Reconfigurable systems in next-gen, heterogeneous HPC platforms: The SOpenCL case

Christos Antonopoulos

Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece

Acknowledgements

- Muhsen Owaida
- Nikolaos Bellas

- Antonis Dimtsas
- Michalis Doulis
- Michalis Spyrou
- Vassilis Vassiliadis

- Charalambos Antoniadis
- Konstantis Daloukas

This was has been partially supported by the Marie Curie International Reintegration Grant IRG-223819

Introduction

- Automatic generation of hardware at the research forefront in the last 10 years.
- Variety of High Level Programming Models have been introduced as Hardware Design Languages: C/C++, C-like Languages, MATLAB
  - Used as a user-friendlier HDL

- Obstacles:
  - Parallelism extraction for larger applications
  - Extensive compiler transformations & optimizations
  - Inability to scale to large applications

Motivation

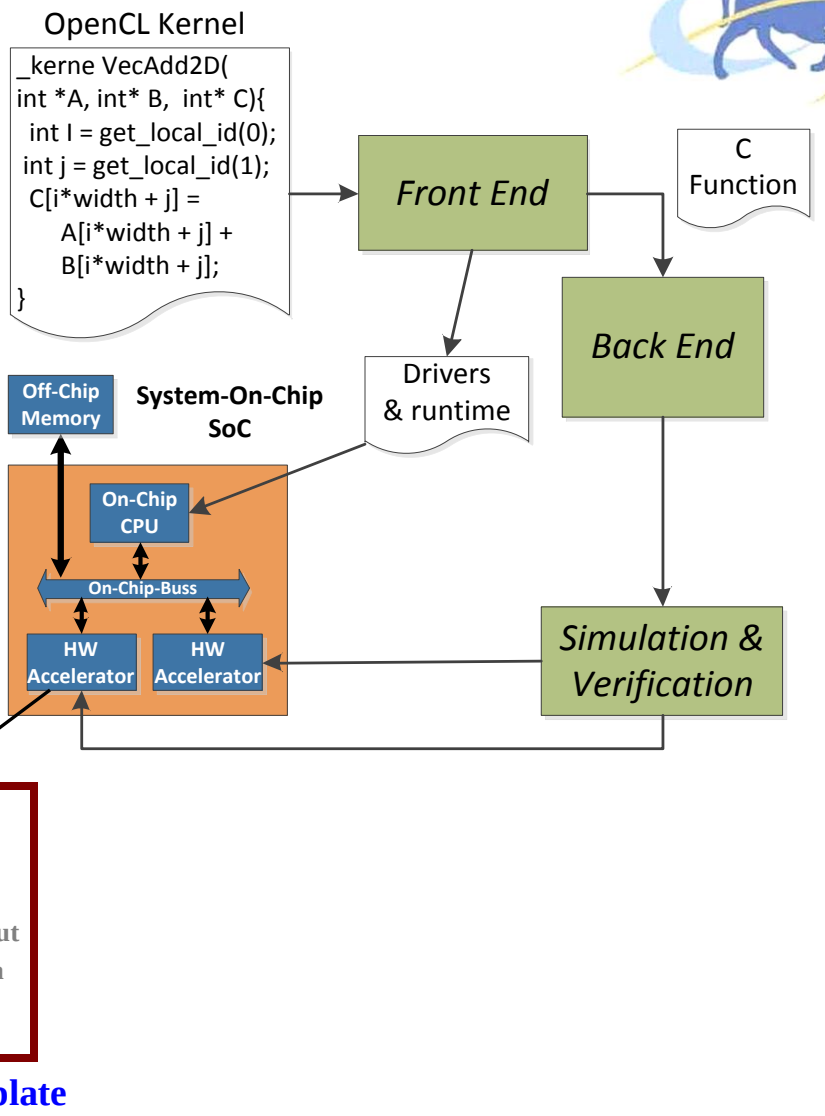
- Lack of parallel programming language for reconfigurable platforms.
- A major shift of computing industry toward many-core computing systems.
- Reconfigurable fabrics bear a strong resemblance to many core systems.

Vision

- “Provide the tools and methodology to enable the large pool of software developers and domain experts, who do not necessarily have expertise on hardware design, to architect whole accelerator-based systems”
  - Borrowed from advances in massively parallel programming models

Contribution

- Silicon-OpenCL “SOpenCL”.
- A tool flow to convert an **unmodified** OpenCL application into a SoC design with HW/SW components.
- A template-based hardware accelerator generation.
- Decouple data movement and computations.



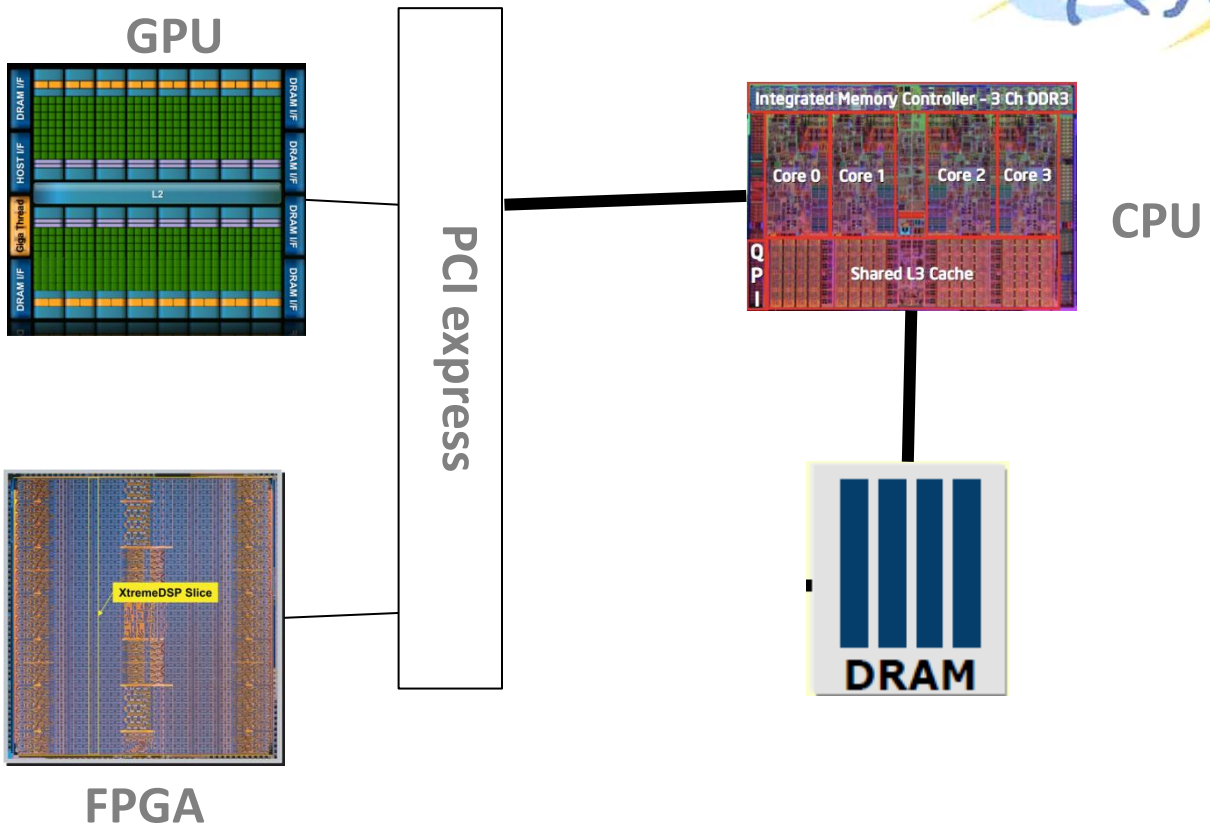
Outline

- Introduction
- OpenCL Programming Model
- Silicon OpenCL (SOpenCL)
  - Front-End
  - Back-End
- Experimental Evaluation
- Upcoming Challenges

OpenCL for Heterogeneous Systems

A modern platform may include:

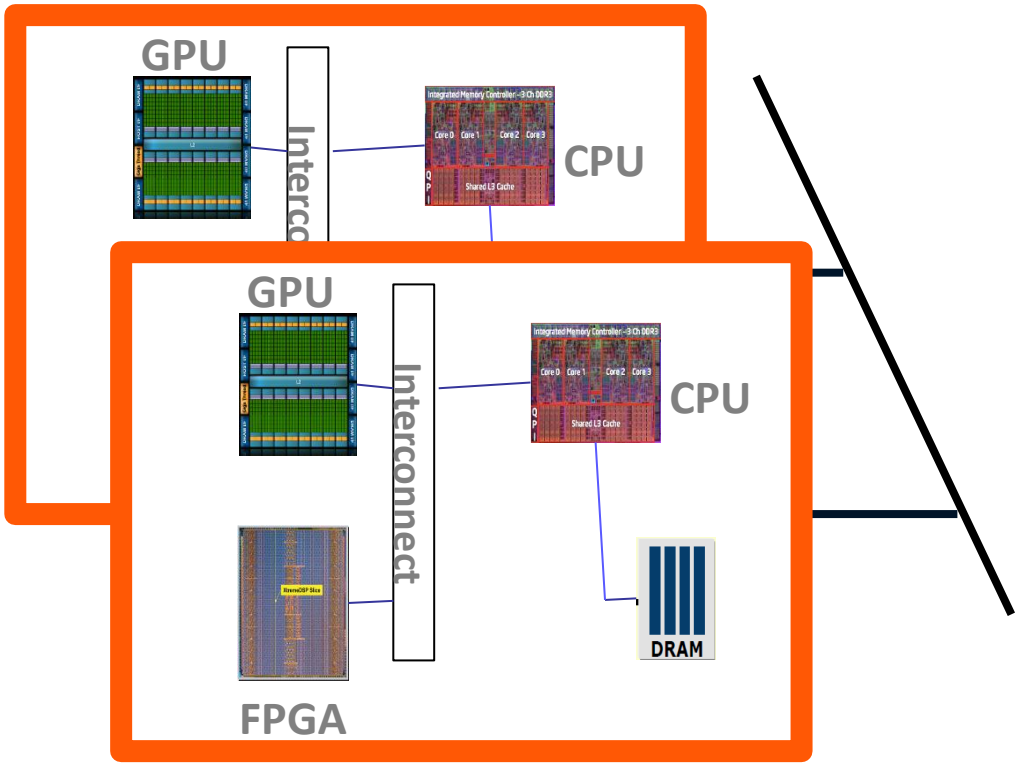
- One or more CPUs
- GPUs
- FPGAs
- DRAMs



Very challenging to program such a system. Multiple programming models may be required.

OpenCL for Heterogeneous Systems

Technology places such heterogeneous systems in an SoC



The future belongs to heterogeneous systems, and SoCs as the standard building block of computing.

OpenCL for Heterogeneous Systems

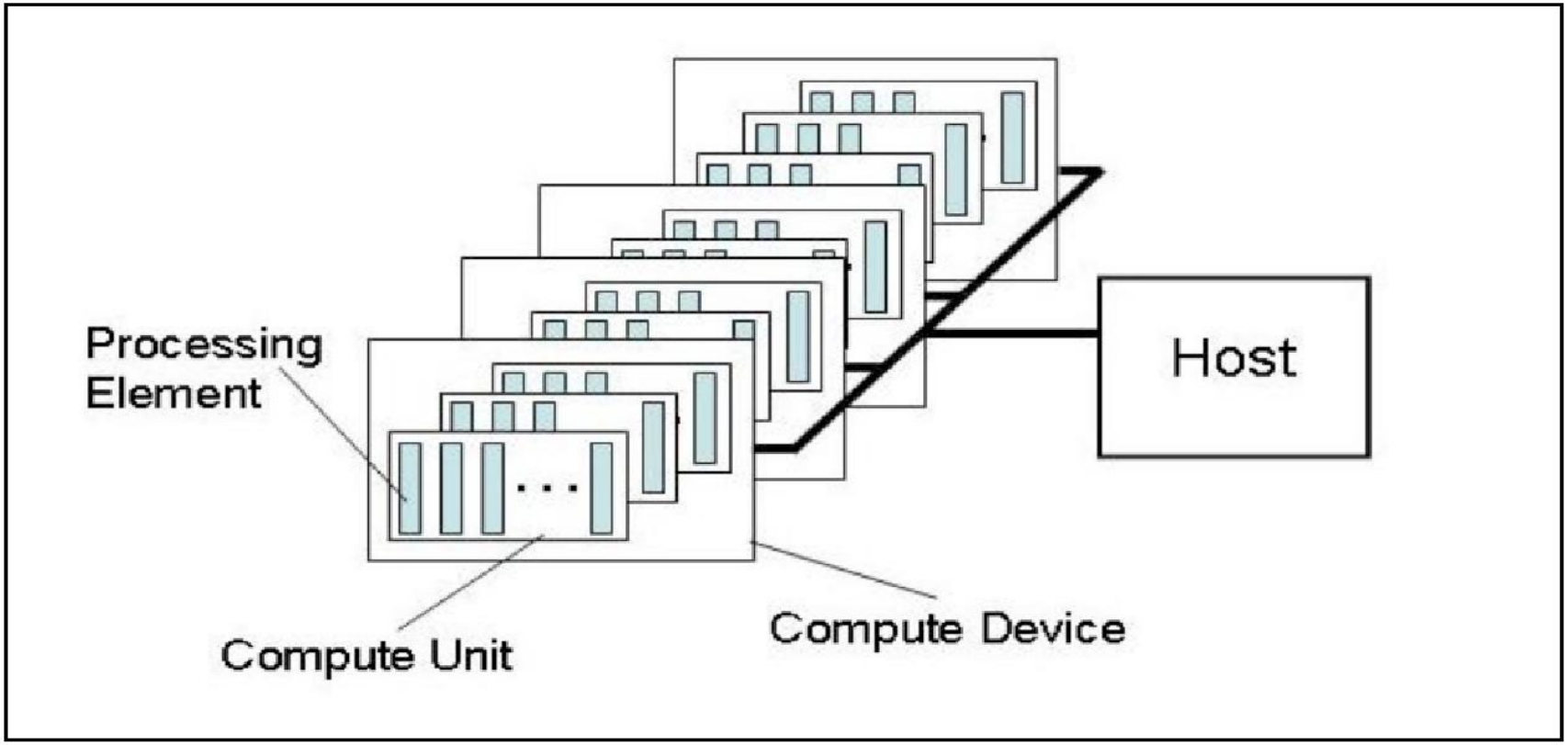
OpenCL (Open Computing Language) aims at letting a programmer write a portable program once and deploy it in any heterogeneous system.

Became an important industry standard after release due to substantial industry support.

OpenCL for Heterogeneous Systems

- OpenCL: a unified programming standard and framework
- Targets:
  - Homogeneous and heterogeneous multicores
  - Embedded Systems (OpenCL-embedded)
  - Accelerator-based systems
- Aims at being platform-agnostic

OpenCL Platform Model



One host and one or more Compute Devices (CD)

Each CD consists of one or more Compute Units (CU)

Each CU is further divided into one or more Processing Elements (PE)

OpenCL Execution Model

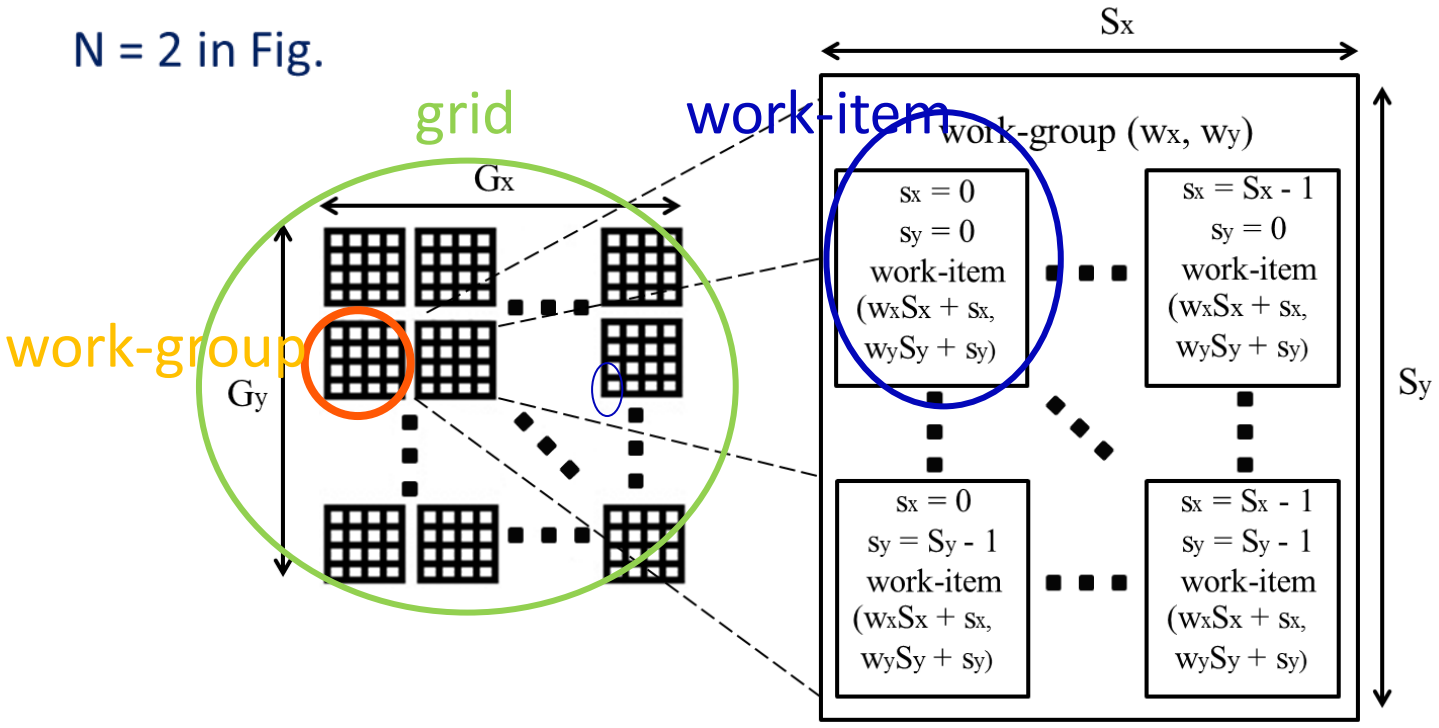
- An OpenCL application consists of two parts:
  - Main program that executes on the host
  - A number of kernels that execute on the compute devices
- Main constructs of the OpenCL execution model:
  - Kernels
  - Memory Buffers
  - Command Queues
- Main program (on the host) submits to command queues:
  - Kernels for execution
  - Memory objects for manipulation

OpenCL Kernel Execution "Geometry"

- OpenCL defines a geometric partitioning of grid of computations
- Grid consists of N dimensional space of work-groups
- Each work-group consists of N dimensional space of work-items

$1 \leq N \leq 3$

N = 2 in Fig.



OpenCL Simple Example

- OpenCL kernel describes the computation of a work-item
  - Finest parallelism granularity
- e.g add two integer vectors (N=1)

Run-time call
Used to differentiate execution
for each work-item

```
void add(int* a,
         int* b,
         int* c) {
for (int idx=0; idx<sizeof(a); idx++)
   c[idx] = a[idx] + b[idx];
}
```

C code

```
__kernel void vadd(
            __global int* a,
            __global int* b,
            __global int* c) {
      int idx= get_global_id(0);
      c[idx] = a[idx] + b[idx];
}
```

OpenCL kernel code

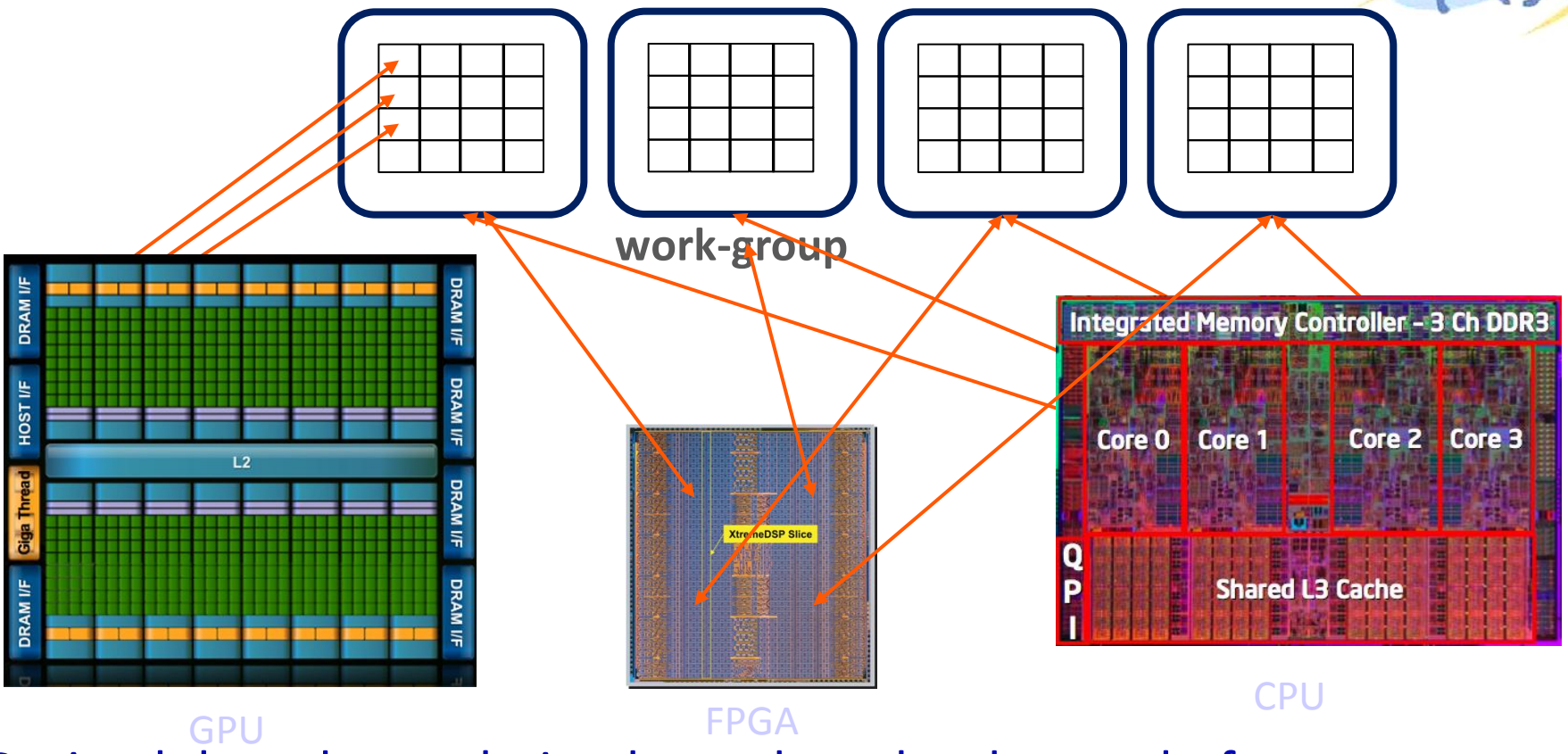
Why OpenCL as an HDL?

- OpenCL exposes parallelism at the finest granularity
  - Allows easy hardware generation at different levels of granularity
  - From coarser to finer accelerators
  - One accelerator per work-item, one accelerator per work-group, one accelerator per multiple work-groups, etc.
- OpenCL exposes data communication
  - Critical to transfer and stage data across platforms
- We target unmodified OpenCL to enable hardware design to software engineers
  - No need for hardware/architectural expertise
  - Write once, deploy everywhere

Outline

- Introduction
- OpenCL Programming Model
- Silicon OpenCL (SOpenCL)
  - Front-End
  - Back-End
- Experimental Evaluation
- Upcoming Challenges

Granularity Management



Optimal thread granularity depends on hardware platform

We select a hardware accelerator to process one work-group per invocation. Smaller invocation overhead

Serialization of Work Items

OpenCL code

```
__kernel void vadd(…) {

  int idx = get_global_id(0);
  c[idx] = a[idx] + b[idx];

}
```

C code

```
__kernel void Vadd(…) {
  int idx;
  for( i = 0; i < get_local_size(2); i++)
   for( j = 0; j < get_local_size(1); j++)
    for( k = 0; k < get_local_size(0); k++) {
      idx = get_item_gid(0);
      c[idx] = a[idx] + b[idx];
    }
}
```

```
idx = (global_id2(0) + i) * Grid_Width * Grid_Height +
      (global_id1(0) + j) * Grid_Width                +
                    (global_id0(0) + k);
```

Elimination of Synchronization Operations

OpenCL code

```
Statements_block1
barrier();
Statements_block2
```


C code

```
triple_nested_loop {
    Statements_block1
    barrier();
    Statements_block2
}
```

Elimination of Synchronization Operations

C code

```
triple_nested_loop {
    Statements_block1
    barrier();
    Statements_block2
}
```


C code

```
triple_nested_loop {
    Statements_block1
}
// barrier();
triple_nested_loop {
    Statements_block2
}
```

Variable Privatization

C code

```
triple_nested_loop {
   Statements_block1
   x = …;
}
triple_nested_loop {
   Statements_block2
   … = … x …;
}
```


C code

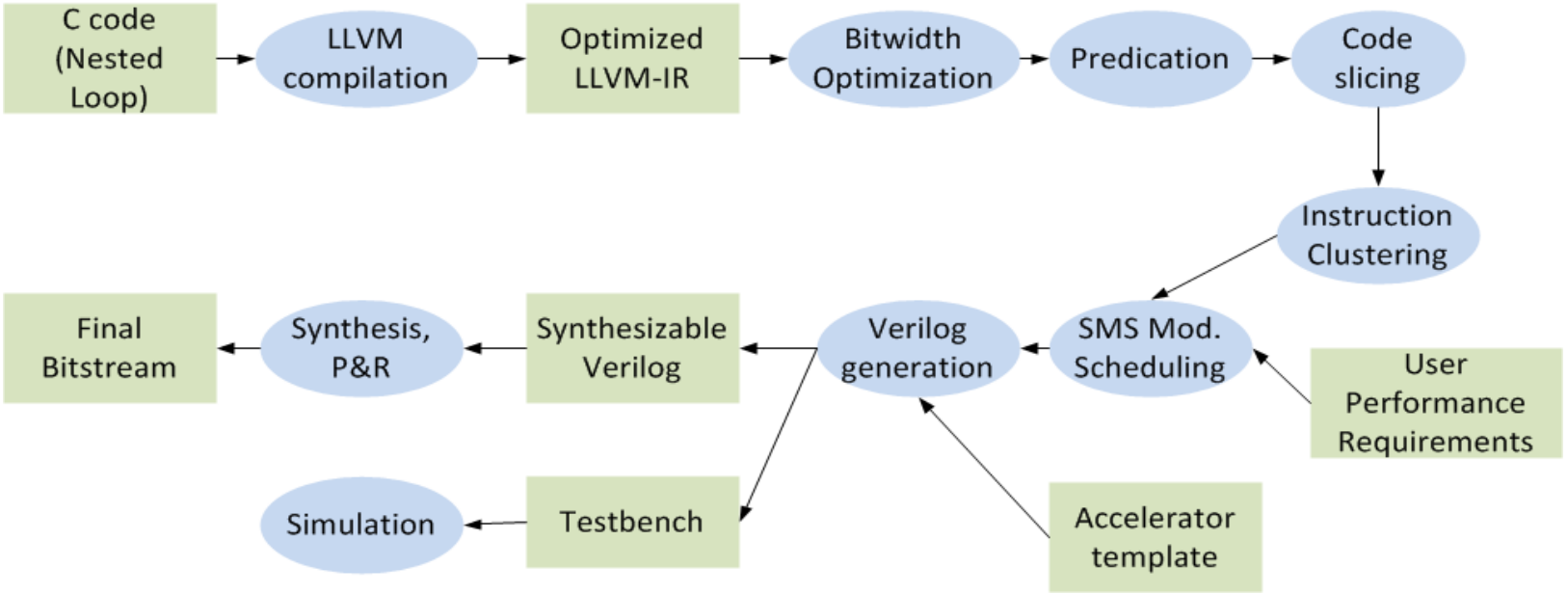
```
triple_nested_loop {
   Statements_block1
   x[i][j][k] = …;
}
triple_nested_loop {
   Statements_block2
   … = … x[i][j][k] …;
}
```

Outline

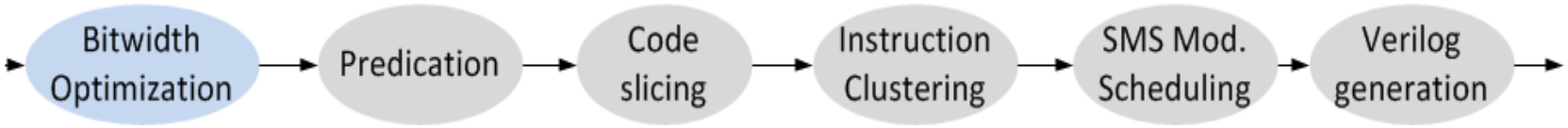
- Introduction
- OpenCL Programming Model
- Silicon OpenCL (SOpenCL)
  - Front-End
  - Back-End
- Experimental Evaluation
- Upcoming Challenges

Hardware Generation

- Perform a series of optimizations and transformations.
  - Uses LLVM Compiler Infrastructure.
- Generate synthesizable Verilog.
- Generate test bench and simulation files.

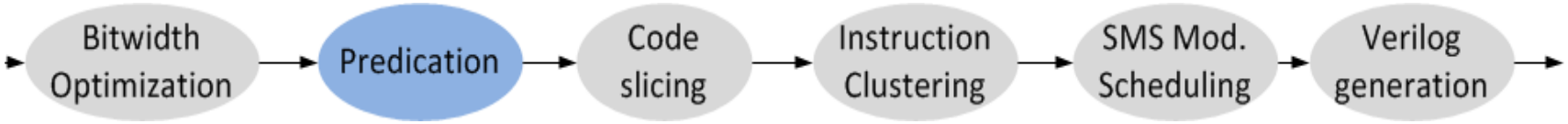


Bitwidth Analysis & Optimization

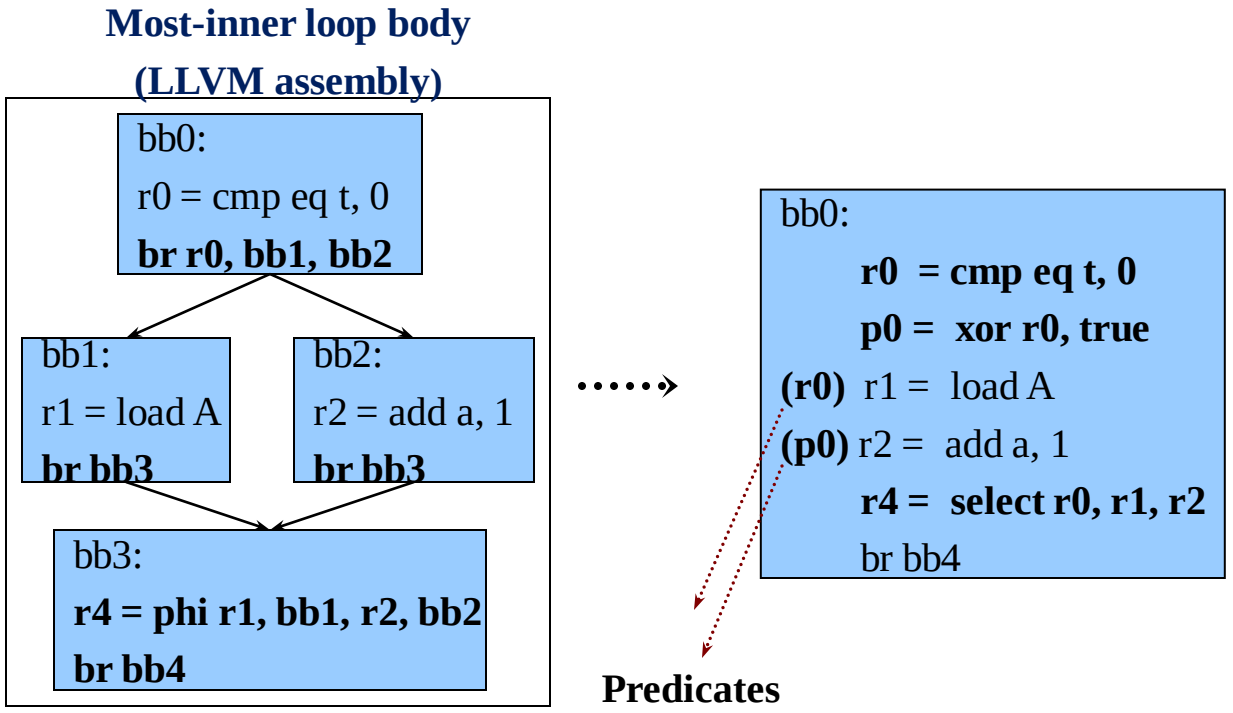


- Detect range of values of program variables to reduce hardware area
  - Programmer can use OpenCL *attributes* to specify range of input values to the kernel
- Important for integer code
  - Especially for computationally intensive code
- Not applicable to FP code due to necessity to be IEEE-754 standard compliant

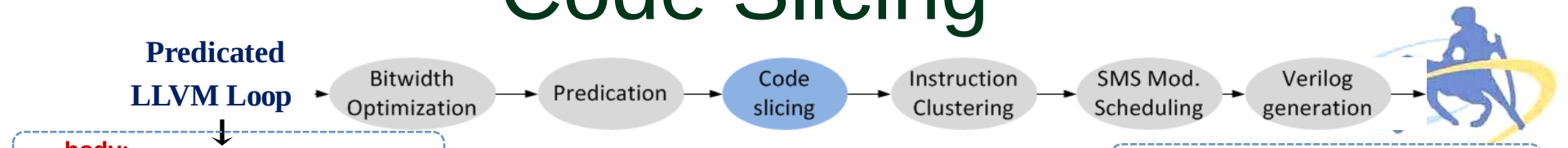
Predication



- Predication: If-conversion necessary for the application of Modulo Scheduling



Code Slicing




```
body:
i46   = phi [true, preh], [i53, body]
ind   = phi [0, preh], [i2, body]
i0    = add a0, ind
i2    = add ind, 1
i3    = add a0, i2
gep0  = getelementptr i8* x0, i0
gep1  = getelementptr i8* x0, i3
i7    = load i8* gep0
i10   = load i8* gep1
i6    = add a2, ind
gep4  = getelementptr i8* x1, i6
i9    = mul i7, a3
i12   = mul i10, a4
i19   = add i9, 32
i20   = add i19, i12
i41   = lshr i22, 6
i44   = cmp slt i41, 0;
i45   = xor i44, true;
i46   = cmp sgt i41, 255;
i47   = xor i46, true;
i48   = and i45, i47;
i50   = select i44, 0, 255;
i51   = select i48, i41, i50;
store i41, i8* gep4
i52   = icmp eq i2, 8
i53   = xor i52, true
br i52, exit, body
```

Sin Kernel:

```
i46   = phi [true, preh], [i53, body]
ind   = phi [0, preh], [i2, body]
i0    = add a0, ind
i2    = add ind, 1
i3    = add a0, i2
gep0  = getelementptr i8* x0, i0
gep1  = getelementptr i8* x0, i3
```

Read Address Computation

```
i7    = load i8* gep0
i10   = load i8* gep1
```

```
i52   = icmp eq i2, 8
i53   = xor i52, true
br i52, exit, body
```

Termination

Sout Kernel:

```
ind   = phi [0, preh], [i2, body]
i0    = add a0, ind
i2    = add ind, 1
i6    = add a2, ind
gep4  = getelementptr i8* x1, i6
```

Write Address Computation

```
store i41, i8* gep4
```

Computational Kernel:

```
i46   = phi [true, preh], [i53, body]
ind   = phi [0, preh], [i2, body]
i2    = add ind, 1
```

```
i7    = pop i8* gep0
i10   = pop i8* gep1
```

```
i9    = mul i7, a3
i12   = mul i10, a4
i19   = add i9, 32
i20   = add i19, i12
i41   = lshr i22, 6
i44   = cmp slt i41, 0;
i45   = xor i44, true;
i46   = cmp sgt i41, 255;
i47   = xor i46, true;
i48   = and i45, i47;
i50   = select i44, 0, 255;
i51   = select i48, i41, i50;
```

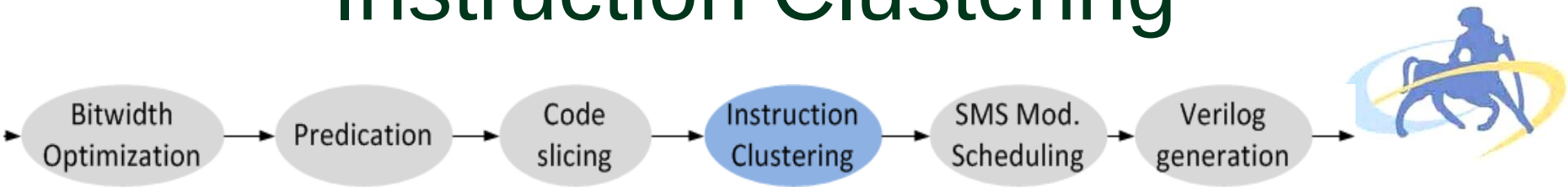
Data Computation

```
push i41, i8* gep4
```

```
i52   = icmp eq i2, 8
i53   = xor i52, true
br i52, exit, body
```

Termination

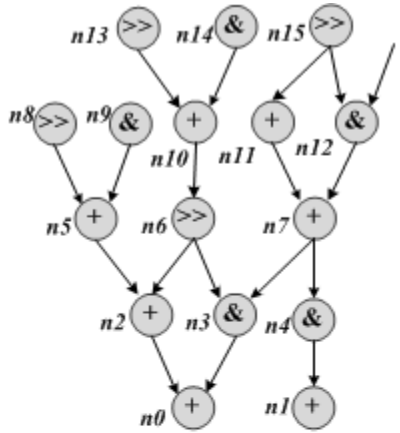
Instruction Clustering

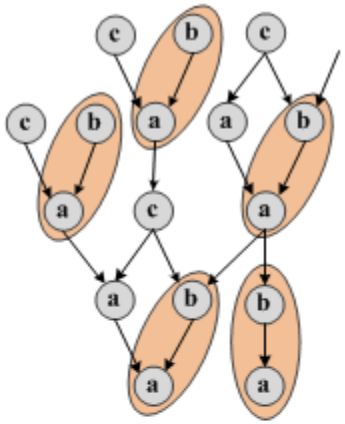


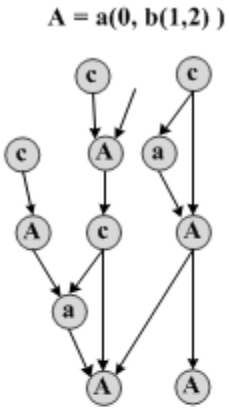
- Create clusters of primitive LLVM instructions
  - Macro instructions executed on synthesized macro functional units (MFUs)
  - For example, $[x + (y >> c_1)] + z \&\& c_2$ on one MFU
- Alleviate routing congestion problems
  - Prevalent in compute bound applications with large number of instructions
- Up to 50% area reduction
- Allows to place & route very complex designs

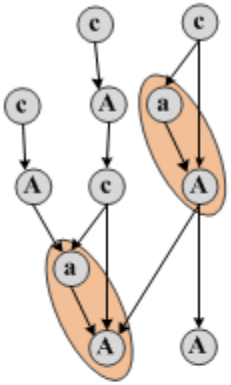
Instruction Clustering

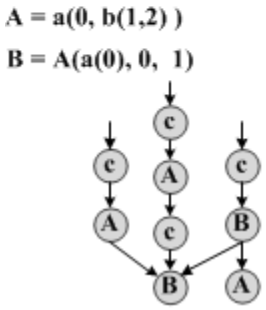
- Grammar-based DFG compression
- Create a hierarchy of potential macro-instructions

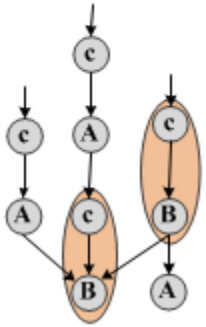


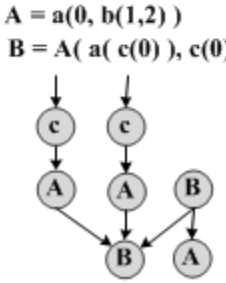


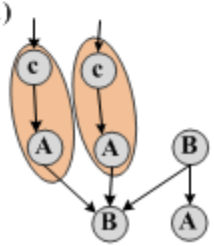


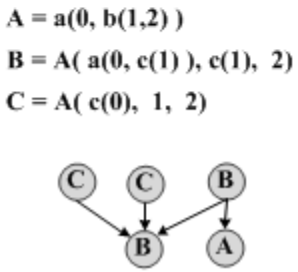


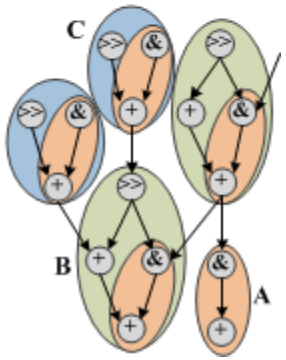






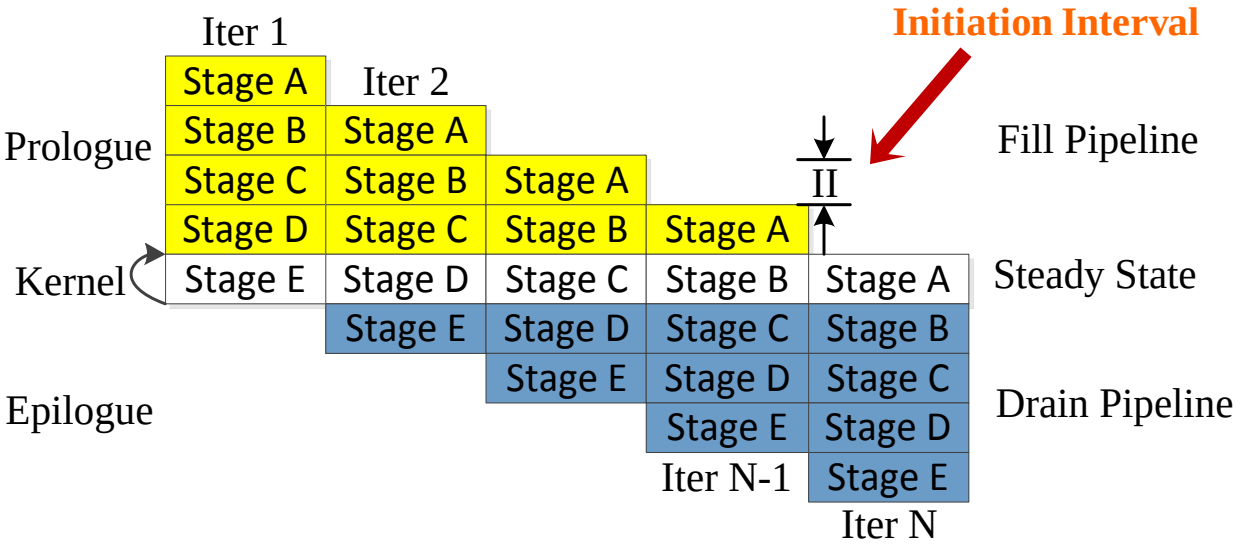




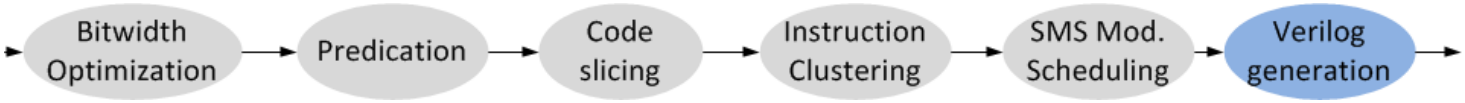


Modulo Scheduling

- Software Pipelining:
  - II: Initiation Interval.
- Swing Modulo Scheduling (SMS).



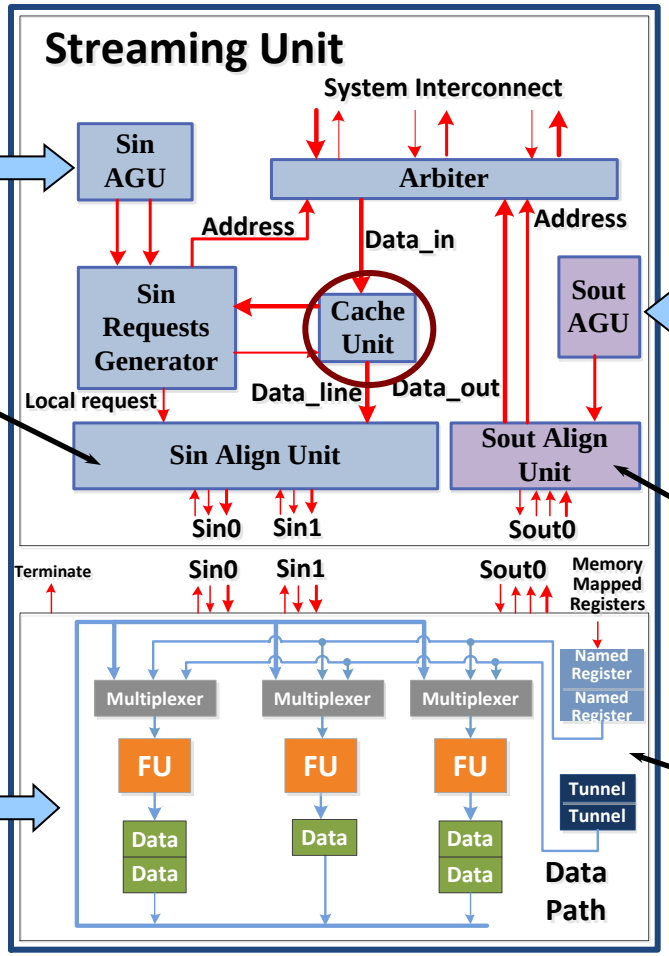
Verilog Generation: PE Architecture



Sin Kernel:

```
ind  = phi [0, preh], [i2, body]
i0   = add a0, ind
i2   = add ind, 1
i3   = add a0, i2
gep0 = getelementptr i8* x0, i0
gep1 = getelementptr i8* x0, i3
i7   = load i8* gep0
i10  = load i8* gep1
```

Feed Data in Order



Sout Kernel:

```
ind  = phi [0, preh], [i2, body]
i2   = add ind, 1
i6   = add a2, ind
gep4 = getelementptr i8* x1, i6
store i23, i8* gep4
```

Write Data in Order

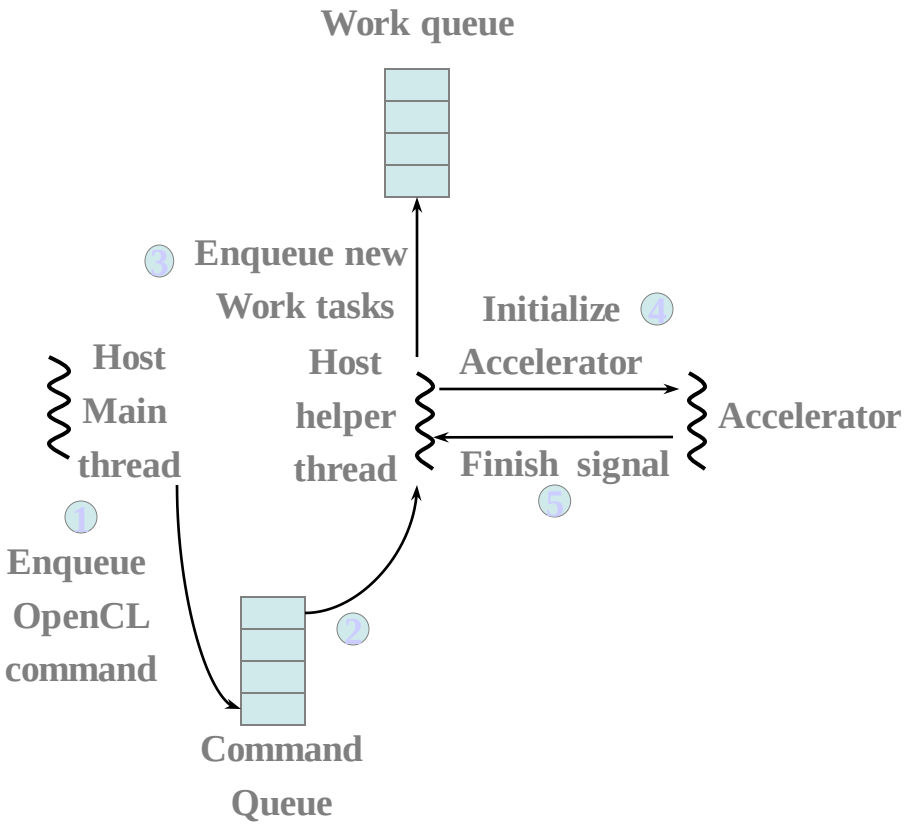
Computational Kernel:

```
i46  = phi [true, preh], [i41, body]
ind  = phi [0, preh], [i2, body]
i2   = add ind, 1
i7   = pop i8* gep0
i9   = mul i7, a3
i10  = pop i8* gep1
i12  = mul i10, a4
i19  = add i9, 32
i20  = add i19, i12
i23  = ashr i22, 6
push  i23, i8* gep4
i40  = icmpeq i2, 8
i41  = xor i40, true
br i40, exit, body
```

FU types, Bitwidths, I/O Bandwidth

Run-Time

- The OpenCL main program is executed as a main thread in the host processor of the platform
- Work-tasks are created by the helper thread.



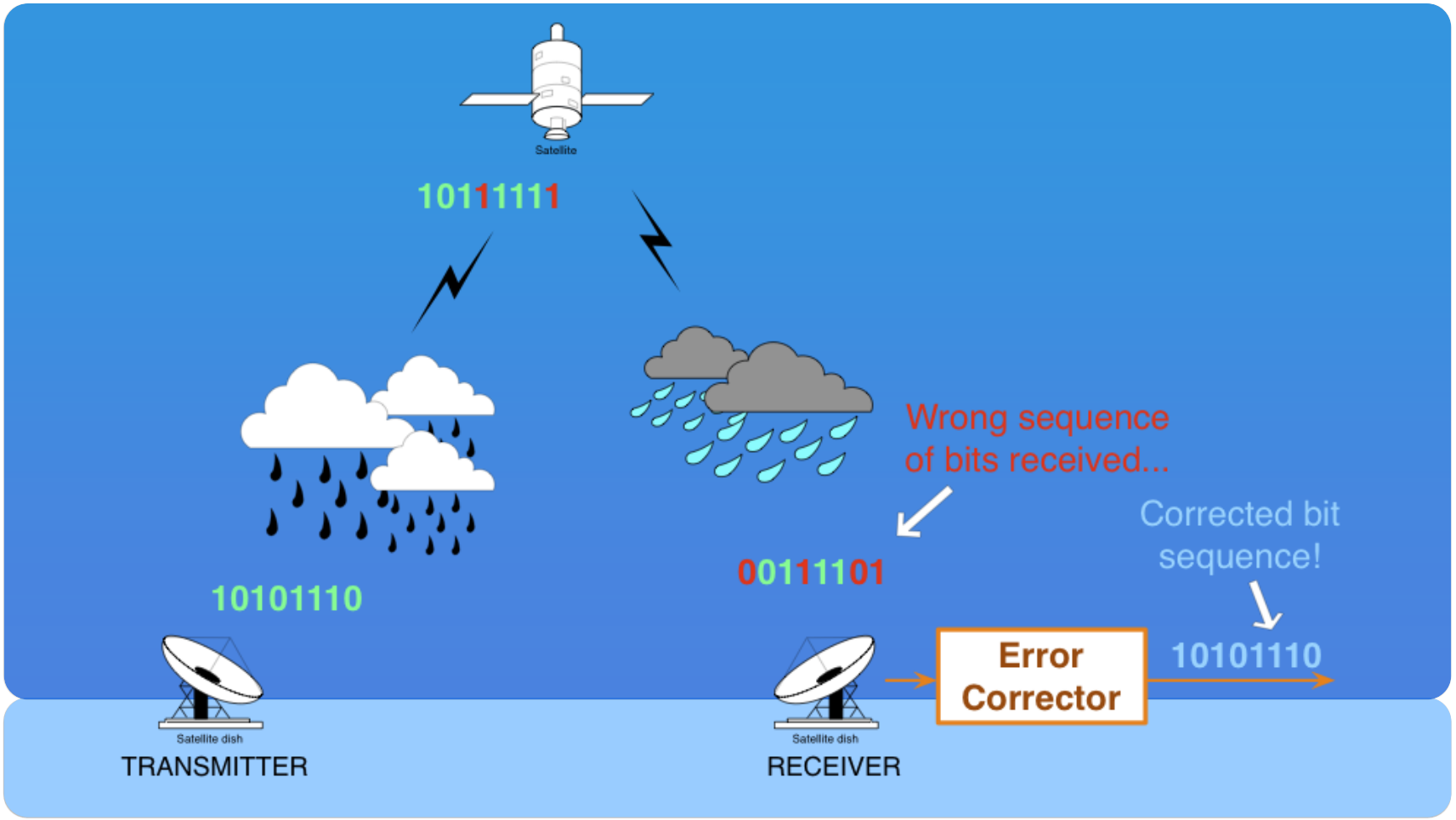
Outline

- Introduction
- OpenCL Programming Model
- Silicon OpenCL (SOpenCL)
  - Front-End
  - Back-End
  - Run-Time
- Experimental Evaluation
- Upcoming Challenges

Experimental Evaluation

- Applied on several OpenCL and C benchmarks:
  - Nvidia SDK
  - Rodinia
  - OpenDwarfs
  - Proprietary OpenCL code
- One interesting test-case: Low density parity check code (LDPC)
- Evaluation Methodology:
  - Three levels of resources availability
    - Minimal resources; one FU of each type (produce large II).
    - Unlimited resources (produce II = 1 in non-recurrent DFGs).
    - Moderate amount of resources (produce average II).

Application case study: Error correcting codes



LDPC decoder

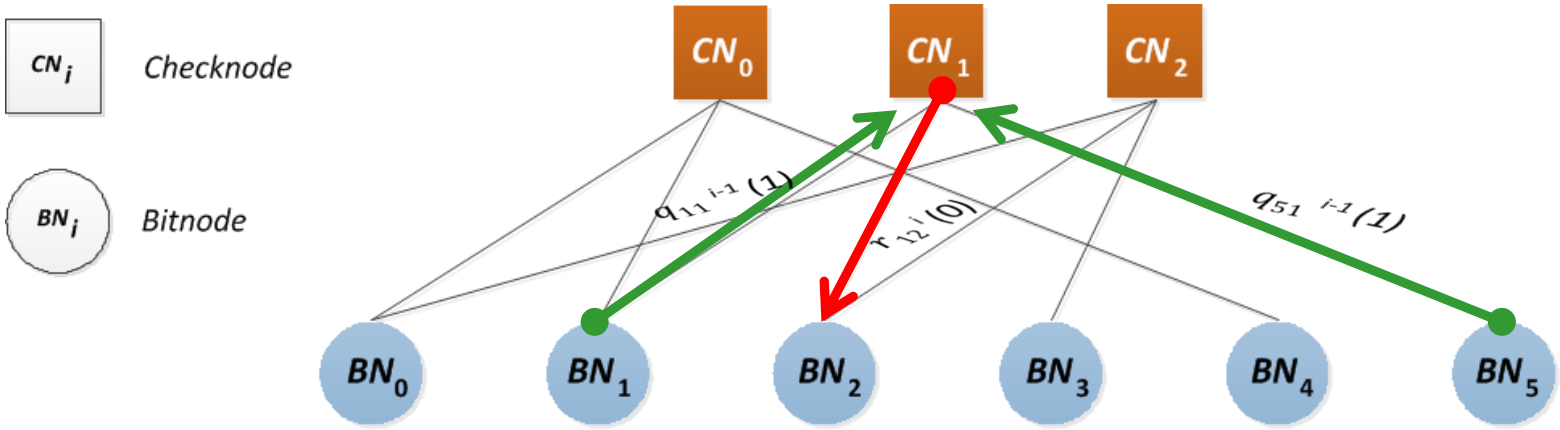
- Low Density Parity Check Code (LDPC) is a linear error correcting code
  - Used for transmitting messages thru noisy channels (e.g. DVB-S2- digital TV transmission over satellite).
  - Obtained from sparse bipartite graphs (coming next…)
  - Very computationally intensive
  - Large amounts of hardware
    - often resort to VLSI implementations

LDPC decoder

BNs and CNs exchange messages (i.e., probabilities) allowing reliable decision on a transmitted bit value

$r_{mn}{}^{i}(0)$ = prob. message from CN_m to BN_n at iteration i is 0

$$r_{mn}^{i}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in N(m) \setminus n} (1 - 2\, q_{n'm}^{i-1}(1))$$



LDPC decoder

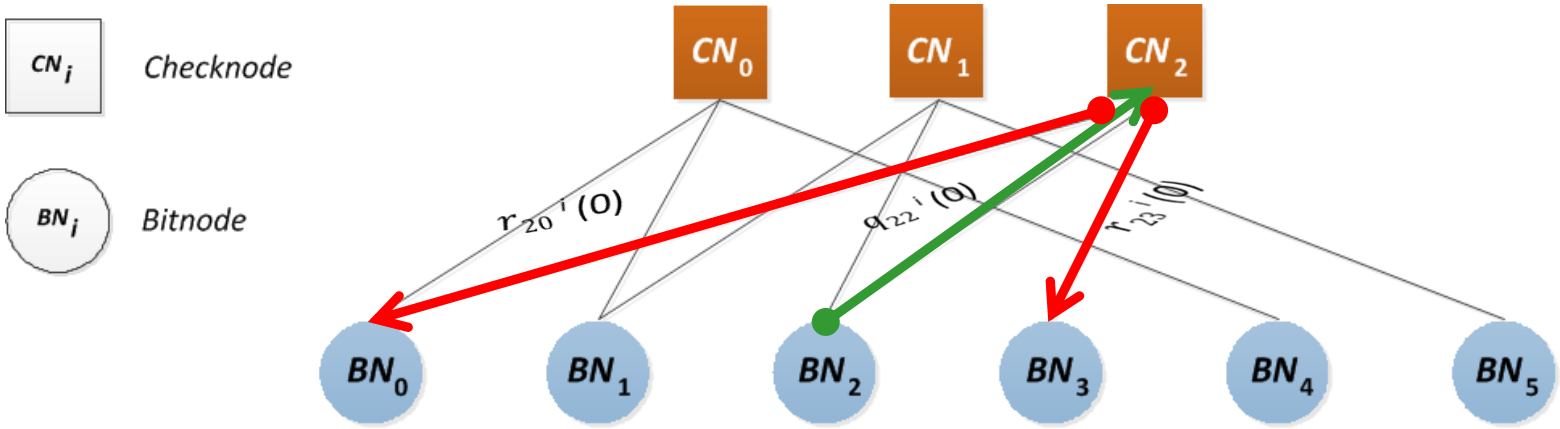
$q_{nm}{}^{i}(0)$ = prob. message from BN_n to CN_m at iteration i is 0

$$q_{nm}^{i}(0) = k_{nm}(1-p_n)\prod_{m'\in N(n)\setminus m} r_{m'n}^{i-1}(0)$$

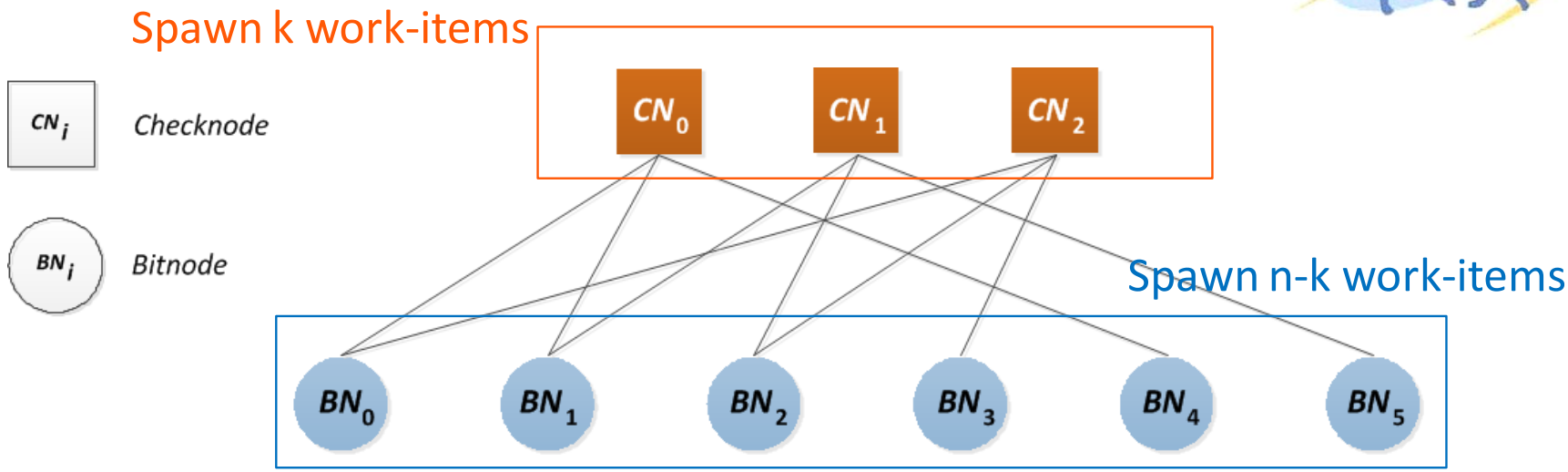
$$Q_n^{i}(0) = k_n(1-p_n)\prod_{m\in M(n)} r_{mn}^{i}(0)$$

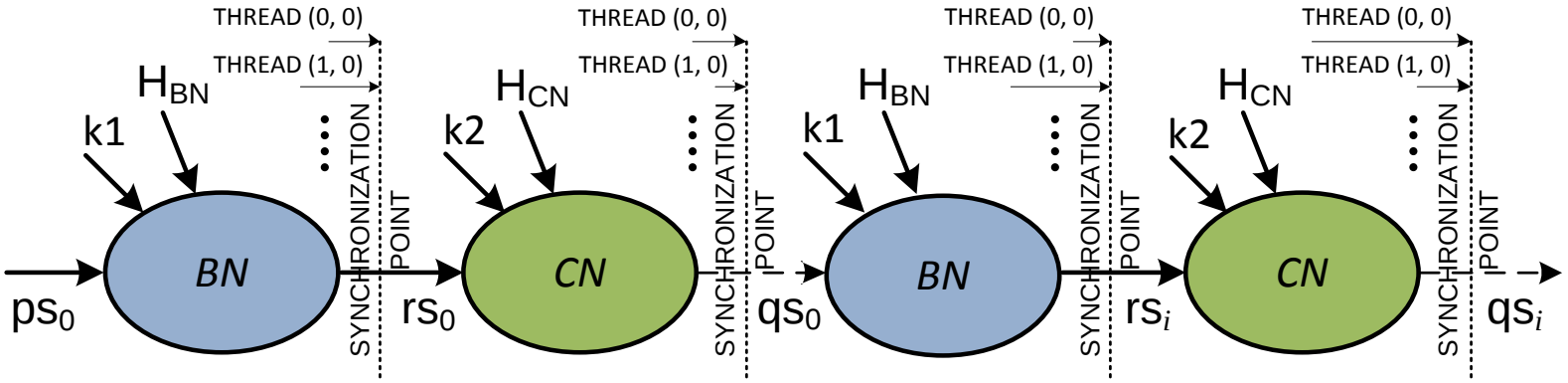
Then computes the a posteriori pseudo-probabilities and performs hard decoding:

$$\forall n, \quad \hat{c}_n = \begin{cases} 1 \Leftarrow Q_n^{i}(1) > 0.5 \\ 0 \Leftarrow Q_n^{i}(1) < 0.5 \end{cases}$$



LDPC (n, k) decoder OpenCL

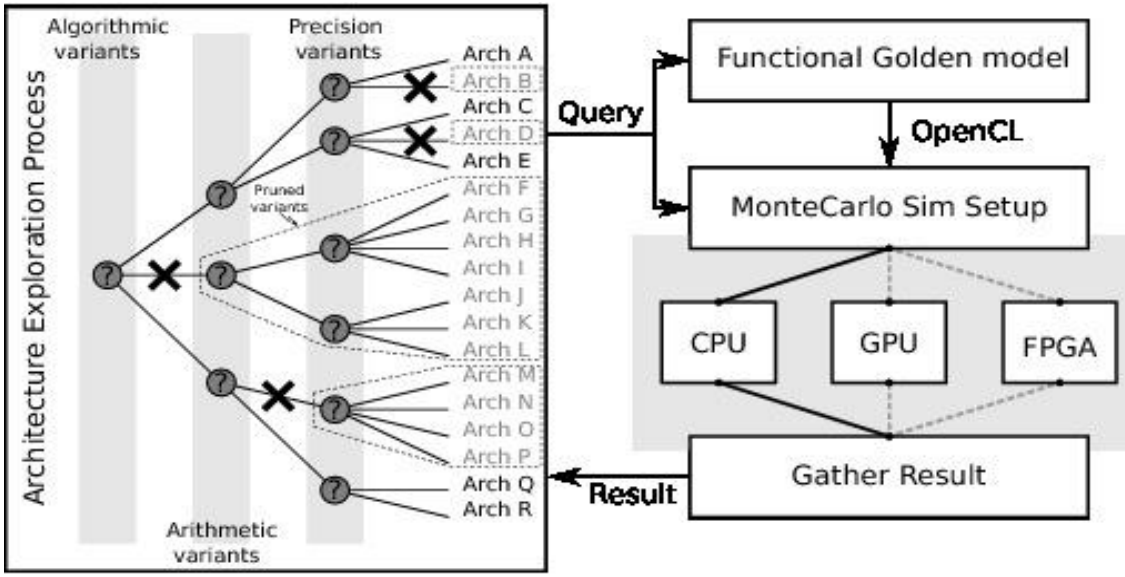




LDPC decoder exploration

System design often based on simulations for parameter space exploration

- Different LDPC codes (matrices) required by application
- Algorithmic variations
- Number of iterations
- Input data bitwidth

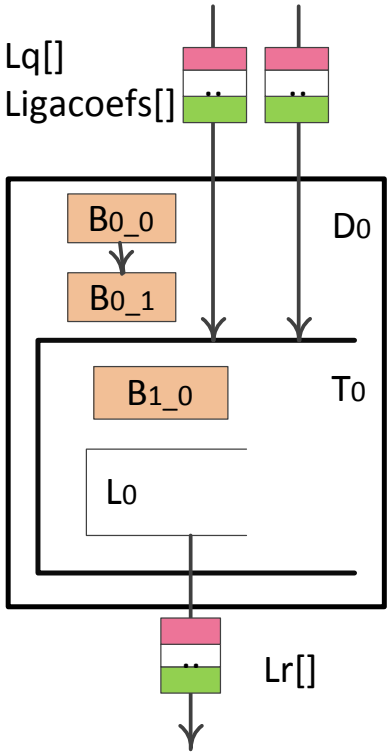


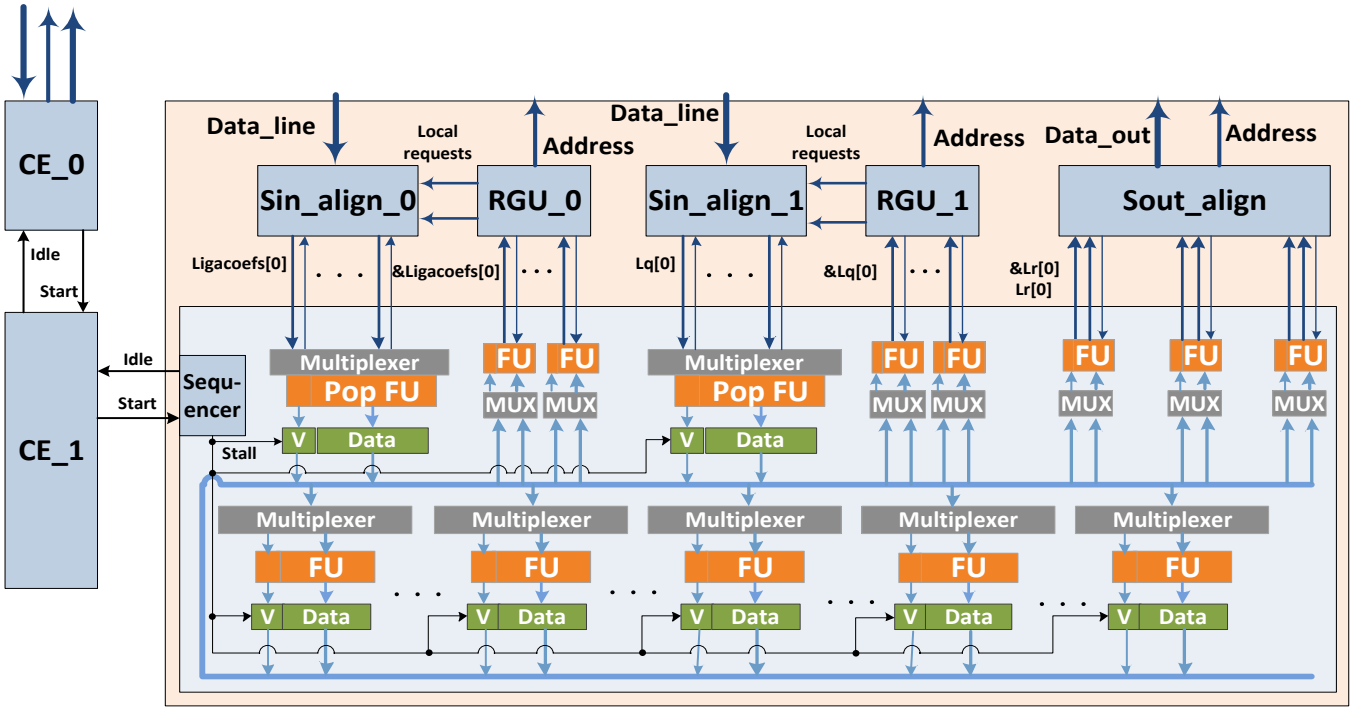
Is a write-once, run-anywhere approach viable, using a common programming model?

Experimental Setup

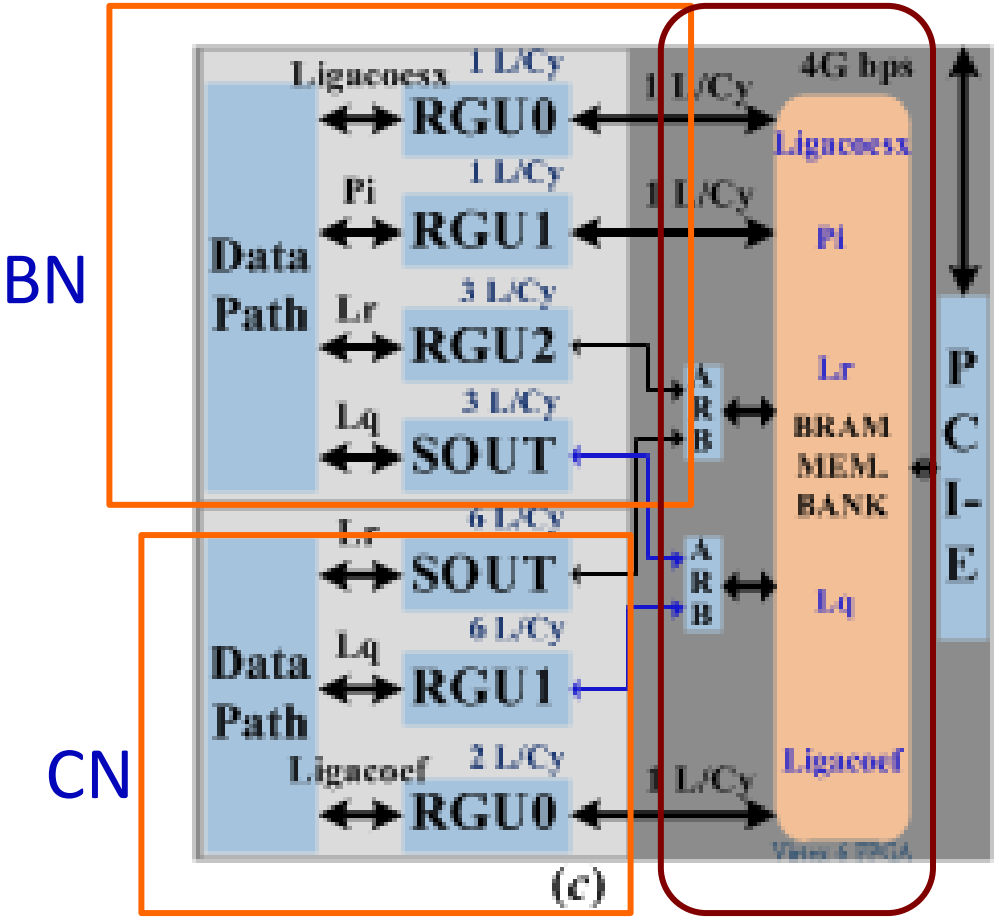
Target Arch.	Specifications
CPU	AMD Phenom X4 945 **Quad-Core** CPU system 3 GHz, 4 GB of DDR3
GPU	ATI Radeon HD 5870 GPU 2720 Single Precision GFLOPS and **1600** stream cores 1.2 GHz, 3 GB DDR5
FPGA	Xilinx Virtex-6 LX760 118,560 slices with four LUTs and eight flip-flops

LDPC CN kernel

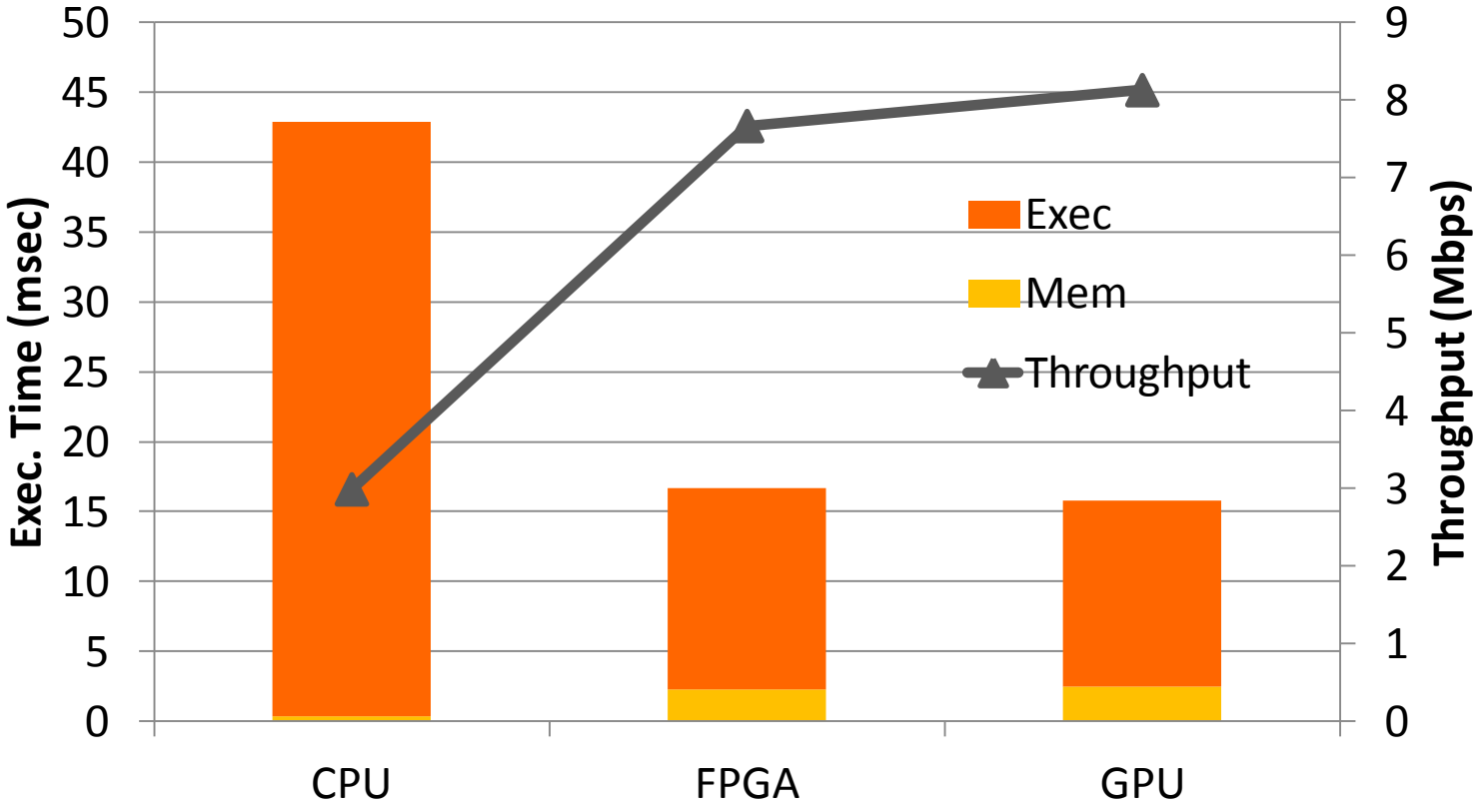




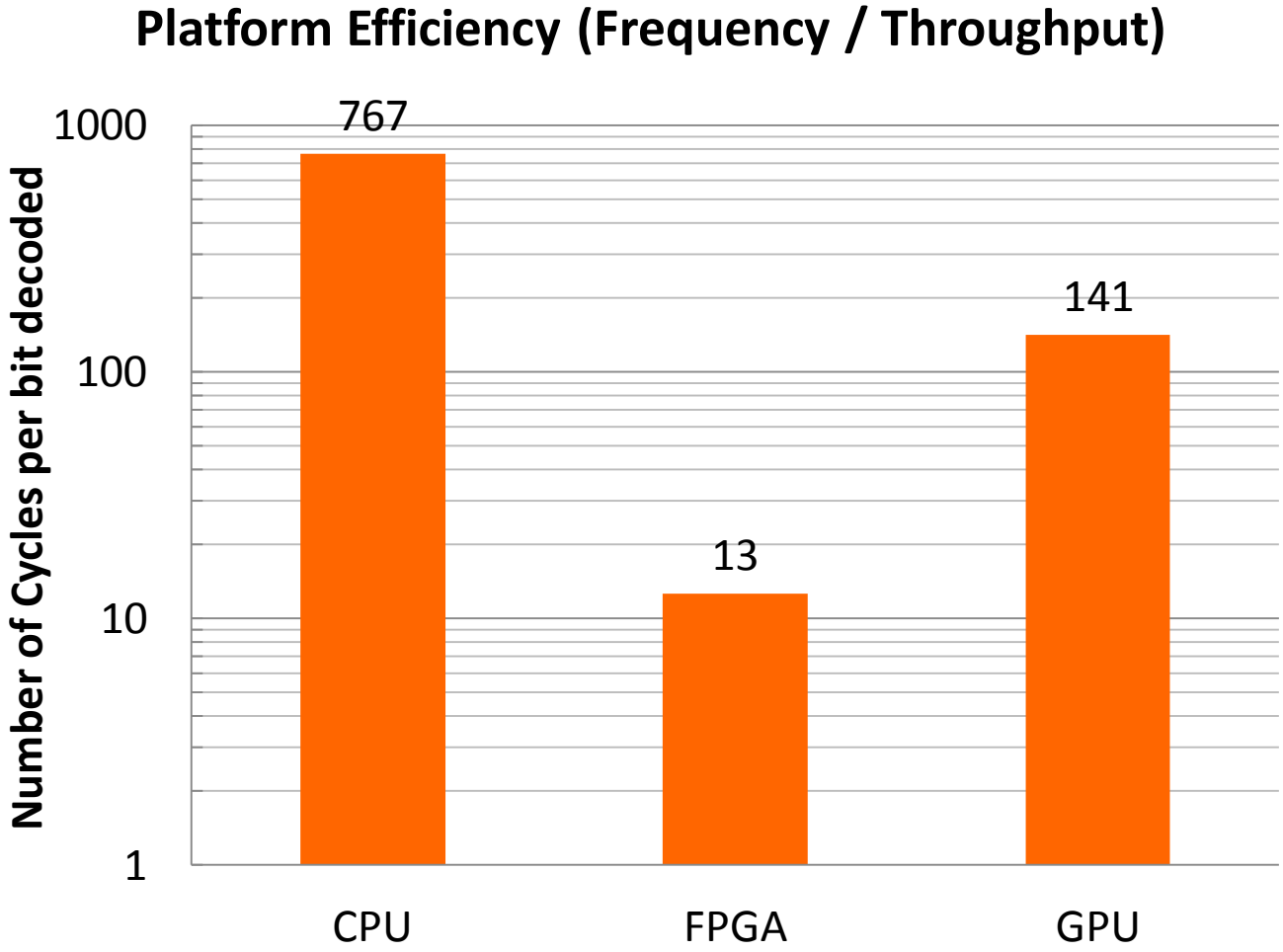
LDPC FPGA system



Performance results: Execution time & Throughput



Performance results: Efficiency



Area results: (8000, 4000) LDPC

CheckNode (CN)	Generic		
Iteration Interval (II)	16	8	1
Slice	17314	17552	24161
LUT	51745	51963	85731
FF	40865	34328	100669
Freq.	92	92	88
Latency	243	130	77
Total P&R Time (mins)	43	42	135

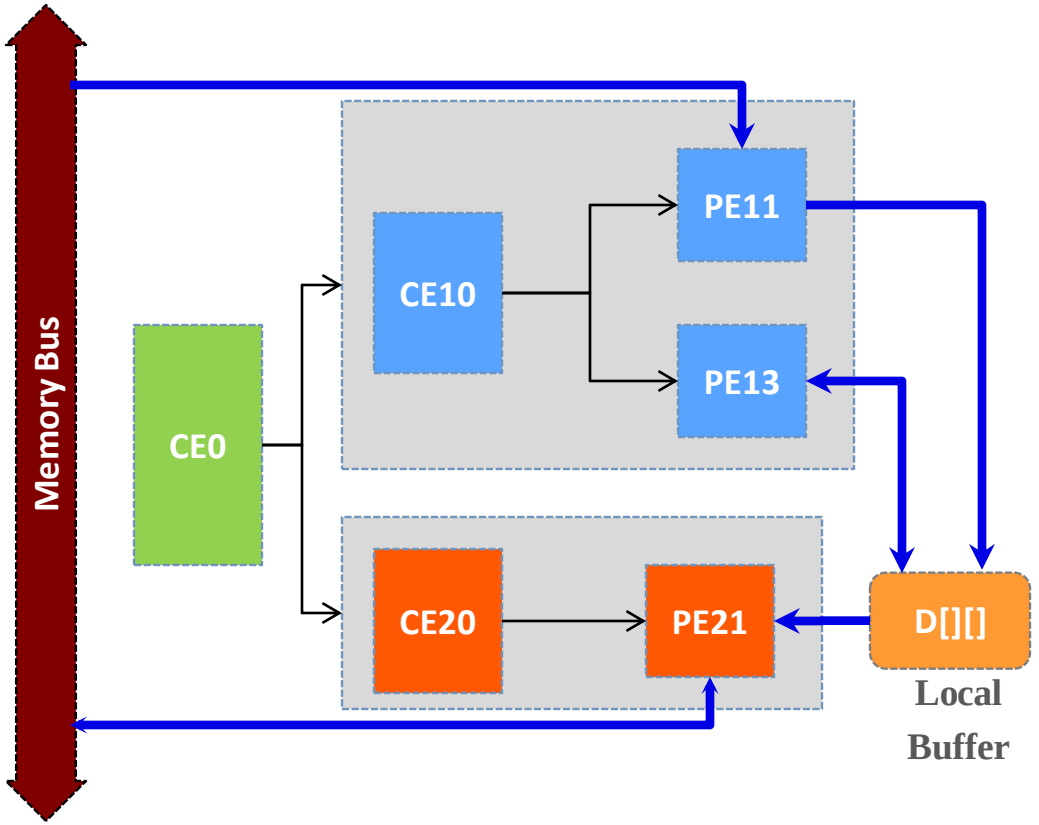
CheckNode (CN)	BitWidth 8		
II	16	8	1
Slice	12427	10160	17404
LUT	38065	29301	54594
FF	28577	23073	47316
Freq.	100	101	103
Latency	280	135	77
Total P&R Time (mins)	31	29	76

BitNode (BN)	Generic		
II	16	8	1
Slice	4605	5373	6269
LUT	13634	18242	21039
FF	15558	13907	24922
Freq.	172	168	169
Latency	112	61	44
Total P&R Time (mins)	19	21.5	27

BitNode (BN)	BitWidth 8		
II	16	8	1
Slice	4044	3731	5075
LUT	12458	11467	16757
FF	13016	12163	21725
Freq.	167	168	169
Latency	109	71	48
Total P&R Time (mins)	16	15	54

LU Decomposition

```
int LUD(int *A, …){
int D[GROUP_SIZE][64];
Basic_Block#00          // CE0
 triple_for( … )          {
   Basic_Block#10       // CE10
   for( n=0; n<64; n++) {
    Statements_Block#11 // PE11
    D[idx][n] = A[idx] – 5*A[idx+n];
   }
   Basic_Block#12       // CE10
   for ( n=0; n<64; n++) {
      Basic_Block#13    // PE13
      D[idx][n] += 7*D[n+1];
   }
 triple_for( … )          {
    Basic_Block#20      // CE20
    for ( n=0; n<64; n++) {
       Basic_Block#21   // PE21
       A[idx]  = D[n][idx] >> 1;
     }
  }
```



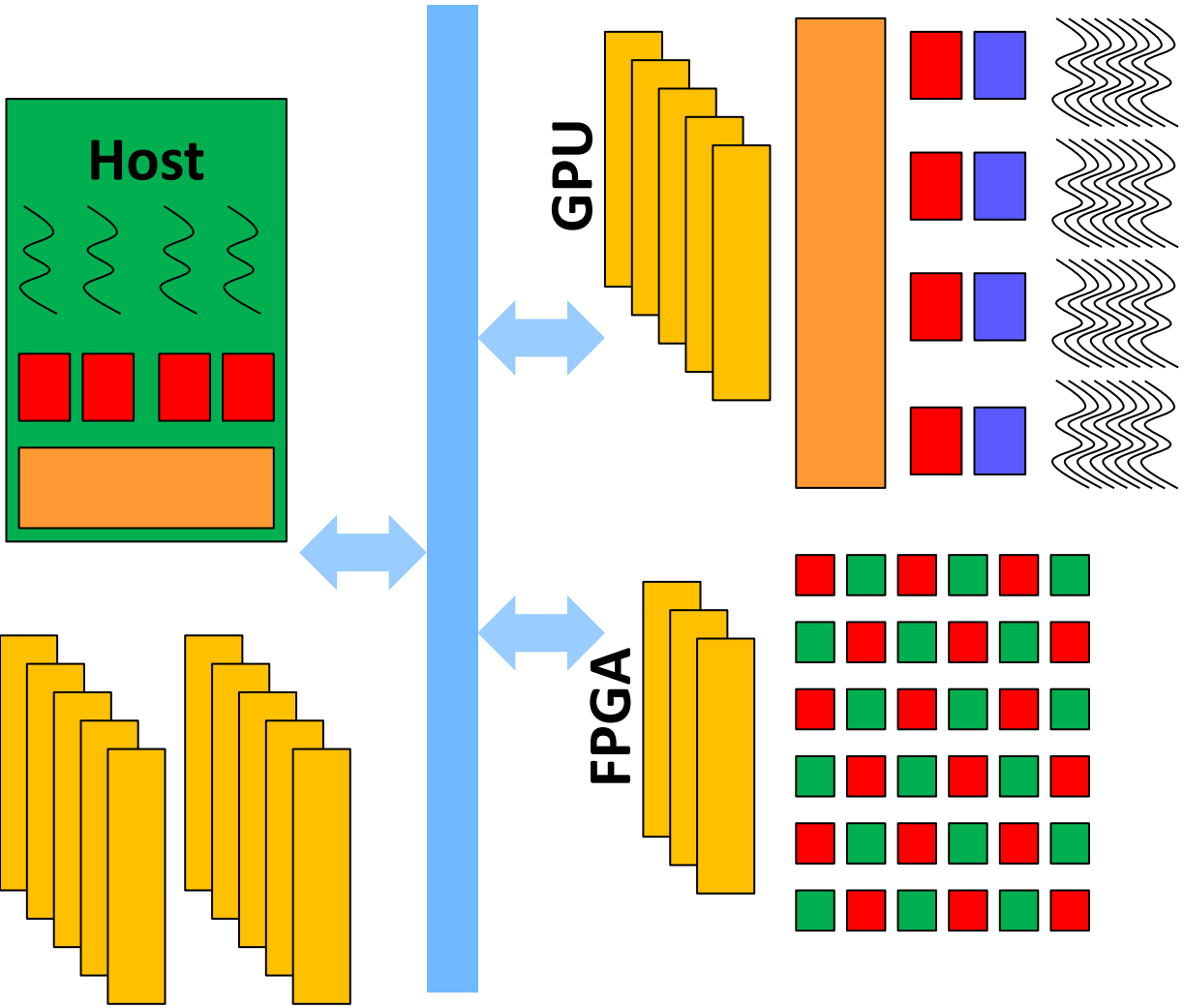
PE: Processing **E**lement.

CE: Control **E**lement.

Outline

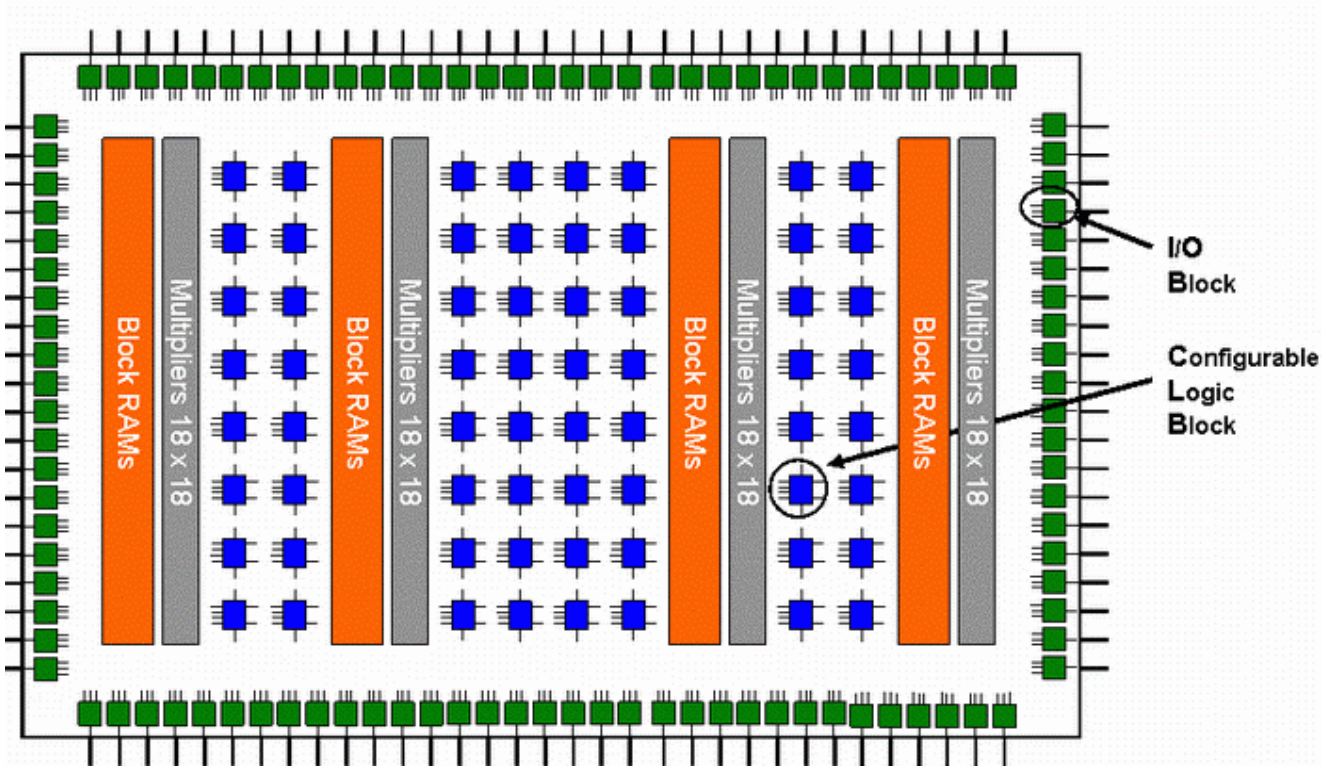
- Introduction
- OpenCL Programming Model
- Silicon OpenCL (SOpenCL)
  - Front-End
  - Back-End
  - Run-Time
- Experimental Evaluation
- Upcoming Challenges

Upcoming Challenges: Memory Analysis



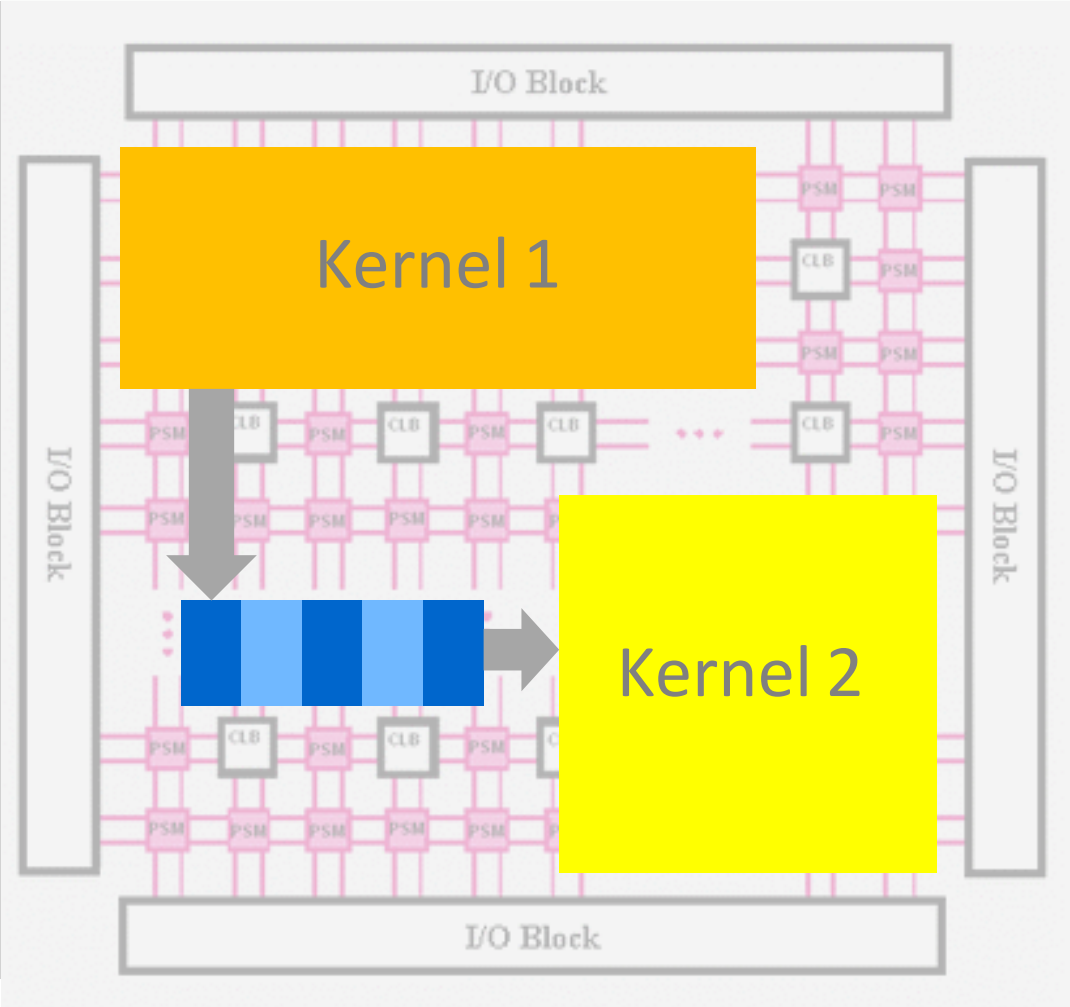
- Complex memory hierarchies
- Multiple address spaces
- Different capacities

Upcoming Challenges: Memory Analysis



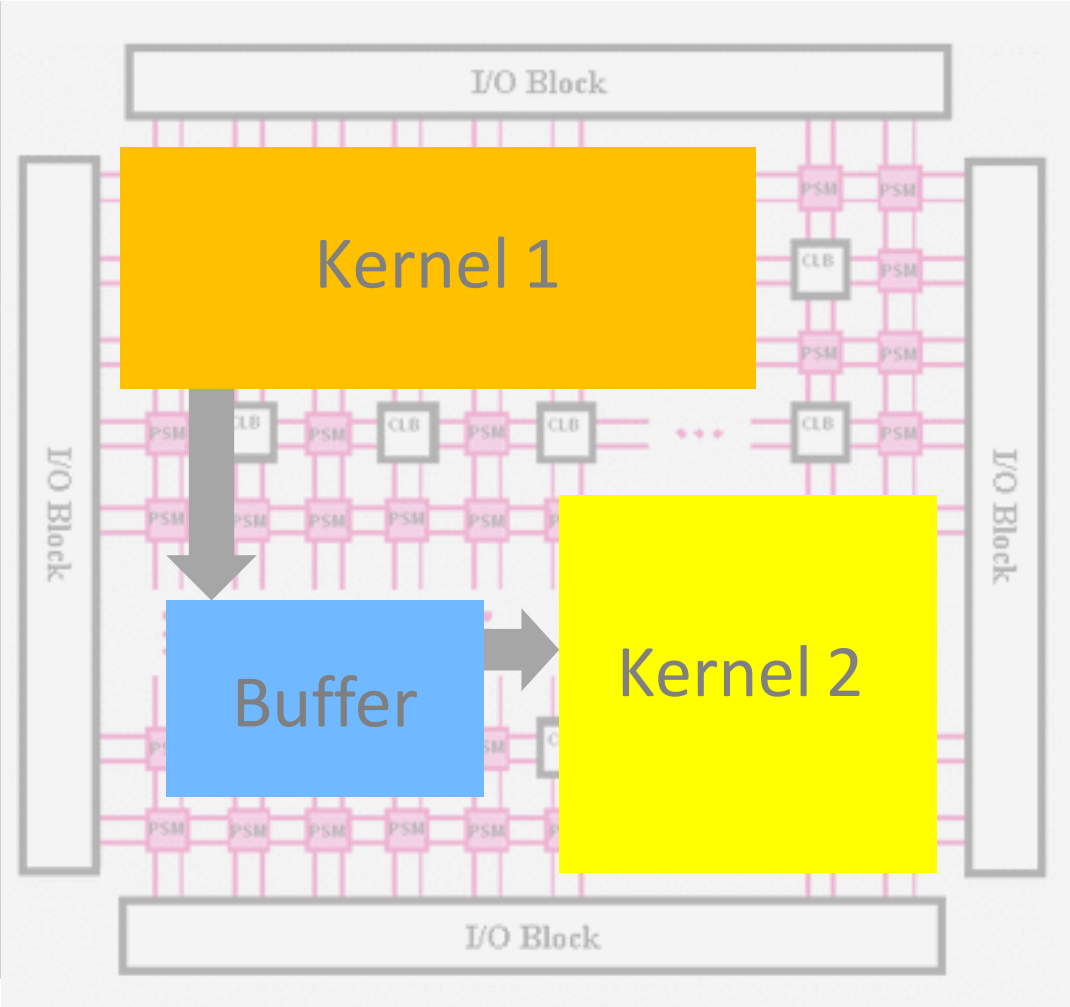
- Distributed memory on the FPGA

Upcoming Challenges: Memory Analysis



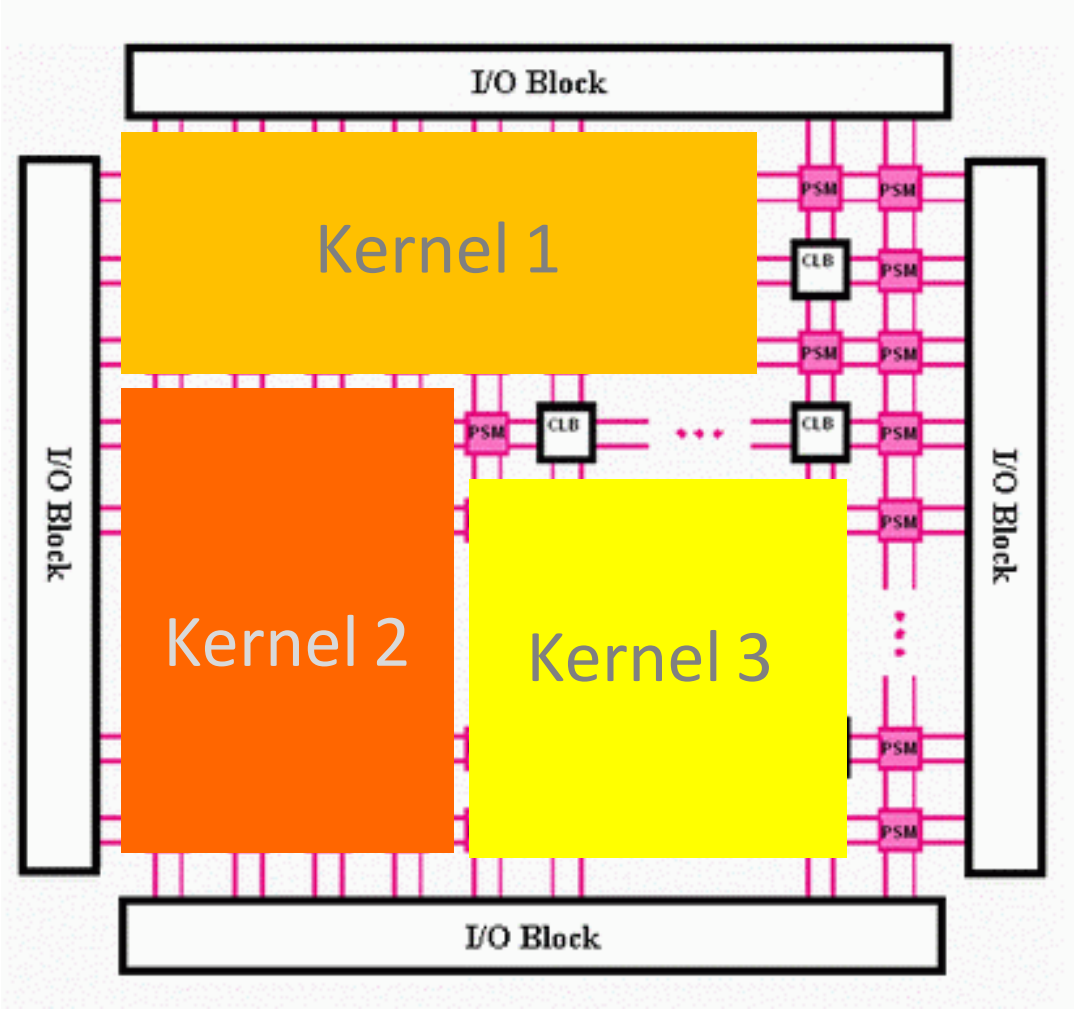
- Configurable mem. hierarchy on the FPGA

Upcoming Challenges: Memory Analysis

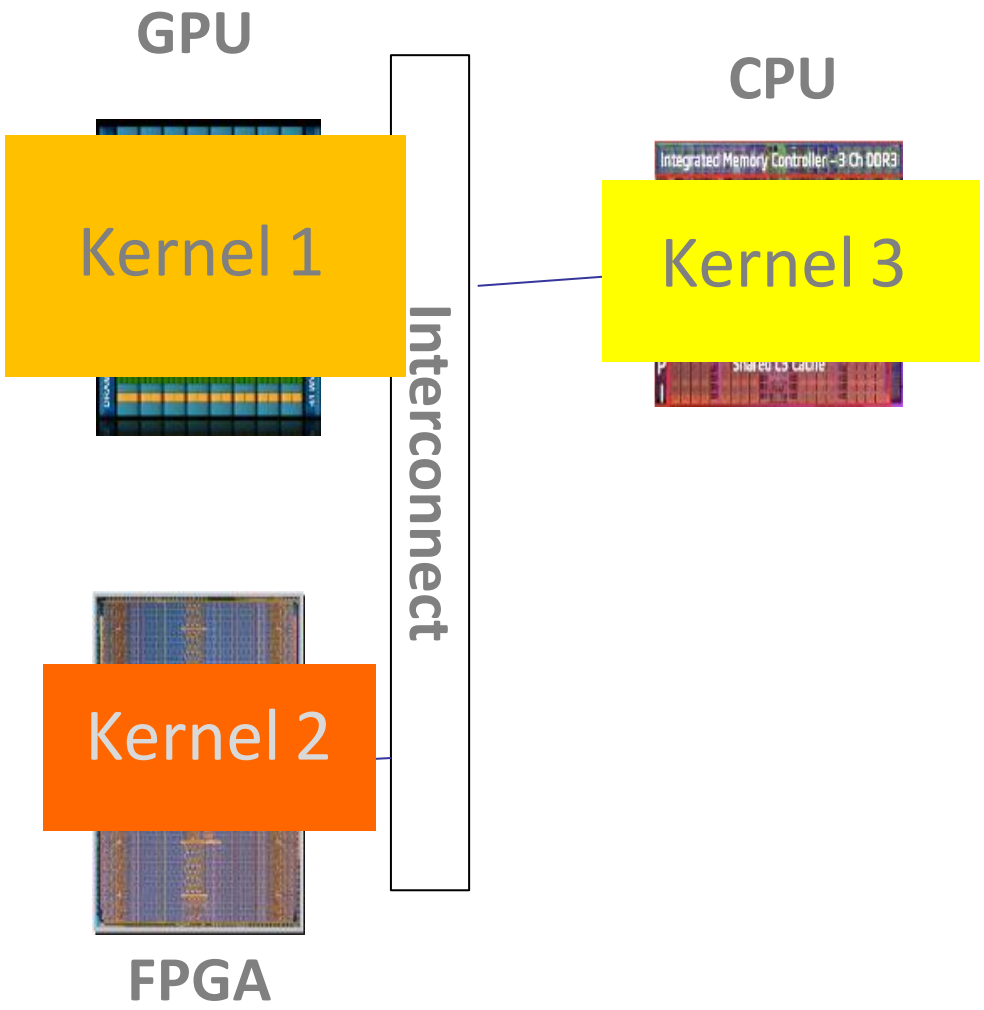


- Configurable mem. hierarchy on the FPGA

Upcoming Challenges: Multiple Accelerators



Upcoming Challenges: Multiple Accelerators



Upcoming Challenges: Language Extensions

- OpenCL
  - Arbitrary bitwidths
  - Kernel-to-kernel communication
- OpenACC

Conclusions

- SOpenCL: Toolflow to generate H/W designs from OpenCL code
  - "Turns" OpenCL to a hardware description language
  - Enables software developers to develop hardware
  - Facilitates the proliferation of FPGAs as HPC accelerator platforms
- Future research directions
  - Automatically partition and schedule software on heterogeneous systems
    - Redefine the H/W – S/W boundary
  - Analyze memory profiles and patterns

Thank you for your attention

cda@inf.uth.gr

http://www.inf.uth.gr/~cda