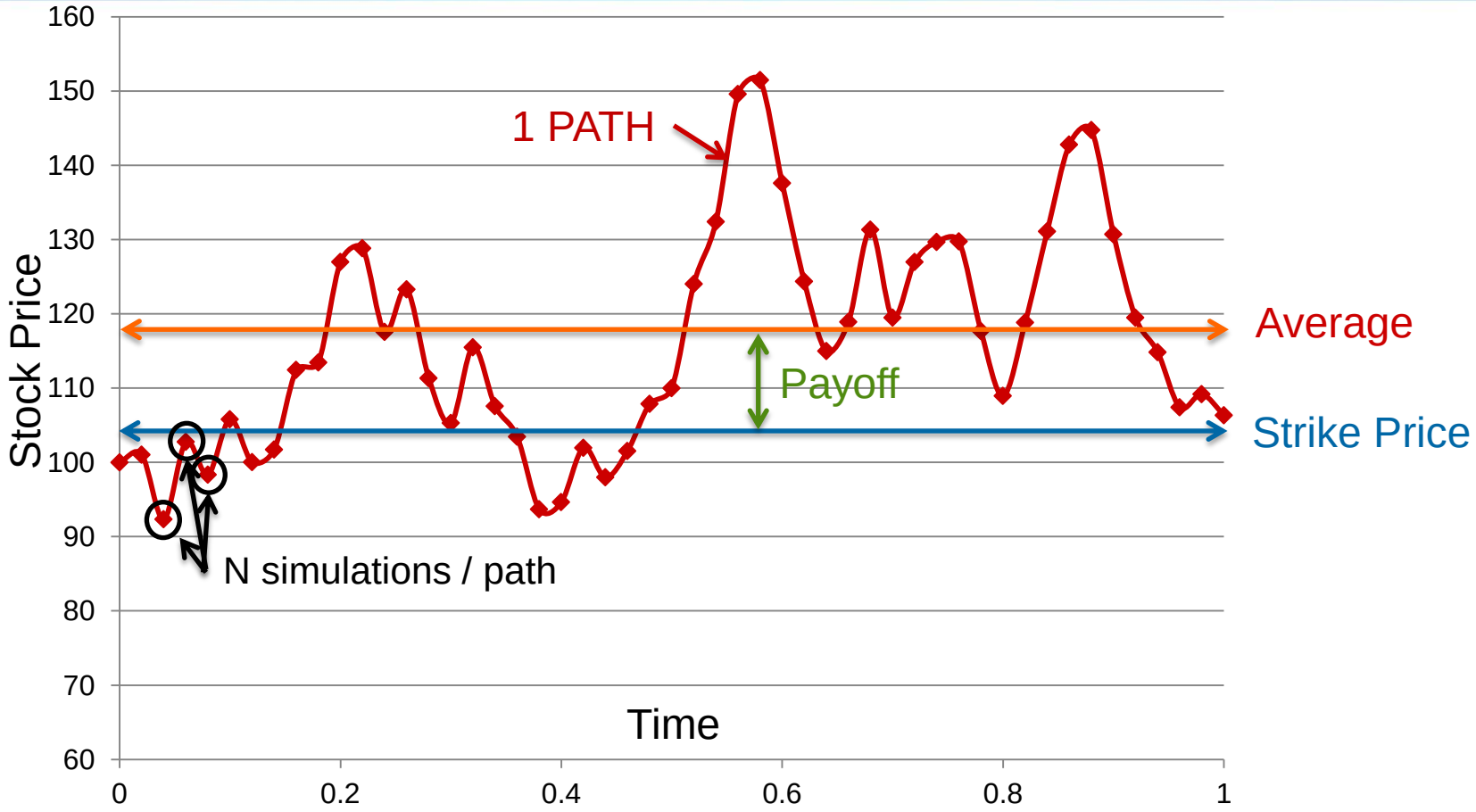
Monte Carlo Pricing of Asian Options on FPGAs using OpenCL

Arithmetically Averaged Asian Options

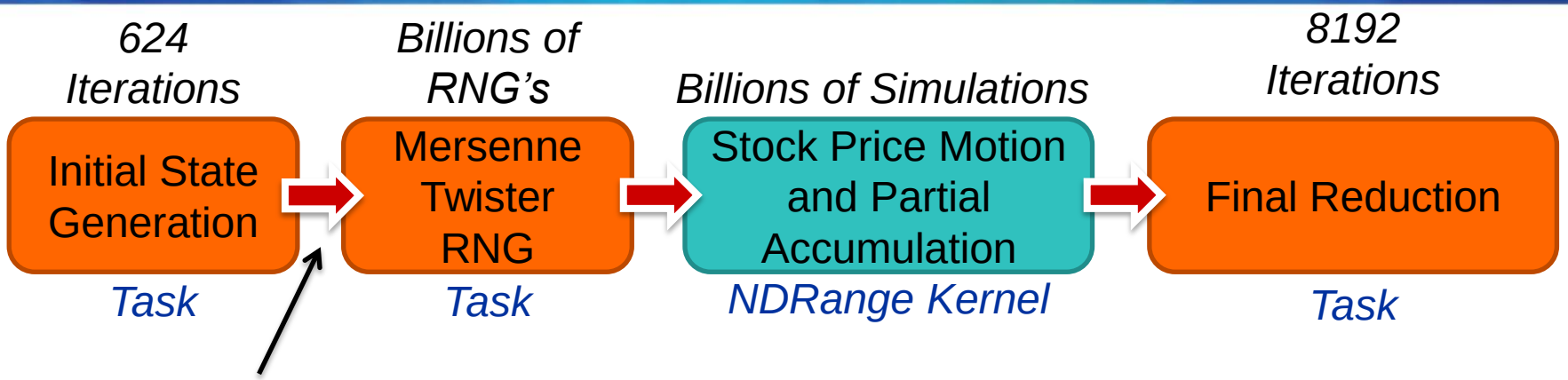


- **At the expiration date, an Asian Option's payoff is the difference between the average price during the time interval and the strike price.**

Efficiently Implementing Monte Carlo Simulation on FPGAs

- **Arithmetic Asian Option pricing is an example of a derivative where no closed form solution is possible**
- **Monte Carlo methods are used to simulate many possible paths and derive an expected value for the payoff**
- **Typical characteristics of Monte Carlo simulations**
  - Requires a very high quality random number generator
  - Many billions of simulations may be required
- ***FPGAs are an ideal platform for addressing these challenges***
  - In this presentation, we address the use of channels and loop pipelining optimizations within OpenCL to create a highly efficient Monte Carlo Simulation of Asian Option Pricing.

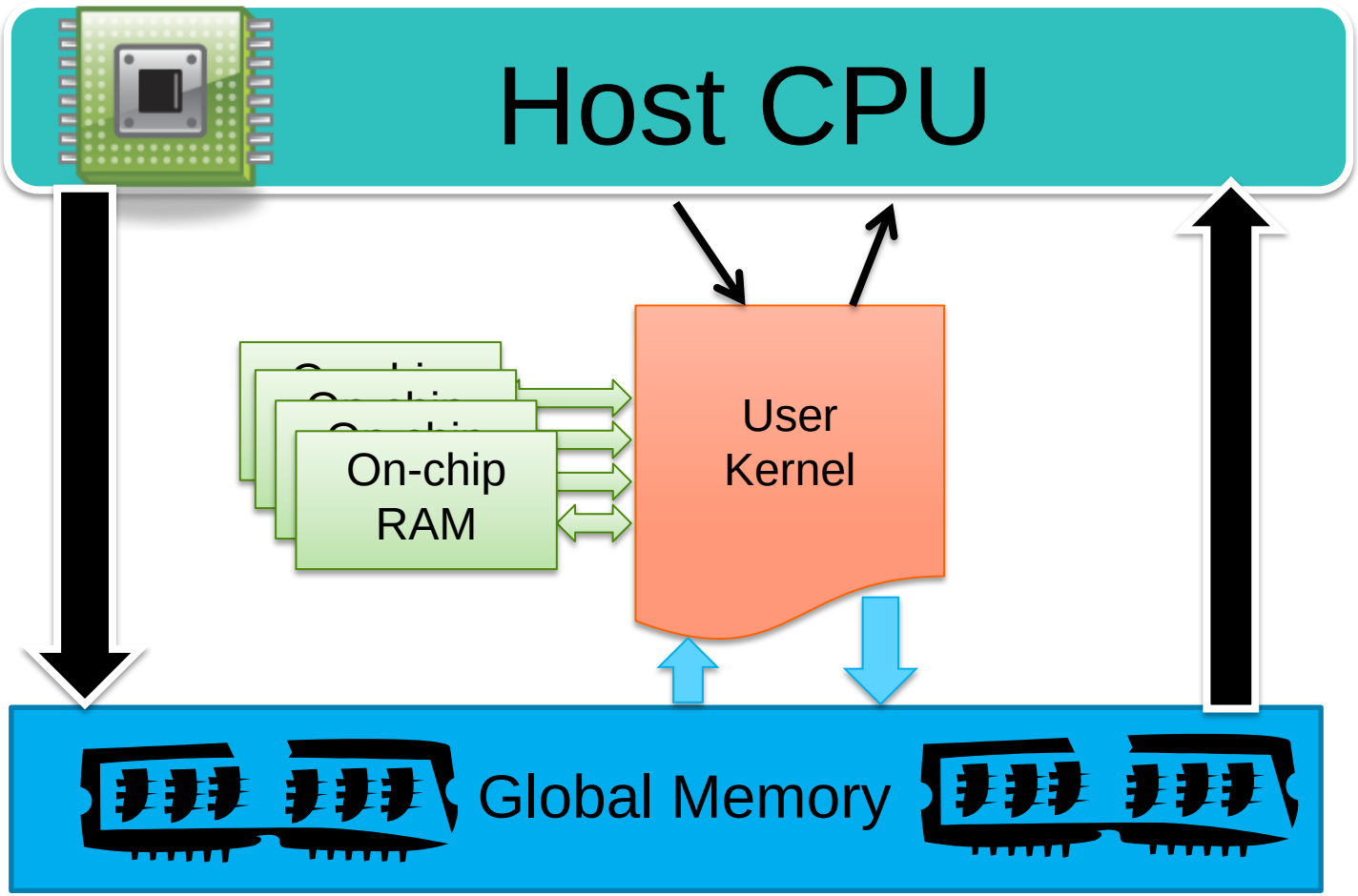
Monte Carlo Asian Option Simulation



- ***Channels are used for direct kernel-to-kernel communication without requiring intermediate global memory buffers*.**
- **Uses both Tasks (single work-item) and ND-range kernels.**
  - Random number generation poses a tremendous challenge in parallelizing using ND-range techniques.
  - *Altera has introduced a new class of loop pipelining optimizations for Task kernels that significantly improves application performance.*
- **Results:**
  - `Altera Stratix V D8: 12.0 Billion Simulations / Second @ 45 Watts`
  - `Leading GPU : 10.1 Billion Simulations / Second @ 212 Watts`

ALTERA MEASURABLE ADVANTAGE™

Host Centric OpenCL Architecture



Host Co-ordinates Kernel Invocations and Data Transfers

ALTERA
MEASURABLE ADVANTAGE™

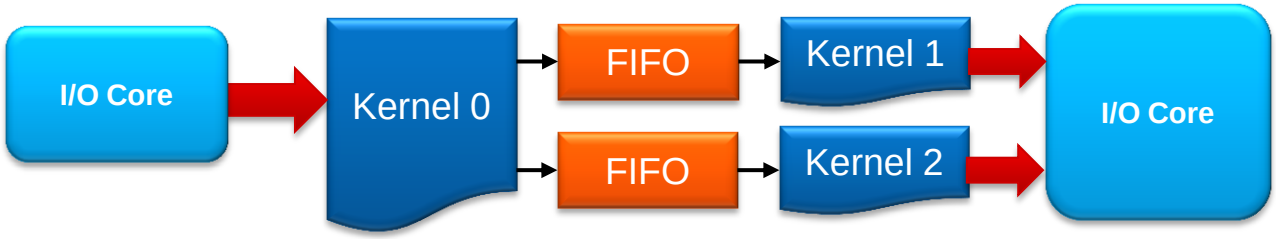
Drawbacks of the Host-Centric Model

- **Intermediate data communicated between kernels must be transferred through global memory**
  - High performance requires high bandwidth and high power !
  - Limited data buffer sizes when problem sizes scale to 100s of billions of simulations

- **Having multiple kernels operating in parallel and communicating requires the host to synchronize and coordinate activities**

New in 13.1: Altera Channels Support

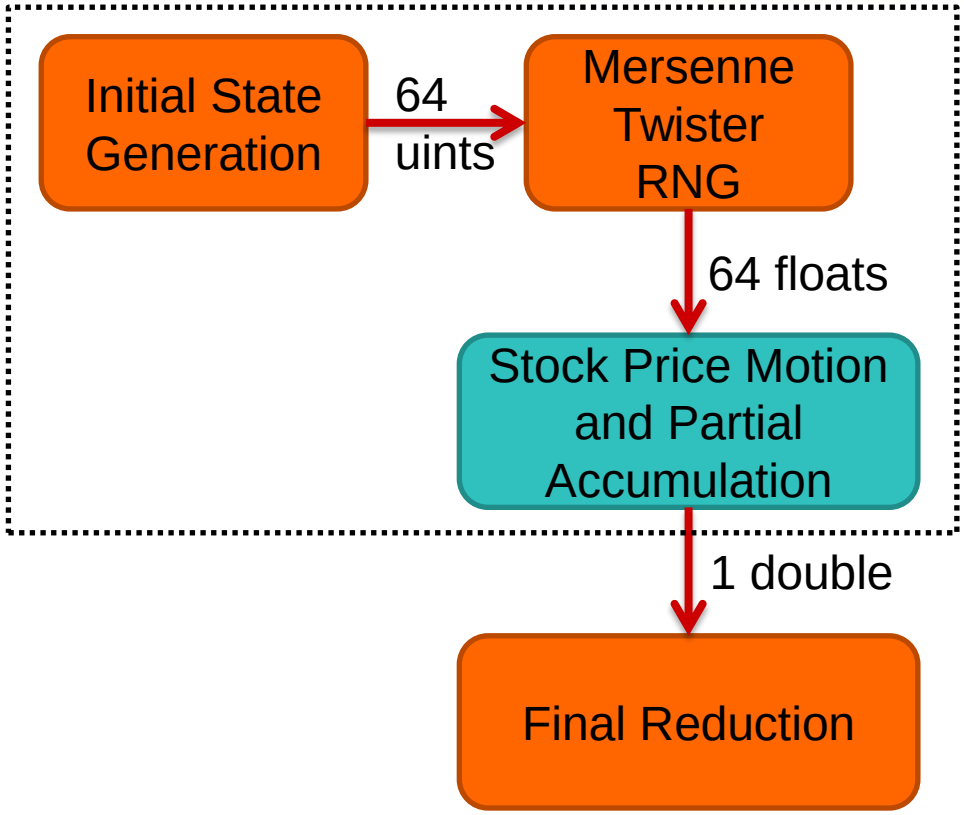
- **Low-Latency and High Bandwidth Channels**
  - Enables **IO → Kernel** and **Kernel → Kernel** Communication



- **Simple and Intuitive API available**
  - Channels are program scope global variables that describes the communication links
    - Ex: __channel float4 FLOATING_POINT_CHANNEL;
  - read_channel_altera
    - Read data from channel endpoint
    - Ex: float4 xvec = read_channel_altera(FLOATING_POINT_CHANNEL);
  - altera_write_channel
    - Write data to channel endpoint
    - Ex: write_channel_altera(FLOATING_POINT_CHANNEL, zvec);

ALTERA
MEASURABLE ADVANTAGE™

Channel Based Decomposition of Asian Option Benchmark



- **Key portion of the computation is generating and consuming 64 random numbers per clock cycle.**
- **At a 200MHz clock, this equates to roughly 12 Billion time-step simulations / second.**

ALTERA
MEASURABLE ADVANTAGE™

The Challenge of High Quality Random Number Generation

- **Consider the typical random number generator:**

```
__kernel void generate_rngs(ulong num_rnds)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<num_rnds; i++) {
    state = generate_next_state( state );
    unit y = extract_random_number( state );
    write_channel_altera(RANDOM_STREAM, y);
  }
}
```

- **Can this kernel be expressed in totally data parallel / ND-range fashion?**

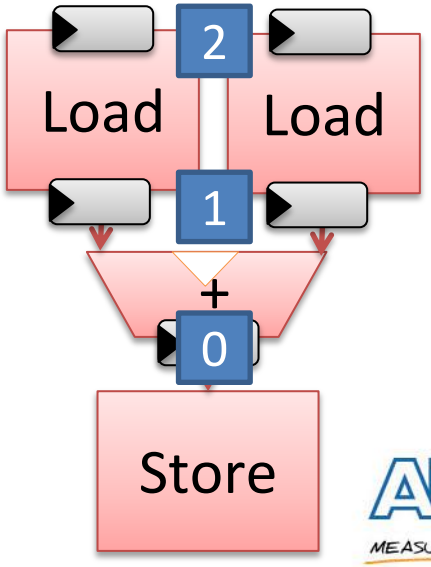
Recall: FPGA Pipeline Parallelism Implementation

- **Very well suited to data parallel problems where all loop iterations can be executed in parallel**

```
for (int i=0; i < n; i++)
{
   answer[i] = a[i] + b[i];
}
```

- **On the FPGA, we use the idea of pipeline parallelism to achieve acceleration**

```
__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];
}
```



Loop Carried Dependencies

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

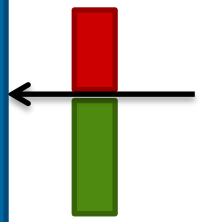
```
__kernel void generate_rngs(ulong num_rnds)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<num_rnds; i++) {
    state = generate_next_state( state );
    unit y = extract_random_number( state );
    write_channel_altera(RANDOM_STREAM, y);
  }
}
```

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

New in 13.1: Loop Pipelining

- **To achieve acceleration, we can *pipeline* each iteration of a loop containing loop carried dependencies**
  - Analyze any dependencies between iterations
  - Schedule these operations
  - Launch the next iteration as soon as possible

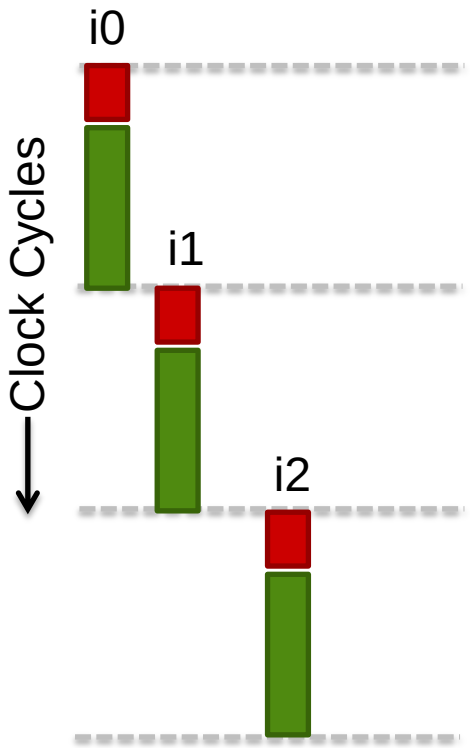
```
__kernel void generate_rngs(ulong num_rnds)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<num_rnds; i++) {
    state = generate_next_state( state );
    unit y = extract_random_number( state );
    write_channel_altera(RANDOM_STREAM, y);
  }
}
```



At this point, we can launch the next iteration

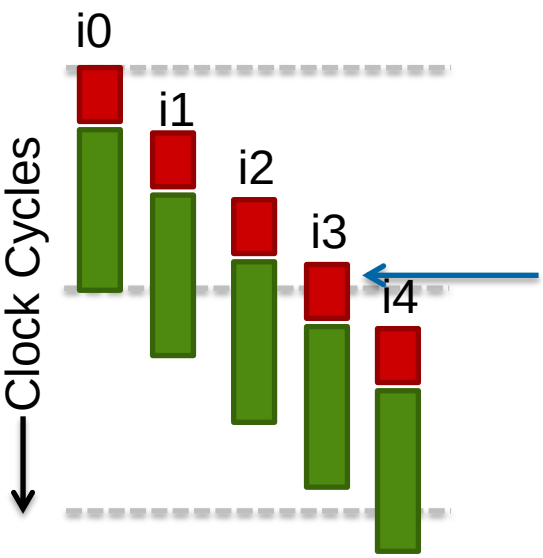
Loop Pipelining Example

- **No Loop Pipelining**



No Overlap of Iterations!

- **With Loop Pipelining**

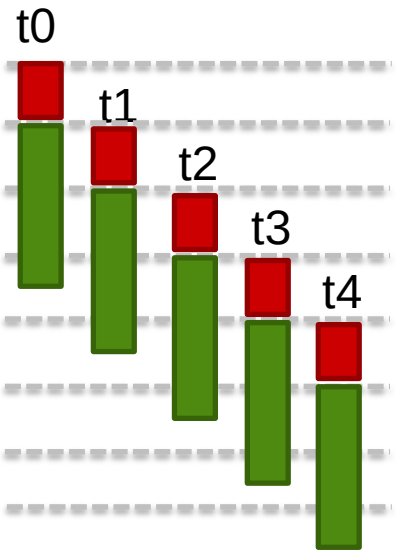


Looks almost like ND-range thread execution!

Finishes Faster because Iterations Are Overlapped

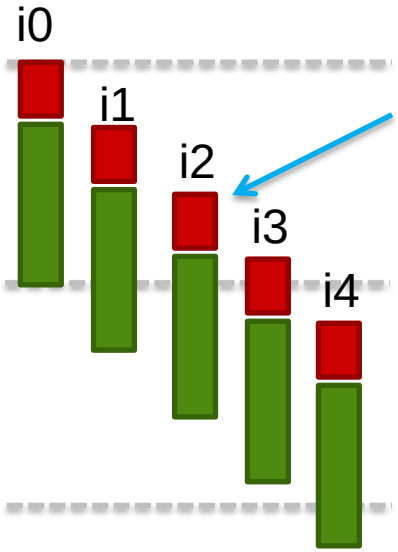
Parallel Threads vs. Loop Pipelining

- **So what's the difference?**



Parallel Threads

Parallel threads launch 1 thread per clock cycle in pipelined fashion



Loop Pipelining

Loop dependencies may not be resolved in 1 clock cycle

- **Loop Pipelining enables Pipeline Parallelism *AND* the communication of state information between iterations.**

Mersenne Twister Random Number Generator

```
__kernel __attribute__((task)) void mersenne_twister_generate(ulong N)
{
   unsigned int mt[MT_N];

   bool read_from_initialization = true;
   ushort num_initializers_read = 0;

   for (ulong n=0; n<N+MT_N; n++) {
      unsigned y;

      bool write_channel = false;
      if (read_from_initialization) {
         y = read_channel_altera(INIT_STREAM);
         if (++num_initializers_read == MT_N) read_from_initialization=false;
      } else {
         y  = (mt[0]&UPPER_MASK)|(mt[1]&LOWER_MASK);
         y = mt[MT_M] ^ (y >> 1) ^ (y & 0x1UL ? MATRIX_A : 0x0UL);
         write_channel = true;
      }

      #pragma unroll
      for (int i=0; i<MT_N-1; i++) {
         mt[i]=mt[i+1];
      }
      mt[MT_N-1]=y;

      // Tempering
      y ^= (y >> 11);
      y ^= (y << 7) & 0x9d2c5680UL;
      y ^= (y << 15) & 0xefc60000UL;
      y ^= (y >> 18);

      float u = (float)y / 4294967296.0f;

      if (u == 0.0f) u = CLAMP_ZERO;
      if (u == 1.0f) u = CLAMP_ONE;

      if (write_channel) write_channel_altera(RANDOM_STREAM, u);
   }
}
```

- *__attribute__((task))* is a compilation style hint used to enable loop pipelining.
- The state of the MT RNG is maintained by 624 unsigned integers. The loop carried dependence is based on generating the next iteration state based on current state.
- *For simplicity, we show code for 1 random number per clock cycle rather than 64 per clock cycle.*

Stock Price Motion and Partial Accumulation Kernel

```
__kernel void black_scholes( int m, int n, float drift, float vol, float S_0, float K) {
   // running statistics -- use double precision for the accumulator
   double sum = 0.0;
   for(int path=0;path<m;path++) {
      float S = S_0;
      float arithmetic_average = 0.0f; // We're not including the initial price in the average
      for (int t_i=0; t_i<n/16; t_i++) {
         barrier(CLK_GLOBAL_MEM_FENCE);
         float Z[16];
         float16 U = read_channel_altera(RANDOM_STREAM);
         ...
         #pragma unroll 8
         for (int i=0; i<8; i++) {
            // Convert uniform distribution to normal
            float2 z = box_muller(U[2*i], U[2*i+1]);
            Z[2*i] = z.x; Z[2*i+1] = z.y;
         }
         #pragma unroll 16
         for (int i=0; i<16; i++) {
            // Simulate the path movement using geometric brownian motion
            S *= drift * exp(vol * Z[i]);
            arithmetic_average += S;
         }
      }
      arithmetic_average /= (float)(n);
      // Check if the average value exceeds the strike price
      float call_value = arithmetic_average - K;
      if (call_value > 0.0f) {
         sum += call_value;
      }
   }
   // send a final result to the accumulate kernel after each thread has already accumulated (m) paths
   write_channel_altera(ACCUMULATE_STREAM, sum);
}
```

- Each thread reads a sequence of random numbers from a channel
- The key computation loop is a fully unrolled floating point datapath
- *For simplicity, we show code for 16 parallel simulations / cycle rather than 64.*

Results

```
Querying platform for info:
==========================
CL_PLATFORM_NAME                         = Altera SDK for OpenCL
CL_PLATFORM_VENDOR                       = Altera Corporation
CL_PLATFORM_VERSION                      = OpenCL 1.0 Altera SDK for OpenCL, Version 13.1

Programming Device(s)
Using AOCX: mcbs_v64_bm_131_s5phq_d8.aocx
Starting Computations
DEVICE 0: r=0.08 sigma=0.30 T=1.0 S0=30.0 K=29.0 : Resulting Price is 3.133114
DEVICE 1: r=0.08 sigma=0.30 T=1.0 S0=30.0 K=28.0 : Resulting Price is 3.746063
2 Devices ran a total of 4.1943e+011 Simulations
Throughput = 24.19 Billion Simulations / second
```

Each Device processes a different strike price (K)

2 Device Result

- **The Asian Option demonstration runs on any board containing an Altera Stratix V D8 device.**
- **Each FPGA in the system is capable of pricing an independent set of options in parallel**

ALTERA MEASURABLE ADVANTAGE™

Results (2)

	Performance	Power	Performance / Power
Altera Stratix V D8	12.0 Billion Sims/s	45 W	266 MSims / s / W
Leading GPU*	10.1 Billion Sims/s	212 W	48 MSims / s / W

- **Altera's FPGA Platforms can provide high-performance, power-efficient and programmer-friendly acceleration of complex financial simulations.**
- **FPGA has 5.5x better performance / watt than GPU based solution.**

*Uses the largest available 28nm GPU and Vendor's example code for Asian option pricing.

Thank You