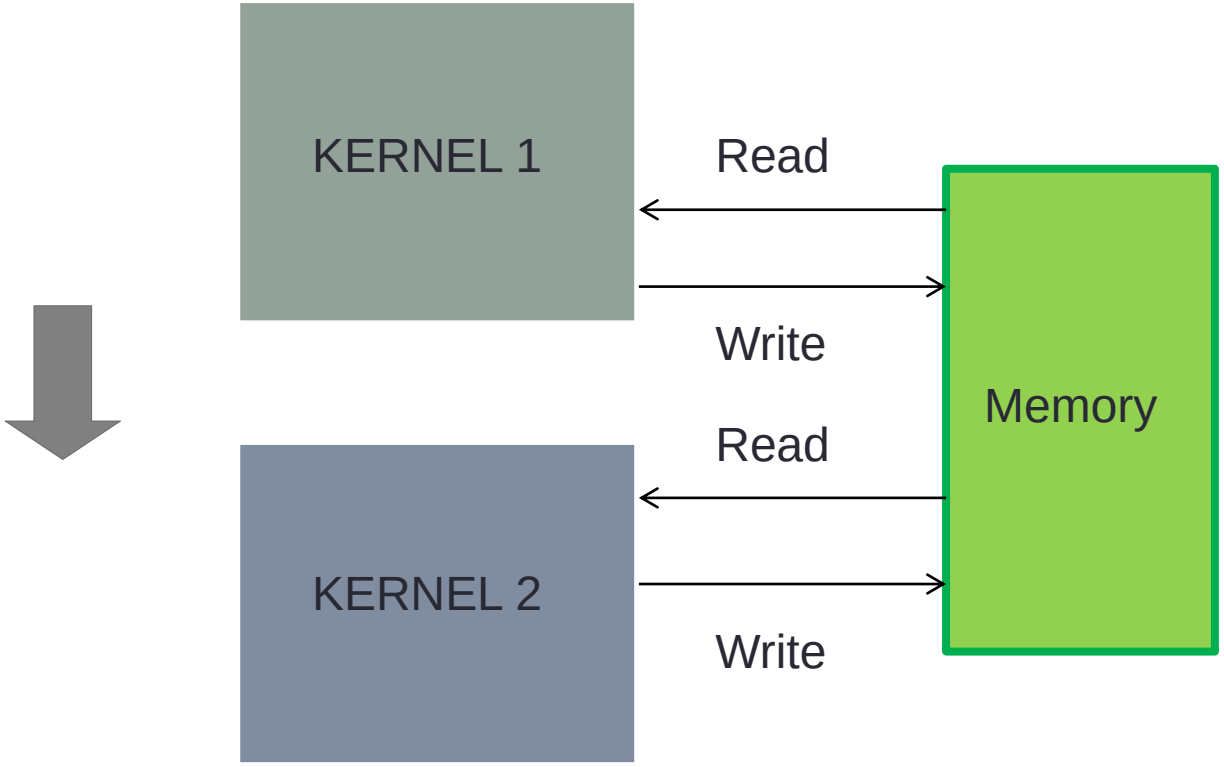
num_cores=2, core_id = 0

num_cores=2, core_id = 1



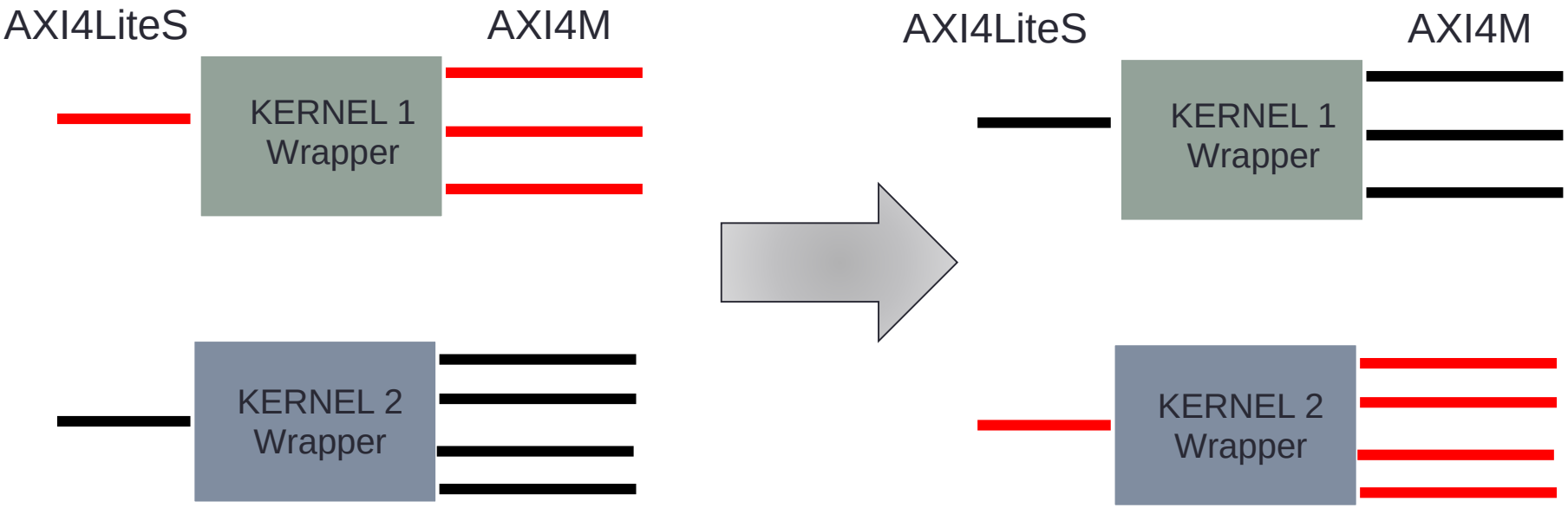
FCUDA SoC – Multi kernels

- Applications which have multiple dependent kernels



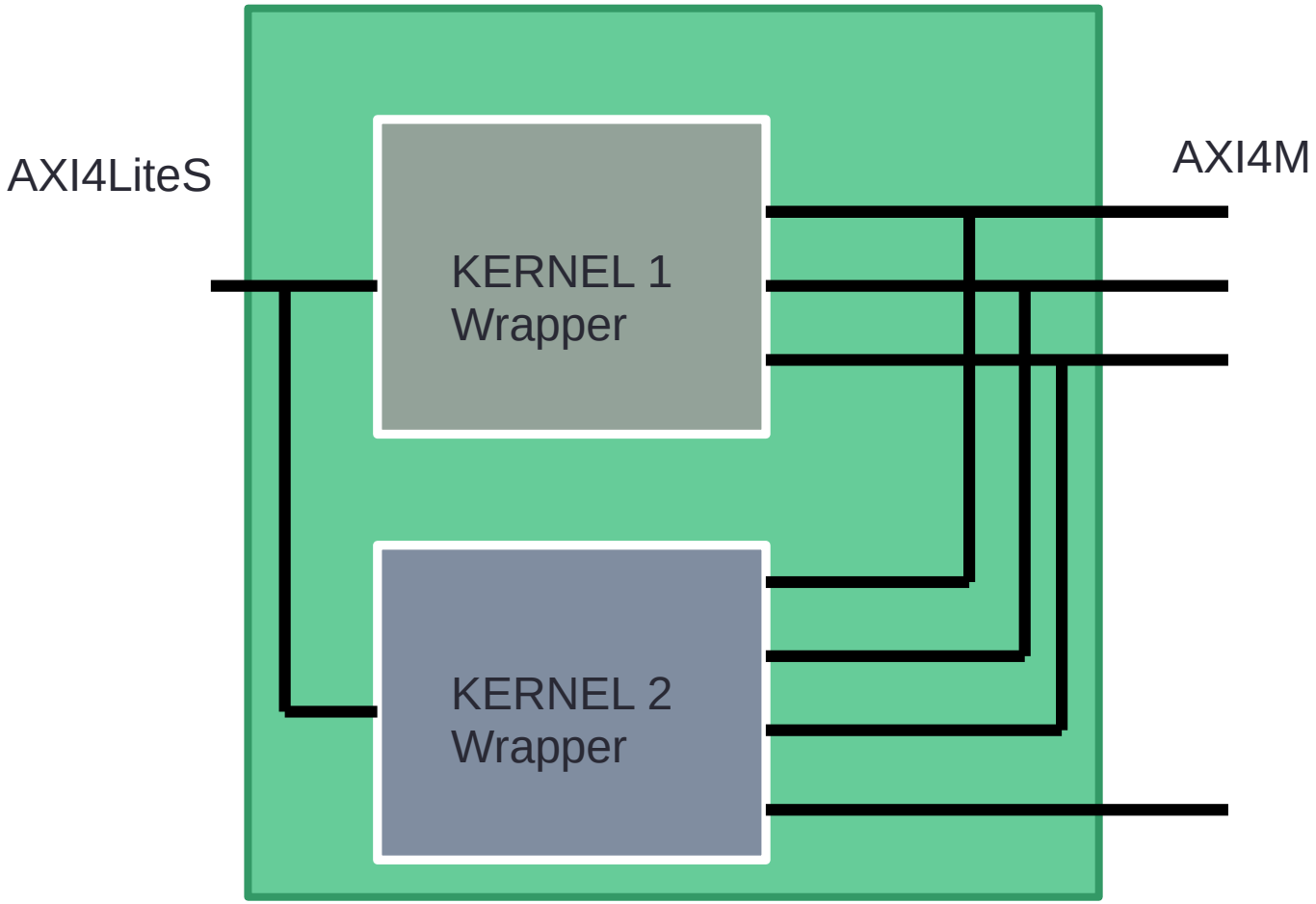
FCUDA SoC – Multi kernels

- Note: we do not use dynamic partial reconfiguration
- Problem:
  - # AXI ports ~ resource consumption
  - Kernel 2's ports are unused when Kernel 1 is active (and vice versa)



FCUDA SoC – Multi kernels

- Solution: merging 2 Kernels to share ports



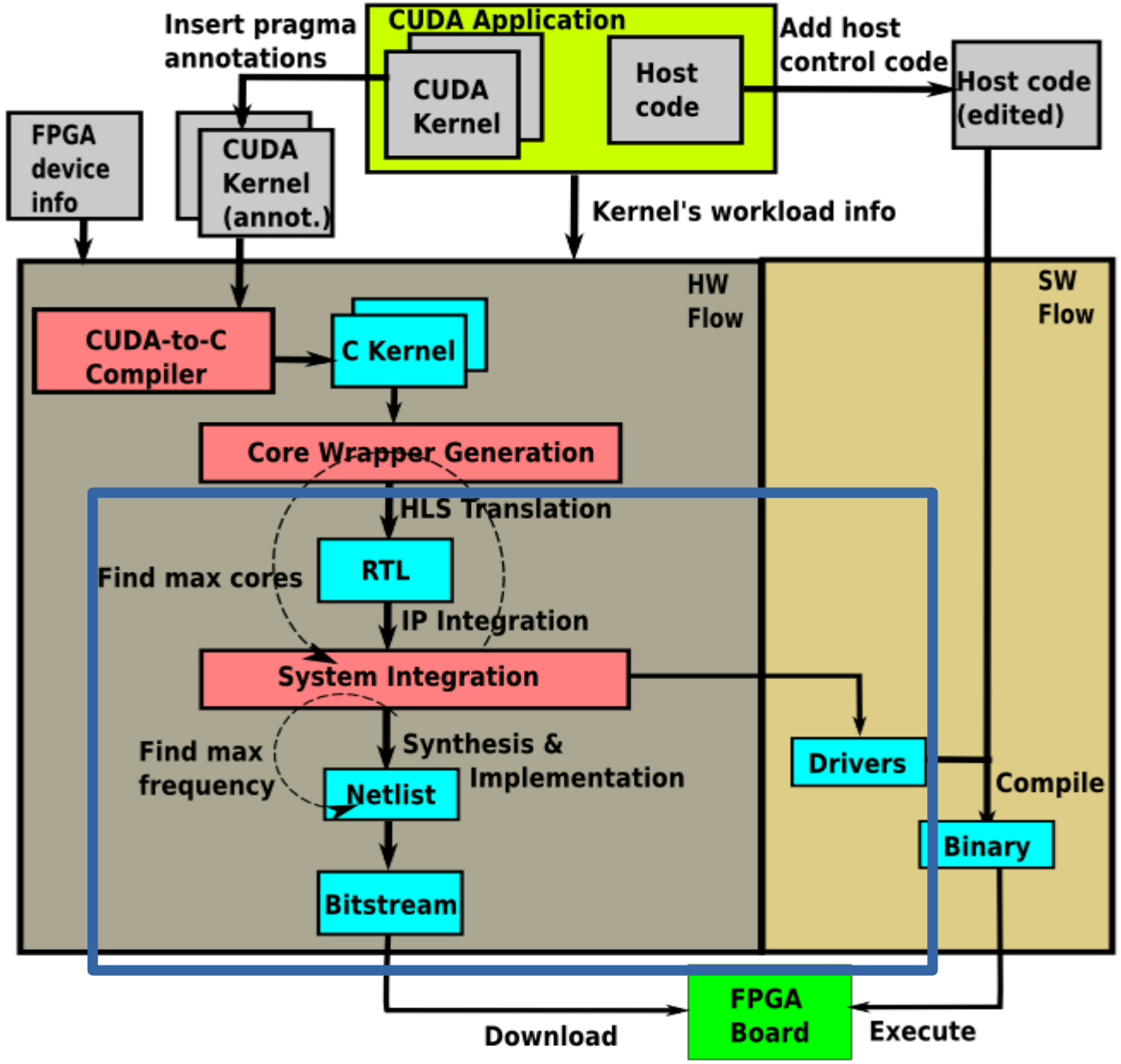
FCUDA SoC – HW flow

5. Generation of HLS IP
   - Find analytical max. number of cores based on HLS report of 1 core and kernels' workloads

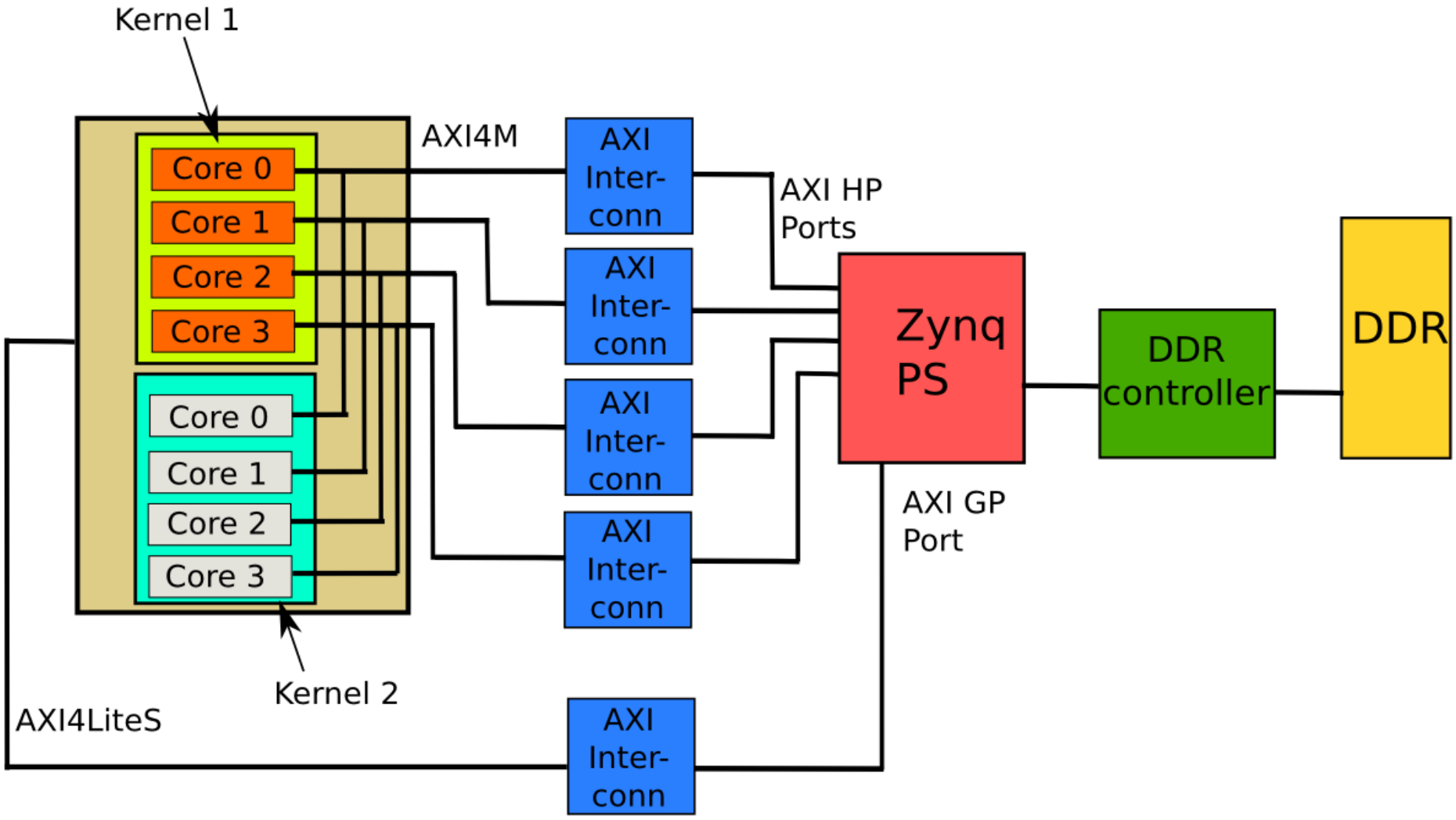
6. System Integration
   - Zynq PS, AXI Interconnect, etc.

7. Perform Synthesis and Place & Route
   - Reiterate until achieving max. frequency

8. Generate bitstream and configure FPGA



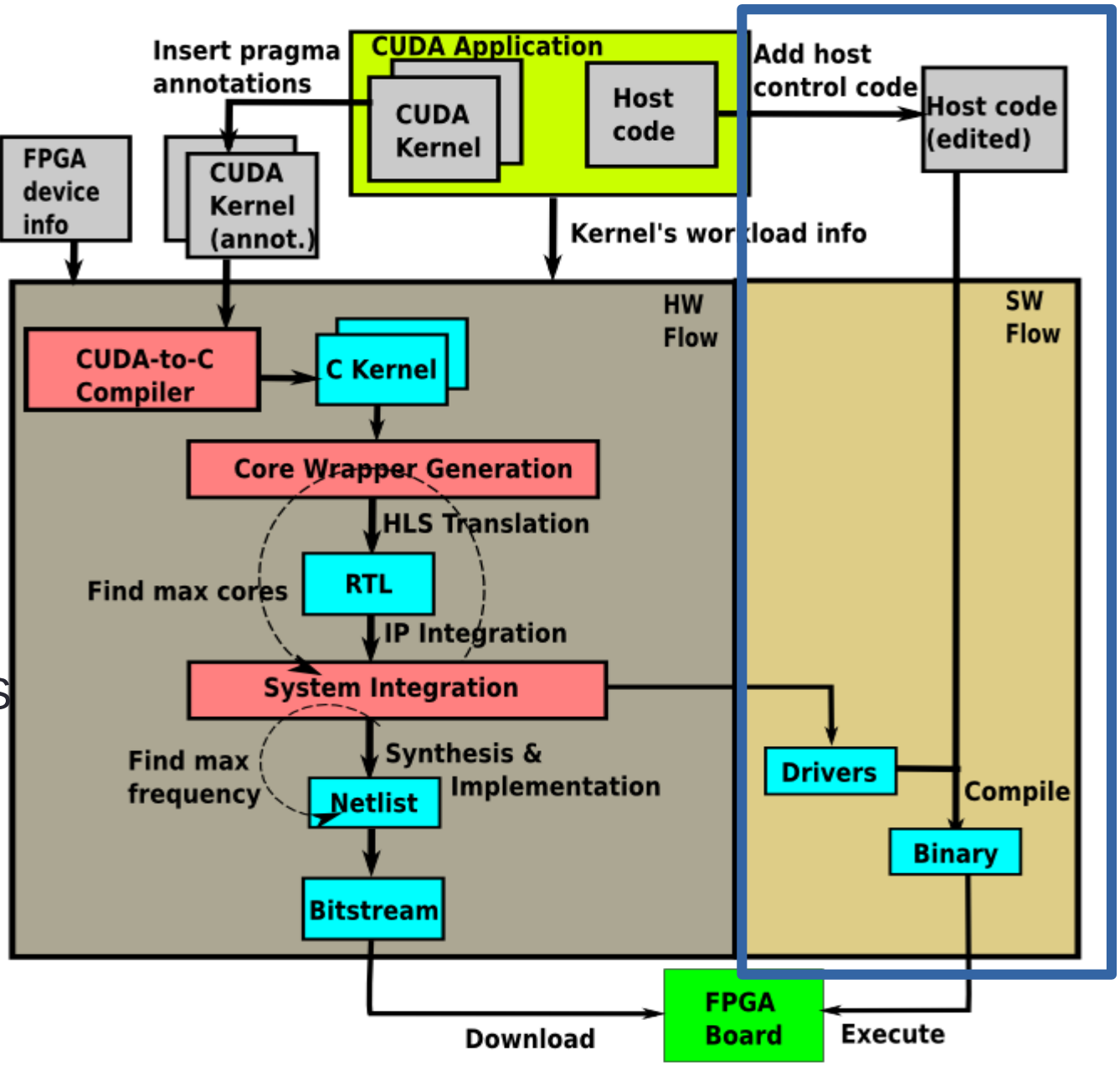
FCUDA SoC – System integration



FCUDA SoC – SW flow

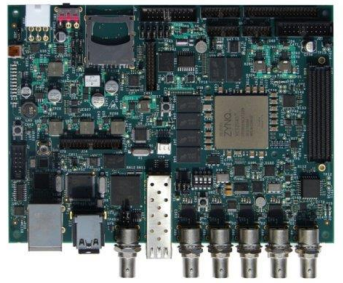
9. Edit host code
   - Remove CUDA's syntax
   - Add driver control code

10. Compile and run on FPGA
   - Bare-metal vs. OS



Experiments

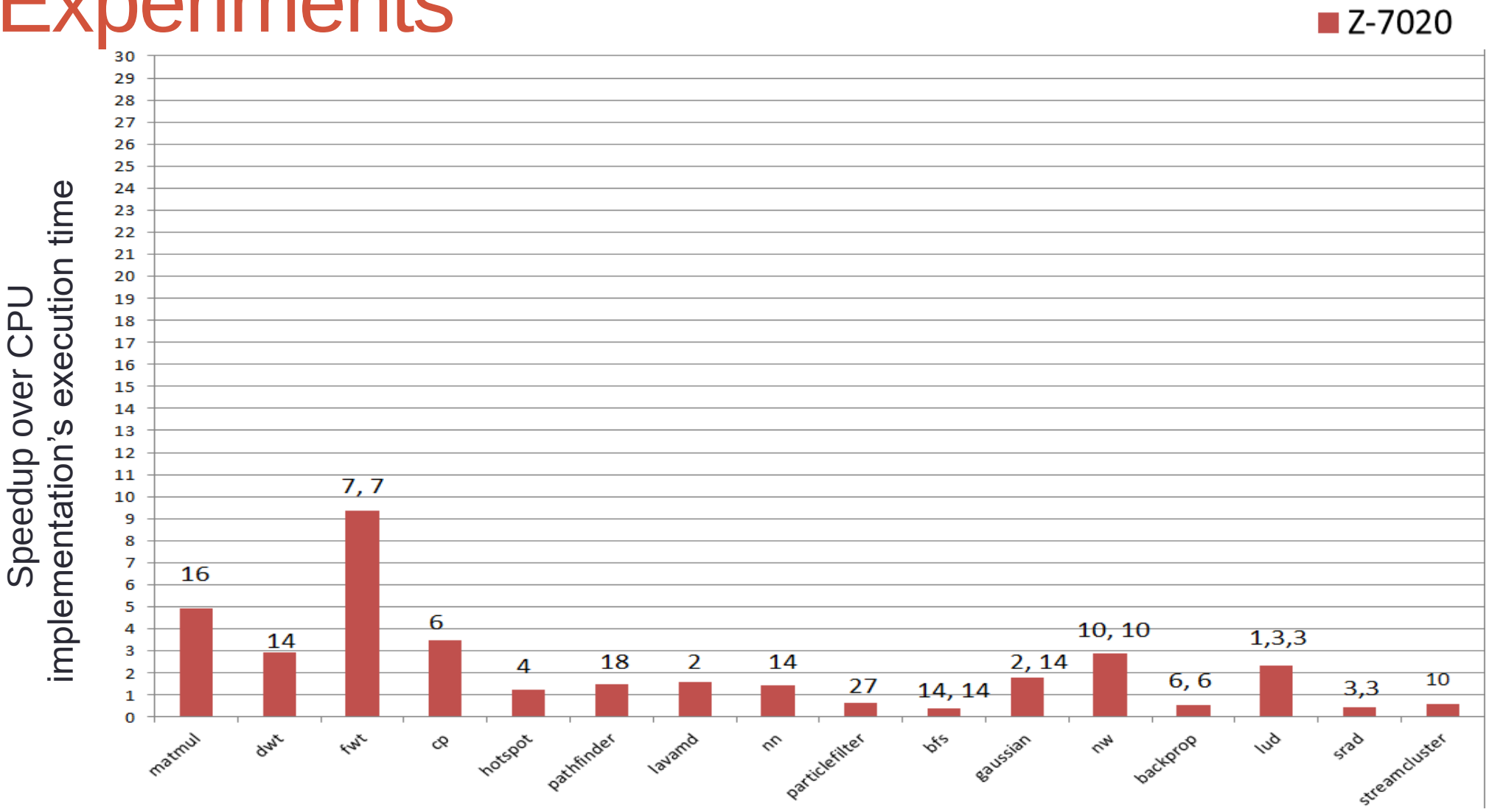
- Two FPGA SoC devices:
  - Z-7020
    - FF: 106400, LUT: 53200, BRAM: 140, DSP: 220
    - ARM Freq: 666 MHz
  - Z-7045
    - FF: 218600, LUT: 437200, BRAM: 545, DSP: 900
    - ARM Freq: 800 MHz



- 16 CUDA benchmarks collected from NVIDIA SDK, Parboil suite, Rodinia suite
- Measure total execution time
- Compare against the CPU implementation

(For GPU's comparison, please refer to our paper)

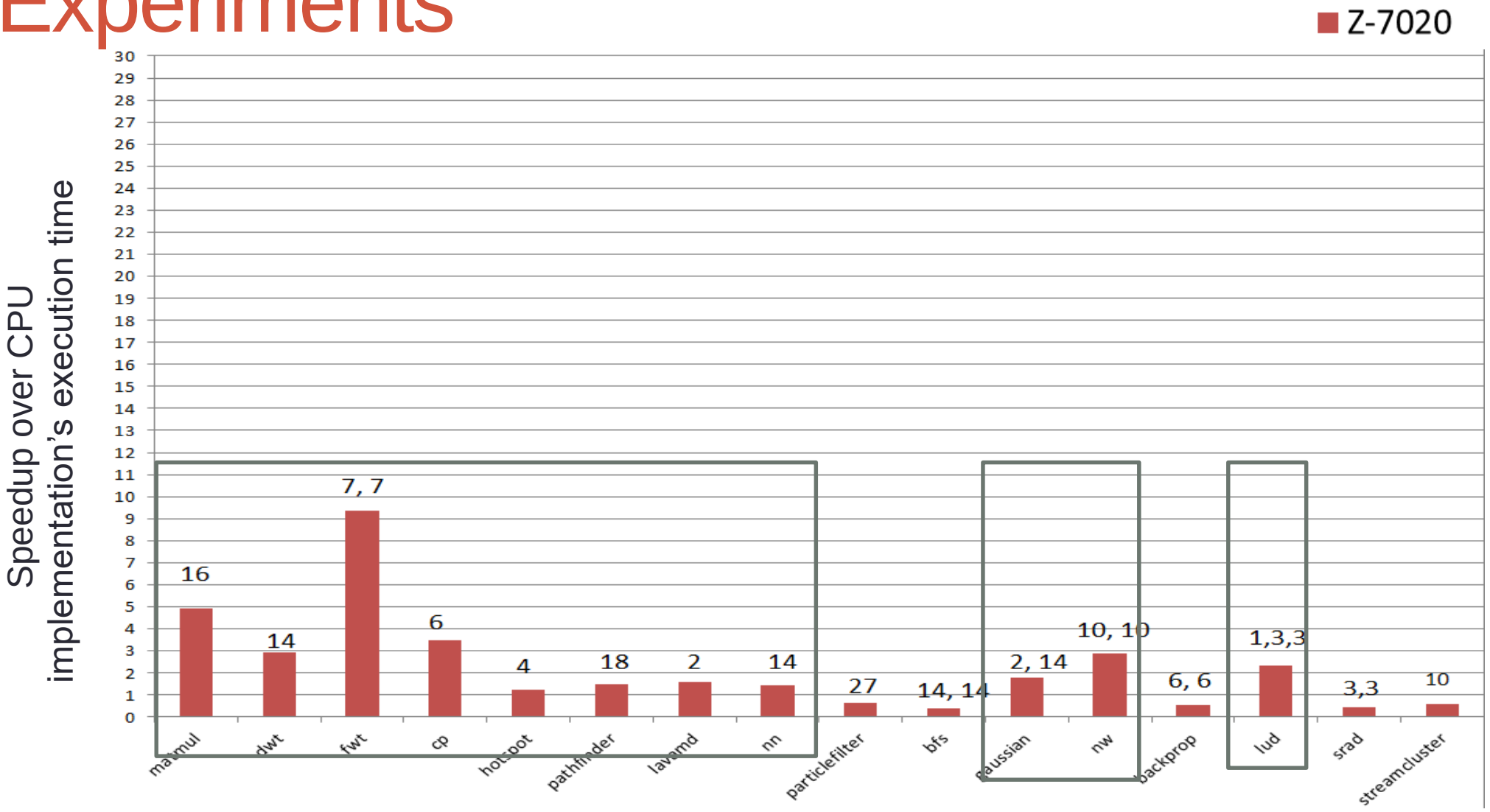
Experiments



Performance Comparison of ARM + FPGA vs. ARM only on Z-7020

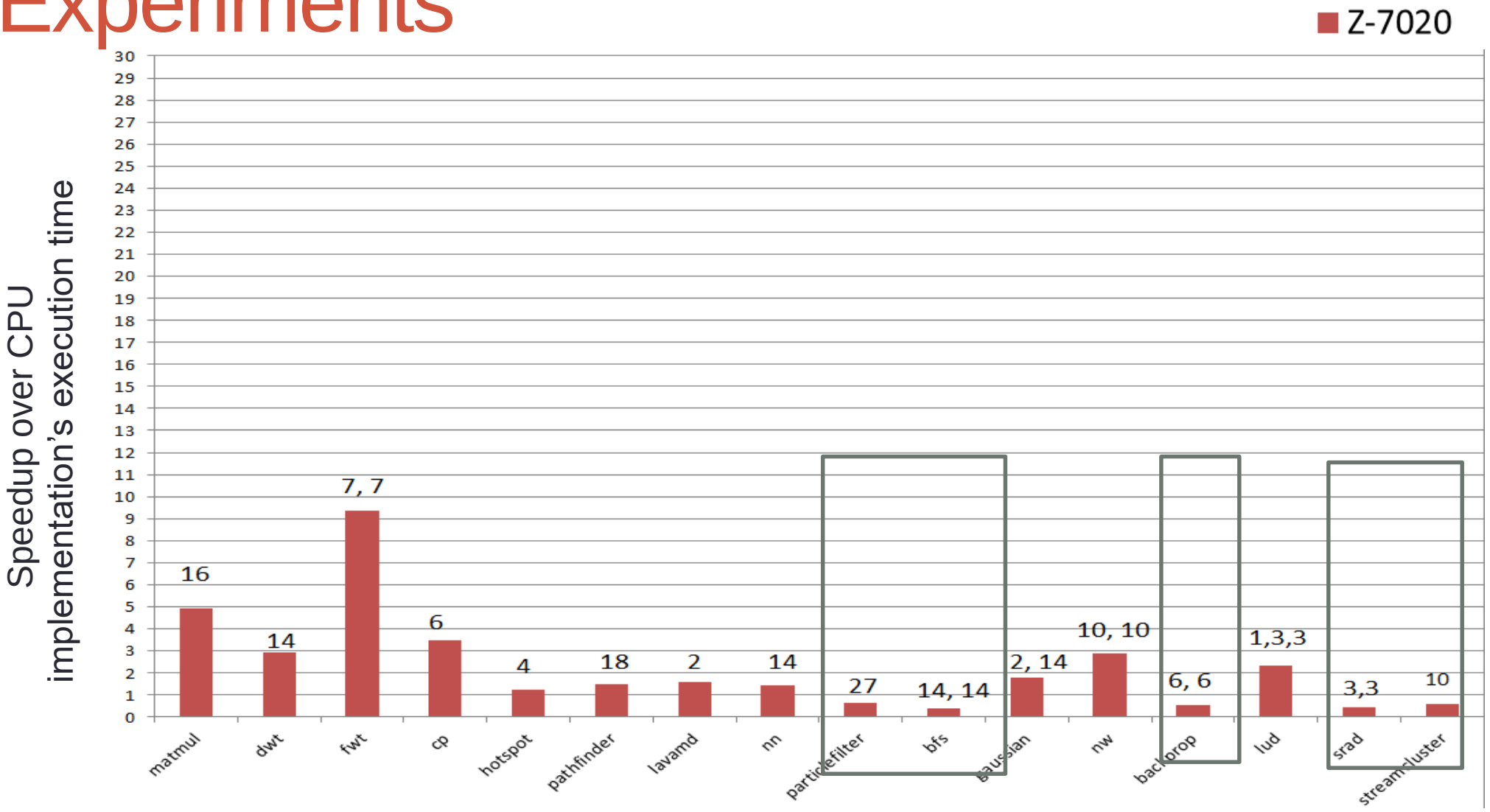
- Numbers above the columns show the total instantiated cores in the design

Experiments



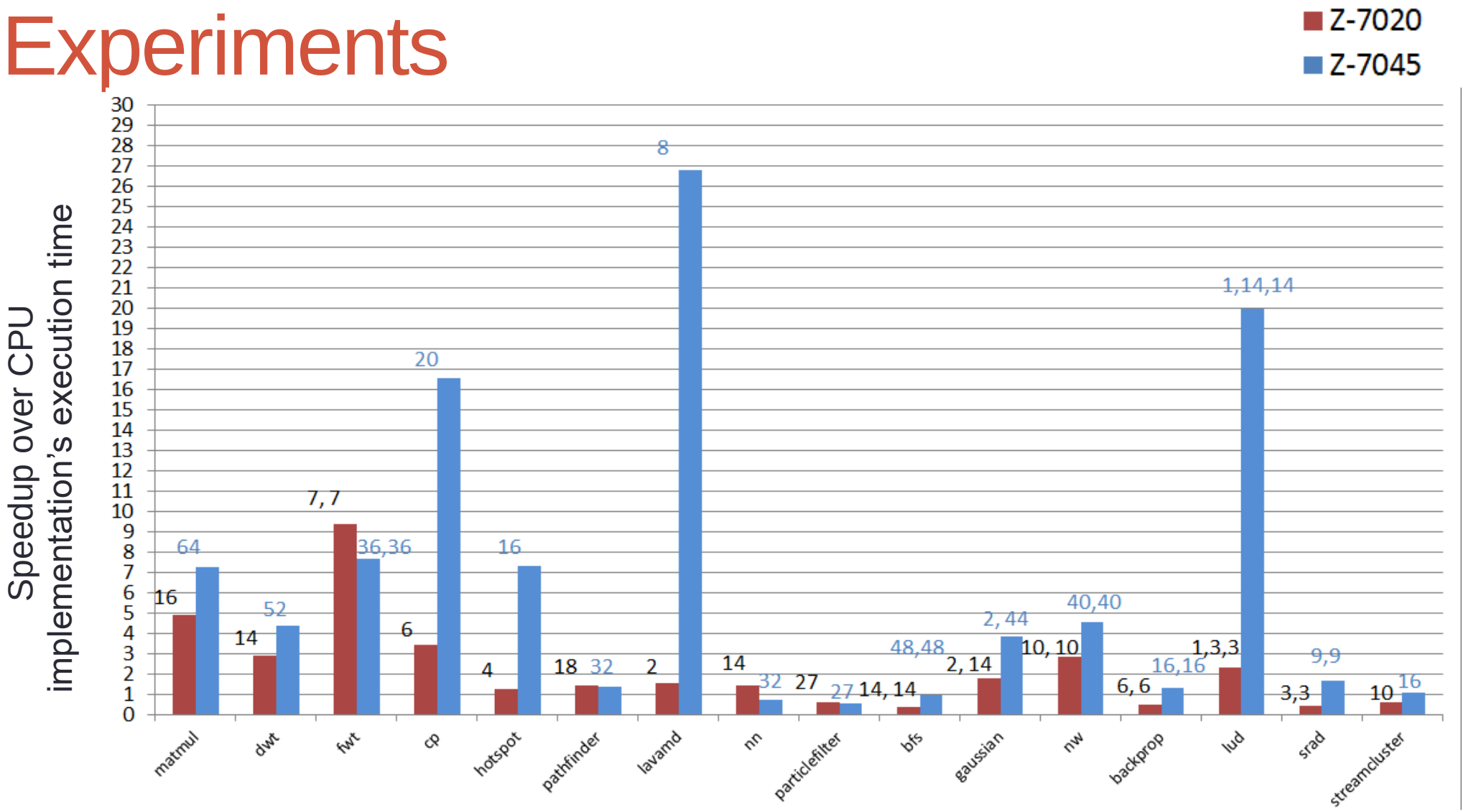
Better than CPU

Experiments



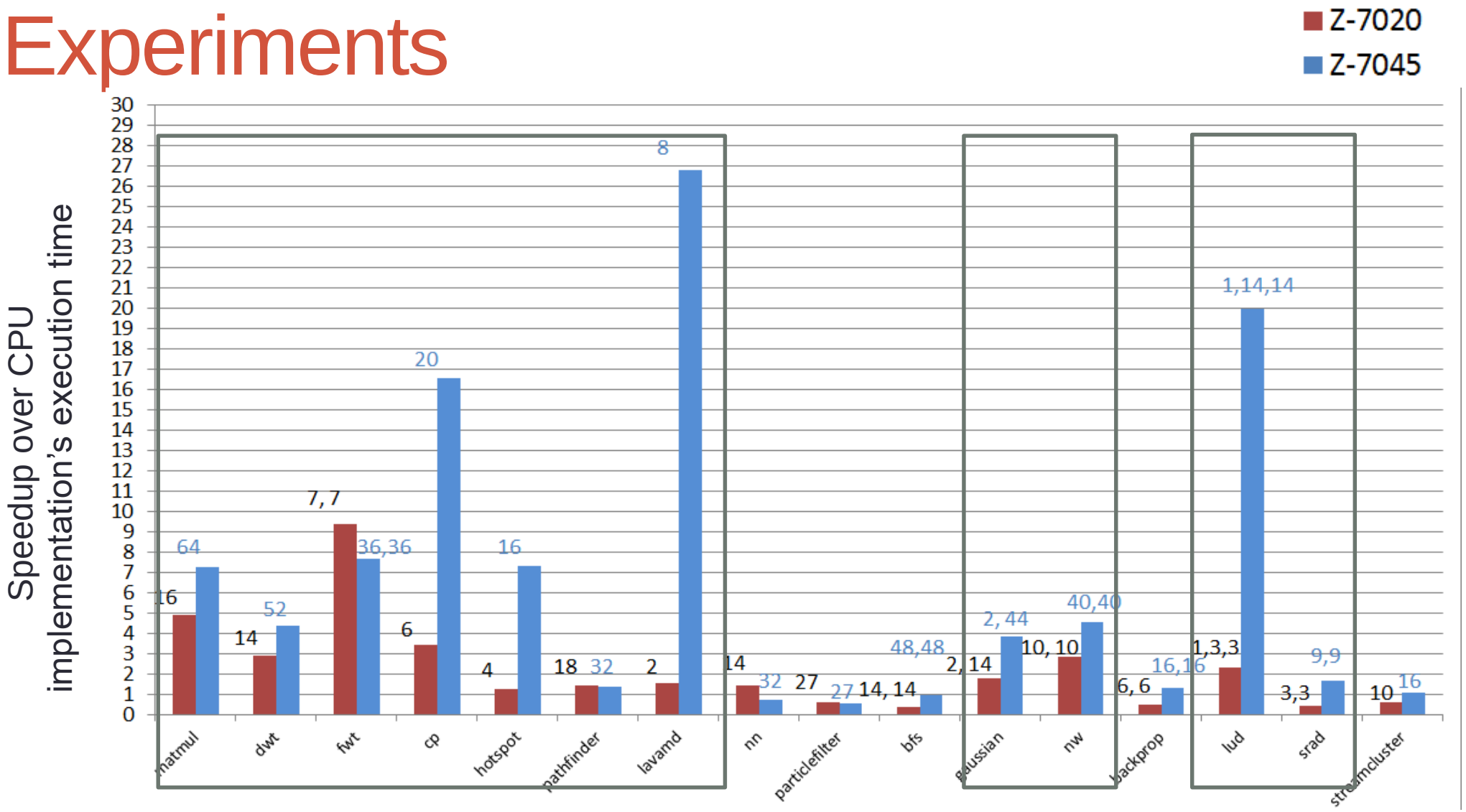
Worse than CPU
+ no shared memory
+ random access
+ limited board resource

Experiments



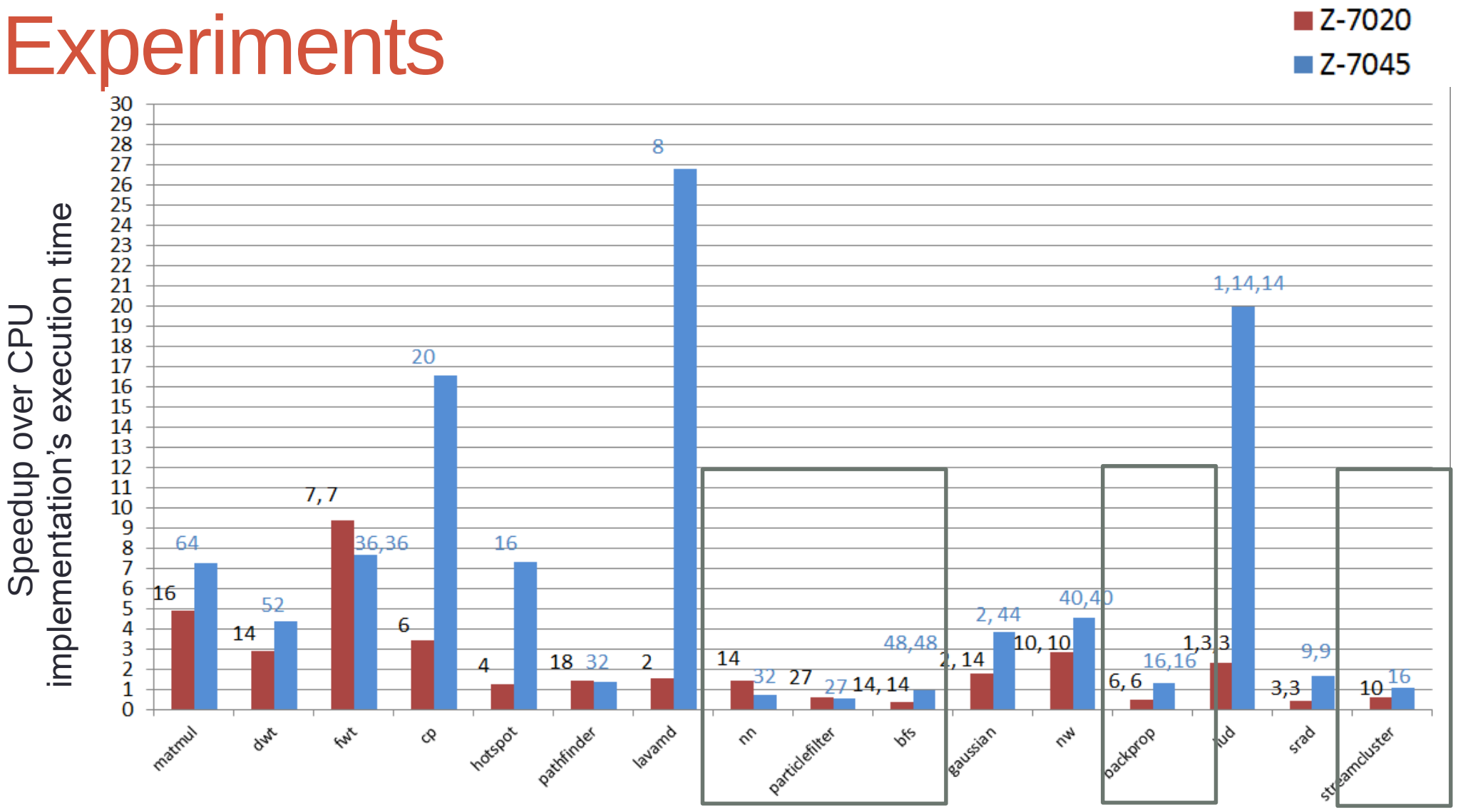
Performance Comparison of ARM + FPGA vs. ARM only on Z-7020 (red) and Z-7045 (blue)

Experiments



Performance Comparison of ARM + FPGA vs. ARM only on Z-7020 (red) and Z-7045 (blue)

Experiments



Performance Comparison of ARM + FPGA vs. ARM only on Z-7020 (red) and Z-7045 (blue)

FCUDA SoC – Best practices

- CPU control:
  - Having a single IP --- keep things simple → reduce management overhead
- Interconnect network consideration
  - Partition signals between interfaces: data ports – High Performance ports, control ports – General Purpose ports
  - Merge data ports of a core, shared AXI ports between kernels
- Good CUDA programming practices
  - Coalescing memory access, local data reused, etc.
- Design Space Exploration should be carried out [FCCM'11]
  - Find the best number of cores

Conclusion

- A design study of how to build a complete hardware/software system of a CUDA-to-FPGA flow targeting SoC FPGA
  - Verified with 16 CUDA benchmarks
  - Discussed key design decisions, best practices
- Our tool flow is open-source and can be found at:

  http://dchen.ece.illinois.edu/tools.html