OpenCL for FPGAs

Dmitry Denisenko
Engineer, High-Level Design Team
Intel Programmable Solutions Group
Outline

(40 min)
- Introduction to FPGAs
- Brief introduction to OpenCL programming language

(10 min break)

(40 min)
- How OpenCL concepts map to FPGA hardware
- Examples of architectures for high-performance applications
Basics of Programmable Logic

FPGA Architecture
Why Programmable Logic?

Solution: Replace External Devices with Programmable Logic
Programmable Logic is Found Everywhere!

- **Consumer Automotive**
  - Entertainment
    - Broadband
    - Audio/video
    - Video display
  - Automotive
    - Navigation
    - Entertainment

- **Test, Measurement, & Medical**
  - Instrumentation
    - Medical
    - Test equipment
    - Manufacturing

- **Communications Broadcast**
  - Wireless
    - Cellular
    - Basestations
    - Wireless LAN
  - Networking
    - Switches
    - Routers
  - Wireline
    - Optical
    - Metro
    - Access
  - Broadcast
    - Studio
    - Satellite
    - Broadcasting

- **Military & Industrial**
  - Military
    - Secure comm.
    - Radar
    - Guidance and control
  - Security & Energy Management
    - Card readers
    - Control systems
    - ATM

- **Computer & Storage**
  - Computers
    - Servers
    - Mainframe
  - Storage
    - RAID
    - SAN
  - Office Automation
    - Copiers
    - Printers
    - MFP
Agenda

- FPGA Architecture
- Design Methodology and Software
FPGA Logic blocks

FPGA logic is made up of Logic Elements (LEs) or Adaptive Logic Modules (ALMs)
Lookup Tables (LUTs)

- Combinational functions created with programmed “tables” (cascaded multiplexers)
- LUT inputs are mux select lines

$$X = \overline{A}B + ABCD + ABCD$$

Programmed levels (EEPROM or SRAM)

= x9889
Programmable register

- Clock typically driven by global clock
- Asynchronous control through other logic or I/O
- Feedback into LUT
- Bypass register or LUT
Chain carry bits between LEs

Register outputs can chain to other LE registers in LAB to form LUT-independent shift registers
Register Packing

- Separate outputs from LUT and register create two outputs from one LE
- Saves device resources
LABs and LEs: A Closer Look

- LUT & carry logic
- Register
Adaptive Logic Modules (ALM)

- Based on LE, but includes dedicated resources & adaptive LUT (ALUT)
- Improves performance and resource utilization
Field Programmable Gate Array (FPGA)

- LABs arranged in an array
- Row and column programmable interconnect
- Interconnect may span all or part of the array
FPGA Routing

- All device resources can feed into or be fed by any routing in device
- Differing fixed lengths to adjust for timing
- Scales linearly as density increases
- Local interconnect
  - Connects between LEs or ALMs within a LAB
  - Can include direct connections between adjacent LABs
- Row and column interconnect
  - Fixed length routing segments
  - Span a number of LABs or entire device
FPGA Embedded Memory

**Memory blocks**

- Create on-chip memory structures to support design
  - Single/dual-port RAM
  - ROM
  - Shift registers or FIFO buffers
- Initialize RAM or ROM contents on power-on
- Memory LABs (MLABs)

**Typical sizes:**

- Single memory block is 20 Kilobits.
- One MLAB is 640 bits.
DSP Block

- Useful for DSP functions
- High-performance multiply/add/accumulate operations
New in latest FPGA family (Arria 10): 32-bit Floating Point Support

- Arria 10 DSP Block can do 32-bit IEEE-compliant floating-point multiply-add.
- Just like LABs, DSPs can be chained to implement large dot products.
FPGA I/O Elements

- Advanced programmable logic blocks connect directly to row or column interconnect
- Control available I/O features
  - Input/output/bidirectional
  - Multiple I/O standards
  - Differential signaling
  - Current drive strength
  - Slew rate
  - On-chip termination/pull-up resistors
  - Open drain/tri-state
  - etc.
Typical I/O Element Logic

- **Output Path**
  - Output Register
  - Output A
  - Output B

- **Input Path**
  - Input Register
  - Input (I)

- **Device Pin**

- **Output Enable Control**
High Speed Transceivers

- High-speed transceivers
  - Used for numerous high speed protocols: Ethernet, PCI Express, etc
FPGA Clocking Structures

- Dedicated input clock pins
- Phase Locked Loops (PLLs) (see next slide)
- Delay Locked Loops (DLLs)
  - Dynamically phase-shift strobes for external memory interfaces
- Clock control block(s)
  - Select clocks to feed clock routing network
  - Enable/disable clocks for power-up/down and for power savings
- Clock routing network
  - Special routing channels reserved for clocks driven by PLLs or clock control blocks
  - Global clock network feeds entire device
  - Regional or hierarchical networks feed certain device areas, such as device quadrants
FPGA PLLs

Based on input clock, programmable blocks that generate clocks (clock domains) for use throughout device with minimal skew.

100 MHz input clock

- **in_clk0** frequency: 100,000 MHz
- **Clk** Ratio Ph (dg) DC (%)
  - c0 1/1 0.00 50.00
  - c1 2/1 0.00 50.00
  - c2 2/1 90.00 50.00

100 MHz clock domain
200 MHz clock domain
90° phase-shifted 200 MHz clock domain
FPGA Programming

Most FPGAs use SRAM cell technology to program interconnect and LUT function levels

Volatile! Must be programmed at power-on!
FPGA Programming (cont.)

- FPGA programming information must be stored somewhere to program device at power on
- Use external EEPROM, CPLD, or CPU to program
- Two programming methods
  - Active: FPGA controls programming sequence automatically at power on
  - Passive: Intelligent host (typically CPU) controls programming
- Also programmable through JTAG connection
Example of Resource Counts for largest part (Arria 10 GX 1150)

- 1,150,000 Logic Elements
  - 1.7 M registers and ~1M 4-input LUTs
- 54,000 Memory Blocks (20Kbits each)
- 1,518 DSP blocks
- 36 17.4 Gbps Transceivers
- 2 Hard PCIe blocks
- 492 general I/O pins
- 12 Hard Memory Controllers
FPGA Advantages

- High density to create many complex logic functions
- Integration of many functions
- Many available I/O standards and features
- Data can go directly from I/O to computation engine and back out, all in one chip.

Altera FPGAs
- Max®, Cyclone®, Arria®, and Stratix® series devices
Agenda

- FPGA Architecture
- Design Methodology and Software
Typical Programmable Logic Design Flow (1/2)

Design entry/RTL coding
- Behavioral or structural description of design
- Possibly with the help of high level tools

RTL functional simulation
- Mentor Graphics ModelSim® or other 3rd party simulators
- Verify logic model & data flow (no timing delays)

Synthesis (Mapping)
- Translate design into device specific primitives
- Optimization to meet required area & performance constraints
- Quartus® synthesis or those available from 3rd party vendors
- Result: Post-synthesis netlist
Typical Programmable Logic Design Flow (2/2)

Place & route (Fitting)
- Map primitives to specific locations inside target technology with reference to area & performance constraints
- Specify routing resources to be used
- Result: Post-fit netlist

Timing analysis
- Verify performance specifications were met
- Static timing analysis

PC board simulation & test
- Simulate board design
- Program & test device on board
- Use on-chip tools for debugging
Quartus Prime Design Software

System Description

Quartus Prime fully-integrated development tool
- Multiple design entry methods
- Logic synthesis
- Place & route
- Timing & power analysis
- Device programming

HDL Code

```verilog
module TwoDepthDelay(
    input dataIn,
    input clk,
    input reset,
    output reg dataOut);

  reg intermediate;
  always @(posedge clk or posedge reset) begin
    if(reset) begin
      dataOut <= 1'b0;
      intermediate <= 1'b0;
    end
    else begin
      intermediate <= dataIn;
      dataOut <= intermediate;
    end
  end
endmodule
```
Brief Introduction to OpenCL
Why OpenCL on FPGAs

OpenCL expands the number of application developers

- Standard CPU Programmers
- Parallel Programmers
- FPGA Programmers
- ASIC Programmers
Utilizing Software Engineering Resources

Altera OpenCL flow abstracts away FPGA hardware flow bringing the FPGA to low-level software programmers

- Software developers write, optimize and debug in familiar environment
- Quartus® II software runs behind the scenes
- Emulator and profiler are software development tools
What is OpenCL?

- A software programming model for software engineers and a software methodology for system architects
  - First industry standard for heterogeneous computing
- Provides increased performance with hardware acceleration
  - Low Level Programming language
  - Based on ANSI C99
- Open, royalty-free, standard
  - Managed by Khronos Group
  - Altera active member
  - Conformance requirements
    - V1.0 is current reference
    - V2.0 is current release
  - http://www.khronos.org
The BIG Idea behind OpenCL

OpenCL execution model …
- Define N-dimensional computation domain
- Execute a kernel at each point in computation domain

Traditional loops
```c
void trad_mul(int n,
const float *a,
const float *b,
float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL
```c
kernel void dp_mul(global const float *a,
global const float *b,
global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over “n” work-items
```

Parallelism Must be Inferred

Parallelism is Explicit

Altera OpenCL SDK for FPGAs supports both styles.
OpenCL Programming Model

```c
#include <CL/cl.h>

int main() {
    // Host code
    read_data( ... );
    manipulate( ... );
    clEnqueueWriteBuffer( ... );
    clEnqueueNDRangeKernel(..., sum, ...);
    clEnqueueReadBuffer( ... );
    display_result( ... );
}

__kernel void sum(__global float *a,
                 __global float *b,
                 __global float *y)
{
    int gid = get_global_id(0);
    y[gid] = a[gid] + b[gid];
}
```

Host and accelerator code are separate
OpenCL Host Program

- Pure software written in standard C / C++
- Communicates with the Accelerator Device via a set of library routines which abstract the communication between the host processor and the kernels

```c
main()
{
    read_data_from_file( ... );
    manipulate_data( ... );
    clEnqueueWriteBuffer( ... );
    clEnqueueNDRangeKernel(..., my_kernel, ...);
    clEnqueueReadBuffer( ... );
    display_result_to_user( ... );
}
```
OpenCL Kernels

Data-parallel function
- Defines many parallel threads of execution
- Each thread has an identifier specified by “get_global_id”
- Contains keyword extensions to specify parallelism and memory hierarchy

Executed by compute object
- CPU
- GPU
- FPGA

```c
__kernel void sum(__global const float *a, __global const float *b, __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

```
float *a = 0 1 2 3 4 5 6 7
float *b = 7 6 5 4 3 2 1 0
float *answer = 7 7 7 7 7 7 7 7
```
Thread ID space for NDRRange kernels

- Each thread knows its “id”. It is used to determine which slice of data the thread should work on.
- Threads are partitioned into work-groups.
  - Only threads within one work-group can share local memory.

**Thread & Group identifiers**

- `get_group_id(0)`
- `get_local_id(0)`
- `get_global_id(0)`

`get_global_id(0) == get_group_id(0) * get_local_size(0) + get_local_id(0)`
Memory Model

- **Private Memory**
  - Unique to thread. Usually registers.

- **Local Memory**
  - Shared within workgroup. On-chip.

- **Global & Constant Memory**
  - Visible to all workgroups and the host.
  - Usually off-chip.

- **Host Memory**
  - On the host CPU. Usually DDR on CPU board.

[Diagram of memory model with arrows connecting between Host Memory, Host CPU, Global Memory, Constant Memory, Local Memory, and Private Memory.]
Compiling OpenCL to FPGAs

__kernel void
sum(__global const float *a, __global const float *b, __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}

int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];

main()
{...}

Kernel Programs

Host Program

PCIe

DDRx
OpenCL CAD Flow

- my_kernel.cl
  - OpenCL Compiler
  - HDL

- my_host.c
  - C++ compiler
  - program.exe
  - ACL runtime Library

- Board Description
- Quartus
  - FPGA Programming bitstream
OpenCL Compiler Builds Complete FPGA

The Altera Offline Compiler (aoc) produces the complete FPGA design:
- accelerators
- data paths
- all memory structures

Everything in this part is generated based on user’s kernel.

OpenCL SDK comes with host-side Run-Time Environment: OS driver, lower-level HAL, OpenCL API implementation library.

This part is fixed by Board Support Package (BSP) vendor.
10 minute break
Outline

(40 min)
- Introduction to FPGAs
- Brief introduction to OpenCL programming language

(10 min break)

(40 min)
- How OpenCL concepts map to FPGA hardware
- Examples of architectures for high-performance applications
Computation in Space

How computation is mapped to FPGAs
Mapping a simple program to an FPGA

High-level code

Mem[100] += 42 * Mem[101]

CPU instructions

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
Execution on a **Simple** CPU

- **Fixed and general architecture:**
  - General “cover-all-cases” data-paths
  - Fixed data-widths
  - Fixed operations
Load constant value into register
CPU activity, step by step

R0 ← Load Mem[100]  
R1 ← Load Mem[101]  
R2 ← Load #42  
R2 ← Mul R1, R2  
R0 ← Add R2, R0  
Store R0 → Mem[100]
On the FPGA we unroll the CPU hardware...

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
... and specialize by position

1. Instructions are fixed. Remove instruction “Fetch”
... and specialize

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]

1. Instructions are fixed. Remove instruction “Fetch”
2. Remove unused ALU ops
... and specialize

1. Instructions are fixed. Remove instruction “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
... and specialize

1. Instructions are fixed. Remove instruction “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly! And propagate state.

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
... and specialize

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]

1. Instructions are fixed. Remove instruction “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly! And propagate state.
5. Remove dead data.
... and specialize

R0 $\leftarrow$ Load Mem[100]

R1 $\leftarrow$ Load Mem[101]

R2 $\leftarrow$ Load #42

R2 $\leftarrow$ Mul R1, R2

R0 $\leftarrow$ Add R2, R0

Store R0 $\rightarrow$ Mem[100]

1. Instructions are fixed. Remove instruction “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly! And propagate state.
5. Remove dead data.
6. Reschedule!
Custom data-path on the FPGA matches your algorithm!

Build exactly what you need:
- Operations
- Data widths
- Memory size & configuration

Efficiency:
- Throughput / Latency / Power

High-level code

\[ \text{Mem}[100] += 42 \times \text{Mem}[101] \]
Data parallel kernel

__kernel void
sum(__global const float *a,
    __global const float *b,
    __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}

float *a = 0 1 2 3 4 5 6 7
float *b = 7 6 5 4 3 2 1 0
float *answer = 7 7 7 7 7 7 7 7
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads. While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored.

Work item IDs

8 work items for vector add example

1 2 3 4 5 6 7
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads
While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Work item IDs

8 work items for vector add example

2 3 4 5 6 7
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads.

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored.

Work item IDs

3 4 5 6 7
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads. While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored.

Silicon used efficiently at steady-state
Example Compiler Optimization: Branch Conversion

Control flow branching is expensive.

Instead, execute both sides of a branch, pick the result for the “true” path, and predicate commands that have side-effects.

If a function has no loops, the whole function loses all branches.

Loops lose all internal branches.

1. \( X = W; \)
2. if (cond) {
3. \( X += 2; \)
4. \( \text{array}[z] = Y; \)
5. }

\( X_{\text{temp}} = X + 2; \)
\( X = \text{cond} ? X_{\text{temp}} : W; \)
\( \text{array}[z] = Y \) only if cond

?: operator is a mux in hardware.

Single operation to store only if condition is true. “cond” is enable on the store unit.
Local memory
Memory systems in hardware

External (off-chip) memory

On-chip memory

FPGA

x86 / External Processor

PCIe

External Memory Controller & PHY

External Memory Controller & PHY

Global Memory Interconnect

Kernel Pipeline

Kernel Pipeline

Kernel Pipeline

Local Memory Interconnect

DDR3

External (off-chip) memory

On-chip memory
1. Register data: Registers in FPGA fabric
2. Private data: Registers in FPGA fabric or on-chip RAMs
3. Local memory: On-chip RAMs
4. Global memory: Off-chip external memory
On-chip memory systems

- “Local” and some “private” memories use on-chip block RAM resources
  - Very high bandwidth, random accesses, limited capacity

- The memory system is customized to your application
  - Huge value proposition over fixed-architecture accelerators

- Memory geometry (width, depth, number of banks, etc.), and interconnect all customized for your kernel
  - Automatically optimized to eliminate or minimize access contention

Caveats:
- Compiler has to understand access patterns to minimize contention efficiently
On-chip memory architecture

Basic memory architectures map to M20Ks
- Each M20K is a dual-ported RAM
- Concurrently, \( \#\text{loads} + \#\text{stores} \leq 2 \)

Kernels require complex accesses

Compiler optimizes the kernel pipeline, the interconnect, and the memory system to achieve this
- Through splitting, coalescing, banking, replication, double-pumping, port sharing
Interconnect

Interconnect includes access arbitration to memory ports

With no optimization, sharing ports destroys performance
- Pipeline stalls due to arbitration for concurrent accesses

Key to high kernel efficiency is never-stall memory accesses
- All possibly concurrent memory access sites in the datapath are guaranteed to access memory without contention
Double pumping

Pros: no M20K increase
Cons: Fmax penalty, register usage increase
Replication

Up to Four ports

1-3 write

Y-read

Memory 2x clock

Memory 2x clock

never-stall
never-stall
never-stall
never-stall
never-stall
never-stall
never-stall
never-stall

store
load
load
load
load
load
load
load

Port0
Port1
Port3
Port0
Port1
Port3
Static coalescing (cont’d)

Original kernel:

```c
kernel void example() {
    local int A[32][2], B[32][2];
    ...
    int lid = get_local_id(0);
    A[lid][0] = B[lid][0];
    A[lid][1] = B[lid + x][1];
    ...
}
```

With coalescing:

```c
kernel void example() {
    local int A[32][2], B[32][2];
    ...
    int lid = get_local_id(0);
    int2 v = (int2)(B[lid][0], B[lid + x][1];
    *(int2*)(&A[lid][0]) = v;
    ...
}
```
Partition memory into logical banks

- An N-bank configuration can handle N-requests per clock cycle as long as each request addresses a different bank
- Uses lower bits of memory access for bank selection
Forcing memory geometry

Compiler attributes can enforce a desired local memory configuration.

```c
int __attribute__((numbanks(1)))
    __attribute__((bankwidth(128)))
    __attribute__((doublepump))
    __attribute__((numwriteports(1)))
    __attribute__((numreadports(4))) MyLocalMem[128];
```
Global Memory
Global Memory in OpenCL

‘global’ address space
- Used to share data between host and device
- Shared between workgroups

Generally allocated on host as cl_mem object
- Created with clCreateBuffer
- Data transferred with clRead/clWrite Buffer
- cl_mem object assigned to global pointer argument in kernel

```c
__kernel void Add(__global float* a,
                 __global float* b,
                 __global float* c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```
Compiler’s View of Global Memory

- Agnostic to the memory technology itself
  - DDR, QDR, HMC, QPI

- Only a few pertinent parameters (provided by BSP)
  - How many interfaces
  - Width of the bus
  - Burst size (affinity for linear access)
  - Latency
  - Bandwidth
Compiler Generated Hardware

```cpp
foo.cl

global int* x;
...
int y = x[k];
```
Global Load/Store Unit (LSU)

- **Width Adaptation**
  - User data (32-bit int) to memory word (512-bit DRAM word)

- **Possible static coalescing** by compiler to avoid wasted bandwidth

- **Burst coalescing**
  - Dynamically coalesces consecutive memory transactions into large burst transaction
Global Load/Store Unit (LSU) Flexibility

- Compiler picks the most appropriate LSU based on static access analysis of user’s kernel.

- **Simple**
  - Passes transactions directly to interconnect from pipeline
  - Used for loads/stores used very infrequently

- **Burst-Coalesced**
  - Most common global memory LSU
  - Specialized LSU to groups loads/stores into bursts
  - Load LSU can a private data cache if compiler determines that it will be beneficial

- **Streaming**
  - Simplified LSU used if compiler can determine access pattern is completely linear

- Other types for special cases: extra-wide, unaligned, etc.
Arbitration

- Arbitrate to physical interfaces
- Distribute (load balance) across physical interfaces
Constant Cache

- Constant buffer resides in global memory but accessed via on-chip cache shared by all work-groups
  - Constant cache optimized for high cache hit performance

- Use for read-only data that all work-groups access
  - E.g. high-bandwidth table lookups

- No `__constant` data, no constant cache.
Complete Picture

Kernel datapath

Load Unit
Load Unit
Coalesce
Load Unit
Cache
Constant Load Unit
Constant Load Unit

Arbitration

Global Memory Controller
Global Memory Controller
Channels
BSP can provide access to arbitrary streaming I/O interfaces as OpenCL channels. A channel is just a FIFO in hardware.
3. Loop Pipelining
On the FPGA, we use pipeline parallelism to achieve acceleration.

Threads execute in an embarrassingly parallel manner.
Ideally, all parts of the pipeline are active at the same time.
Data-Parallel Execution - drawbacks

Difficult to express programs which have partial dependencies during execution

```
kernel void
sum(global const float *a,
    global const float *b,
    global float *c)
{
    int xid = get_global_id(0);
    c[xid] = c[xid-1] + b[xid];
}
```

Would require complicated hardware and new language semantics to describe the desired behavior
Solution: Tasks and Loop-pipelining

- Allow users to express programs as a single-thread

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

- Pipeline parallelism still leveraged to efficiently execute loops in Altera’s OpenCL
  - Parallel execution inferred by compiler
  - Loop Pipelining
Loop Pipelining Example

No Loop Pipelining

Clock Cycles

i0
i1
i2

No Overlap of Iterations!

With Loop Pipelining

Clock Cycles

i0
i1
i2
i3
i4
i5

Finishes Faster because Iterations Are Overlapped

Looks almost like multi-threaded execution!

Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.
Parallel Threads vs. Loop Pipelining

So what’s the difference NDRange and loop pipelining?

- **Parallel Threads:** Launch 1 thread per clock cycle in pipelined fashion.

- **Loop Pipelining:** Sometimes loop iterations cannot be started every cycle.
Loop-Carried Dependencies

Loop-carried dependencies are dependencies where one iteration of the loop depends upon the results of another iteration of the loop.

```c
kernel void state_machine(ulong n) {
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        write_output(y);
    }
}
```

The variable state in iteration 1 depends on the value from iteration 0. Similarly, iteration 2 depends on the value from iteration 1, etc.
Loop-Carried Dependencies

To achieve acceleration, we pipeline each iteration of a loop with loop-carried dependencies

- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as possible

```c
kernel void state_machine(ulong n)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        write_output(y);
    }
}
```

At this point, we can launch the next iteration.
Design Examples
FFT – Example of Feed-Forward Pipeline Architecture

Instead of loops, data is fed into feed-forward pipeline
- Two complex points per clock cycle is the input and output data rate.

Delay elements in the pipeline, together with rotators, ensure that all

Architecture and Image above are taken from Mario Garrido, Jesús Grajal, M. A. Sanchez, Oscar Gustafsson “Pipeline Radix-2k Feedforward FFT Architectures.”, IEEE Trans. VLSI Syst. 21(1): 23-32 (2013)
Matrix Multiply in OpenCL – Small 4x4 variant

- Systolic array architecture.
- Great match for FPGAs.
- Scalable to smaller and larger FPGA devices by simply enlarging the grid.

Arria 10
GX1150

Load A
Load B
Drain C

DDR4