ECE ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Porting Performance across GPUs and FPGAs

Deming Chen, ECE, University of Illinois

In collaboration with Alex Papakonstantinou[1], Karthik Gururaj[2], John Stratton[1], Jason Cong[2], Wen-Mei Hwu[1]

1: ECE Department at University of Illinois, Urbana–Champaign

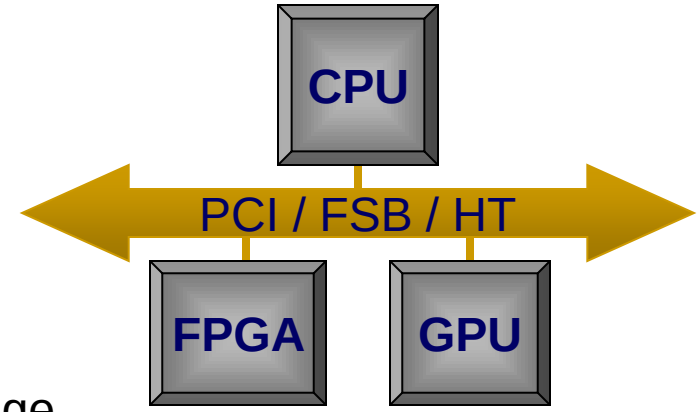
2: CS Department at University of California, Los-Angeles

Outline

- Motivation
- FCUDA Objectives
- FCUDA Flow
- Future Direction – FPGA in OpenCL
- Conclusions

Motivation

- The shift towards parallel computing has emphasized the potential of devices that can offer massive compute parallelism:
  - GPUs (NVIDIA, AMD-ATI)
  - FPGAs (Xilinx, Altera)
  - Many-core processors (IBM Cell, Tilera TILE64, Intel 48-core SCC)
- High performance computing is moving towards heterogeneous systems that combine
  - Multi-core CPUs with
  - Accelerators to extract more parallelism at a **lower power** footprint
- Other examples
  - Fusion: CPU+GPU on the same die
  - Stellarton: CPU+FPGA in the same package
  - Convey HC 1: CPU+Multi-FPGA sharing memory



Computing on FPGAs - Advantages

- Versatile mapping of application-specific parallelism:
  - Coarse & fine grained parallelism
  - Data and task parallelism
  - Flexible pipelining schemes
- Low power high-performance computation
  - High computational density per Watt compared to CPUs and GPUs
  - High reliability due to lower operating temperatures
- Deployment flexibility
  - Deployed as CPU accelerator (co-processor) or
  - Autonomous System on Chip implementation

Computing on FPGAs – Challenges

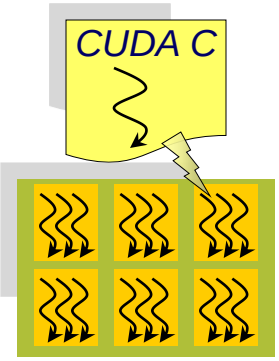
- FPGA programming abstraction is low
  - Knowledge on hardware design and device details required (e.g. in VHDL, Verilog)
- Time consuming synthesis flow increases complexity of identifying performance-optimal implementation
  - Interdependencies between cycles, frequency and concurrency
  - Identifying optimal mapping of application parallelism onto hardware is not trivial
- High-level synthesis (HLS) tools help raise the abstraction, but
  - Parallelism extraction may be limited by programming model
  - May not offer evaluation and selection of best parallelism extraction for performance

FCUDA: CUDA-to-FPGA

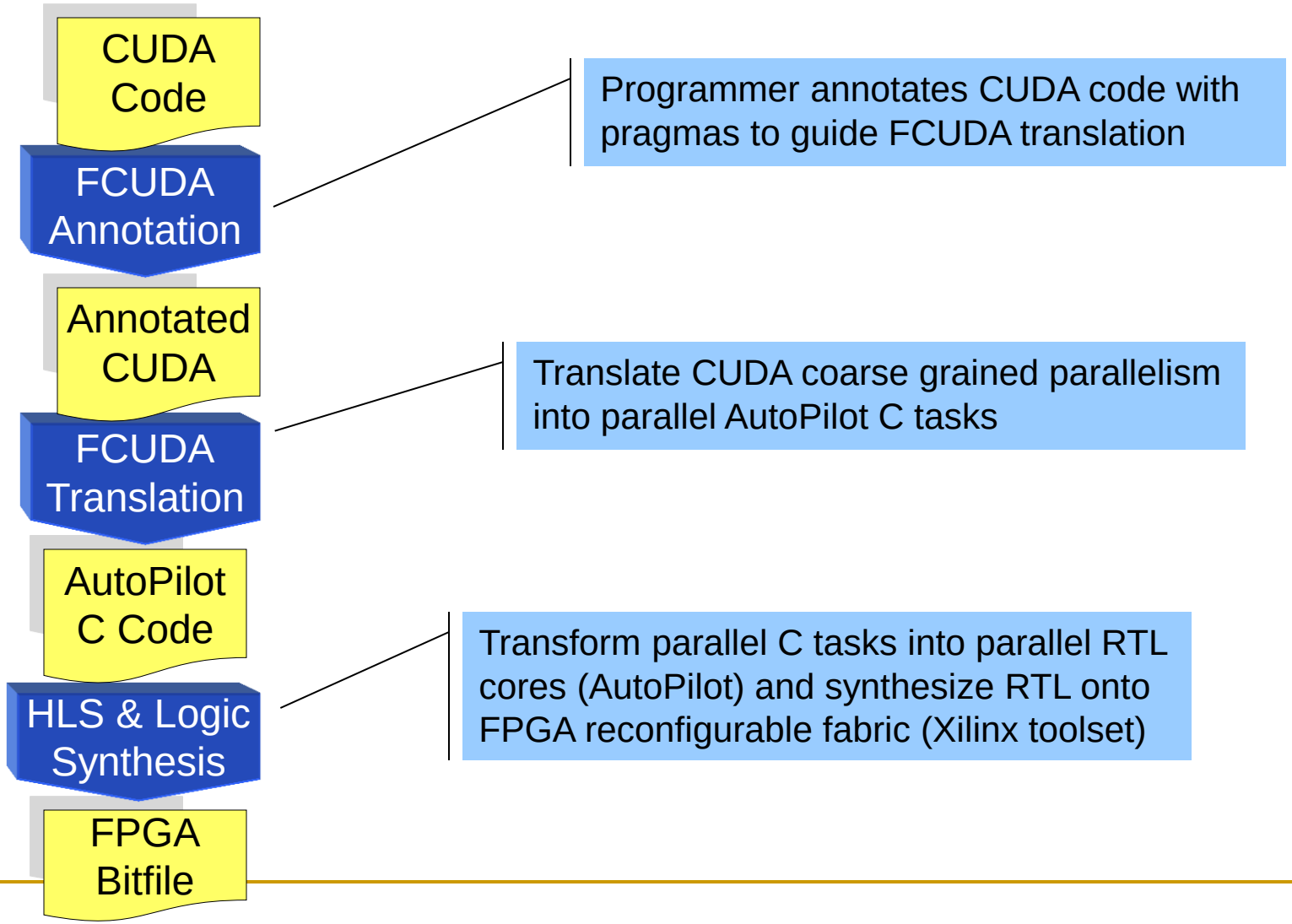
- Use CUDA code in tandem with HLS to:
  - enable high abstraction FPGA programming
  - leverage different types of parallelism during hardware generation

- CUDA: C-based parallel programming model for GPUs
  - Concise expression of coarse grained parallelism
  - Large amount of existing applications
  - Good model for providing common programming interface for kernel acceleration on GPUs & FPGAs

- AutoPilot: Advanced HLS tool (from AutoESL, now Xilinx)
  - Automatic fine-grained parallelism extraction
  - Annotation-driven coarse-grained parallelism extraction

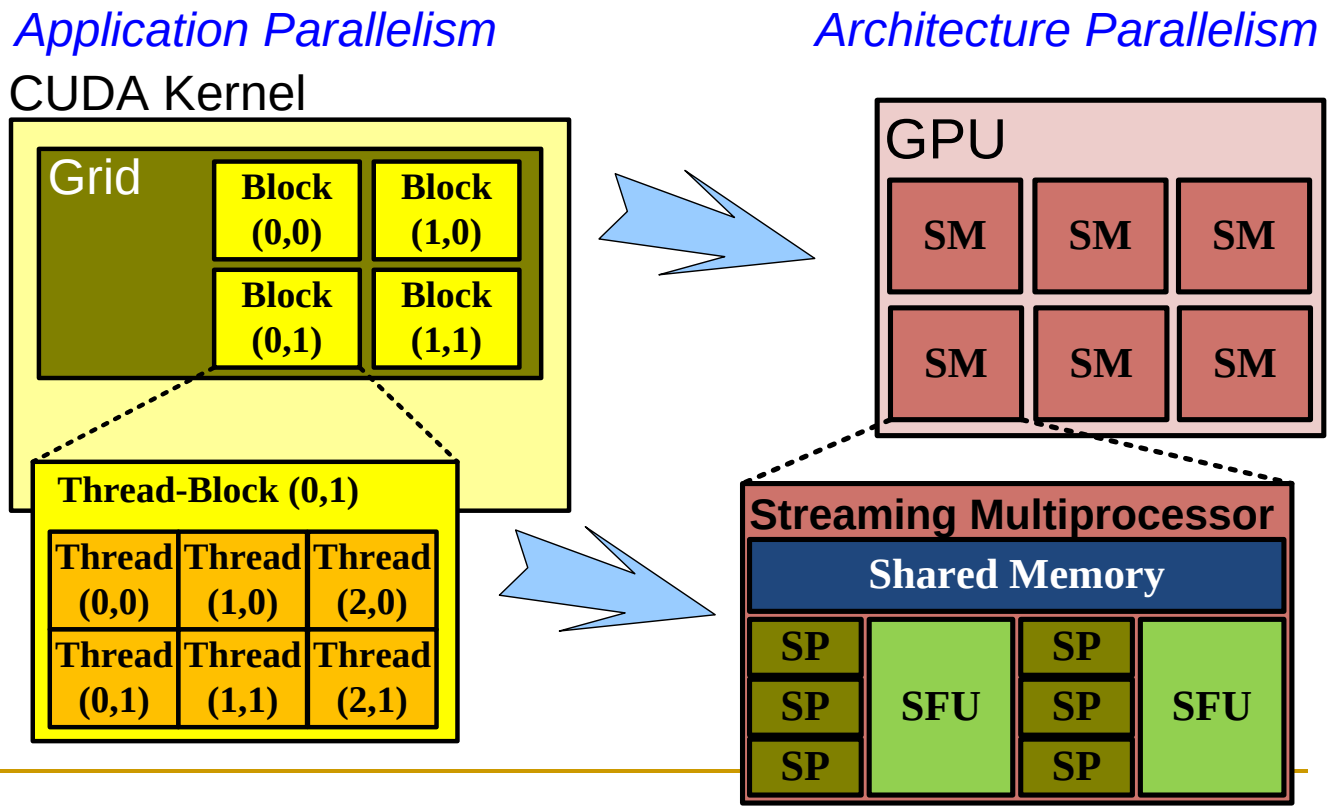


FCUDA Flow


CUDA Code
FCUDA Annotation
Annotated CUDA
FCUDA Translation
AutoPilot C Code
HLS & Logic Synthesis
FPGA Bitfile
Programmer annotates CUDA code with pragmas to guide FCUDA translation
Translate CUDA coarse grained parallelism into parallel AutoPilot C tasks
Transform parallel C tasks into parallel RTL cores (AutoPilot) and synthesize RTL onto FPGA reconfigurable fabric (Xilinx toolset)

CUDA Programming Model

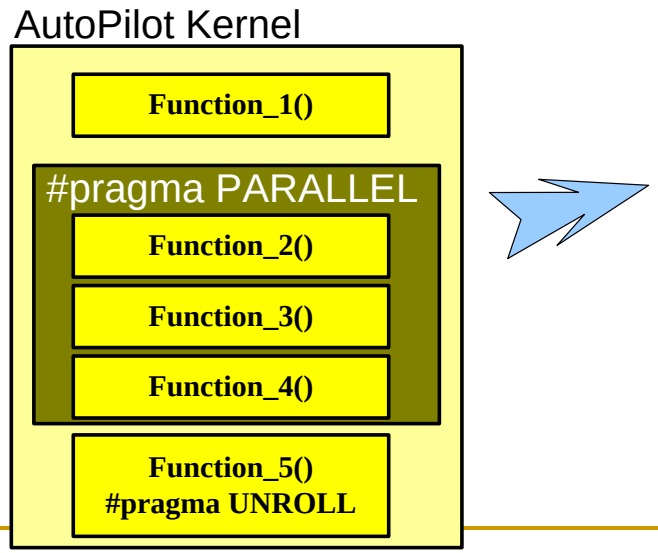
- Threads are clustered into thread-blocks
  - Each thread-block is assigned to one Streaming Multiprocessor (SM)
  - Each thread runs on a Streaming Processor (SP)

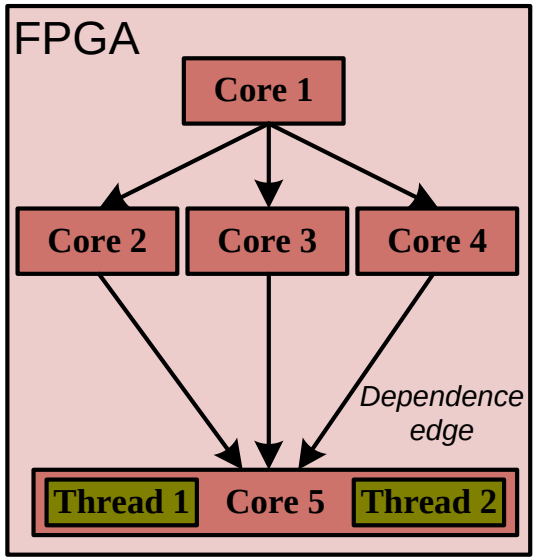


AutoPilot Programming Model

- Coarse-grained parallelism is represented at the function level
  - Each function is transformed into a custom core*
  - Functions annotated with the PARALLEL pragma are transformed into concurrently executing cores
  - Non-annotated functions are transformed into sequentially executing cores (represented by dependence edges)

Application Parallelism *Architecture Parallelism*

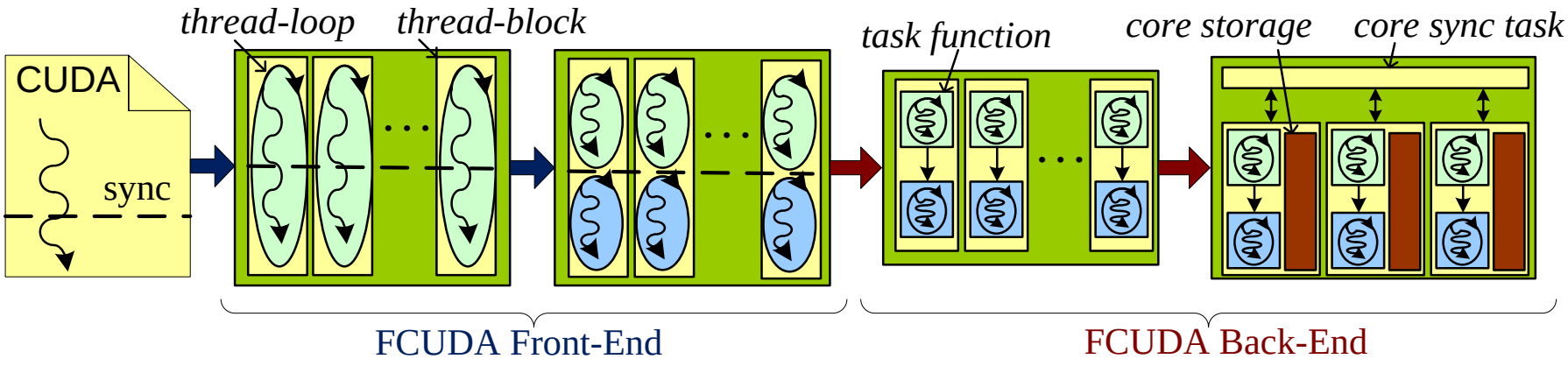




* ***Core*** := allocated set of resources required to execute corresponding procedure computation

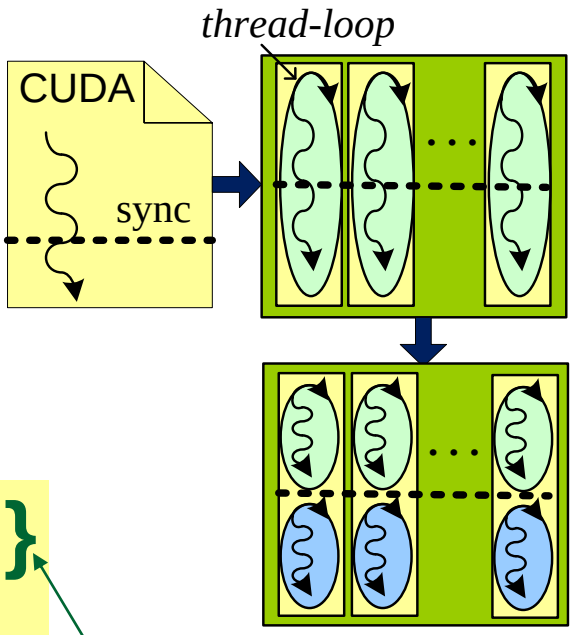
FCUDA Implementation Overview

- The FCUDA translation consists of two main stages:
  - FCUDA Front-End stage:
    - Convert **logical threads** into **explicit thread-loops**
    - Based on the MCUDA framework (John Stratton et al., “MCUDA: An efficient implementation of CUDA kernels on multi-core CPUs”)
  - FCUDA Back-End stage:
    - Extract **coarse grained parallelism** at the thread-block level
- Implemented with the Cetus compiler infrastructure
  - S. Lee et al., “Cetus - An extensible compiler infrastructure for source-to-source transformation,” 2003.



Front-End Transformations

- Serialize logical threads in **thread-loops**
  - Thread-blocks are a good granularity for coarse-level parallelism extraction on the FPGA
- Handle intra-block synchronization at:
  - CUDA thread-block sync statements
  - Annotated FCUDA task boundaries



Input CUDA code

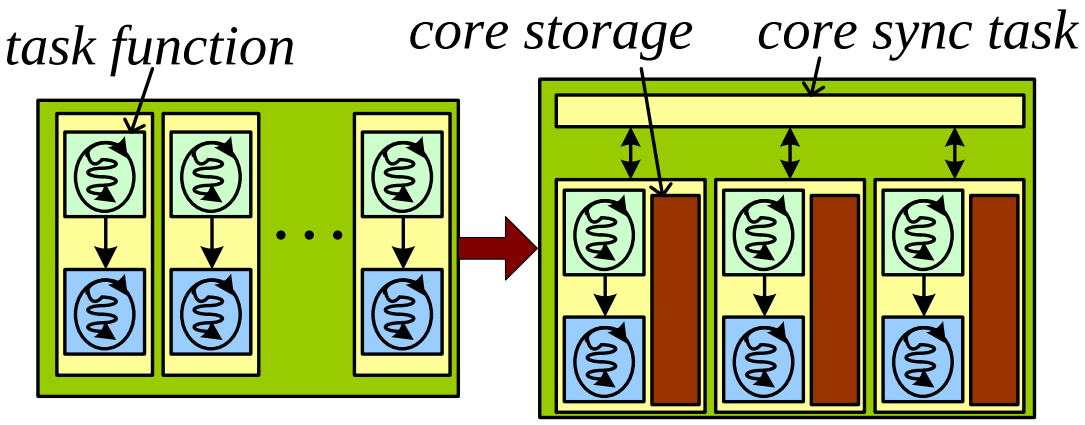
```
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

__syncthreads();

for ( k = 0; k < BLOCK_SIZE; ++k)
  Csub += AS(ty, k) * BS(k, tx);
```

```
for (ty=0; ty<blockDim.y; ++ty)
  for (tx=0; tx<blockDim.x; ++tx) {
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
} }

for (ty=0; ty<blockDim.y; ++ty)
  for (tx=0; tx<blockDim.x; ++tx) {
    for ( k = 0; k < BLOCK_SIZE; ++k)
      Csub += AS(ty, k) * BS(k, tx);
} }
```

thread-loops

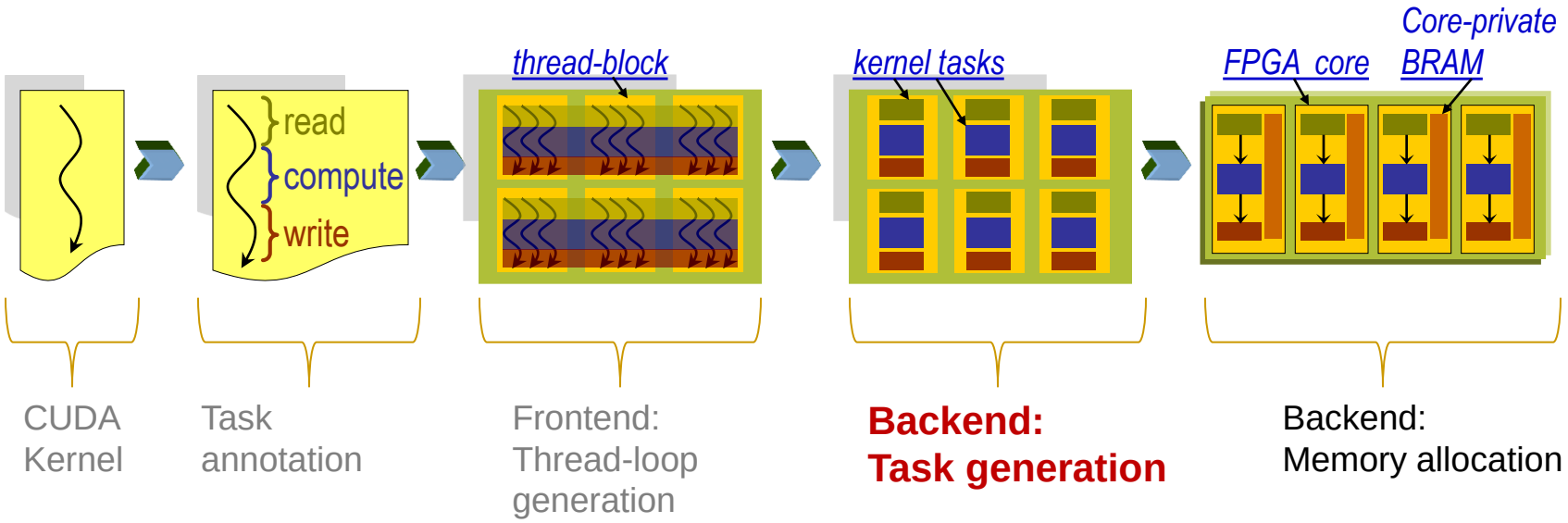
Back-End Overview

- Generate task functions based on FCUDA
- Leverage task synchronization and thread-block scheduling
- Manage data storage allocation and data communication between generated functions



Task Generation

- Kernel decomposition into compute & data-transfer tasks
  - Aggregate off-chip transfers into coalesced blocks
  - Transform data transfer blocks into DMA bursts
- Coarse Grain Parallelism Exposure
  - Threadblock → Core (or PE)

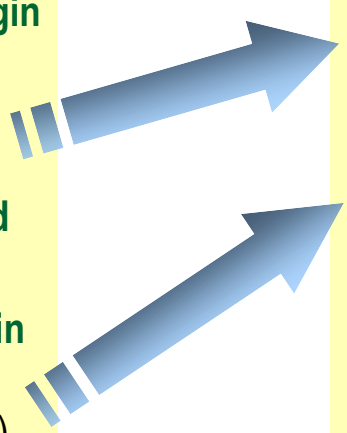


Task Generation Code Example

- Identify FCUDA annotated tasks and generate task functions for them
  - Analyze data accesses within task and pass necessary variables through task function parameters list
    - Identify off-chip and on-chip allocated variables
  - Replace FCUDA annotated task code in kernel with task function call

Front-End generated code

```
#pragma FCUDA TRANSFER begin
  for ( <thread-loop> )
    As[ ]=A[ ];
    Bs[ ]=B[ ];
#pragma FCUDA TRANSFER end

. . .
#pragma FCUDA COMPUTE begin
  for ( <thread-loop> )
    for (k=0; k<BLOCK_SIZE; ++k)
      Csub += As(ty,k) * Bs(k, tx);
#pragma FCUDA COMPUTE end
```



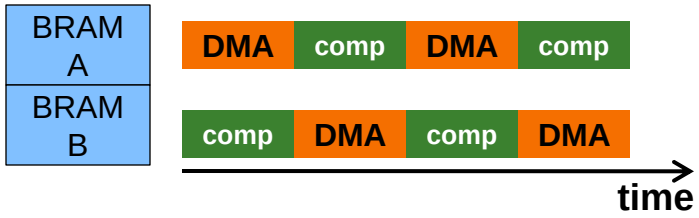
```
void fetch (volatile int* A, volatile int* B, int As[ ],
            int Bs[ ], …) { … }

void compute (int Csub[ ], int As[ ], int Bs[ ],
              dim3 blockDim, … ) { … }

void matrixMul (volatile int* A, volatile int* B, … ) {
  …
  fetch (A, B, As, Bs, …);
  compute (Csub, As, Bs, blockDim, …);
  …  }
```

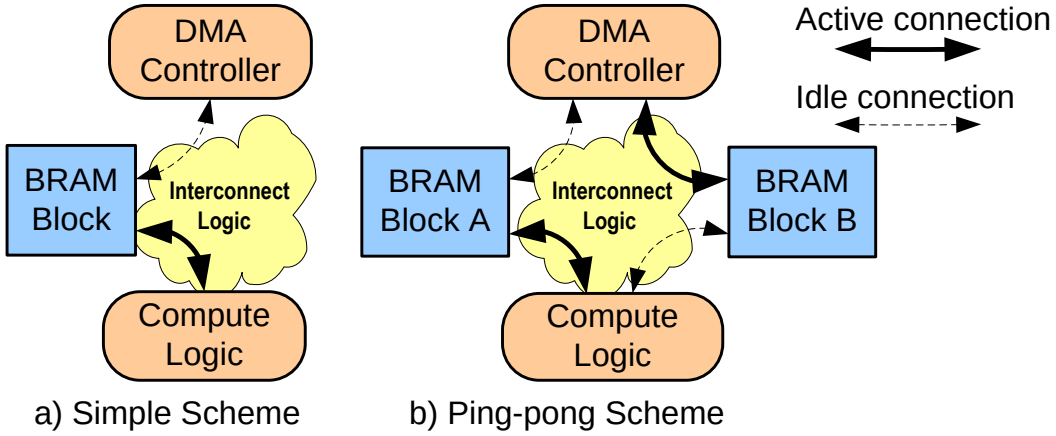
Task Synchronization

- Pragma-driven source code transformation
  - Sequential: temporally interleave compute & transfer
  - Ping-Pong: temporally overlap compute & transfer
    - Higher BRAM cost

Sequential scheme

Ping-Pong scheme

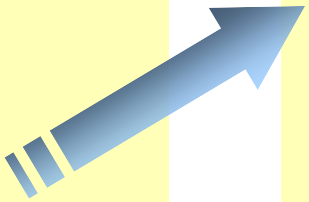




Task Synchronization Code Example

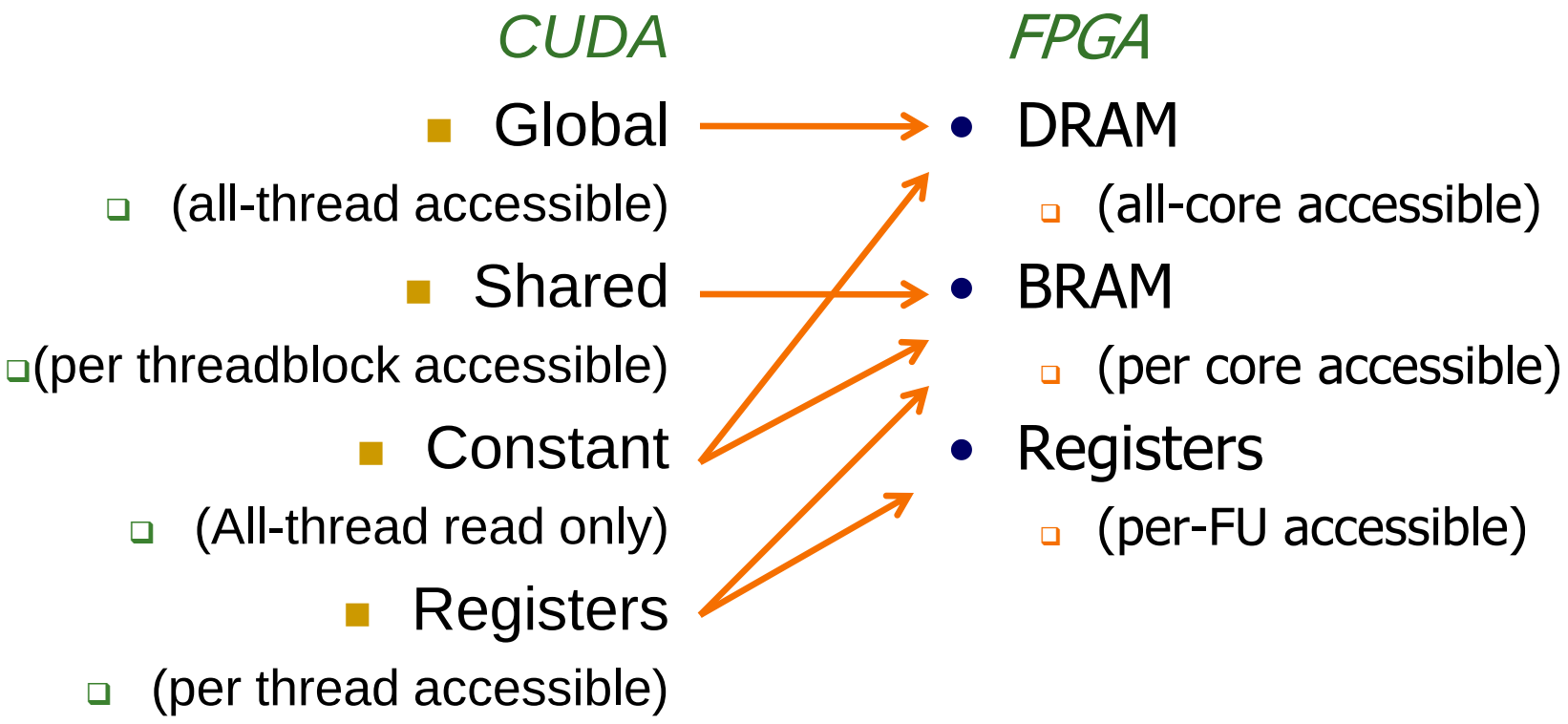
- Statically schedule the execution of thread-blocks based on parallelism info provided in programmer-specified annotation
  - Replicate task function calls according to required concurrency
  - Annotate concurrent task function calls with AutoPilot PARALLEL pragmas
  - Update stride of loop over thread-block grid

```
#pragma FCUDA COMPUTE cores=2
#pragma FCUDA BLOCKS start_x=0 end_x=63
#pragma FCUDA SYNC type=simple
void matrixMul (…) {

 for (by=0; by<gridDim.y; ++by) {
   for (bx=0; bx<gridDim.x; ++bx) {
      …
} } }
```



```
void matrixMul (int * C, int * A, int * B, …) {
 for (by=0; by<gridDim.y; ++by) {
   for (bx=0; bx<gridDim.x; bx +=2) {
     …
     #pragma AUTOPILOT REGION begin
     #pragma AUTOPILOT PARALLEL
      matrixMul_compute(Csub1, As1, Bs1, … );
      matrixMul_compute(Csub2, As2, Bs2, … );
      #pragma AUTOPILOT REGION end
    …
} } }
```

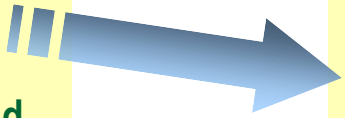
CUDA Memory Spaces Mapping



Data Transfer Code Example

- In data transfer task functions, merge single off-chip accesses into DMA bursts
  - DMA bursts are inferred by *memcpy* calls in AutoPilot
    - Compute array offsets and lengths
    - Arrange bursts for multiple partial rows
- In compute task functions, replace direct accesses to off-chip memory arrays by on-chip memory-block accesses
  - Update task function parameter list
  - Currently, this transformation is based on info provided in the annotation inserted by the programmer

```
#pragma FCUDA TRANSFER begin
  for ( <thread-loop> )
    As[ ]=A[ ];
    Bs[ ]=B[ ];
#pragma FCUDA TRANSFER end
```



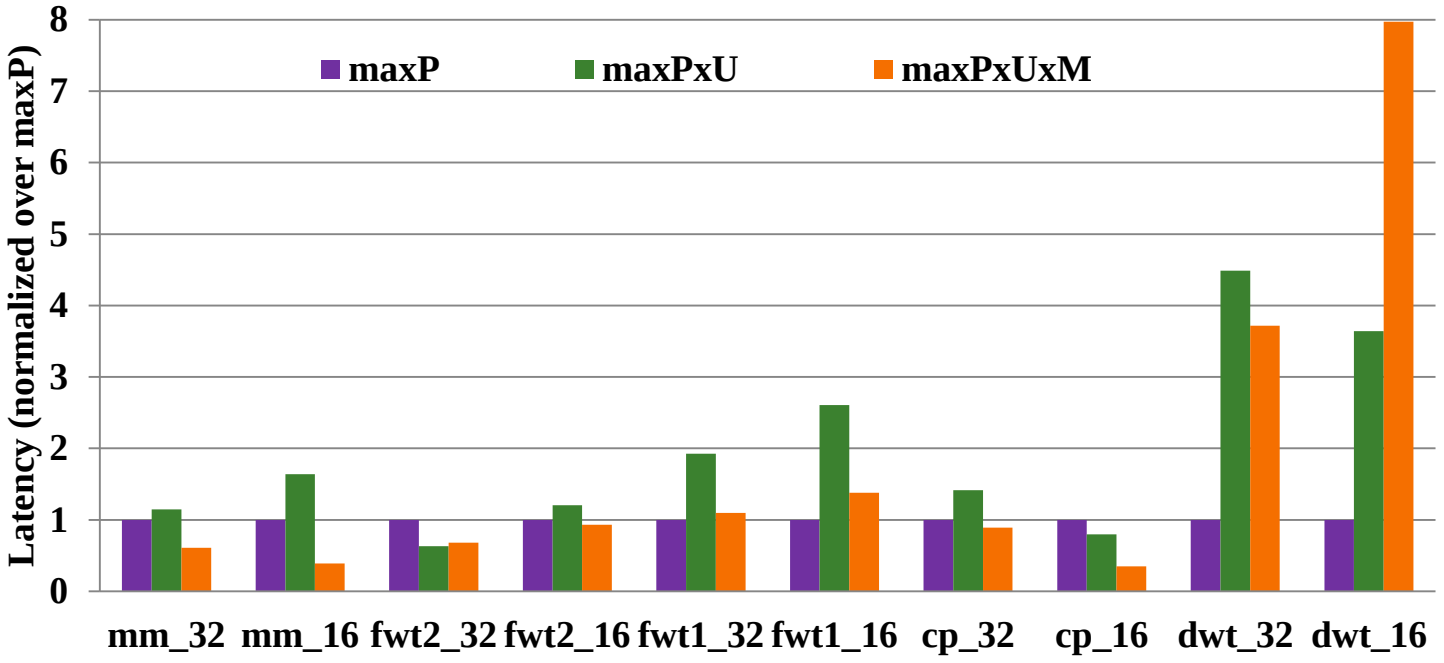
```
void fetch (volatile int* A, volatile int* B, int As[ ],
                int Bs[ ], … ) {
  for (<# rows > )
    memcpy(As+tIdx.y, A+a+tIdx.y*wA, bDim.x*sizeof(int));
    memcpy(Bs+tIdx.y, B+b+tIdx.y*wB, bDim.x*sizeof(int));
}
```

CUDA Kernels

Kernel	Data Dimensions	Description
Matrix Multiply (matmul)	4096x4096	Computes multiplication of two arrays (used in many applications)
Coulombic Potential (cp)	4000 atoms, 512x512 grid	Computation of electrostatic potential in a volume containing charged atoms
Fast Walsh Transform (fwt1)	32 Million element vector	Walsh-Hadamard transform is a generalized Fourier transformation used in various engineering applications
Fast Walsh Transform (fwt2)		
Discreet Wavelet Transform (dwt)	120K points	1D DWT for Haar wavelets and signals

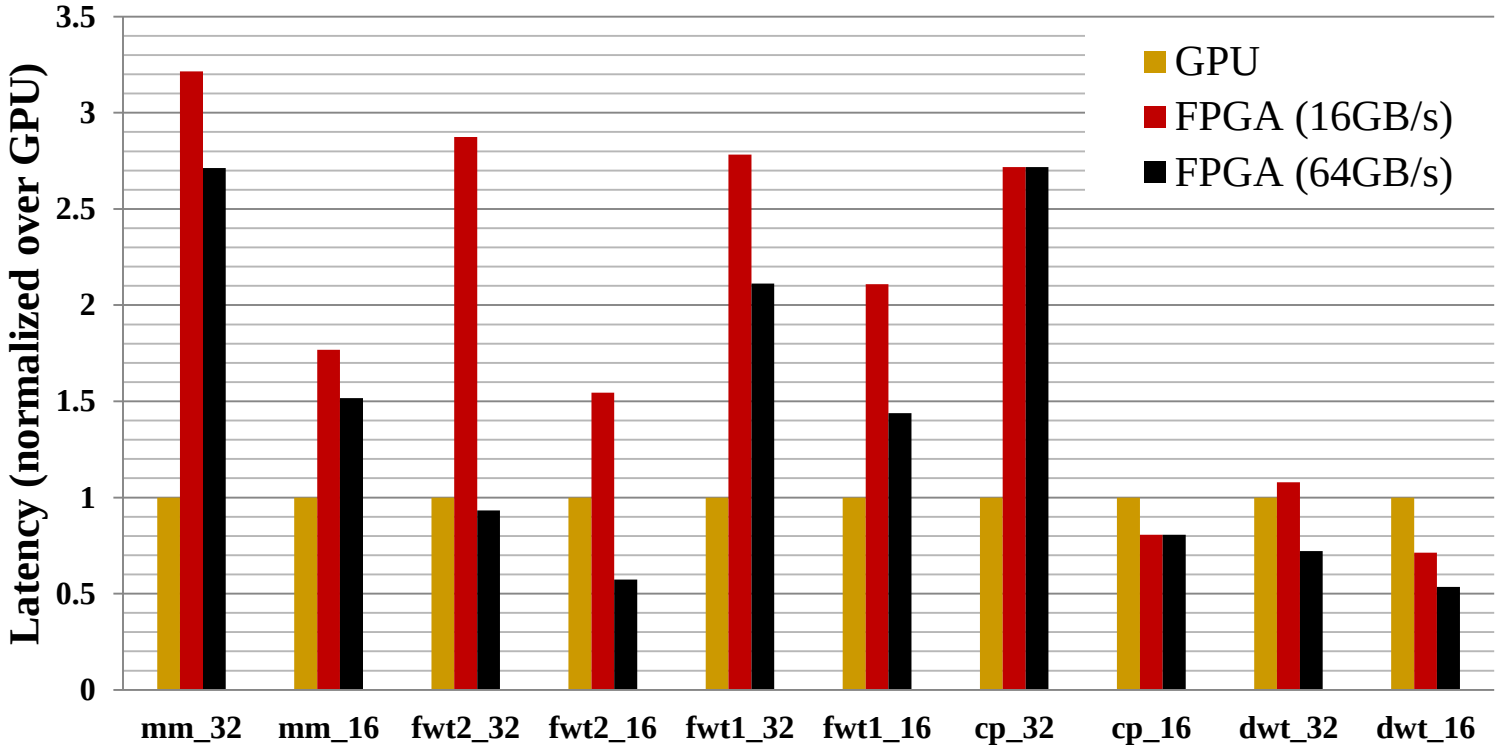
Parallelism Impact on FPGA performance

- *maxP:* maximum PE (core) count – total PEs
- *maxPxU:* maximum (PE*Unroll) – total threads
- *maxPxUxM:* maximum PE*Unroll*Partition – balanced



FPGA vs. GPU – Latency

- Nvidia G92 (65nm)
- Xilinx SX240T Virtex-5 (65nm)

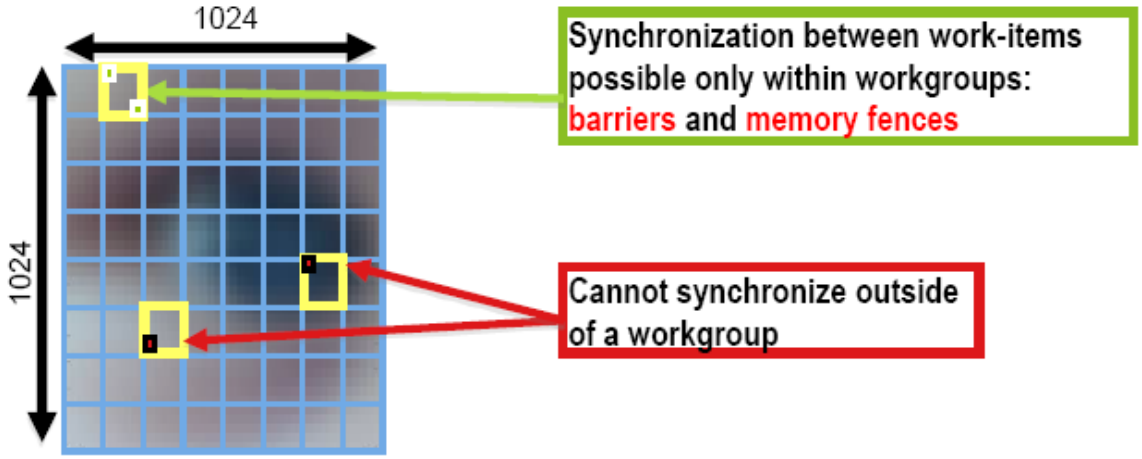


Introducing OpenCL

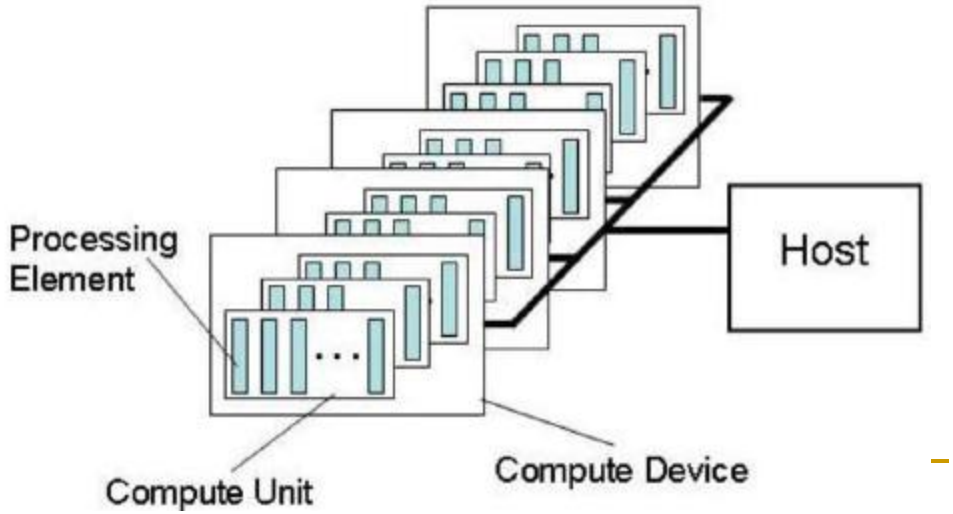
- Open Computing Language (Open Standard)
  - Royalty free
  - Khronos OpenCL working group (driven by industry)
- Provide single programming model for heterogeneous devices
  - Support all compute resources in system
  - Provide portability
  - Exploit data and task parallelism
- C99 subset
  - Missing Function pointers, recursion, variable length arrays, etc.

Data-Level Parallelism

- Hierarchical N-dimensional compute domain (N=1,2 or 3)
  - Work-item
  - Work group

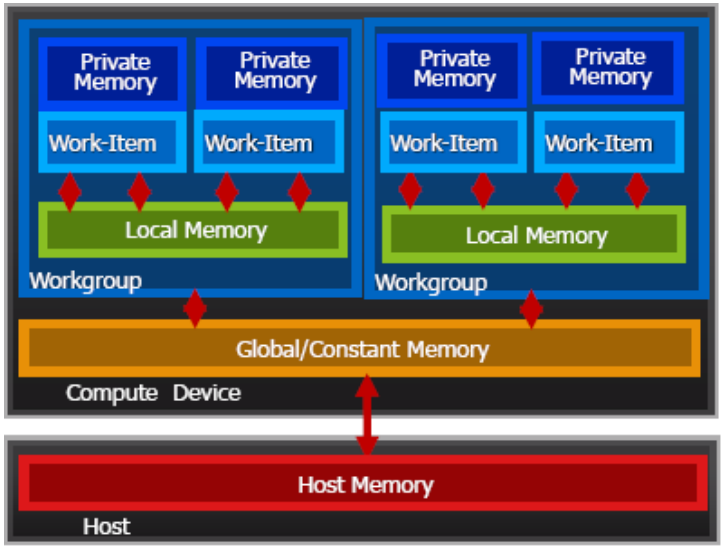


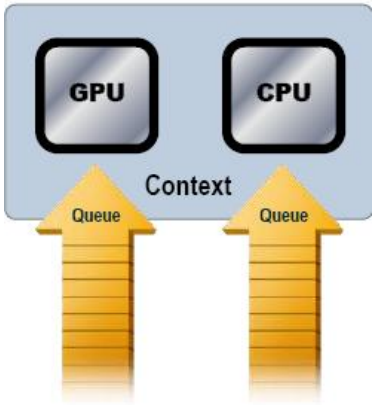
- Platform model
  - 1 Host
  - 1 or more devices

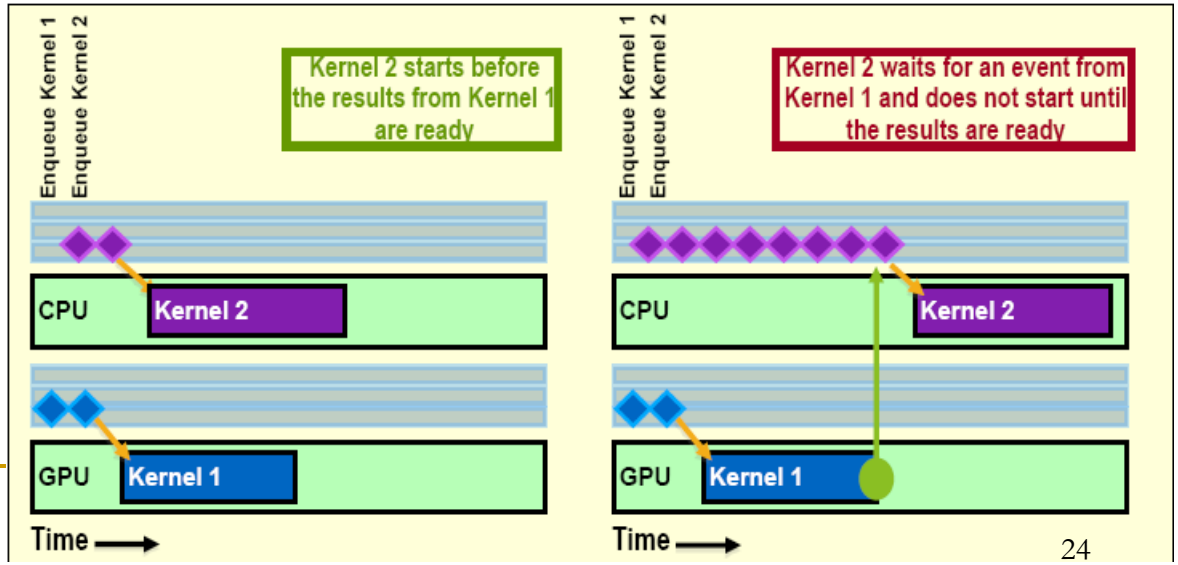


Memory Model & Host-Device IF

- Multiple address spaces
- Command queues
  - Data transfers
  - Kernel invocations
  - In-order execution
  - Out of order exec.

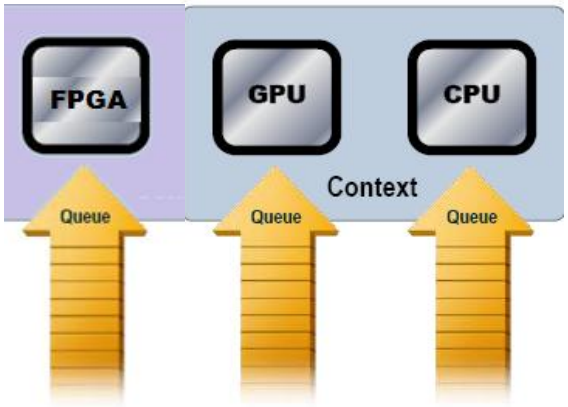






OpenCL Support - Two Main Tasks

- Static compilation
  - FOpenCL flow
- Run time API implementation
  - Support pre-compiled bitfiles
    - Due to lengthy synthesis runtimes
  - Implement queues for
    - Data transfer commands
    - Kernel invocation commands
      - Download a new bitfile
      - Use previously downloaded bitfile
      - Use embedded hard CPUs, e.g., handle sequential computation



Challenges – Flow and Command Queue

- FOpenCL
  - OpenCL can use a single kernel to target different devices
  - Although it is portable, performance may not exploit the maximum potential of every platform
  - For FPGA, we will explore the following
    - Data structure adjustment targeting FPGA specific features
    - Computation adjustment to take advantage of customization capability of FPGA
- Runtime API implementation
  - Need to work with low-level FPGA device drivers
  - Coordinate commands between kernels on FPGA fabric and sequential computation on embedded hard CPU

Challenges – System Level Issues

- Performance Driven Kernel Mapping
  - In heterogeneous systems with multiple accelerators
    - Different platforms, e.g., FPGA & GPU devices
    - Different types, e.g., different FPGA devices
  - Analyze kernel compute & data patterns to find good workload partitioning

- Multi-FPGA application acceleration
  - Scenario 1:
    - Map a single kernel to multiple FPGA devices
  - Scenario 2:
    - Map kernels connected through data streaming to different devices and eliminate traffic to global memory

Conclusions

- FPGAs are becoming increasingly attractive in heterogeneous multi-processor environments
  - FPGAs can provide application specific parallelism with high computational density per Watt
  - However, the devil is in programming the thing
- FCUDA aims to contribute in bridging compilation and high-level-synthesis techniques
  - overcoming the hurdle of programmability
  - easy parallelism mapping on the reconfigurable fabric at high abstraction
  - efficient extraction of different levels of parallelism in applications
  - enabling common frontend language for heterogeneous platforms
  - initial results promising
- Support FPGA in OpenCL

Acknowledgement

- We acknowledge the support of the following funding agencies
  - GSRC
  - NSF

Thank You Very Much