Molecular Simulations with Hardware Accelerators: A Portable Interface Definition for FPGA Supported Acceleration

Eric Stahlberg[1], Michael Babst[2], Daryl Popig[3], Debbi Ryle[3], Mohamed Taher[2], Kelly Anderson[4], and Thomas Steinke[5]

[1] Ohio Supercomputer Center, 1224 Kinnear Road, Columbus, OH, 43212, USA, eas@osc.edu
[2] DSPlogic, 13017 Wisteria Dr., STE 420, Germantown, MD, 20874, USA, mike.babst@dsplogic.com
[3] Accelerated Data Concepts, LLC, 3916 Marsha Drive, Columbus, OH, 43207, USA
{dpopig, dryle}@acceleratedata.com
[4] The Procter and Gamble Company, 11810 East Miami Road, Cincinnati, Ohio,USA anderson.kl.1@pg.com
[5] Zuse Institute Berlin (ZIB), Dept. Computer Science, Takustrasse. 7, Berlin, Germany, steinke@zib.de

Abstract. Recent widespread interest in the use of configurable hardware accelerators has brought to light the need for a portable application programmer interface (API) to achieve widespread adoption. Recent activities defining a candidate common generic API for field programmable gate arrays have facilitated the definition of an application specific API for accelerating molecular dynamics programs. Using the LAMMPS application as a prototype implementation platform, both the general FPGA API and application specific molecular dynamics API are presented with preliminary results confirming the viability of the portability of both a general and functionally specific API across reconfigurable hardware and development environments.

Keywords: hardware accelerator, FPGA, API, molecular dynamics.

1 Introduction

Interest has risen significantly in the past two years in seeking new approaches for accelerating scientific applications. This is particularly evident in several efforts focused on speeding up molecular dynamics applications using technologies ranging from multiple core CPUs, graphical processing units, cell processors and field programmable gate arrays (FPGAs). This paper extends the base of research needed to fully exploit field programmable gate arrays in this capacity, including presentation and discussion of application programmer interfaces (APIs) needed to facilitate the use of FPGAs in a hybrid accelerated computing environment.

1.1 Challenges of Force Field Based Molecular Simulations in Practice

Molecular dynamics is a method for investigating the behavior of chemical systems at a molecular and atomic level. The ability of molecular dynamics to describe, explain, and differentiate among complex interactions at a molecular level involving systems of thousands and even millions of atoms has lead to the development of several approaches to molecular dynamics, including Carr-Parinello molecular dynamics (CPMD) [1], all atom classical force field molecular dynamics [2], and increasingly hybrid approaches involving mixed quantum and classical molecular dynamics [3] to name a few. This ability of molecular dynamics to describe complex interactions and three-dimensional shape, particularly at a protein level, and the computational challenges in achieving solutions for very large peta-scale systems is summarized in a recent article by Agarwal and Alam. Their investigations, conducted at Oak Ridge National Lab, identified existing limitations in current applications and programming models for highly parallel systems needed to model these extremely large systems, with specific discussion of new approaches needed to incorporate emerging FPGA and Graphical Processing Unit (GPUs) application accelerators [4].

For large molecular systems with 10,000 of particles and more, still approximate and experimentally tuned interaction potentials must be used to reduce the computational demands to an acceptable limit. Furthermore, for large in-silico screening “experiments” the time-to-answer is very important so that desirable turn-around times for simulations are over night runs even if this implies again further approximations in the simulation method.

At its foundation, molecular dynamics involves the evaluation of forces on atoms within a molecular system, employing Newtonian equations of motion to compute time-dependent displacement the atoms, integrating over time to advance the system to future states. A commonly accepted method for integrating the molecular system in time is the *Velocity Verlet* algorithm [5], which is defined as:

$$r(t+\Delta t) = r(t) + v(t)\Delta t + \frac{1}{2m} F(t)(\Delta t)^2 \quad (1)$$

$$v(t+\Delta t) = v(t) + \frac{\Delta t}{2m}\left[F(t) + F(t+\Delta t)\right] \quad (2)$$

where equation (1) describes the update of the position in space *r*, and equation (2) the update of the velocity *v* at the next time step $t+\Delta t$ with *F* and *m* being the force acting on the particle and *m* its mass. Usually, the time step Δt is in the order of femtoseconds leading high computational demands for simulating the dynamics of large-scale particle systems, e.g. complexes of biomolecular compounds in solution or nano-scale probes of inorganic solid-state materials. Excepting the computation of the actual forces, the Velocity Verlet algorithm update involves a computational complexity of *O(N)* for the system, with *N* being the number of atoms.

With a broad acceptance and relatively low computational cost of the Velocity Verlet algorithm, it is understandable that evaluation of the forces among atoms within interacting molecules is where approaches differentiate themselves. Limiting the scope of these methods to those already proven viable for FPGA acceleration, the focus of this paper is to examine the nature, feasibility and implications of standard application programmer interfaces by proposing a common API for classical force field molecular dynamics implementations on field programmable gate arrays.

1.2 Forces and Energies from Classical Force Fields with FPGAs: Related Work

The evaluation of accurate system energies and forces on each atom is composed of several individual contributions, each receiving investigative attention from the reconfigurable computing research community. These investigations have been critical in establishing the context for a portable molecular dynamics API for reconfigurable computing, highlighting both capabilities and limitations in these solutions. Limiting the scope of these methods to those already proven viable for FPGA acceleration, the focus of this paper is to examine the nature, feasibility and implications of standard application programmer interfaces by proposing a common API for classical for force field molecular dynamics implementations on field programmable gate arrays

Short-range inter-atomic interactions: Exploiting specialized hardware accelerators for computational challenging problems are today's common approach to meet the user's requirements. In the area of molecular dynamics simulation the most prominent example is the evolution of the MDGRAPE system [6] with latest version breaking the PetaFlop/s barrier recently [7].

Kindratenko and Pointer [8] reported their effort in porting the molecular dynamics code NAMD to the SRC-6 platform. On the SRC-6 system, they achieved an overall speedup of 3 compared to a 2.8 GHz x86-CPU which is remarkable for a highly optimized production code.

For FPGAs, early investigations by Scrofano and Prasanna [9] showed favorable results for a tuned Lennard-Jones potential and force computation with reconfigurable hardware. While the investigations focused on only a limited problem description, the results nevertheless showed the capabilities for FPGAs to accelerate computation of short-range interactions using reconfigurable computing hardware available at the time.

Long-range Coulomb interactions: Cordova and Smith [10, 11] have done some research in improving the performance of large-scale molecular dynamics calculations using reconfigurable supercomputers. After analyzing FFT, they found that for the parallel systems, the communication costs become significant. As a result, they suggested implementing the FFT kernel in single or closely coupled FPGAs. This will reduce the overhead of communications since data does not need to be transferred to other processors.

Hemmert and Underwood [12] have analyzed the implementation of double-precision floating-point FFT on FPGAs. They have explored three different implementations for the Fast Fourier Transform (FFT) on FPGAs. The algorithms are compared in terms of sustained performance and memory requirements for various FFT sizes and FPGA sizes. The results show that the low FPGA clock rate and high latency floating-point units make current FPGAs inferior to microprocessors for a single, small FFT. For large FFTs, FPGAs show a trend toward dramatically outperforming microprocessors.

Recent investigations by Alam et al. [13] of the Particle Mesh Ewald (PME) summation successfully illustrated speed-up of the AMBER molecular dynamics applications using SRC FPGA computing systems. Using the SRC system with its integrated Carte environment, the investigations clearly demonstrated speedup for molecular dynamics applications when exploiting deep pipelining, concurrent execution, and data streaming to accentuate capabilities of FPGAs. The investigations exploited FFTs in the evaluation of the Ewald summation to achieve high performance and speedup on reconfigurable computing hardware.

2. Design Principles for Application APIs for Reconfigurable Computing

Development of common APIs is not a simple task. Either a group must have market dominance whereby de facto APIs are developed as a result of unilateral decisions, or, one must develop an API with the input and support of a community in response to their evolving needs. With many molecular dynamics applications in use within industry and academia, and no single dominant organization, the latter scenario must be pursued to develop such an API. The proposed molecular dynamics API for reconfigurable computing has been developed within the applications library (APPLIB) working group for OpenFPGA, the result of a sponsored project to investigate and demonstrate the feasibility of such an API.

In a recent special issue of Compute devoted to reconfigurable computing, Herbordt et. al. have proposed several guidelines for achieving high-performance FPGA-based computing [14]. These methods are briefly summarized as follows:

- Use an algorithm optimal for FPGAs
- Use a computing mode appropriate for FPGAs
- Use appropriate FPGA structures
- Live with Amdahl's Law
- Hide latency of independent functions
- Use rate-matching to remove bottlenecks
- Take advantage of FPGA-specific hardware
- Use appropriate arithmetic precision
- Use appropriate arithmetic mode
- Minimize use of high-cost arithmetic operations
- Create families of applications, not point solutions
- Scale application for maximal use on FPGA hardware

The design and specification of this API has taken into account this key insight for achieving high-performance results on reconfigurable computing hardware. In addition, the proposed API incorporates additional design criteria into the specification to ease incorporation into a broad range of applications and support across multiple technologies. These additional criteria include:

- High portability across technology and application platforms
- Resource discovery
- Ease of adoption for software developers
- Existence of verifiable reference implementation
- Extensibility of the API to adapt to new innovations in the reconfigurable computing and accelerator computing space

An overview of the key features for the API serves to illustrate the support for the design criteria required to achieve high-performance for FPGA devices while concurrently providing for ease of incorporation and retrofitting of existing applications.

Selectable precision – As pointed out by Herbordt et al., the ability to select computing precision is key to achieving high-performance in reconfigurable computing. The importance of this element was further emphasized by Dongarra et al. in their SC06 presentation examining the use of single precision computation in a double precision environment [15]. *The API developed for molecular dynamics achieves selectable precision by specifically separating the precision of the data representation for input and output elements from the preferred precision for the selected method.* The selectable precision provides the programmer the ability, if desired, to optimize the balance between precision required of the implementation and parallelism enabled by the computing environment.

Asynchronous operation – Highlighted as a key design criteria for achieving FPGA performance, and further supported by Underwood et al. in a presentation at SC06 [16], asynchronous operation is inherent to enable hiding latency in the underlying implementation. The developed API incorporates a method for asynchronous execution accessible to the application programmer in a manner familiar to implementers of MPI.

Easy specification of implementation methods – Rather than define a specific API for each implementation method and combination, the developed API incorporates a selectable list for identifying a given method of interest for transforming the input to output. Implemented in a hierarchical manner, the list supports specification

of defaults and delayed binding preferences such as 'fastest available' for a dynamic computing environment. This approach to *delayed binding* also enables high-level application portability. An early list of implementation methods is listed in Appendix B.

Common Algorithmic Specification – Reinforcing the point for developing application families, specific API methods and structures are defined for the most commonly implemented and needed algorithms for molecular dynamics. It is expected that this set will expand as new algorithms become commonly accepted.

Layering – The developed API abstraction supports layering of methods within the fabric of the API. Layering is an important construct, enabling early and long-term adoption as computing environments evolve. Support for 'tiers' of abstraction is required to enable early adopters to integrate the API at a low-level into their applications with minimal disruption while also supporting growing composite operations that become increasingly possible with advances in hardware.

The cumulative criteria are approached in the following manner evident in the developed molecular dynamics API. It is important to note that the functionality behind the proposed API is not yet fully implemented and supported, but that it will increasingly be implemented with the cumulative contributions of the community.

3. Portable Application Programmer Interfaces for Reconfigurable Computing

The proposed APIs for evaluating FFT contributions and non-bonding interactions are based on the OpenFPGA GenAPI definition for FPGAs. A standard GenAPI interface will increase adoption of reconfigurable computing technologies. Currently, all vendors of reconfigurable hardware provide customized APIs for interfacing to their hardware and software. All software vendors are then required to port their tools to each hardware vendors APIs by writing additional middleware. Should all hardware vendors support the OpenFPGA GenAPI, then reconfigurable platforms would automatically be supported by any compliant software vendor's tools.

Based on an earlier OpenFPGA draft document[1] we have implemented the following Generalized API for using FPGA in molecular dynamics simulations. The summary list of functions and their purpose are listed below. (Details of the full specification are available in the Appendix A and at the OpenFPGA website, www.openfpga.org.) The features of this generic API are quickly summarized as follows:

- Allocation of necessary resources and initialization of FPGA device and its infrastructure
- Managing of FPGA algorithms (bitstream files) and its mapping to FPGA devices.
- Allocation of aligned memory segments for optimal data transfers from host memory to FPGA attached memory banks and vice versa.
- Explicit interface to (blocked) data transfer functions.

Our proposed GenAPI extends the API draft of the OpenFPGA GENAPI Working Group by

i) introducing the concept of an *Algorithm Registry* and *metadata* about algorithm requirements, and
ii) support for *encoding* and *querying* the FPGA hardware configuration

for higher level application specific APIs, see Figure 0 for illustration of the relationship of these components and the internal control flow.

The basic idea for introducing an algorithm registry is to optimize the configuration time and to support multiple algorithms on the FPGA within one application. If supported by the vendor hardware, the registry interface can improve the time to reconfigure a FPGA device during runtime by pre-fetching the bitstream into a specific memory location with an optimal upload path to the FPGA device. Another advantage can be seen if applications require multiple and different algorithms to be used.

Properties of the FPGA infrastructure such as the type of the FPGA device itself, type and size of attached memory banks, average configuration time, or quality of the connection link to a host system have to be processed by applications to make decisions for options of data mapping or multiple run-time reconfigurations. These properties can be queried by an application code. For the description of these FPGA infrastructure properties we propose a XML based schema [17].

[1] OpenFPGA GENAPI Working Group document prepared by Stefan Möhl, Mitrionics.

The Generic FPGA interface abstraction is a key element to achieve easy and rapid underlying portability of the Molecular Dynamics OpenFPGA API. Designed to be supportable in most FPGA environments, the OpenFPGA GenAPI was used in the development and validation of early elements for the proposed molecular dynamics API.

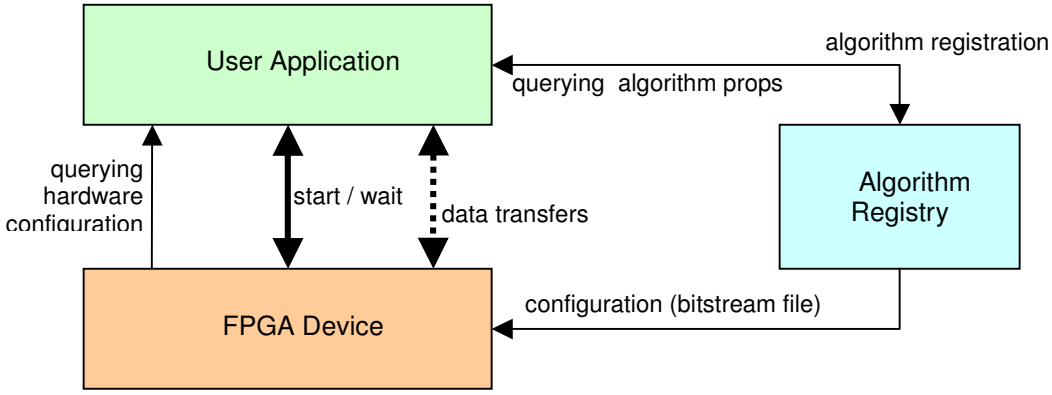

Figure 0: Relationship of the user's application, the FPGA device infrastructure and the algorithm registry to support faster reconfiguration and for multiple bitstream support. Major control and information flow as realized in the proposed OpenFPGA GenAPI are shown.

3.2 Function Definitions

For a general interface between a high level application and the low level API for specific hardware, we propose adding a layer of abstraction that can be coded in a way to be vendor specific for low level hardware API calls but very general to any application.The application can be written in Java, C, C++, C# or any language that can share the standard API library for FPGAs. We propose that this library is written in C since most of the APIs that are currently on the market are using the C language.To differentiate the specific hardware, the API library would use a vendor table that is used to distinguish to what hardware the generic interface is talking to. The vendor ID will be assigned and tracked by OpenFPGA.

Table 1 gives an overview of the GenAPI, a more detailed description can be found in Appendix A.

3.3 Advantages and Limitations of the Proposed GenAPI

The proposed GenAPI provides a vendor neutral interface for the communication of the host system to the FPGA device. Thus, it assumes the co-processor model as system architecture: The FPGA device usually with attached control logic for communication and memory moduls is connected to the host system consisting of one or more general purpose CPU cores and its memory subsystem. This architectural model can be mapped on most of the today's general purpose computers equipped with FPGAs[2]. The GenAPI represents a subset of functions to be found in a convergent set of vendor-specific interfaces of SGI, Cray, DSPlogic and Nallatech, for example. From these interfaces a common and necessary minimal subset of functionality was derived.

A newly introduced feature is the *algorithm registry*. Although a support for multiple FPGAs exists only in a few of the today's systems, it is intended to provide a basic functionality for using multiple FPGAs in a compute node within one application. Hence, it makes sense to support the applicability of more than one algorithmic kernel to be used on different FPGAs per node. The algorithm registry keeps track of registered algorithms (FPGA bitstream files) complemented with metadata describing the algorithm[3].

[2] Note, that in the future the re-configurable functionality of data processing devices can be a integral part of a general purpose CPU device, and part of the functionality of the proposed API has to be handled by the operating system than.

[3] An XML schema for algorithmic description is developed and pending release.

Table 1: Preliminary General API for Reconfigurable Computing Devices

`ofpga_init`	The function opens the specified FPGA device(s) on a node and performs low-level system initialization
`ofpga_device_prop`	The function returns hardware configuration data of the FPGA device, e.g. types, numbers and sizes of attached memory modules, bitstream load bandwidth, and so on
`ofpga_close`	Data and FPGA device resources are cleared.
`ofpga_register_algorithm`	This function registers the bitstream files, i.e. an algorithm in a registry per node. If supported, this pre-loads the bitstream into a dedicated memory location near the FPGA chip. Multiple calls should be supported if we want to support deep scaling per node or reconfiguring the FPGA during runtime within one application.
`ofpga_load_algorithm`	The function loads the registered algorithm into the FPGA, i.e. the FPGA is configured
`ofpga_registered_malloc`	The function allocates memory with an optimal alignment (e.g. on memory page boundaries) for optimal data transfer between host memory and FPGA
`ofpga_run`	The function starts the algorithm asynchronously on the configured FPGA and returns control immediately
`ofpga_status`	The function provides status information about the running algorithm as much as it supported by the platform.
`ofpga_bwait`	The function call, a blocked wait, returns if the run on the FPGA is finished.
`ofpga_read_register`	The function transfers one word from a FPGA "register" to host memory. Register is an algorithm dependent interface to one word wide parameters.
`ofpga_write_register`	The function writes one word from host to a FPGA register destination.
`ofpga_send`	This function sends a data block to the FPGA device.
`ofpga_receive`	This function receives a data block from a FPGA device.

To match application requirements with effectively available hardware resources (e.g. size of attached memory) the properties of the FPGA hardware has to be described as well and can be queried by means of a function [17].

Maintaining a narrow scope for definition of initial common functionality, the currently defined GenAPI has deferred common specification for the following features:

- The explicit access to DMA capabilities.
- A streaming model for data transfer.
- The automatic partitioning of data, i.e., distributing user data across FPGA internal BRAM or to FPGA attached memory modules within one compute node is application specific.
- An automatic partitioning of multiple kernels on FPGAs. The programmer has to keep track which kernel (bitstream file) is loaded on which FPGA device.
- Automatic wide and deep scaling[4].
- Support for direct FPGA to FPGA communication.

Consequently, a functionally specific API must be defined in anticipation of future extensions to the initial general API.

[4] These terms are used by SGI in its RASC library definition to annotate

4. Development of a Reconfigurable Computing API Family for Molecular Dynamics

Validation of the GenAPI presented in the prior section was accomplished in the development of a functionally specific API for molecular dynamics specifically for Lennard-Jones interactions and FFT evaluations. The following reconfigurable computing API for molecular dynamics illustrates the utility of the GenAPI as well as highlight key elements for effective acceleration of a molecular dynamics application. While preliminary, the molecular dynamics API for reconfigurable computing has been developed within the applications library working group (APPLIB) for OpenFPGA, driven with input from industrial users of the applications

The current status of the API for molecular dynamics functions on reconfigurable computing devices is in its bootstrapping phase. Based on well-known examples in the literature [6, 8, 9, 18 - 20] and our own experiences, an initial representative set of functionality has been implemented:

- **Lennard-Jones forces and energies** as a common model for short-range *non-bonding interactions*, and
- **3D** and **1D FFT** used by the *Particle-Particle Particle-Mesh* (PPPM) method for long-range Coulomb interactions.

The decision for this initial selected was rationalized by runtime execution profiles obtained with LAMMPS [21] using representative input data sets provided by a project sponsor. For typical large benchmark cases, 75 percent of the runtime is spent in the computation of Lennard-Jones (LJ) pair wise forces and energies in a single CPU LAMMPS run on a Cray XD1 node (2.2 GHz Opteron, 2 GB RAM per node). Other program sections where substantial parts of the runtime were spent are the FFT for long-range Coulomb interactions and the building of the neighbor list.

The cumulative criteria are approached in the following manner evident in the developed molecular dynamics API. It is important to note that the functionality behind the functionality behind the proposed API is not yet fully implemented nor supported at this time. However, the API, even in the current early state, serves as a template to guide further implementations incorporating reconfigurable hardware. A quick summary of the preliminary API is presented below, with full details of the molecular dynamics family API available in Appendix B.

4.1 API for Short-Range Non-bonding (Lennard-Jones) and Long-Range Coulomb Interactions

As outlined in section 2, the API for non-bonding interactions will support multiple precision variants of an LJ implementation. One of the strengths of FPGAs is that they can support implementations with varying bit allocations for data representation tuned to optimize data transfer, emphasize precision, or yield best performance. This extends to the use of non-IEEE data specifications which can yield effective results even if all underlying operations do not employ full IEEE floating point data representations.

Examples for precision types are (subject to availability of implementation):

- *LJ_612_FastestAvailable*: fastest available algorithm for computing the answer
- *LJ_612_BestPrecision*: best precision algorithm available for computing the answer
- *LJ_612_ALL_SingleExact*: IEEE single precision exact implementation for both force and energies
- *LJ_612_ALL_DoubleExact*: IEEE double precision exact implementation for both force and energies
- *LJ_612_ALL_SingleInterp1*: IEEE single precision linear interpolation for both force and energies
- *LJ_612_ALL_DoubleInterp1* : IEEE double precision linear interpolation for both force and energies
- *LJ_612_ALL_SingleBestFixed*: best fixed point implementation for at least single precision accuracy for both force and energies
- *LJ_612_ALL_DoubleBestFixed*: best fixed point implementation for at least double precision accuracy for both force and energies

Ideally, to minimize necessary data transfers between host memory and FPGA, top-level functions that call compute-intensive functions has to be migrated to the accelerator device. For a pure LJ interaction model, this means the integrator for the equation of motion, e.g. Velocity Verlet is best migrated to the FPGA. Unfortunately, for algorithms requiring heavily floating-point operations, this is still a difficult task on today's available FPGA devices. Nonetheless, the molecular dynamics API anticipates future improvements in individual FPGA capabilities and further extensions to the GenAPI, and follows a top-down decomposition of

the computational workflow in each functional family. At the next level, the computation of total forces acting on each particle and the total non-bonding energy are defined. This function usually calls the computational kernel in any LJ simulation – the function which computes a pairwise energy and forces for a given particle pair, and thus the bottom-level function in the hierarchy is also considered.

Table 2: Overview of the preliminary OpenFPGA Molecular Dynamics API

`ofpgaMD_LJ_init`	The function initializes required resources for the specified FPGA device. For a defined type of precision and problem size given by the maximum number of particles and maximum number of particle pairs, respectively, a matching to available hardware resources is attempted internally, and a data distribution topology is return in the *LJ_handle*.
`ofpgaMD_LJ_setparam`	This function loads the LJ parameters ε and σ for *n_types* particle pair types in a certain precision format to a storage location compatible with the algorithm already registered.
`ofpgaMD_LJ_set_masses`	This function loads the masses for each of the particle types in a certain precision format to a storage location compatible with the algorithm already registered.
`ofpgaMD_LJ_reg_buffer1D`	This registers buffers for coordinates and forces in vector representation; if not yet allocated buffer space is made available.
`ofpgaMD_LJ_reg_buffer2D`	This function registers buffers for coordinates and forces in array representation; if not yet allocated buffer space is made available.
`ofpgaMD_LJ_run`	This function starts the calculation of Lennard-Jones energies and/or forces for data allocated.
`ofpgaMD_LJ_VVrun`	This function starts the Velocity Verlet integration with the parameter and data supplied using the LJ potential for the particle interaction.
`ofpgaMD_LJ_status`	This is a non-blocking call to test the status of a running LJ computation
`ofpgaMD_LJ_wait`	This is a blocking call waiting for completion of a LJ computation
`ofpga_1dfft_config`	The function sets the FFT size and FFT direction (forward or reverse) and direction of the FFT.
`ofpga_3dfft_config`	This function configures the FPGA for computing a 3 dimensional FFT

For an implementation of the Velocity Verlet algorithm the intention of using the propose API is illustrated with the following pseudo code:

```
// Schematic velocity Verlet code with GenAPI and MD_API
// initialize FPGA infrastructure
fpga = ofpga_init(device_id,msglvl)

// retrieve hardware configuration information
fpga_prop = ofpga_device_prop(device_id,msglvl)

// select appropriate algorithm
If ( fpga_prop.maxSRAM >= 4 MB ) then
    alg_prop ← xml_string_propierties
    alg_id = ofpga_register_algorithm(fpga,bitfile,alg_prop,msglvl)
else   return SORRY   // bad luck

// load the bitstream file into the FPGA
status = ofpga_load_algorithm(fpga,alg_id,msglvl)

// now lets check we have the resources we need for the LJ part
LJ_handle = ofpgaMD_init (device_id,alg_id,max_part,max_pairs,
                          LJ_612_FastestAvailable, IEEE32, IEEE32, msglvl)
If ( LJ_handle == NULL ) return SORRY

// set the LJ parameters
status = ofpgaMD_LJ_setparam(LJ_handle,IEEE32,n_types,*epsilon,*sigma,
                             return_code,msglvl)
```

```
// same for particle masses
status = ofpgaMD_LJ_set_masses(LJ_handle,IEEE32,n_types,*Masses,
                               return_code,msglvl)

// now allocated the buffer for the coordinates, velocities, forces
status = ofpgaMD_LJ_reg_buffer2D(LJ_handle,IEEE32,n_part,n_pairs,*XYZ,
                                 *type,*nb,*Vxyz,*Fxyz,return_code,msglvl)

// fill initial data, relax, and thermalize
*XYZ = random(); *Vxyz = Boltzmann(temperature); *Fxyz = random();
relax(); thermalize();

// --- the big loop over a couple of MD steps
While ( I_am != tired ) {

   // if no error, we have everything to start n sweeps on FPGA
   status = ofpgaMD_LJ_VVrun(LJ_handle,n_sweeps,delta_t,cutoff_rs,
                             cutoff_tbl,(energy&&forces),return_code,
                             msglvl)

   // wait to be completed; another thread can do something useful...
   status = ofpgaMD_LJ_wait(LJ_handle,return_code,msglvl )

   // calculate properties and e.g. check for conservation of energy on CPU
   energy = ofpga_read_register( fpga,e_register, msglvl
   status = analyse_data (*XYZ, *Vxyz, energy)
   // print stuff out
}
// well done
  status = ofpga_close ( fpga, msglvl )
// END
```

The FFT API considers two possible program partitions, at the *fft_1d* and the *fft_3d* function. The API for the *fft_1d* function consists of the GenAPI, plus one additional function, *ofpga_1dfft_config*. The purpose of this function is to set the FFT size and FFT direction (forward or reverse) and direction of the FFT. In order to compute an FFT, the user would invoke the following pseudo code using GenAPI commands:

```
//Program to compute FFT
//Initialize FPGA
fpga   = ofpga_init ( device_id, msglvl )
// Load FFT Algorithm
status = ofpga_load_algorithm ( fpga, fft1d_algorithm_id, msglvl )

// Configure FFT length and direction
status = ofpga_1dfft_config (fpga, size, direction, msglvl)

// Send data message to FPGA (may include multiple FFTs)
status = ofpga_send ( fpga, data,  dest_id, length, msglvl )

WHILE message not received
// Check to see if FFT is complete
status = ofpga_receive ( fpga, data, src_id, length, msglvl )

// INSERT OTHER PROCESSING TASKS HERE IF DESIRED

END

// Close FPGA
status = ofpga_close ( fpga, msglvl )
```

If required by a particular hardware platform, the *ofpga_registered_malloc* function should be used to create a memory space for FPGA I/O.

5. Early API Implementation and Validation

The aim of this paper is to present a viable common and portable API for molecular dynamics applications employing reconfigurable computing devices as application accelerators. A reference implementation of the extended GenAPI for a first set of FPGA platforms is in progress. A stub implementation with a test program written in C is available on request[5].

The primary goal of the investigations were to determine the overall feasibility of the APIs for application development across several development environments. In fulfillment of this aim, development environments examined and explored included Mitrion-C, Dime-C, Nallatech H101 platforms, Cray XD1 supercomputers with Virtex4 LX100 FPGAs, and DRC FPGA plugins. While still in progress, the initial results are very promising for a feasible and portable API. The authors of this paper have exercised the API with an initial choice of the LAMMPS molecular dynamics application, examining the potential for the API for accelerating this application. The evaluations have exercised both the FFT and Lennard-Jones elements of the molecular dynamics API as well as the functions of the vendor neutral GenAPI.

Convertibility of the LAMMPS application to acceleration with reconfigurable computing hardware was demonstrated using existing FFT routines readily converted to the API specification presented. Within the LAMMPS application, the Particle-Particle Particle-Mesh (PPPM) method uses the FFT in computations. These early results, conducted on a Cray XD1 supercomputer with Virtex4 LX-100 FPGAs were successful in demonstrating the viability of the API for introduction into the LAMMPS application. Not unexpectedly, performance of the 1D FFT was not adequate to deliver compelling performance improvement and necessitated the development of the 3D FFT API. Early efforts in determining the viability of the 3D FFT API are very positive with a determination of ample space on the FPGA device for necessary transformations and cores and projected speed-up for the FFT evaluation within the application in excess 10-fold for the average case.

The authors have also implemented the molecular force and move functions with a test program into a single Xilinx Virtex-4 LX-100 FPGA being clocked at 100 MHz on a Nallatech H101-PCIXM board placed in an AMD Opteron server. The interface between the AMD CPU and FPGA was a standard PCIX running 133MHz. Development tools were Nallatech DimeC for the compiler and DimeTalk for building the support devices such as memory interfaces, on-chip routers, and the PCIX interface. DimeC was used to create hardware VHDL. The Xilinx ISE 8.2 tool was then used to do the synthesis, place and route and to generate the bitmap. The C code is arranged so that API DMA calls were added to send data from the CPU to the FPGA at n step intervals. After the n step iterations on the FPGA, the data block was read back into the CPU for further analysis.

Some observed challenges require discussion. The first challenge to note is that of placing most of the molecular dynamics calculation into the FPGA. This includes the Lennard-Jones potentials and the Verlet algorithm together with support functions to handle the pair-wise interaction potentials. The FPGA must be of sufficient size to support tiers of implementation. The second challenge was the amount of memory required for the coordinates, velocities, and forces of each of the atoms. As the number of atoms grows, the data array grows well beyond the available onboard FPGA BLOCK RAM and forces the design to use large blocks of external but directly coupled SDRAM. As the system size increases, memory requirements will increase and these issues will need to be resolved such that the common molecular dynamics API is not restricted by the amount of external or internal RAM available. Ideally, the molecular dynamics API should be defined to be transparent to the underlying layer that supports the manufacturer's native API calls, employing portable general options. In this case, the Nallatech FUSE API calls used for the DMA data transfers from the CPU to the H101-PCIXM board. Not unexpectedly, a third challenge is in keeping hardware multiplies to a minimum in order to fit the algorithm onto the LX-100 FPGA device.

Preliminary results look very promising and demonstrate that a general molecular dynamics API is feasible for FPGAs. The necessary calculations do fit on a single Xilinx Virtex-4 LX-100 FPGA using a commercially available tool set. An increase in speed is expected where the limiting factor will be dependent on the volume of data transfers that are required for a given implementation. Specific and comparable results across a variety of reconfigurable systems is anticipated as implementations are completed and validated for the presented general and molecular dynamics specific API.

[5] contact OpenFPGA or Thomas Steinke at steinke@zib.de

6. Conclusions and Outlook

This paper presents a candidate for a common API for molecular dynamics calculations in environments with reconfigurable computing hardware. While still in early stage development, the API has been proven viable for implementation and development across several reconfigurable computing systems and application development environments. The goal for increased performance of the API in practice across multiple platforms remains still to be demonstrated, although early indications are very positive for meaningful acceleration in the two dominant areas of interest based on the conclusions of prior work and our own experience.

Further work is needed in several areas, not the least of which is expanding the successful demonstration of the API across more reconfigurable computing application development and execution environments. Optimizations within each implementation are expected to result in high-performance accelerated results portably supported across several environments. A well selected set of standard molecular dynamics benchmarks will be extremely valuable to confirm the performance of the multitude of reconfigurable computing development and execution environments.

The proposed APIs have implications beyond the reconfigurable computing application domain and transcend into the broader use of accelerators. The abstraction, with its decoupling of accelerator specifics from the method request is equally viable for use with specialized Graphical Processor Units, Cell processor implementations and more creative hybrid combinations including multiple CPU cores.

Acknowledgments. The authors would like to acknowledge the efforts of the OpenFPGA GenAPI working group for its extremely valuable efforts in developing the basis for the general FPGA API. The authors would further acknowledge the valuable contributions of the pioneers in the use of reconfigurable computing for molecular dynamics applications, without which the proposed API would not have been possible.

References

1. Carr, R. and Parrinello M.,: *Unified approach for molecular dynamics and density-functional theory*, Phys. Rev. Letters, **55** (1985), 2471
2. Rappe, A., Casewit, C., Colwell, K., Goddard, W., *UFF, a Full Periodic Table Force Field for Molecular Mechanics and Molecular Dynamics Simulations*, J. Am. Chemical Society, **114** (1992), 10024-10035
3. Singh U. and Kollman, P.*: A combined ab initio quantum mechanical and molecular mechanical method for carrying out simulations of complex molecular systems*: Applications to the CH3Cl + Cl- exchange reaction and gas phase protonation of polyethers, Journal of Computational Chemistry, **7**(1986), 718-730
4. Agarwal, P., and Alam, S.: *Biomolecular simulations on petascale: promises and challenges*, Journal of Physics: Conference Series, **46** (2006), 327-333
5. Swope, W. C., Andersen, H. C., Berens, P. H., and Wilson, K. R., J. Chem. Phys. **76** (1982) 637
6. Taiji, M., Narumi, T., Ohno, Y., Futatsugi, N., Suenaga, A., Takeda, N., Konagaya, A.: *Protein explorer: A petaflop special-purpose computer system for molecular dynamics simulations*. in: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, New York, ACM Press, 2003
7. In various news forums it was reported that the final ProteinExplorer installation (aka MDGRAPE3) with a PetaFlops performance was finished in July, 2006.
8. Kindratenko, V., Pointer, D.: *A case study in porting a production scientific supercomputing application to a reconfigurable computer*, In: Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), April 2006, Napa, California
9. Scrofano, R., and Prasanna, V.K.: *Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware*. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), June 2004
10. Cordova, L., Alam, S., Smith, M., and Vetter, J.: *A High Performance Programming Model for Large-Scale Molecular Dynamics Calculations on Reconfigurable Supercomputers*. 9th Annual Workshop on High Performance Embedded Computing (HPEC), 2005.
11. Smith, M., Vetter, J., and Alam, S.: *Scientific Computing Beyond CPUs: PGA implementations of common scientific kernels*. 2005 MAPLD International Conference.
12. Hemmert, K. S., Underwood, K.D.: *An Analysis of the Double-Precision Floating-Point FFT on FPGAs*. FCCM 2005: 171-180
13. Alam, S., Agarwal, P., Smith, M., Vetter, J., Caliga, D.: *Using FPGA Devices to Accelerate Biomolecular Simulations*. Computer, **30**(3), March 2007, 66-73
14. Herbordt, M., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., DiSabello, D.: *Achieving High Performance with FPGA-Based Computing*. Computer, **30** (3), March 2007, 50-57
15. Langou, J., Langou, J., Luszczek, P., Kurzuk, J., Buttari, A., Dongarra, J.; *Exploiting the Performance of 32-Bit Floating Point Arithmetic in Obtaining 64-Bit Accuracy*. SC'06, Tampa, 2006; Kurzak, J., Dongarra, J.: *Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the CELL Processor*. Tech. Report UT-CS-06-580

16. Underwood, K.D., Hemmert, K.S., Ulmer, C.: *Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications*. SC'06, Tampa, 2006
17. Peick, M, and Steinke, Th., FPGA hardware description schema, May 2007
18. Azizi, N., Kuon, I., Egier, A., Daribiha, A., Chow, P.: *Reconfigurable Molecular Dynamics Simulator*. IEEE Intl. Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2004
19. Kuon, I., Azizi, N., Darabiha, A., Egier, A., and Chow, P.: *FPGA-Based Supercomputing: An Implementation for Molecular Dynamics*, Poster, ACM Symposium on Field Programmable Gate Arrays (FPGA), Feb. 2004
20. Scrofano, R., Prasanna, V.K.: *Preliminary Investigation of Advanced Electrostatics in Molecular Dynamics on Reconfigurable Computers*. Supercomputing 2006, November, 2006
21. Plimpton, S.J.: *Fast Parallel Algorithms for Short-Range Molecular Dynamics*. J. Comp. Phys. **117** (1995) 1-19, LAMMPS, http://lammps.sandia.gov/

Appendix A: OpenFPGA GenAPI Function Definitions

This Appendix includes the definition of functions of the GenAPI required for the proposed API for molecular simulations.

Note: Most of the proposed function has the argument *msglvl* to control the verbosity for printouts inside the function. In the tables of the argument description, this argument is omitted for simplicity.

Function: **`ofpga_init`**
Usage: `fpga = ofpga_init ( device_id, msglvl )`

Parameter	I/O	Description
fpga	O	FPGA handle structure or NULL pointer
device_id	I	Vendor & platform specific ID

Description: The function opens the specified FPGA device(s) on a node and performs low-level system initialization.

Function: **`ofpga_device_prop`**
Usage: `fpga_prop = ofpga_device_prop ( device_id, msglvl )`

Parameter	I/O	Description
fpga_prop	O	FPGA property structure or NULL pointer
device_id	I	Vendor & platform specific ID

Description: The function returns hardware configuration data of the FPGA device, e.g. types, numbers and sizes of attached memory modules, bitstream load bandwidth, and so on.

Function: **`ofpga_close`**
Usage: `status = ofpga_close ( fpga, msglvl )`

Parameter	I/O	Description
fpga	I	FPGA handle structure
status	O	Status = OFPGA_OK | error code

Description: Data and FPGA device resources are cleared.

Function: **`ofpga_register_algorithm`**
Usage:
```
algorithm_id = ofpga_register_algorithm ( fpga,
                    bitstream_file, algorithm_properties, msglvl )
```

Parameter	I/O	Description
Fpga	I	FPGA handle structure
bitstream_file	I	path to bitstream file
algorithm_properties	I	metadata of the algorithm including precision information
algorithm_id	O	unique ID of the registered algorithm

Description: This function registers the bitstream files, i.e. an algorithm in a registry per node. If supported, this pre-loads the bitstream into a dedicated memory location near the FPGA chip. Multiple calls should be supported if we want to support deep scaling per node or reconfiguring the FPGA during runtime within one application.

Function: **`ofpga_load_algorithm`**
Usage: `status = ofpga_load_algorithm ( fpga, algorithm_id, msglvl )`

Parameter	I/O	Description
Fpga	I	FPGA handle structure
algorithm_id	I	ID of an registered algorithm
Status	O	OFPGA_OK | error_code

Description: The function loads the registered algorithm into the FPGA, i.e. the FPGA is configured.

Function: **`ofpga_registered_malloc`**
Usage: `buff_ptr = ofpga_registered_malloc ( fpga, memid, size, mode, msglvl )`

Parameter	I/O	Description
fpga	I	FPGA handle structure
memid	I	FPGA memorz bank identifier
size	I	size of the memory space to be allocated
mode	I	access mode for the memory area, e.g. read-only, write-only or read-and-write for FPGA device
returns	O	valid memory adress for FPGA data transfer or NULL pointer in error case

Description: The function allocates memory and registers the allocated buffer for a given memorz bank identifier. The memory aligned to be optimal for data transfers, e.g. on memory page boundaries.

Function: **`ofpga_run`**
Usage: `status = ofpga_run ( fpga, msglvl )`

Parameter	I/O	Description
fpga	I	FPGA handle structure
Status	O	OFPGA_OK | err_code

Description: The function starts the algorithm asynchronously on the configured FPGA, i. e., it should return immediately.

Function: **`ofpga_status`**
Usage: `fpga_stat = ofpga_status ( fpga, msglvl )`

Parameter	I/O	Description
Fpga	I	FPGA handle structure
fpga_stat	O	structure with status information

Description: The function provides status information about the running algorithm as much as it supported by the platform.

Function: **`ofpga_bwait`**
Usage: `status = ofpga_bwait ( fpga, msglvl )`

Parameter	I/O	Description
Fpga	I	FPGA handle structure
Status	O	OPFGA_OK | error_code

Description: The function call, a blocked wait, returns if the run on the FPGA is finished.

Function: **`ofpga_read_register`**
Usage: `value = ofpga_read_register ( fpga, register_id, msglvl )`

Parameter	I/O	Description
Fpga	I	FPGA handle structure
register_id	I	identifies register
value	O	returned value

Description: The function transfers one word from a FPGA "register" to host memory. Register is an algorithm dependent interface to one word wide parameters.

Function: **`ofpga_write_register`**
Usage: `status = ofpga_write_register ( fpga, register_id, value, msglvl )`

Parameter	I/O	Description
fpga	I	FPGA handle structure
register_id	I	identifies destination register
Value	I	word to be written
Status	O	OPFGA_OK | error_code

Description: The function writes one word from host to a FPGA register destination.

Function: **`ofpga_send`**
Usage: `status = ofpga_send ( fpga, data, dest_id, length, msglvl )`

Parameter	I/O	Description
Fpga	I	FPGA handle structure
data	I	buffer address of source data
dest_id	I	data destination identifier for FPGA
length	I	number of bytes to be sent
status	O	OPFGA_OK | error_code

Description: This function sends a data block to the FPGA device.

Function: **`ofpga_receive`**
Usage: `status = ofpga_receive ( fpga, data, src_id, length, msglvl )`

Parameter	I/O	Description
Fpga	I	FPGA handle structure
Data	I	destination address
src_id	I	data source identifier for FPGA
Length	I	number of bytes sent
Status	O	OPFGA_OK | error_code

Description: This function receives a data block from a FPGA device.

Appendix B: OpenFPGA MD API Function Definitions

Note: Most of the proposed function has the argument *msglvl* to control the verbosity for printouts inside the function. In the tables of the argument description, this argument is omitted for simplicity.

Function: **`ofpgaMD_LJ_init`**
Usage:
```
LJ_handle = ofpgaMD_init ( device_id, algorithm_id, max_part,
            max_pairs,   algoritm_precision,   input_precision,
            output_precision, msglvl )
```

Parameter	I/O	Description
device_id	I	FPGA device identifier
algorithm_id	I	Algorithm identifier obtained from registry
max_part	I	Maximum number of particles to be processed
max_pairs	I	Maximum number of particle pairs to be processed
algorithm_precision	I	Requested precision tag of the algorithm
input_recision	I	Type of precision for input data
output_precision	I	Type of precision for output data
LJ_handle	O	Handle to structure carrying LJ status information

Description: The function initializes required resources for the specified FPGA device for the calculation of Lennard-Jones properties. For a defined type of precision and problem size given by the maximum number of particles and maximum number of particle pairs, respectively, a matching to available hardware resources is attempted internally, and a data distribution topology is return in the *LJ_handle*.

Function: **`ofpgaMD_LJ_setparam`**
Usage:
```
status = ofpgaMD_LJ_setparam (LJ_handle, precision, n_types,
epsilon, sigma, return_code, msglvl )
```

Parameter	I/O	Description
LJ_handle	I	Handel from LJ_init
precision	I	Precision type of LJ constants data to be submitted
n_types	I	Number of particle pair types
epsilon	I	LJ ε constants for n_types
sigma	I	LJ σ constants for n_types
return_code	O	Return information structure
status	O	OK or ERROR

Description: This function loads the LJ parameters ε and σ for *n_types* particle pair types in a certain precision format to a storage location compatible with the algorithm already registered.

Function: **`ofpgaMD_LJ_set_masses`**
Usage:
```
status = ofpgaMD_LJ_set_masses (LJ_handle, precision, n_types,
ptr_Mass, return_code, msglvl )
```

Parameter	I/O	Description
LJ_handle	I	Handel from LJ_init
Precision	I	Precision type to be submitted
n_types	I	Number of particle types
ptr_Mass	I	Pointer to particle masses
return_code	O	Return information structure
status	O	OK or ERROR

Description: This function loads the masses for each of the particle types in a certain precision format to a storage location compatible with the algorithm already registered.

Function: **ofpgaMD_LJ_reg_buffer1D**

Usage:
```
status = ofpgaMD_LJ_reg_buffer1D (LJ_handle, precision,
n_particles, n_pairs, ptr_X, ptr_Y, ptr_Z, ptr_type, ptr_nb,
ptr_Vx, ptr_Vy, ptr_Vz, ptr_Fx, ptr_Fy, ptr_Fz, return_code,
msglvl )
```

Parameter	I/O	Description
LJ_handle	I	Handle from LJ_init
Precision	I	Precision type of LJ constants data to be submitted
N_particles	I	Number of particles
N_pairs	I	Number of particle pairs, upper limit estimate
ptr_{X, Y, Z}	I	Pointer to vectors of X, Y, and Z coordinates
ptr_{Vx, Vy, Vz}	I	Pointer to vectors of velocities in X, Y, and Z direction
ptr_{Fx, Fy, Fz}	I	Pointer to vectors of forces in X, Y, and Z direction
ptr_type	I	Pointer to particle types
ptr_nb	I	Pointer to neighbour list structure
return_code	O	Return information structure
Status	O	OK or ERROR

Description: This registers buffers for coordinates and forces in vector represenation; if not yet allocated buffer space is made available.

Function: **ofpgaMD_LJ_reg_buffer2D**

Usage:
```
status = ofpgaMD_LJ_reg_buffer2D (LJ_handle, precision,
      n_particles, n_pairs, ptr_XYZ, ptr_type, ptr_nb,
      ptr_Vxyz, ptr_Fxyz, return_code, msglvl )
```

Parameter	I/O	Description
LJ_handle	I	Handle from LJ_init
Precision	I	Precision type of LJ constants data to be submitted
n_particles	I	Number of particles
n_pairs	I	Number of particle pairs, upper limit estimate
ptr_XYZ	I	Pointer to array of coordinates
ptr_Vxyz	I	Pointer to array of velocities
ptr_Fxyz	I	Pointer to array of forces
ptr_type	I	Pointer to particle types
ptr_nb	I	Pointer to neighbour list structure
return_code	O	Return information structure
Status	O	OK or ERROR

Description: This function registers buffers for coordinates and forces in array representation; if not yet allocated buffer space is made available.

Function: **ofpgaMD_LJ_run**

Usage:
```
status = ofpgaMD_LJ_run (LJ_handle, cutoff_rs, cutoff_tbl,
comp_flag, return_code, msglvl )
```

Parameter	I/O	Description
LJ_handle	I	Handle from LJ_init
cutoff_rs	I	Cut off value in real space
cutoff_tbl	I	Cut off value for neighbour table
comp_flag	I	Bitfield with flags to indicate which properties has to be calculated: energy, forces, virial, …
return_code	O	Return information structure
status	O	OK or ERROR

Description: This function starts the calculation of Lennard-Jones energies and/or forces for data allocated.

Function: **`ofpgaMD_LJ_status`**
Usage: `status = ofpgaMD_LJ_status (LJ_handle, return_code, msglvl )`

Parameter	I/O	Description
LJ_handle	I	Handle from LJ_init
return_code	O	Return information structure
status	O	OK or ERROR

Description: This is a non-blocking call to test the status of a running LJ computation.

Function: **`ofpgaMD_LJ_wait`**
Usage: `status = ofpgaMD_LJ_wait (LJ_handle, return_code, msglvl )`

Parameter	I/O	Description
LJ_handle	I	Handle from LJ_init
return_code	O	Return information structure
status	O	OK or ERROR

Description: This is a blocking call waiting for completion of a LJ computation.

Function: **`ofpgaMD_LJ_VVrun`**
Usage:
```
status = ofpgaMD_LJ_VVrun (LJ_handle, n_sweeps, delta_t,
      cutoff_rs, cutoff_tbl,  comp_flag, return_code, msglvl )
```

Parameter	I/O	Description
LJ_handle	I	Handle from LJ_init
n_sweeps	I	Number of sweeps to be run
delta_t	I	Time step for integration
cutoff_rs	I	Cut off value in real space
cutoff_tbl	I	Cut off value for neighbour table
comp_flag	I	Bitfield with flags to indicate which properties has to be calculated: energy, forces, virial, …
return_code	O	Return information structure
status	O	OK or ERROR

Description: This function starts the Velocity Verlet integration with the parameter and data supplied using the LJ potential for the particle interaction.

Function: **`ofpga_1dfft_config`**
Usage: `int ofpga_1dfft_config (fpga, size, direction, msglvl)`

Parameter	I/O	Description
fpga	I	FPGA handle structure
size	I	Size of the transform, N=16, 32, 64, 128
direction	I	Direction: Forward (1) or Backward (0) FFT
returns	O	status := OFPGA_OK | error_code

Description: The purpose of this function is to set the FFT size and FFT direction (forward or reverse) and direction of the FFT.

Function: **`ofpga_3dfft_config`**

Usage: `int ofpga_3dfft_config (fpga, size, direction, msglvl)`

Parameter	I/O	Description
fpga	I	FPGA handle structure
size	I	Size of the transform, N=16, 32, 64, 128
direction	I	Direction: Forward (1) or Backward (0)
returns	O	status := OFPGA_OK | error_code

Description: The API for the *fft_3d()* is essentially the same as that for the *fft_1d* function.