OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics

Chen Yang* Jiayi Sheng* Rushi Patel* Ahmed Sanaullah* Vipin Sachdeva† Martin C. Herbordt*
*Department of Electrical and Computer Engineering, Boston University
†Silicon Therapeutics

***Abstract*—FPGAs have emerged as a cost-effective accelerator alternative in clouds and clusters. Programmability remains a challenge, however, with OpenCL being generally recognized as a likely part of the solution. In this work we seek to advance the use of OpenCL for HPC on FPGAs in two ways. The first is by examining a core HPC application, Molecular Dynamics. The second is by examining a fundamental design pattern that we believe has not yet been described for OpenCL: passing data from a set of producer datapaths to a set of consumer datapaths, in particular, where the producers generate data non-uniformly. We evaluate several designs: single level versions in Verilog and in OpenCL, a two-level Verilog version with optimized arbiter, and several two-level OpenCL versions with different arbitration and hand-shaking mechanisms, including one with an embedded Verilog module. For the Verilog designs, we find that FPGAs retain their high-efficiency with a factor of 50× to 80× performance benefit over a single core. We also find that OpenCL may be competitive with HDLs for the straightline versions of the code, but that for designs with more complex arbitration and hand-shaking, relative performance is substantially diminished.**

I. Introduction

FPGAs have emerged as a cost-effective accelerator alternative in clouds [1]–[3] and clusters [4]–[8]. A well-known problem remains, however. Programming FPGAs for performance is still significantly more difficult than other accelerators, such as GPUs and Xeon Phi. The reasons for this are complex and have been discussed in detail in many places (e.g., [9]–[11]), but boil down to the fact that an FPGA program is not a program at all, but rather a hardware specification. Still, for many applications, there is sufficient commonality between these specifications for there to be promise that a High Level Language can be used for FPGA accelerators. Moreover, there is a realization that, whatever the difficulties, HLLs are an essential component in solving the programmability problem for FPGAs in HPC, even if compromises must be made.

OpenCL is currently the leading candidate to be the language that spans hardware and software specifications. Previous studies in use of OpenCL for FPGAs have compared performance with HDLs [12], implemented certain applications [13], and evaluated benchmark kernels and optimization methods [14], [15]. There is still much work to be done. First, the applications studied have mostly been simple code fragments. Second, the optimizations have focused on applying standard practices. Third, the performance obtained appears to be a fraction of what can be achieved with HDLs. Finally, we need to better understand the constructs that map well using OpenCL and those that do not.

In this work we seek to advance the use of OpenCL for HPC on FPGAs in two ways. The first is by examining a core HPC application, Molecular Dynamics (MD). In particular, we look at the range-limited force pipeline, the part of the computation responsible for 90% of the FLOPs in MD. Our goal is to determine whether the OpenCL version is competitive with an HDL version (e.g., [16]–[19]) and, if not, what modifications need to be made to make it so. The obvious benefits of an OpenCL implementation are portability and maintainability, especially with a target of integration into production systems such as NAMD [20], Amber [21], and OpenMM [22].

The second is to examine a basic design pattern that we believe has not yet been described for OpenCL: passing data from a set of producer pipelines to a set of consumer pipelines where the producers generate data non-uniformly. This is an example of a function (or pattern) where hardware and software solutions are both well understood, but are different from one another. The hardware created from the HLL specification is therefore unlikely to be resemble that which a hardware designer would use, potentially resulting in diminished performance.

The best hardware implementations for the main MD computation have two levels of pipelines (e.g., [17]–[19]). The first level computes the distance between pairs of particles. If the distance is less than a cutoff, which is true for about 15.5% of the pairs, then the particle pair is sent to one of a much smaller number of much more expensive force computation pipelines. We evaluate several designs: single-level versions in Verilog and in OpenCL (no filter), a two-level Verilog version with optimized arbiter, and several two-level OpenCL versions with different arbitration mechanisms including one with an embedded Verilog arbiter.

For the Verilog designs, we find that FPGAs retain their high-efficiency for this application with a factor of 50× to 80× performance benefit over a single core. For the OpenCL codes, we find that the straightline non-filtered versions are competitive with the Verilog. For designs with more complex arbitration and hand-shaking, relative performance is substan-

This work was supported in part by the National Science Foundation through Awards CNS-1405695 and #CCF-1618303/7960; by a grant from Microsoft; by a grant from Red Hat; by Altera through donated FPGAs, tools, and IP; and by Gidel through discounted FPGA boards and daughter cards.
Email: (jysheng|cyang90|sanaulah|herbordt)@bu.edu, vipin@silicontx.com

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

tially diminished. Moreover, the source of this diminished performance is not readily understandable. The significance of these findings is as follows. Since the largest benefit of using large, FPGA-centric, clusters for MD is reduced communication latency [3], using simple versions of the code is likely to be acceptable with respect to performance. And the improved programmability and maintainability from using OpenCL will be highly beneficial. On the other hand, it appears that substantial opportunities are being lost and that much work remains in optimizing channels and hand-shaking among datapaths.

II. OpenCL Basics

The power of FPGAs comes not just from the raw compute logic, but from the flexibility with which that logic can be configured. Basic blocks are synthesized directly into custom pipelined datapaths, which can then be replicated. Data can be streamed among data paths as well as to/from local (BRAM) and global memories. Compute structures are replicated to maximally use compute resources. This flexibility is also what makes FPGAs challenging to program. The goal of OpenCL for FPGAs is to facilitate development, especially for HPC tasks; the challenge is to retain performance with a (necessarily) restricted programmer's model.

OpenCL [23] was originally created to support heterogeneous parallel processing platforms such as CPUs with GPU accelerators. An OpenCL application consists of two parts: a host program written in C/C++ that handles control-intensive tasks, and a kernel which is compiled into a dedicated accelerator. The latter is to offload computational-intensive parts from the host. Much of the syntax follows GPU architecture, e.g., support for work groups (CUDA blocks) and work items (CUDA threads).

Recently Altera released an OpenCL SDK that supports FPGAs. Each computational task is work-item. A certain number of work-items can be grouped together into a work-group; this is determined by the host program at runtime. A work group is the unit for dispatching work to the computation units (CU) declared in the kernel. Work-items inside the same work-group can share their data through shared memory. But data sharing is prohibited among work-groups.

OpenCL has a three level memory architecture: global memory resides in on-board DRAM and is used for exchanging data between the host program and FPGA computational units; local memory uses the on-chip BRAMs for data exchange across multiple work-items belonging to the same work-group; and private memory, which is implemented as registers and used for passing data at run-time.

Due to the complexity of place and route for FPGAs, OpenCL compile time for FPGA is long. Hence, unlike CPU and GPU where compilation is done at runtime, users need to compile their design offline and load the generated programming file (.aocx) onto the FPGA before launching the host code and starting execution. OpenCL has two levels of parallelism on FPGAs (Figure 1). The first is pipelining; the second is replication of the kernels and having them run concurrently [24].

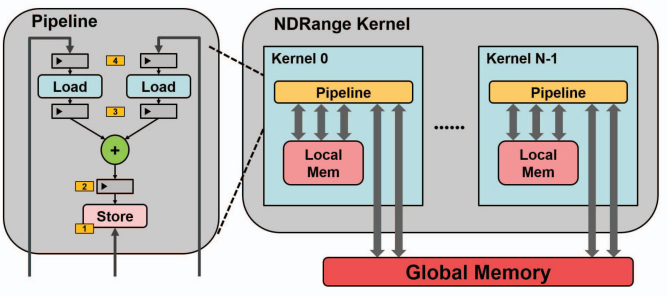

Fig. 1. Two levels of OpenCL parallelism (after [25])

OpenCL has two types of kernels, NDRange and task. An NDRange kernel is the default model which assigns multiple work-items to a single work-group. It gets performance improvements by having multiple work-items working in different stages of the pipeline at the same time [25]. The task kernel executes the entire job as a single work-item on a single kernel by implementing the outer loop inside that work-item.

Traditionally, data exchange between different kernels is done through global-memory. This incurs huge delays and requires the host program to coordinate. The Altera SDK provides an efficient mechanism, the channel, for passing data between kernels and for synchronizing kernels with high efficiency and low latency [26]. Channels are implemented as FIFOs residing in on-chip BRAM. Users can specify the channel (FIFO) depth with the command *depth()*. Each channel is assigned a unique id and can only be read by and written to a dedicated kernel.

III. Model Application: MD

```
1  Execute Timesteps UNTIL DONE
2   Integrate Motion
3   Compute Forces
4      Compute Bonded
5      Compute Non-Bonded
6         Compute Long-Range
7         Compute Range-Limited
8            FORALL Cells,
                  Compute forces on particles in cell
9              FORALL Particles in Cell,
                     Compute force on reference particle
10               FORALL Other particles in cell neighborhood,
11                  Compute distance between particles in pair
12                  IF (distance < cut-off) Compute force
```

Fig. 2. Pseudocode for Molecular Dynamics. Emphasis is on the range limited part of the non-bonded force calculation, steps 7-12.

There are many reasons to use Molecular Dynamics (MD) as a model application, including its centrality in High Performance Computing (HPC), the breadth of computations it

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

contains, the challenges in strong scaling, and the particularly good fit for FPGA clusters. MD is an iterative application of Newtonian mechanics to ensembles of atoms and molecules. We show high-level pseudocode in Figure 2. The outer loop is the timestep (Line 1), each of which consists of two phases, motion integration (Line 2) and force computation (Lines 3-12). The forces consist of bonded (hydrogen bond, covalent – Line 4) and non-bonded (LJ, Coulomb – Lines 5-14) terms:

$$\mathbf{F}^{total} = F^{bond} + F^{angle} + F^{torsion} + F^{HBond} + F^{non-bonded} \quad (1)$$

Motion integration and bonded force computation are generally $< 1\%$ of the computation and not discussed further [27]. The non-bonded forces are all based on the electrostatic interactions between particle pairs. The LJ force for particle i can be expressed as:

$$\mathbf{F}_i^{LJ} = \sum_{j \neq i} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 12 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{14} - 6 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{8} \right\} \mathbf{r}_{ji} \quad (2)$$

where the ϵ_{ab} and σ_{ab} are parameters related to the types of particles, i.e. particle i is type a and particle j is type b. The Coulombic force can be expressed as:

$$\mathbf{F}_i^{C} = q_i \sum_{j \neq i} \left(\frac{q_j}{|\mathbf{r}_{ji}|^3} \right) \mathbf{r}_{ji} \quad (3)$$

For computational convenience, the non-bonded forces are generally partitioned into long-range (Line 6) and range-limited (Lines 7-12) components, where the latter has a cut-off radius r_c. These components are optimized separately. The long-range force computation is based on a 3D FFT and discussed elsewhere [28]–[30]; we concentrate here on the range-limited interaction (RL). RL between (all) pairs of particles i and j can be approximated with

$$\frac{\mathbf{F}_{ji}^{short}}{\mathbf{r_{ji}}} = A_{ab} r_{ji}^{-14} + B_{ab} r_{ji}^{-8} + QQ_{ab} (r_{ji}^{-3} + \frac{g'_a(r)}{r}). \quad (4)$$

where A_{ab}, B_{ab}, and QQ_{ab} are distance-independent coefficient look-up tables indexed with atom types a and b [17]–[19].

The cut-off radius r_c is used in two ways to optimize RL: cell lists and neighbor lists (see Figure 3). With cell lists, the simulation space is partitioned into cubes with edge-length equal to r_c. Non-zero forces on the *reference particle* P can then only be applied by other particles in its *home cell* and in the 26 neighboring cells (the 3x3x3 *cell neighborhood*). We refer the second particle of the pair as the *partner particle*. With neighbor lists, P has associated with it a list of exactly those partner particles within r_c. We now compare these methods.

- **Efficiency.** Neighbor lists are by construction 100% efficient: only those particle pairs with non-zero mutual force are evaluated. Cell lists as just defined are 15.5% efficient with that number being the ratio of the volumes of the cut-off sphere and the 27-cell neighborhood.
- **Storage.** With cell lists, each particle is stored in a single cell's list. With neighbor lists, each particle is typically stored in 400-1000 neighbor lists.

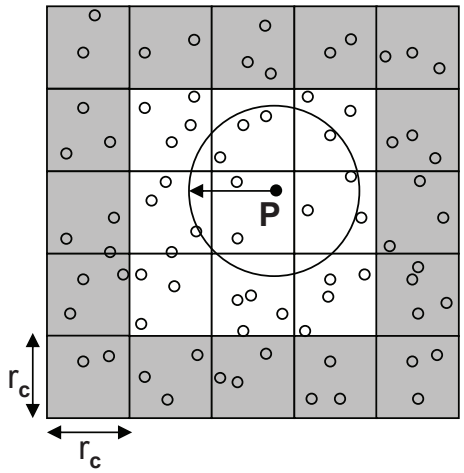

Fig. 3. Shown is part of the simulation space about particle P [17]. Its two dimensional *cell neighborhood* is shown in white; cells have edge size equal to the cut-off radius. The cut-off circle is shown; particles within the circle are in P's neighbor list and will be passed by the filter.

- **List creation complexity.** Computing the contents of each cell requires only one pass through the particle array. Computing the contents of each neighbor list requires, naively, that each particle be examined with respect to every other particle: the distance between them is then computed and thresholded. In practice, however, it makes sense to first compute cell lists anyway. Then the neighbor lists can be computed using only the particles in each reference particle's cell neighborhood.

IV. Application Design

We begin describing our design with the innermost loop (Lines 11-12): The pairwise force pipeline is shown in Figure 4. This is the bulk of the logic for the RL interaction; the rest is used to route particle data. Note that the first two computations, adjusting for periodic boundary conditions and obtaining r^2, can be done in fixed point. In that case, r^2 must be converted to floating point before being combined with the coefficients (on the smoothing side) and divided (on the Coulomb + LJ side). The conversion at the output is similar. As is usual with FPGA designs, high performance is obtained first by creating a custom pipelined datapath (as just shown) and then by replicating the datapath as many times as possible.

In previous work [17] we proposed a further optimization. We observed that the r^2 calculation, indicated with the gray dashed line, is exactly what is needed to compute the neighbor list. Putting this another way, while the r^2 calculation (Line 11) needs to be performed for every particle pair in the cell list, the balance of the computation (Line 12) only needs to be performed if $r^2 < {r_c}^2$. If follows that each force pipeline will only be active for 15.5% cycles and thus can be shared among several distance calculation *filters* (8 turns out to be a good number). The hardware module to be replicated is now a single force pipeline shared by 8 filters.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

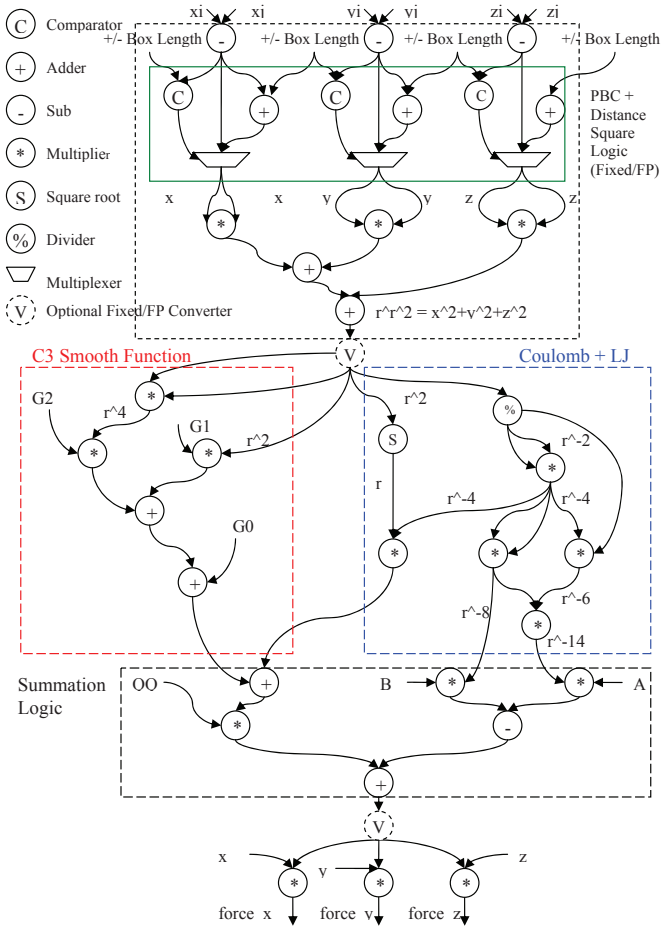

Fig. 4. Shown is the detailed datapath for the direct computation. For hybrid integer/floating point, the computation is the same, but with data before the first conversion and after the second being integer. Design is from [16]; for an alternative based on table look-up with interpolation see [19].

We now describe two further issues critical to obtaining good performance: taking advantage of Newton's 3rd Law (N3L) and load balancing. For illustration, we first describe the ApoA1 benchmark and its processing. ApoA1 has 92,224 particles, a bounding box of $108\mathring{A} \times 108\mathring{A} \times 78\mathring{A}$, and a cut-off radius of 12Å. Our simulation space thus has $9 \times 9 \times 7$ cells with an average of 175 particles per cell with a uniform distribution [18]. For ease of implementation and memory access, all filters share the same reference particle but calculate the distance with different neighboring particles. For a given reference particle from the current cell (Line 9), the partner particles within its cut-off radius must reside in the 26 neighboring cells plus the home cell.

Since by N3L particle interaction is mutual, the force calculated on the reference particle is the same for the partner particle. It is therefore standard to update both particles after each calculation and to use a geometric partition to ensure that each particle is only updated once. Figure 5 shows a simple partition scheme in 2-D. For any particles in the home cell, we only need to examine a semicircle in 2-D, i.e., cells 1-5 plus the home cell. Hence, in 3-D, the number of cells we need to examine is reduced from 27 to 18.

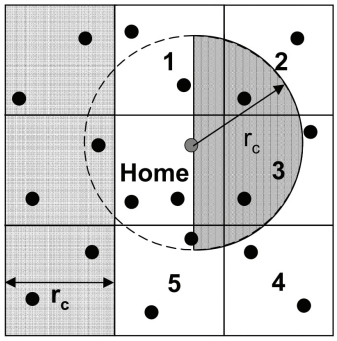

Fig. 5. Shown is a simple way to partition the computation to account for Newtons's 3rd Law. For any reference particle, the only forces that must be computed are in the forward hemisphere. Thus the shaded cells can be ignored. More efficient and complex solutions include [31], [32].

The load balancing issue arises because we would like each filter to process a similar number of particles within the cut-off and so simplify arbitration and force pipeline load. The obvious mapping of cells to filters leads to load *imbalance*: particles in different cells have different probablities of falling within the cut-off range. For example, particles in the home cell are more likely to be in range than particles from cells 4 and 5. To improve distribution, we divide each cell into 12 slices with roughly 15 particles/slice. This makes a total of $18 \times 12 = 216$ slices; each of the 8 filters is responsible for particles from 27 of the 216 slices. The calculation is analogous for more multiple force pipelines. We map slices to filters based on an empirical model. We find that the pass percentage for all filters lies within the range of 14% to 17% with high probability.

V. Implementation

In this section, we present the details of our OpenCL implementations for the RL calculation. The inner loop is shown in Figure 6 with the most important parts as follows: the 8 independent filter kernels, which share the same functionality, but with different output channels; a single force-evaluation pipeline kernel that calculates the non-bonded electrostatic force between a pair of particles if they are within the cut-off radius; and 8 channels that bridge the connection between 8 data generators (filters) and a single consumer (force pipeline). In previous work this structure was replicated eight times (for an Altera Stratix-III [17]). With more recent technology, the Altera Arria-10, it can be replicated 20 times or more.

In our case study we examine execution of particles in a single cell and surrounding cell set. This is easily extended to full RL with logic that transfers cell sets in the background. In the testbench, the host program initializes the data, launches the kernels, collects the data on completion, and collects statistics. For the designs with filters, each filter kernel inputs pairs of particle coordinates and parameters, computes the distance, and compares the result with a threshold. If they are within the cut-off radius, then the distance and the parameters are sent to channel dedicated to this filter using blocking channel

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

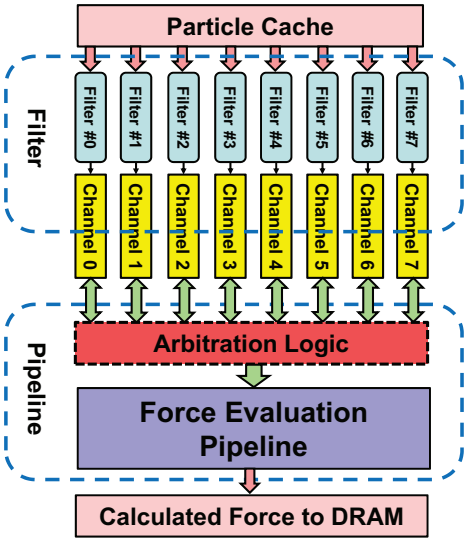

Fig. 6. Basic architecture of filter-based RL processing.

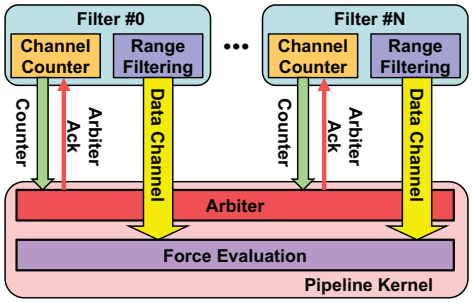

Fig. 7. Channel handshake mechanism between filter and pipeline

write command **write_channel_altera(channel_id, data)**. If the channel is full, the write command keeps executing until the data is successfully written to it. The force evaluation pipeline (consumer) computes the force without interruption.

We now describe the arbitration mechanisms. Previous OpenCL work (e.g., [25], [33], [34]) has used channels to pass data among different kernels. Our work differs in having multiple producers and a single consumer.

Recall that the filters all receive inputs nearly continuously, that they collectively output (on average) one at a time, but that on any given cycle from 0 to 8 filters may be generating outputs and thus input to the force pipeline. The problem that the arbiter must solve is as follows: for maximal efficiency the force pipeline must receive a new particle pair on every cycle for which one is available (from any filter), while at the same time keeping the channel buffers small and applying minimal back pressure to the filter inputs (due to full input buffers).

In previous work [18] we implemented in VHDL the following near-optimal algorithm. If zero or one filter have data, then the action is trivial. If multiple channels have data, then they are serviced round-robin, unless one is nearly full, in which case that one has priority. In OpenCL, the representation is less direct and so it is not obvious which arbiter can be implemented nor what the resulting efficiency will be. We propose several candidate designs. We present results in the next section.

1. No filter. As a baseline, we implement a design with no filters (Figure 4. That is, all particle pairs are processed whether their distance is below the cut-off threshold or not.

2. Round Robin. Altera OpenCL SDK provides two sets of channel read&write operations, blocking and non-blocking. Blocking operations execute until they succeeds. The non-blocking return a boolean value indicating success or failure and then move on to the next instruction [26]. We use the non-blocking read command **read_channel_nb_altera(channel_id, status)** to successively try the different channels. On failure, the arbiter moves on to read the next channel. On success, the arbiter jumps out of the loop and performs the force evaluation. This design has no guarantee of avoiding channel congestion or of keeping the force pipeline maximally active.

3. Explicit Handshaking. To prevent stalls we want to read from the channel that is almost full. However, the channel function provided by Altera OpenCL SDK returns no information regarding the channel's full status. We introduce a channel handshaking mechanism that provides channel status information to the consumer and thus enables full channels to be given priority. To make this work, we add two extra channels of depth 0 (shown in Figure 7): a counter channel and a single bit acknowledgment.

Each filter has its own channel counter. When a particle pair passes the filter and is written to the channel, the counter increments and its value is sent to the arbiter using the same blocking write command. When the arbiter has received counter values from all the 8 filters, it chooses the channel from which to read. The arbiter writes a 1 to the ack channel of the selected channel and 0 to others. The filter receiving the ack decrements its counter. In this way, a handshake between multiple filters and pipeline is performed. As the channel depth for the handshake is 0, the handshake is done in each cycle. Thus, synchronization across all kernels is achieved as described in [34].

The details of the logic are as follows. The arbiter channels are grouped into three sets, those that are close to full, those not full but have data, and those that are empty. Priority is given in the obvious order. Within each set channels are selected round-robin.

4. Explicit Handshaking with Custom Logic. The previous design is implemented with an unrolled for loop, which may be inefficient. Here we implemented the design in Verilog and incorporate it into OpenCL as a custom library. The algorithm is shown in Figure 8. The input and output are both an 8-bit priority mask with the selection result set high. The mask maintains arbitration status from the previous iteration. We rotate the input mask to find the next high bit and return that bit's location. Lines 1-5 update the input mask to push the arbitration to the left side. Lines 6-7 find the right-most 1

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

1 Get priority mask based on channel count
2 If internal mask is 10000000, skip to Step 6
3 Internal mask shift left 1 bit and subtract 1
4 Not Step 3
5 And Step 3 with input mask (Thus remove the previously selected ones)
6 2's complement on Step 5 (or input mask if jumped from Step 2)
7 And Step 5 and Step 6, this is the output result
8 Keep the current result as the internal mask

Fig. 8. Pseudocode for arbiter #4.

bit indicating the selected channel. In Verilog, lines 2-7 are combinational logic executed within a single cycle.

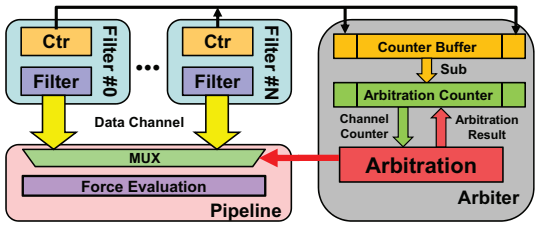

Fig. 9. Logic for design #5 with no handshaking.

5. Distributed control. In the OpenCL version of *explicit handshaking*, the pipeline needs to read eight channel counters and write eight acknowledgments. Since channels are implemented as FIFOs, these operations incur non-negligible delays. Also, the pipeline needs to wait for the counter values while the filters are calculating distances, and the filters need to wait for the ack signal during arbitration. We propose a new design which keeps track of channel data without handshaking (see Figure 9).

As in the previous design, the arbiter still reads from 8 filter channels. But this time the arbiter itself keeps track of the number of items inside each channel. The filter now proceeds without waiting; also, the filter only needs to record the total number of items that have passed. The arbiter now keeps track of the number of times each channel has been selected; the number of items in each channel is calculated by subtracting the two counters. By keeping all channel-related counters inside a single kernel (arbiter), data exchange is no longer necessary. In this design the arbiter makes decisions faster reducing the chance of stalls.

6., 7. Verilog without and with filtering. For another control we implement designs 1 and 3 directly in Verilog (corresponding the Figures 4 [16] and 6 [17], respectively).

VI. Results

We have implemented our design on a Nallatech 385A FPGA card, which hosts an Altera Arria 10 FPGA (10AX115N3F40E2SG), has an 8GB DDR3 on-card memory, and Altera OpenCL SDK 16.0 support. Performance depends on five things (see Table I).

1) Number of times the design can be replicated on the FPGA. This depends on the efficiency of resource utilization and is shown under column *Reps*.
2) Are filters used? Ideally (e.g., in design 7) one particle pair can be issued to each filter each cycle. If there are filters, then we multiply the number in Reps by eight (our current best design); this is shown under column *Filters*.
3) How efficient is the logic? That is, for what fraction of cycles is the force pipeline active? Note that bubbles/stalls can occur either because a full filter stalls the particle fetch or because there are no particle pairs available that have passed the filter.
4) How efficient is the timing? For example, the Verilog designs have been constructed so that there is a continuous pipeline from particle fetch through the filters and force pipeline. With the OpenCL designs we have no such control. For example, the arbitration implementation could take multiple cycles and so create extra bubles in the force pipeline.
5) How efficient is the lay-out? The result here is the operating frequency shown in column *freq*. The designs have so far only undergone basic optimization: using standard EDA methods the Verilog designs could probably be improved by 50% or more. Improving OpenCL operating frequency is less well understood.

We make a number of general observations.

- For the baseline Verilog design (#7), which is the state-of-the-art, the expected run-time for an entire iteration of ApoA1 is around 20ms. This demonstrates scaling over implementations using older technologies (e.g., [18]) and retains a 50× to 80× benefit over running on a single CPU core.
- For the simplest OpenCL design (#1), the straight-line datapath with no filtering and no arbitration, performance is within 30% of the corresponding Verilog design (#6).
- While filtering is an effective design method in Verilog, resulting in a 6× performance benefit, it is much less so in OpenCL. There the performance benefit is only 25%.

Some specific observations related to arbitration in OpenCL are as follows.

- For the OpenCL designs, the most successful is #5 with no handshaking and the arbiter keeping track of the counts. The hybrid design was particularly disappointing with substantial overhead in integrating HDL with OpenCL.
- Handshaking in OpenCL leads to a dramatic decrease in performance, slowing down execution by a factor of 10× to 20×. Clearly a great deal of care, not to mention additional rounds of optimization, are required to make this approach viable.
- For the baseline *no filter* designs (1 & 6), the Verilog design needs 2/3rds the resources (40 versus 24) of the OpenCL.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE I
COLLECTED RESULTS

Design		LEs	FFs	BRAMs	DSPs	Reps	Filters	Freq (mhz)	Time (ms)
1 OpenCL No filter	Total	79124(9%)	160932(15%)	304(11%)	89(6%)	24	24	151.3	3.9
	Kernel	5704(1%)	14092(1%)	80(3%)	22(1%)				
2 OpenCL Round-Robin	Total	114515(13%)	230994(21%)	740(27%)	139(9%)	4	32	246.1	32.0
	Kernel	40655(5%)	79698(7%)	484(18%)	72(5%)				
3 OpenCL Standard	Total	114488(13%)	218123(20%)	727(27%)	138(9%)	4	32	245.2	22.1
	Kernel	40308(5%)	66507(6%)	471(17%)	71(5%)				
4 OpenCL Hybrid	Total	162033(19%)	337235(31%)	1156(43%)	138(9%)	2	16	241.1	17.0
	Kernel	87853(10%)	185619(17%)	900(33%)	71(5%)				
5 OpenCL Distrib.	Total	118733(14%)	256017(24%)	784(29%)	123(8%)	4	32	256.5	3.1
	Kernel	39709(5%)	92788(9%)	470(17%)	56(4%)				
6 Verilog No filter	Total	486(1%)	795(1%)	46(2%)	29(2%)	40	40	194.8	3.0
7 Verilog Standard	Total	654(1%)	1236(1%)	53(2%)	85(6%)	14	112	195.2	0.5

Resource utilization for the various designs as enumerated in Section V. Resources are for a single instance of the design. *Total* includes overhead logic for OpenCL, *Kernel* does not. *Reps* is the number of times that the instance can be replicated and still fit on chip and is equal to the number of force pipelines. *Filters* is the total number of filters and gives the upper bound on throughput; this is nearly achieved for the Verilog designs (6 & 7). *Time* refers to time to process a single entire cell using a single replication.

- For the filter-based designs (2, 3, 4, 5, &7), the OpenCL versions are dramatically less efficient, with the Verilog design fitting from 3.5× to 7× more logic.

There remains to be discussed logical efficiency. Are there cycles when one design, but not another, has no data from the filters to feed to the force pipeline? We find that in general the answer is no: all designs keep the force pipeline full nearly all the time. This is because we have intentionally overprovisioned the filter bank by passing (on average) slightly more particle pairs than can be consumed. The filter channels (FIFOs) thus gradually fill as the computation proceeds guaranteeing the availability of data.

VII. CONCLUSION

With the widespread availability of FPGAs for HPC, general use depends on programmability. The most likely tool flow for the next few years will include OpenCL. We have examined a critical application that should be a good fit: the range-limited component of the MD force calculation. This has the benefit of both being a high-value target (in terms of compute cycles globally), but also containing an interesting design pattern. We believe that neither of these has been explored beyond immediate and unexamined compilation.

We have implemented the computation datapaths, but more significantly, we have explored various methods of coupling producer and consumer datapaths. We make several observations. One is that for "straightline" code, OpenCL appears to be within a factor of two of HDL in terms of resource utilization. But with increased control complexity, such as with handshaking, comes drastic inefficiency.

We now conjecture about integration into production MD on a large-scale FPGA-based system. We have found that communication dominates [3]. Having less compute capability (due to OpenCL inefficiencies) could not only be acceptable, but completely hidden. In any case, we need to better understand the inefficiency in the OpenCL handshake designs. In a way these issues mirror the more general problems and opportunities of OpenCL versus HDL: OpenCL designs are much easier create, while HDL designs are easier to understand and optimize.

REFERENCES

[1] A. Caulfield, et al., "A cloud-scale acceleration architecture," in *49th IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.

[2] A. Putnam, et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. Int. Symp. on Computer Architecture*, 2014, pp. 13–24.

[3] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt, "HPC on FPGA Clouds and Clusters: 3D FFTs and Implications for Molecular Dynamics," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2017.

[4] R. Sass, et al., "Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2007, pp. 127–138.

[5] T. Guneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp, "Cryptanalysis with COPACABANA," *IEEE Trans. on Computers*, vol. 57, no. 11, 2008.

[6] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang, "Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects," in *IEEE High Perf. Extreme Computing Conf.*, 2016.

[7] J. Sheng, C. Yang, and M. Herbordt, "Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study," in *Proc. Highly Efficient and Reconfigurable Technologies*, 2015.

[8] J. Sheng, Q. Xiong, C. Yang, and M. Herbordt, "Collective Communication on FPGA Clusters with Static Scheduling," *Computer Architecture News*, vol. 44, no. 4, 2016.

[9] M. Gokhale and P. Graham, *Reconfigurable Computing: Accelerating Computation with Field Programmable Gate Arrays*. Springer, 2005.

[10] T. VanCourt and M. Herbordt, "LAMP: A tool suite for families of FPGA-based application accelerators," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2005.

[11] ——, "Families of FPGA-based algorithms for approximate string matching," in *Proc. Int. Conf. on Application Specific Systems, Architectures, and Processors*, 2004, pp. 354–364.

[12] K. Hill, S. Cracium, A. George, and H. Lam, "Comparative Analysis of OpenCL vs. HDL with Image-Processing Kernels on Stratix-V FPGA," in *Proc. Int. Conf. on Application Specific Systems, Architectures, and Processors*, 2015, pp. 189–193.

[13] J. Zhang and J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Networks," in *Proc. ACM Symp. on Field Programmable Gate Arrays*, 2017.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

[14] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. Tahoori, "Energy Efficient Scientific Computing on FPGAs using OpenCL," in *Proc. ACM Symp. on Field Programmable Gate Arrays*, 2017.
[15] H. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating aand Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis – Supercomputing*, 2016.
[16] M. Chiu, M. Herbordt, and M. Langhammer, "Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems," in *Proceedings High Performance Reconfigurable Technology and Applications*, 2008.
[17] M. Chiu and M. Herbordt, "Efficient filtering for molecular dynamics simulations," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2009.
[18] ——, "Molecular dynamics simulations on high performance reconfigurable computing systems," *ACM Trans. Reconfigurable Tech. and Sys.*, vol. 3, no. 4, pp. 1–37, 2010.
[19] M. Chiu, M. Khan, and M. Herbordt, "Efficient calculation of pairwise nonbonded forces," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2011.
[20] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *J. Comp. Chemistry*, vol. 26, pp. 1781–1802, 2005.
[21] D. Case, T. Cheatham III, T. Darden, H. Gohlke, R. Luo, K. Merz, Jr., A. Onufriev, C. Simmerling, B. Wang, and R. Woods, "The Amber biomolecular simulation programs," *J. Comp. Chemistry*, vol. 26, pp. 1668–1688, 2005.
[22] P. Eastman and V. Pande, "OpenMM: A Hardware Independent Framework for Molecular Simulations," *Computing in Science and Engineering*, vol. 12, no. 4, pp. 34–39, 2010.
[23] Khronos OpenCL Working Group, *The OpenCL Specification*, 2012.
[24] Altera. Altera SDK for OpenCL: Best Practices Guide. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
[25] Z. Wang, B. He, and W. Zhang, "A study of data partitioning on OpenCL-based FPGAs," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2015, pp. 1–8.
[26] Altera. Intel FPGA SDK for OpenCL: Programming Guide. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
[27] Q. Xiong and M. Herbordt, "Bonded Force Computations on FPGAs," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2017.
[28] B. Sukhwani and M. Herbordt, "Acceleration of a Production Rigid Molecule Docking Code," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2008, pp. 341–346.
[29] J. Sheng, B. Humphries, H. Zhang, and M. Herbordt, "Design of 3D FFTs with FPGA Clusters," in *IEEE High Perf. Extreme Computing Conf.*, 2014.
[30] B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M. Herbordt, "3D FFT on a Single FPGA," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2014.
[31] K. Bowers, R. Dror, and D. Shaw, "Zonal methods for the parallel execution of range-limited n-body simulations," *J. of Computational Physics*, vol. 221, no. 1, pp. 303–329, 2007.
[32] M. Snir, "A note on N-body computations with cutoffs," *Theory of Computing Systems*, vol. 37, pp. 295–318, 2004.
[33] Z. Wang, J. Paul, B. He, and W. Zhang, "Multikernel Data Partitioning With Channel on OpenCL-Based FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–13, 2017.
[34] S. Settle, "High-performance dynamic programming on FPGAs with OpenCL," in *High Performance Embedded Computing Workshop*, 2013, pp. 1–6.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE