2017 IEEE International Parallel and Distributed Processing Symposium Workshops

# Exploiting Decoupled OpenCL Work-Items with Data Dependencies on FPGAs: A Case Study

Javier Alejandro Varela, Norbert Wehn
Microelectronic Systems Design Research Group
University of Kaiserslautern, Germany
{varela, wehn}@eit.uni-kl.de

Qian Liang, Songyin Tang
BearingPoint GmbH, Germany
{qian.liang, songyin.tang}@bearingpoint.com

***Abstract*—In the field of high performance heterogeneous computing systems, field programmable gate arrays (FPGAs) have shown great advantages in terms of acceleration and energy efficiency. And with the inclusion of the OpenCL framework for parallel programming, the design complexity has been greatly reduced. However, the parallel implementation of applications containing data-dependent branches usually experiences an important loss in performance, which affects all platforms alike. This data dependency leads the execution of parallel threads, also called work-items in OpenCL, to diverge. Whereas fixed architectures like CPU, GPU and Xeon Phi cannot efficiently cope with this divergent execution, the flexibility offered by FPGAs in terms of architecture can be exploited to tackle this problem.**

**In this work, we present a new approach for FPGA implementations that decouples the parallel OpenCL work-items, avoiding the interference of data-dependent branches between them. We also demonstrate the necessary workarounds to obtain the maximum performance in a pipelined design, when unpredictable for-loop exit conditions are caused by the data dependency. Furthermore, we show how to efficiently interleave computation with transfers to device global memory in each work-item.**

**This approach is then evaluated with a real-life case study from Finance, with four different configurations implemented on FPGA with Xilinx SDAccel, and compared to the optimized implementation on CPU, GPU, and Xeon Phi. Our results show that FPGAs can deliver up to 5.5x speedup, whereas the system-level energy efficiency increases between 2x and 9.5x in all cases.**

## I. Introduction

An FPGA is a computing device that offers an empty canvas of hardware resources and interconnects that can be customized to the target application [1], providing four main advantages over traditional architectures: pipelined data paths, infinite bit-level parallelism, custom-precision data types, and customized memory hierarchy. These characteristics are gaining increasing attention in high performance and cloud computing, in particular with the recent inclusion of FPGAs in the new Amazon Elastic Compute Cloud (EC2) F1 instance [2]. Maxeler Technologies, one of the largest providers of FPGA-based systems in the Finance industry, has recently announced it will exploit these Amazon instances to expand its services for compute-intensive financial risk simulations [3].

In terms of heterogeneous computing systems, the Open Compute Language (OpenCL) [4] has become the standard framework for general purpose parallel programming [5], enabling the migration between different platforms due to its code portability. Although performance might not be portable, the code can be optimized to the underlying architecture. This also allows for a fair comparison among different platforms in terms of runtime, energy consumption, or even both.

When an algorithm is mapped onto multiple parallel threads, which are called *work-items* in OpenCL, data-dependent branches should always be avoided, as they reduce the overall performance. The reason is that these work-items are physically grouped in hardware [6], [7], or implicitly vectorized [8], especially in accelerators with fixed architecture. This is the case for example on central processing unit (CPU), graphics processor unit (GPU), and Intel's Xeon Phi, a many integrated core (MIC) architecture.

Exploiting the mentioned FPGA characteristics, we present a new design approach to decouple the OpenCL work-items on FPGAs, in such a way that the divergent branches in different work-items are independently executed. Besides, the necessary memory transfers to device (accelerator) global memory are efficiently interleaved with the fully pipelined computations, in order to achieve maximum throughput.

Additionally, we tackle applications with data-dependent branches that require loops with dynamically-modified limits. One such case is a limit based on a counter whose value is incremented inside a divergent branch. This is a challenge in pipelined architectures on FPGAs to achieve the maximum throughput, i.e. an initiation interval of one clock cycle.

One type of algorithms with these characteristics are rejection-based methods, where an output is accepted/rejected based on a data-dependent rule condition, and where the loop iterations depend on a counter. Therefore, we evaluate our new approach by implementing on FPGA a nested rejection-based algorithm used in real-life financial applications, and we compare its performance in terms of runtime and system-level energy efficiency, against CPU, GPU, and Xeon Phi.

In summary, we provide detailed analysis and results on:

- a new generic approach to implement decoupled OpenCL work-items on FPGAs,
- how to handle dynamically-modified for-loop exit conditions in pipelined architectures with an initiation interval of one clock cycle,
- interleaving the pipelined work-items with transfers to device global memory,
- the comparison to CPU, GPU, and Xeon Phi in terms of runtime and energy efficiency.

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.34

## II. Background and Related Work

OpenCL is an open, royalty-free standard for general purpose parallel programming that can be ported to multiple platforms, and it supports both data- and task-based parallel programming models [4]. This framework assumes the presence of a host (usually a CPU) and an accelerator (device) connected to it via Peripheral Component Interconnect Express (PCIe). A device contains *compute units*, each subdivided into *processing elements*. This structure, as well as the memory hierarchy, is shown in Fig. 1.

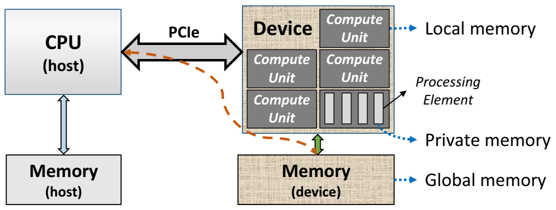


Fig. 1: OpenCL basic hardware and memory structure.

Kernels are enqueued by the host as a *Task* (basically a single-threaded kernel), or as an *N-Dimensional Range* (NDRange) with a defined number of *work-items* (*globalSize*) grouped into *work-groups* of *localSize work-items*. The host transfers data (read/write) to device global memory, by pre-declaring the necessary buffers.

In this work we consider four OpenCL accelerators. First, a CPU is the most general-purpose platform with good performance per single core/thread [9]. Although regularly used as a host in OpenCL applications, a CPU can also be used as an accelerator [7]. Second, Intel's Xeon Phi is a many integrated core (MIC) architecture, where each core includes a 512-bit vector arithmetic unit for wide single instruction multiple data (SIMD) instructions [8]. Third, on GPU we can execute a very large number of parallel threads on highly optimized data paths, hiding this way the latency of memory accesses [6]. Fourth, an FPGA offers a set of hardware resources that can be configured and interconnected based on the target application [1], namely flip-flops (FFs), lookup tables (LUTs), digital signal processors (DSPs), and Block RAMs (BRAMs).

### A. OpenCL on FPGAs

Whereas CPUs, GPUs, and Xeon Phi have a fixed architecture that executes the compiled OpenCL code, the hardware configuration on FPGAs is contained in a file called *bitstream*, which is generated with vendor-specific tools that translate the OpenCL code. In our case we employ the Xilinx SDAccel tool (version 2015.4) [1], which provides the development environment and the necessary hardware interface via PCIe, delivering an increase in the overall design productivity [10].

*Compute units* on FPGAs are instantiated at design time. SDAccel maps each *work-group* of an *NDRange* kernel to one *compute unit*, and inside the latter the corresponding *work-items* are mapped to a single pipeline using a set of nested for-loops [11]. In this regard, spatial parallelization of *work-items* is then achieved by instantiating several *compute units*.

To transform the OpenCL code into the required register-transfer level (RTL) implementation, SDAccel uses Xilinx Vivado high-level synthesis (HLS) compiler [12]. Certain features and libraries inherited from Vivado HLS are only available in .c kernels launched as *Tasks*. This includes arbitrary precision data types (*ap_int.h*) and arbitrary precision fixed point types (*ap_fixed.h*) [12], which are necessary in our test case application. Working at .c level also offers low-level control of the architecture, including the HLS *pragmas* that allow us to manually instantiate parallel *work-items*.

### B. Divergent OpenCL Work-Items

On accelerators with fixed architecture, parallel OpenCL *work-items* are physically grouped in hardware [6], [7], or the *work-groups* are implicitly vectorized [8]. This can be seen as having hardware partitions of fixed size. For example, Nvidia GPUs schedule *warps* (hardware partitions) of 32 threads (*work-items*) that execute the same kernel computation on different data, while Intel Xeon Phi uses a 512-bit implicit vectorization unit (SIMD) on each *work-group*.

The ideal execution of *work-items* in a hardware partition can be exemplified as in Fig. 2a, which includes static branches that take the same side in all *work-items*. However, data-dependent branches lead the *work-items* to diverge, as exemplified in Fig. 2b. When this happens, the hardware partition needs to evaluate both sides of each branch, which can only be done sequentially. As a result, the *work-items* not executing the current side of the branch (or the current operation) become idle, causing a significant loss in performance (Fig. 2b).

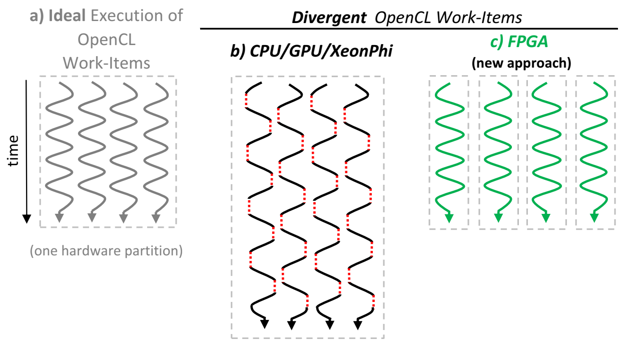


Fig. 2: (a) all *work-items* execute the same instruction at any given time; (b) divergent *work-items* become idle (red dots) on fixed architectures; (c) our new approach on FPGAs.

### C. New Approach: Decoupled OpenCL Work-Items on FPGAs

To avoid the loss in performance caused by data-dependent branches (Fig. 2b), meaning branches whose condition has a *true dependency* on the results from preceding instructions (see e.g. [13]), we exploit the flexibility of FPGAs to generate fully decoupled OpenCL *work-items* that do not interfere with each other. This can be seen as having multiple hardware partitions of one *work-item* each, as exemplified in Fig. 2c.

To maximize throughput on FPGAs, these *work-items* must be fully pipelined with an initiation interval of one clock cycle,

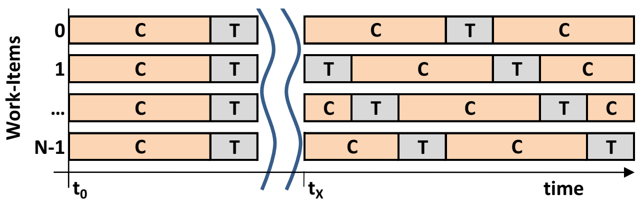


Fig. 3: OpenCL *work-items* schedule in time: $C$ = Computation, $T$ = Transfer to device global memory.

and all transfers to device global memory must be interleaved with the pipelined computations, as shown in Fig. 3. By exploiting the maximum width in the memory interface, the output data can be packed and transferred into bursts of fixed length. Therefore, the time consumed by each transfer is smaller than the corresponding computation. Following Fig. 3, all *work-items* are triggered at $t_0$, but the transfers to memory can only occur one at the time on a single memory channel. Then at a later time $t_X$ the *work-items* become shifted in time, efficiently overlapping computation and transfers.

Following Section II-A, our new design approach is developed for the general case of .c kernels launched as a *Task*, with guidelines on how to adapt it to the .cl *NDRange* case. Details on the *work-group* granularity are discussed in Section III-A.

### D. Test Case Application: Basics

Our representative test case is the Marsaglia-Tsang rejection-based algorithm to generate gamma-distributed random numbers (RNs) [14], shown in Fig. 4, which is a *nested* random number generator (RNG) that requires three input RN sequences: one with normal and one with uniform distribution to generate and accept/reject the output RN, and a second uniform distribution to correct this output under certain input values. Both the rejection and the correction rules have a *true data dependency* (see for example [13]) on the underlying RNGs (see later Listing 2). Details of the algorithms mentioned in Fig. 4, as well as the application, are given below:

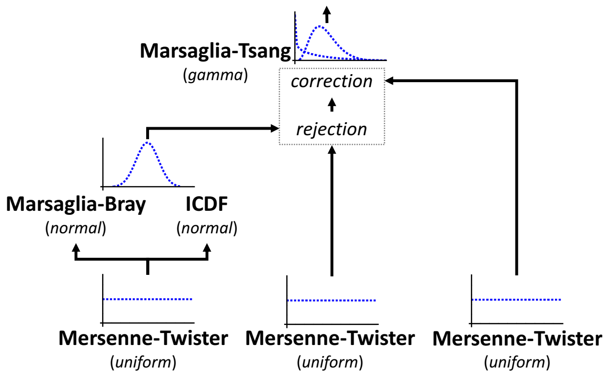


Fig. 4: Test case application: Marsaglia-Tsang gamma RNG.

*1) Mersenne-Twister:* it is widely used to obtain uniform distributions [15], and on FPGAs it requires a small amount of resources. It needs a vector of internal states and a set of logic operations (such as *shift* and *xor*) to obtain the output RN. Each state is only used once and updated accordingly.

*2) Marsaglia-Bray Transformation:* one of the several ways of converting from uniform to normal distributed RNs is by means of rejection methods [16]. Although the Marsaglia-Bray method avoids the heavy trigonometric math operations used in the well-known Box-Muller method [17], it still runs complex floating-point operations such as *log*, *sqrt*, and division. Its rejection rate becomes a challenge in terms of implementation, and it also needs two input uniform RNs to generate one output. If necessary, the two input sequences can be split into two parallel Mersenne-Twisters following [18].

*3) Inverse cumulative distribution function (ICDF) Transformation:* it is an alternative method to convert from uniform to normal distributed RNs [16], that only requires logic operations. Our implementation on FPGA follows [19], which is optimized for bit-level logic operations. For the other three platforms, these operations need to be replaced by their equivalent 32-bit unsigned integer version using *shifts* together with *and* and *or* masking operations. However, we later show that this modification becomes inefficient in terms of runtime, especially on CPU and Xeon Phi. As a replacement, we implement a modified version of Nvidia's *_curand_normal_icdf* function (*curand_normal_static.h*, available in Compute Unified Device Architecture (CUDA)), with one important modification: we replace the *erfcinv* function with a more appropriate version that minimizes divergent branches [20], together with the identity $erfcinv(x) = erfinv(1-x)$.

*4) Application of Gamma RNs:* CreditRisk+ is a financial model to perform credit risk analysis in a portfolio of loans, and it is the only such model that focuses on the event of default [21]. Under compute-intensive Monte Carlo (MC) simulations, the economy state is simulated by the combination of sectors, which are assumed to be stochastically independent gamma-distributed RNs with expectation $E(S_k) = 1$ and variances $Var(S_k) = \sigma_k^2 = v_k$, $k = 1, ..., N$. Therefore, $S_k \sim \text{Gamma}(a_k, b_k)$, where $a_k = 1/v_k$ and $b_k = v_k$. The larger the simulated gamma variable is, the worse is this financial sector in the current simulation run.

### E. Test Case Application: Challenges on FPGAs

Generated gamma-distributed RNs are validated or rejected based on a stochastic process. Therefore, a valid output will not be available in every clock cycle, which is a challenge in pipelined architectures. The same applies to the Marsaglia-Bray algorithm and to the ICDF function's implementation.

Following Fig. 4, every time Marsaglia-Bray (or ICDF) invalidates its output, the following two Mersenne-Twisters should stop; and every time a gamma RN is rejected, the Mersenne-Twister used for gamma correction should also stop. If this was not the case, we would be incorrectly discarding RNs, causing a distortion in the uniform distributions.

Furthermore, the complete process is inserted inside one or more for-loops, whose limits cannot be determined at design time. Therefore, the limit of the main (inner) for-loop needs to be dynamically adapted to the underlying stochastic process.

## *F. Related Work*

The implementation challenges associated with rejection methods in parallel programming are discussed in [22]. To the best of our knowledge, our new approach to decouple OpenCL work-items on FPGAs is exploited for the first time, especially in a generalized way with algorithms that contain data-dependent branches. This also includes the handling of dynamically-modified for-loop exit conditions, and the full implementation of our test case application (see Section II-D).

## III. FPGA Implementation

Following Section II, here we present all implementation details, including: how to decouple OpenCL work-items in general, specific implementation details on the test case application, the necessary modification to the Mersenne-Twister pseudo random number generator (PRNG) implementation, the transfers from each work-item to device global memory, and how to efficiently handle the buffers in device global memory.

### *A. Decoupled Work-Items Generation*

In Listing 1 we present the function that packs *N parallel work-items*, which is inlined in the main kernel function.

Listing 1: Decoupled Work-Items

```
inline void DecoupledWorkItems  ( ...,
                    // repeat N times for index _i
                    ap_uint<512>* gammaValues_i )
{
  #pragma HLS DATAFLOW
  // --------- WORK ITEM ----------------
  // repeat N times for index _i
  hls::stream<float> gS_i;
  #pragma HLS STREAM variable=gS_i depth=2048
  GammaRNG (_i, ... , gS_i);
  Transfer (_i, ... , gS_i, gammaValues_i);
  // ------------------------------------
  return;
}
```

To provide more flexibility at design time, in each work-item we separate computations from the transfers to memory (see Fig. 3). In doing so, we need the *hls::stream* interface [12] to introduce blocking communication between generation (*GammaRNG*) and the corresponding *Transfer* function.

The *DATAFLOW* pragma [11], [12] schedules the work-items in parallel, under the constraint that each variable has a single producer-consumer pair. This prevents us from instantiating a single pointer to device global memory. However, we offer a workaround by assigning each work-item its own pointer to memory, following Sections III-D and III-E.

Each work-item in Listing 1 is given manually a unique *id* (*_i*) at design-time, the same way OpenCL would assign them in a .cl kernel. Notice that whereas each work-group in an *NDRange* kernel (.cl) would be completely mapped to a single pipeline (see Section II-A), here we are directly instantiating each work-item in parallel inside a single *Task*. Although the latter limits *localSize* to 1 (the size of a work-group), an *NDRange* has a flexible work-group granularity. In either case, what directly affects the overall runtime is the number of pipelines (work-groups) instantiated in parallel.

### *B. Test Case Application: Design and Workarounds*

Following the implementation challenges discussed in Section II-E, we propose an efficient workaround by creating a single block of code which is fully pipelined, and where each gamma RN is computed, corrected, and only validated afterward. The three Mersenne-Twisters are placed in a sequence, using a set of flags to modify their behavior on-the-fly (details are covered in Section III-C). Only validated (and possibly corrected) gamma RNs are written to an output *hls::stream*, transferring them to the next step, as shown in Listing 2.

Every time the function outputs a validated gamma RN, a *counter* is incremented, which is used to dynamically determine the break point of *MAINLOOP*. This dependency on the value of the *counter* hinders an initiation interval of one clock cycle. To overcome this effect, we delay the count as many iterations as necessary using a completely partitioned array and a *breakId* (index). This index is kept as low as possible, and here it suffices to use zero (meaning a delay of one cycle).

Listing 2: Test Case Application

```
inline void GammaRNG (..., const unsigned int wid,
                 hls::stream<float> &gammaStream )
{
  // ... all initializations ...
  // ------------------------------------
  SECLOOP: for(sector=0;sector<limitSec;++sector)
  {
    bool alphaFlag = (alpha<=1.0f)?true:false;
    // ------------------------------------
    unsigned int counter = 0;
    const short breakId = 0;
    unsigned int prevCounter[breakId+1];
    // ------------------------------------
    MAINLOOP: for(unsigned int k=0; (k<limitMax)
        &&(prevCounter[breakId]<limitMain); ++k)
    {
      #pragma HLS pipeline II=1
      // ------------------------------------
      UpdateRegUI(breakId, counter, prevCounter);
      // Normal RN
      float n0;
      //bool n0_valid = ICDF(&n0,MT0(true,...));
      bool n0_valid = M_Bray(&n0,MT0(true,...));
      // Uniform RN (for rejection)
      float u1 = uint2float (MT1(n0_valid,...));
      // Rejection Method
      float gRN;
      bool gRN_valid  = GammaRN(&gRN, n0, u1);
      bool gRN_ok     = n0_valid && g_valid;
      // Uniform RN + for correction
      float u2 = uint2float(MT2(gRN_ok,...));
      float gRN_ = Correct(gRN, u2, alpha);
      // Output
      float gamma = (alphaFlag)?gRN_:gRN;
      if(gRN_ok && (counter<limitMain))
      { gammaStream.write(gamma);
        ++counter;
  } } }
  return;
}
```

### *C. Mersenne-Twister Adapted Implementation*

As mentioned before, the Mersenne-Twisters should be stopped whenever necessary. But in order to achieve an initiation interval of one clock cycle, these blocks are allowed to run continuously, using an external flag to enable the internal state update, see Listing 3. Once the current state is finally used and updated, the state index is incremented by one.

Listing 3: Mersenne-Twister (Adapted)

```
inline unsigned int MT ( bool updateFlag,...)
{
  unsigned int currI0 = stateI0[*i0]; ...;
  // ------------------------------------
  unsigned int y = Update(currI0,...);
  // ------------------------------------
  if(updateFlag) stateI0[*i0] = y; ...;
  // ------------------------------------
  unsigned int U = Tempering(currI0,...);
  // ------------------------------------
  unsigned short inc = (updateFlag)?1:0;
  *i0 = ((*i0)>=(MT_N-1))?0:(*i0)+inc; ...;
  return U;
}
```

## D. Transfers to Memory: Device Side

The transfers to memory are more efficient when the full width of the interface is used, and when the transfers are generated in bursts. The interface on SDAccel and the given board (Section IV-A) is 512 bits, equivalent to 16 single-precision floating point values [11]. The gamma RNs are read one after the other and packed in groups of 16, which is the equivalent of *float16* in an *NDRange* kernel. The bursts are obtained by means of the *memcpy* function, and an offset is used to allocate the blocks of data consecutively in memory. Listing 4 shows the corresponding code.

Listing 4: Transfers

```
inline void Transfer ( ...,
          const unsigned int wid,
          hls::stream<float> &gammaStream,
          ap_uint<512>* gammaValues)
{
  // ------------------------------------
  ap_uint<512> gamma512;
  ap_uint<512> transfBuf[LTRANSF];
  #pragma HLS DEPENDENCE variable=transfBuf false
  unsigned short i = 0;
  // ------------------------------------
  unsigned int offset = blockOffset * wid;
  // ------------------------------------
  SECLOOP: for(sec=0; sec<limitSec; ++sec)
  {
    REPLOOP: for(rep=0; rep<limitRep; ++rep)
    {
      TLOOP: for(path=0; path<SXTRANSF; ++path)
      {
        #pragma HLS pipeline II=1
        #pragma HLS LOOP_FLATTEN off
        float gamma = gammaStream.read();
        bool  tFlag = g512(&gamma512, gamma, ...);
        if(tFlag) transfBuf[i] = gamma512;
        if(tFlag) i=(i>=(LTRANSF-1))?0:i+1;
      }
      memcpy((ap_uint<512>*) (gammaValues+offset),
          transfBuf, sizeof(float)*SXTRANSF);
      offset += LTRANSF;
  } } // REPLOOP -- // SECLOOP ----------
  return;
}
```

## E. Transfers from Memory: Host Side

Assume we have $N$ work-items. Following Section III-D, assigning each transfer function (therefore, each work-item) with its own pointer to memory would imply the allocation of $N$ OpenCL buffers, and $N$ *read* requests from device memory. In general, the host would preferably handle a single buffer in host memory, which in turn means the reads should be combined into one. To this end, we present two solutions:

*1) Combining buffers at host level:* The host allocates $N$ buffers in device global memory with length $L/N$, being $L$ the total length, whereas a single buffer with length $L$ is allocated in host global memory. Then $N$ *read* requests are enqueued, each with an offset $wid * L/N$ on the destination host buffer, being $wid$ the id of the corresponding work-item.

*2) Combining buffers at device level:* We handle a single buffer in device global memory, and a single buffer in host global memory, both with length $L$. Then the same device buffer is assigned $N$ times to the kernel, and each work-item determines its transfer offset based on their $wid$ (see Listing 4). Note that this approach does not reduce the performance on the device side (less than $1\%$ loss for the setup in Section IV-B). Because it requires a single *read* request from device global memory, this is the chosen approach in this project.

## IV. Results

Hardware and simulation setups are defined, followed by results on resources utilization, runtime, and energy efficiency.

## A. Hardware Setup

For this project we make use of a high-performance workstation with the following setup: SuperMicro Superserver 7048GR-TR: dual-socket X10DRG-Q motherboard, 2000W high-efficiency redundant power supplies (Titanium level), cooling kit MCP-320-74701-0N-KIT (for GPU and Xeon Phi); *CPUs*: (2x) Intel Xeon Processor E5-2670 v3 (Haswell architecture, technology node 22 nm, 12 cores (24 threads), base frequency 2.3 GHz), 64GB DDR4; *Xeon Phi*: (1x) Intel Xeon Phi Coprocessor 7120P (Many Integrated Core (MIC) architecture, technology node 22 nm, 61 cores, base frequency 1.238 GHz, 16 GB, passive cooling); *GPU*: (x1) Nvidia Tesla K80 GPU Accelerator (dual GK210, Kepler architecture, technology node 28 nm (2x) 2596 CUDA Cores, base frequency 560 MHz, (2x) 12 GB, passive cooling); *FPGA*: (x1) Alpha Data ADM-PCIE-7V3 (Xilinx Virtex-7 XC7VX690T-2 (FFG1157C) FPGA, technology node 28 nm, SDAccel frequency 200 MHz, 16 GB, active cooling/small fan). Operating system: Linux RedHat 6.6 (Linux 2.6.32-504.el6.x86_64); room temperature is controlled at $\sim 23°C$.

In this work we evaluate and compare four combinations *host+accelerator*, which are named:

- CPU: *CPU+CPU* (dual-socket motherboard),
- GPU: *CPU+GPU*,
- PHI: *CPU+Xeon Phi*,
- FPGA: *CPU+FPGA*.

## B. Simulation Setup

We evaluate four configurations of the test case application, as shown in Table I. These configurations affect the resources utilization, runtime and energy consumption.

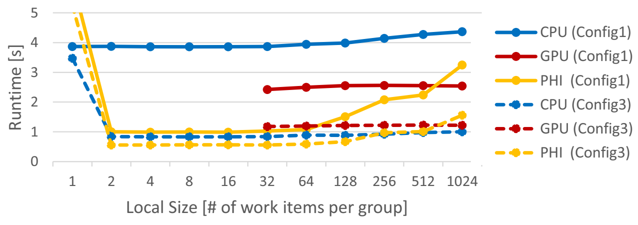
TABLE I: Simulation Setup: Application Configurations

| Config. Name | Uniform to Normal Transformation | Mersenne-Twister [18] Exponent | Period | States |
|---|---|---|---|---|
| Config1 | Marsaglia-Bray | 19937 | $2^{(19937-1)}$ | 624 |
| Config2 | | 521 | $2^{(521-1)}$ | 17 |
| Config3 | ICDF | 19937 | $2^{(19937-1)}$ | 624 |
| Config4 | | 521 | $2^{(521-1)}$ | 17 |

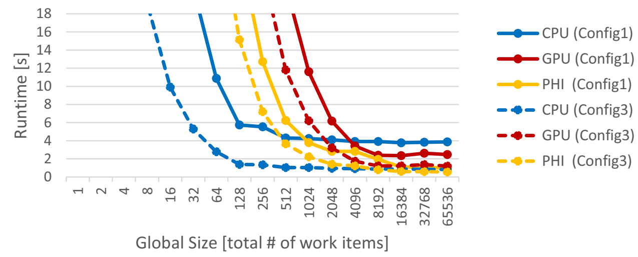
The code running on CPU, GPU and PHI is also optimized: 1) memory accesses on GPU/PHI are coalesced, whereas each work-item on CPU writes to consecutive addresses; 2) their ICDF implementation is optimized for fixed-architectures, which differs from the optimal implementation on FPGA (see Section II-D3). We later show in Section IV-E the runtime of both versions. Although using platform-specific programming frameworks could deliver slightly better performance, OpenCL enables a fair comparison under similar conditions.

We assume from now on the generation of large sequences, where the length can be set at runtime. We also assume that the four accelerators send the gamma RNs back to the host. Therefore, we focus on reducing the kernel runtime.

To define *localSize* for CPU, GPU and PHI, we run the application and plot the profile shown in Fig. 5a. From here we can derive $localSize_{CPU} = 8$, $localSize_{GPU} = 64$, and $localSize_{PHI} = 16$. Given the optimal *localSize* per platform, in Fig. 5b we confirm the choice of *globalSize* set to 65536.



(a) Runtime vs *localSize* for *Config1,3*, at $globalSize = 65536$.



(b) Runtime vs *globalSize* for *Config1,3*, at the optimal $localSizes$.

Fig. 5: Measured runtime vs *localSize* and *globalSize* for *Config1,3*. The remaining configurations yield a similar plot.

In the case of FPGAs, the number of parallel work-items depends on the number of available hardware resources (detailed in Section IV-C). Achieved: 6 work-items with *Config1,2* and 8 work-items with *Config3,4*. This corresponds to the number of work-groups in a .cl *NDRange* kernel (see Section III-A).

Additional parameters: $numScenarios = 2,621,440$ and $numSectors = 240$ (see Section II-D4), which corresponds to a total of $\sim 2.5\ GB$ of generated data (in single-precision floating point) per simulation run; the representative financial sectors variance is $\upsilon = 1.39$ (see later Section IV-E).

## C. FPGA Resources Utilization

For our final FPGA implementations we have iteratively increased the number of parallel work-items in steps of one, as far as the place-and-route process allowed. Table II shows that in all cases the design is limited by the number of slices.

TABLE II: FPGA P&R Resources Utilization Report

| Resources[1] Name | Available | Utilization [%][2] Config1 | Config2 | Config3 | Config4 |
|---|---|---|---|---|---|
| Slice[3] | 107400 | 53.43 | 52.75 | 52.92 | 52.72 |
| DSP | 3600 | 23.67 | 23.67 | 21.56 | 21.56 |
| BRAM | 1470 | 20.31 | 20.31 | 24.05 | 24.05 |

[1] Includes: 1) reconfigurable OCL (OpenCL) region; 2) static region (PCIe).
[2] Information on regions' size is not available. After several trial-and-error tests we estimate the available OCL region at approx. 2/3 of the total resources. The corrected utilization for slices is estimated at 80%.
[3] Each slice contains 4 LUTs and 8 FFs.

## D. Results Validation

In Fig. 6 we show two representative gamma-distributed RN sequences generated on FPGA, under the given setup. For the same input parameters, the distributions look very similar to the benchmark distribution generated in Matlab, and become closer and closer to it as the number of samples increases.

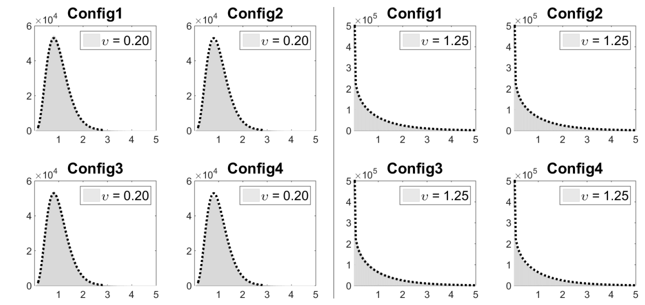


Fig. 6: FPGA gamma distribution (gray color area) vs Matlab benchmark *gamrnd* function (dotted black line); $\upsilon$ corresponds to the financial sector variance (see Section II-D4).

## E. Runtime

Table III summarizes the measured runtime under the given setup (see Section IV-B), averaged over several iterations for a minimum of 100 seconds. In the case of *Config1* the FPGA achieves the best performance in terms of runtime, reaching a $5.5x/3.5x/1.4x$ speedup vs CPU/GPU/PHI respectively, with comparable runtime to PHI under *Config2*. Here the combination of the Marsaglia-Bray algorithm to obtain normally distributed RNs and the Marsaglia-Tsang algorithm to generate the gamma-distributed outputs (see Fig. 4) achieves a rejection rate of 30.3% for the given setup. From a Matlab benchmark simulation, this rate ranges from 27.8% for a sector variance $\upsilon = 0.1$, to 33.7% for $\upsilon = 100$.

TABLE III: Runtime [ms] for the given Setup

| **Setup** | **CPU** | **GPU** | **PHI** | **FPGA** |
|---|---|---|---|---|
| Config1 | 3825 | 2479 | 996 | 701 |
| Config2 | 3883 | 1011 | 696 | 701 |
| Config3: ICDF CUDA-style[1] | 807 | 1177 | 555 | - |
| Config3: ICDF FPGA-style[1] | 2794 | 1181 | 2435 | 642 |
| Config4: ICDF CUDA-style[1] | 839 | 522 | 460 | - |
| Config4: ICDF FPGA-style[1] | 2776 | 521 | 2294 | 642 |

[1] Refer to Section II-D3 for details on the ICDF versions. On CPU, GPU and PHI, we later use ICDF CUDA-style, due to its higher performance.

In the case of the ICDF function (see Fig. 4), the combined rejection rate is 7.4% for the given setup, and it varies from 5.3% to 10.2% for $\upsilon = 0.1$ tand $\upsilon = 100$ respectively. A lower rejection rate implies less divergent branches for CPU, GPU and PHI, reducing the runtime. In this regard, the FPGA still shows $\sim 2x$ better performance than CPU under *Config3* and *Config4*. Nevertheless, it just achieves $0.9x$ and $0.7x$ of the PHI performance for *Config3* and *Config4* respectively, whereas compared to GPU these values read $1.8x$ and $0.8x$.

The theoretical runtime on FPGA can be approximated by:

$$t \approx \left( \frac{numScenarios * numSectors}{numWorkItems * f_{FPGA}} * (1 + r) \right) \tag{1}$$

where $f_{FPGA}$ is the operating frequency, $r$ refers to the combined rejection rate seen previously (in absolute value), and we exclude the overhead outside the main pipelined for-loop. For the given setup, we obtain: $t_{Config1,2} \approx 683ms$ and $t_{Config3,4} \approx 422ms$ Although former is close to the measured runtime in Table III, the later differs by approximately 35%.

If we now remove the computations from our kernel, leaving only the transfers to device memory, we obtain Fig. 7 for different burst sizes and the number of parallel work-items.

Comparing Fig. 7 with Table III, we find that the transfers to memory determine the overall runtime, with a measured bandwidth equal to $3.58GB/s$ for *Config1,2*, and $3.94GB/s$ for *Config3,4*. Further customizations of the memory controller inside the tool would improve the performance.

## *F. Power and Dynamic Energy Consumption*

In our setup, power measurements were possible at the power plug. To this end, we have used a Voltcraft VC870 digital multimeter, which takes one sample per second. This sample rate is enough in our case, provided the measurement time is kept high enough. The multimeter transmits the results via a dedicated Universal Serial Bus (USB) cable to an external PC, where they are conveniently stored and post-processed.

The measurements are shown in Fig. 8. The host triggers the gamma kernel at the first (vertical) marker, and it enqueues

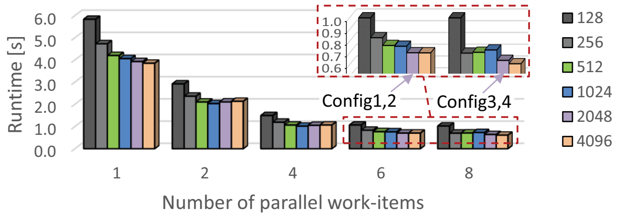


Fig. 7: Measured runtime of transfers-only (using dummy data) with different burst lengths (number of RNs per burst). The memory interface is the maximum available: 512 bits [11].

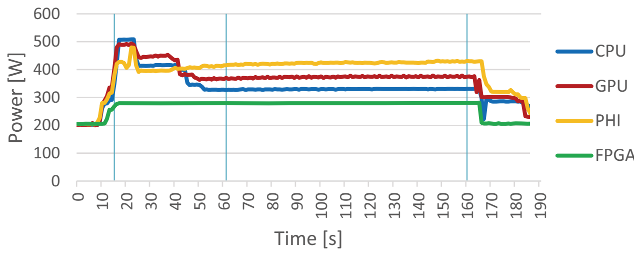
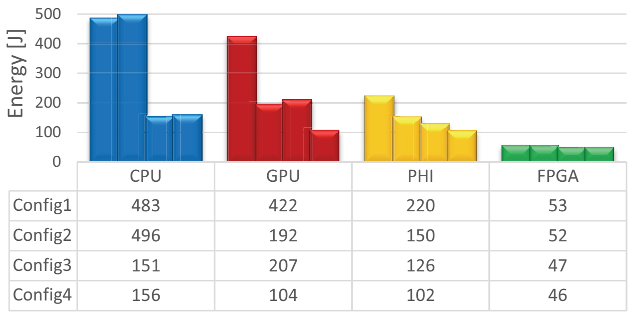


Fig. 8: Power consumption using *Config1*. The measurements of the remaining configurations yield similar plots.

the kernel several times in order to reach over 150 seconds. The power consumption spike seen at system-level at the first marker includes the host, device, PCIe communications, and the cooling system set to dynamically adapt to the workload (*optimal* mode). Because the process of enqueuing the kernel is asynchronous from the host side, after some time the host will remain idle waiting for the *cl_events* to complete (one per kernel invocation), minimizing its power consumption. Therefore, only the time interval of 100 seconds between the last two markers is considered for integration purposes. From this result we subtract the static energy consumption (corresponding to an idle power consumption of $\sim 204W$ in Fig. 8), to obtain the dynamic energy consumption.

This is in fact a fair way of comparing the different platforms under the given hardware setup: we start from a workstation with all devices in idle mode, including the host, all accelerators, and the cooling system, and we are then



| | CPU | GPU | PHI | FPGA |
|---|---|---|---|---|
| Config1 | 483 | 422 | 220 | 53 |
| Config2 | 496 | 192 | 150 | 52 |
| Config3 | 151 | 207 | 126 | 47 |
| Config4 | 156 | 104 | 102 | 46 |

Fig. 9: Derived system-level dynamic energy consumption per kernel invocation.

interested in finding the solution that minimizes the extra energy required for the simulation. It is also fair in terms of the consumption arising from the cooling system, since different devices have different external cooling requirements that should be taken into account at system-level.

Dividing the integrated dynamic energy consumption by the times a kernel is repeated inside the 100-second interval (the number of repetitions is no longer an integer value), we obtain the averaged dynamic energy consumption per kernel invocation. The results are presented in Fig. 9 for all configurations (see Section IV-B).

The FPGA solution shows the best energy efficiency in all cases, ranging from a maximum of $9.5x/7.9x/4.1x$ vs CPU/GPU/PHI under *Config1*, to a minimum of approximately $2.2x$ vs GPU and PHI under *Config4*. This results from a low power consumption combined with an efficient pipelined architecture that delivers a very good runtime performance.

## V. Conclusion

In this work we have tested four configurations of a representative financial application. Our results show that the FPGA implementation achieves very good runtime performance compared to optimized implementations on CPU (used as an accelerator), GPU and Xeon Phi, only slightly underperforming the latter when the memory transfers become the bottleneck. Further customizations of the memory controller inside Xilinx SDAccel would improve the performance. Based on their low power consumption, FPGAs increase the system-level dynamic energy efficiency in all our test configurations, by up to $9.5x$.

The new design approach presented here to decouple OpenCL work-items on FPGAs, can be extended to other algorithms that resemble the rejection methods, with data-dependent branches and dynamic for-loop exit conditions.

In this regard, the *DecoupledWorkItems* function in Listing 1, as well as the *Transfer* block in Listing 4, can be easily reused or customized to any application. The designer just needs to rewrite the application function in Listing 2, the same way it is done on other platforms like GPUs and Xeon Phi. Besides, the adapted implementation of the Mersenne-Twister algorithm in Listing 3 shows how small modifications give more external control on an existing block inside a pipelined architecture, without affecting the maximum throughput.

## Acknowledgment

We gratefully acknowledge financial support from the Deutsche Forschungsgemeinschaft (DFG) within the research training group 1932 ”Stochastic Models for Innovations in the Engineering Sciences”, and from the Center of Mathematical and Computational Modelling (CM)$^2$ of the University of Kaiserslautern. The authors alone are responsible for the content of this paper.

## References

[1] Xilinx, *SDAccel Development Environment User Guide: Features and Development Flows*, 2015.4 ed., Xilinx, Feb. 2016, UG1023, Last access: 27 Nov. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug1023-sdaccel-user-guide.pdf

[2] J. Barr. (2016, Nov) Developer Preview - EC2 Instances (F1) with Programmable Hardware. https://aws.amazon.com/de/blogs/aws/developer-preview-ec2-instances-f1-with-programmable-hardware/. Amazon Web Services. Last access 2016-12-22.

[3] Maxeler Technologies. (2016, Dec) New Amazon EC2 F1 instance bringing Maxeler Maximum Performance Computing to The Cloud. https://www.maxeler.com/f1/. Last access 2016-12-22.

[4] L. Howes and A. Munshi, *The OpenCL Specification*, online, Khronos OpenCL Working Group Std. 2.1, Rev. 8, Jan 2015. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf

[5] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. Waltham: Morgan Kaufmannn, 2012.

[6] D. A. Patterson and J. Hennessy, *Computer Organization and Design*, 4th ed., 4, Ed. Morgan Kaufmann, 2009.

[7] V. Kartoshkin and T. Mattson. (2011, Mar) Tutorial: OpenCL - Introduction for HPC Programmers. Intel. Last access: 03 Dec 2016. [Online]. Available: https://software.intel.com/en-us/articles/tutorial-opencl-introduction-for-hpc-programmers

[8] Intel. (2014, Jan) OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor. Last access: 03 Dec 2016. [Online]. Available: https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor?language=ru

[9] J. Reinders. (2012) An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors. Intel. Last access: 03 Dec 2016. [Online]. Available: http://download.intel.com/newsroom/kits/xeon/phi/pdfs/overview-programming-intel-xeon-intel-xeon-phi-coprocessors.pdf

[10] G. Guidi, E. Reggiani, L. Di Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, “On How to Improve FPGA-Based Systems Design Productivity via SDAccel,” in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 247–252.

[11] Xilinx, *SDAccel Development Environment Methodology Guide: Performance Optimization*, v2.0 ed., Xilinx, August 2016, UG1207 https://www.xilinx.com/support/documentation/sw_manuals/ug1207-sdaccel-performance-optimization.pdf. Last access: 27 Nov. 2016.

[12] ——, *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*, v2014.3 ed., Xilinx, oct 2014.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[14] G. Marsaglia and W. W. Tsang, “A Simple Method for Generating Gamma Variables,” *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 363–372, sep 2000.

[15] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[16] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor, “Gaussian Random Number Generators,” *ACM Comput. Surv.*, vol. 39, no. 4, p. 11, Oct. 2007.

[17] G. Marsaglia and T. Bray, “A convenient method for generating normal variables,” *Siam Review*, vol. 6, no. 3, pp. 260–264, 1964.

[18] M. Matsumoto and T. Nishimura, “Dynamic Creation of Pseudorandom Number Generators,” Online: http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/DC/dgene.pdf, 1998, last access: 20/05/2016. [Online]. Available: {http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/DC/dgene.pdf}

[19] C. de Schryver, D. Schmidt, N. Wehn, E. Korn, H. Marxen, A. Kostiuk, and R. Korn, “A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision,” *International Journal of Reconfigurable Computing (IJRC)*, vol. 2012, Mar. 2012, article ID 675130, 11 pages.

[20] M. Giles, “Approximating the erfinv Function,” in *GPU Computing Gems*, Jade ed., W.-M. W. Hwu, Ed. Morgan Kaufmann, 2012, vol. 2, ch. 10, pp. 109–116.

[21] Credit Suisse First Boston International, “CreditRisk+: A Credit Risk Management Framework,” Tech. Rep., 1997, http://www.csfb.com/institutional/research/assets/creditrisk.pdf.

[22] D. B. Thomas, “Rejection methods on GPUs,” Online: http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-rejection.html, last access: 27 Nov 2016.