Programming Framework for Clusters with Heterogeneous Accelerators

Kuen Hung Tsoi, Anson H. T. Tse, Peter Pietzuch and Wayne Luk
Department of Computing, Imperial College London
{khtsoi, htt08, prp, wl}@doc.ic.ac.uk

ABSTRACT

We describe a programming framework for high performance clusters with various hardware accelerators. In this framework, users can utilize the available heterogeneous resources productively and efficiently. The distributed application is highly modularized to support dynamic system configuration with changing types and number of the accelerators. Multiple layers of communication interface are introduced to reduce the overhead in both control messages and data transfers. Parallelism can be achieved by controlling the accelerators in various schemes through scheduling extension. The framework has been used to support physics simulation and financial application development. We achieve significant performance improvement on a 16-node cluster with FPGA and GPU accelerators.

1. INTRODUCTION

Dedicated hardware accelerators can achieve orders of magnitude speedup when compared with CPU based implementations. The processing element (PE) of these accelerators usually contains tens to hundreds of ALUs such as graphics processing units (GPUs) or deeply pipelined data paths on the field programming gate array (FPGA) devices. At the expense of program control flexibility, these PEs can outperform the modern CPUs by massive parallelization and off load the most computational intensive task from the CPUs. The large amount of on-chip fast caches and tightly coupled off-chip memory also contribute to performance improvements. High performance clusters (HPCs) with dedicated accelerators offer practical solutions in real world applications [3, 6].

Despite these obvious advantages of accelerator clusters, the number of systems and applications is not comparable to the number of CPU-centric HPCs. These accelerators are usually more expensive than CPU in terms of unit price. To maximize the cost effectiveness of a system, the potential of the PEs must be fully exploited. This can be a challenge to application developers since, unlike CPU programming, it requires information of the PE's internal hardware architecture, different vendor-specific tool chains and even different programming languages.

For example, VHDL programming skills and knowledge of the internal resources of the FPGA are critical for optimizing the kernel performance on FPGA accelerators. The local memories of these accelerators form a complex memory hierarchy. Without a unified flat memory space, the programmers need to explicitly manage the memory systems and move data between them. This overhead reduces productivity, constrains the achievable speedups, and increases the difficulty in kernel optimization. Integrating different types of accelerators in the system worsen the situation. To avoid this complexity, most applications utilize a single type of accelerator even when other types of accelerators sit idle in the system. These difficulties and complexities appear repeatedly in many applications and systems. A general programming framework is desirable to unify the application development practice and improve developer productivity for heterogeneous clusters.

The proposed research targets a cluster with FPGA and GPU accelerators [14]. Based on this hardware architecture, we study the software stack of heterogeneous clusters. Our main aim of this work is to create an easy to use, flexible and efficient framework for developing distributed applications. The framework should be supported by a set of application programming interfaces (APIs) such that developers can utilize the accelerators in a familiar software environment. Abstraction and modularization of the framework will help to minimize the effort of including or excluding different types of accelerators in the applications. More importantly, these different accelerators can work collaboratively on a single application to maximize the efficiency of the system. The major contributions of this work include:

- A general programming framework for heterogeneous accelerators in cluster systems. This framework includes an execution model for applications to distribute the computation to collaborative accelerators. It also includes a modular programming model such that new types of accelerators can be utilized by applications without recompiling the original program code.
- A set of APIs for developing distributed applications on heterogeneous clusters. These APIs enable various schemes of control flow and data transfer between the accelerators. It also provides extension interfaces for static and dynamic load balancing. These interfaces enhance both productivity and efficiency.
- Several applications targeting the Axel cluster using the proposed programming framework. To demonstrate its feasibility and capability, we develop ap-

plications for physics simulation, financial simulation and event-driven message filtering using the framework. The performance of these applications is evaluated on a 16-node cluster with GPU and FPGA accelerators.

The rest of this paper is organized as follows. Section 2 reviews the development framework in previous accelerator clusters. Section 3 presents the architecture of the cluster platform in this work. Section 4 explains the structure and execution model of the distributed applications in the programming framework. Section 5 presents the application programming interfaces for building applications. Section 6 evaluates the results and performance. Finally, section 7 summaries the findings and achievements.

2. RELATED WORK

In 2005, SGI announced the RC100 Blade which can support CPU blades and FPGA blades in the same host system. The SGI proprietary NUMAlink technology provided a flat memory view to the programmers and function calls to the RASC Abstraction Layer to enable data transfer between PEs. The computation kernels running on the Xilinx Virtex-4 LX200 FPGAs were developed separately using high level language synthesizers.

In 2007, a cluster with 64 Virtex-4 FPGA devices was built in the Maxwell project [11]. Linux systems are running independently on each node and communicate through MPI [8]. A custom framework called Parallel Toolkit (PTK) [2] was used as middle-ware between user applications and vendor specific drivers. This toolkit is based on the request-grant model of abstract hardware types to manage FPGA resources and to transfer data. It relied on the assumption that the program is already parallelized in CPU level and requires explicit CPU-FPGA communication.

In 2009, the Quadro Plex (QP) Cluster [6] was built by NCSA in UIUC. For each of the 16 nodes in the QP prototype, there are two AMD Opteron CPUs, four nVidia G80GL GPUs and one Xilinx Virtex-4 LX100 FPGA. This system can theoretically achieve 23 TFlops (single precision). A runtime framework called Phoenix [10] was proposed to schedule the threads to vitalized CPUs. This framework did not manage the FPGA and GPU accelerators.

In 2010, the Axel cluster [14] demonstrates collaboration between heterogeneous accelerators. The development environment, however, requires programmers to manually and statically assign the work load to Xilinx Virtex-5 FPGAs and nVidia C1060 GPUs.

All these development frameworks and programming models require an efficient way of utilizing the accelerators and re-compiling the code when the accelerator types change. Also, the workload distribution are often managed by the runtime systems which have little knowledge of the computation and communication pattern of the application.

3. HARDWARE ARCHITECTURE

The proposed framework can be adapted to different clusters with arbitrary heterogeneous accelerators. To realize and evaluate the framework, we perform our experiments on the Axel cluster. The Axel cluster adopts the nonuniform nodes uniform system (NNUS) approach in which heterogeneous PEs are hosted in a single node. All nodes are uniform with the same abilities. The single program multiple data (SPMD) programming paradigm is suitable for this NNUS architecture. Since all the nodes are identical at system level, multiple instances of user application can easily be distributed to the multi-node cluster.

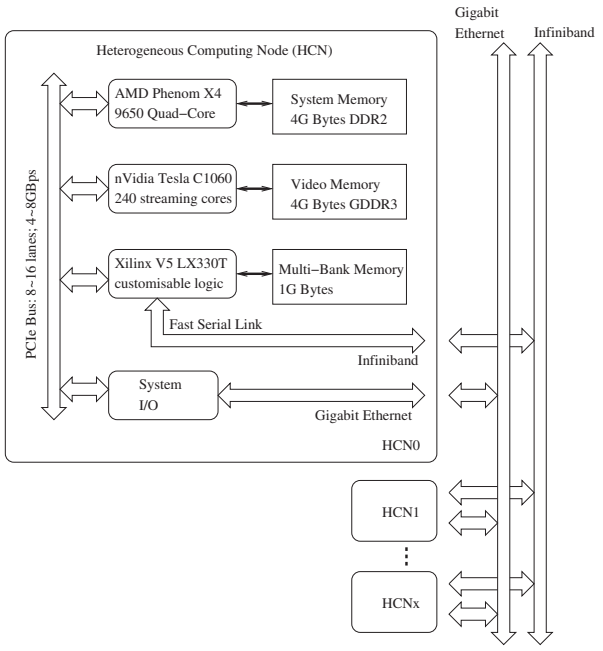

Figure 1: Axel Hardware Architecture.

Figure 1 is an overview of the Axel cluster which currently includes 16 nodes. There are three different types of PEs in a single node: an AMD Phenom Quad-Core CPU, an nVidia Tesla C1060 card [9], and a Xilinx Virtex-5 LX330 FPGA hosted on an ADM-XRC-5T2 card [1]. Each node independently runs a copy of the Linux operating system. Although only GPUs and FPGAs are currently included in the cluster, it is possible to support other type of accelerators in a similar system architecture.

The intra-node communication between PEs is based on PCIe system bus where the GPU and FPGA use separate channels and thus can transfer data simultaneously without blocking. The cluster wide communication is based on Gigabit Ethernet through the NIC port on each node. The versatility and flexibility of Gigabit Ethernet is at the expense of high latency and indeterministic communication. To address this problem, a second inter-node network is added using the four GTP interfaces in the FPGA platform.

There are large amounts of local memory associated with each PE. This creates a physically non-uniform memory hierarchy within a node. The bandwidth between the PEs and their associated local memory is much higher than the communication bandwidth between PEs. Thus data partition and distribution are critical for performance in the cluster.

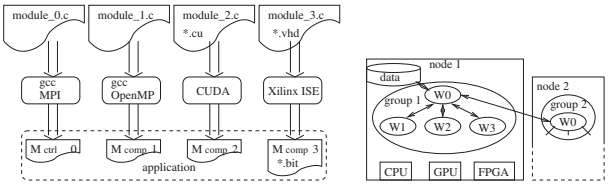

Figure 2: Distributed Application Structure.

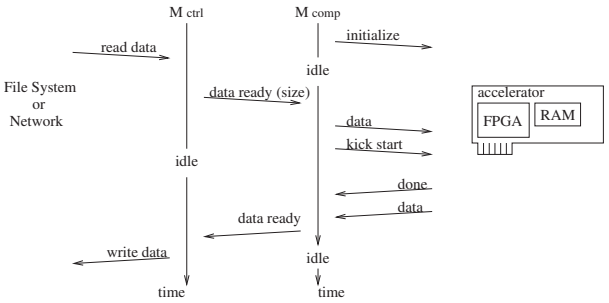

Figure 3: Module interactions.

4. APPLICATION STRUCTURE

4.1 Distributed Modules

The most distinct character of this proposed framework from most other systems is the multiple executables structure in a single application. One *module* in the framework is a full featured executable for a specific task targeting a specific PE. The active instance of a module is called a *worker*. A *group* of workers collaborate on a single node. A running *application* is a collection of groups of workers. Parallelism is achieved by distributing the workload to the active workers of the application. Figure 2 illustrates this distributed application structure.

A module can be either for computation or for control. The computation modules, $M_{comp}1..M_{comp}3$, carry the application kernels which are accelerated by the targeted PEs. The control module, M_{ctrl}, commands and synchronizes the M_{comp} to complete the application. An application consists of at least one M_{ctrl} and one M_{comp}.

Similar to a stand alone application targeting a single accelerator, M_{comp} interfaces to the vendor specific device driver to initialize and configure the PE; download the data to the accelerator local memory; monitor and synchronize the hardware; and collect the results from the accelerator. The major difference of M_{comp} from a stand alone application is its interaction with M_{ctrl}. Figure 3 shows an example of the interaction during an application execution using the FPGA accelerator. Instead of reading data and storing results actively, M_{comp} starts being idle. M_{ctrl} prepares the data and signals M_{comp} to start working. After finishing accelerator computation, M_{comp} reports the status to M_{ctrl} and becomes idle again. Besides controlling the M_{comp} in the group, M_{ctrl} also communicates with M_{ctrl} in other groups and performs data read/write. A single M_{ctrl} can master multiple M_{comp} although only one M_{comp} is shown in Figure 3. Another important task of M_{ctrl} is to launch the associated M_{comp} as described in section 4.2.

Since each module targets a single type of accelerator, the management of adding and removing accelerators to and from the application is similar to managing files in a directory. This modular structure ensures that a module will continue to function correctly regardless of other modules. This feature provides flexibility in application development. For example, the application can start functioning as soon as any of the accelerator modules is ready, and it can be gradually improved as new modules become ready. Unlike other systems which hardwire the application in a single executable, re-compilation of the whole application is not necessary in this framework when updating the accelerators.

Another benefit of this distributed module structure is that the original tool chains of the accelerators are preserved as shown in Figure 2, improving productivity. Integrating the accelerators in a unified program description usually requires an abstraction layer involving source-level translation or syntax extension. In this framework, the absence of this abstraction layer improves ease of use and efficiency. Application developers for different accelerators can use the most familiar tool chains to develop and optimize modules, based on the unified communication API.

4.2 Auxiliary Information

Isolating the accelerator kernels in independent modules introduces an integrity problem of the application, since none of the modules have knowledge about the capability or even existence of other modules. This information, excluded from program code, is captured in a auxiliary meta file call the Application Configuration File (ACF). The ACF plays an essential part in this framework.

The ACF is in XML format and is placed in a networked file system such that all M_{ctrl} can access it. It is created by application programmers to chain up all the required modules in the applications. The ACF provides the controlling modules three types of important information: (i) the available modules, (ii) the groups and workers with them, and (iii) the communication between groups. An example of ACF is shown below. Its content forms a recipe showing how the application will be executed in the cluster.

```
<application name="myapp">

  <module id="0" name="./myapp_m0"
    type="io" pe="cpu"> </module>
  <module id="1" name="./myapp_m1"
    type="compute" pe="gpu"> </module>
  <module id="2" name="./myapp_m2"
    type="compute" pe="fpga"> </module>

  <group id="0" ofst="0" size="5">
    <worker module="0" ofst="0" size="0"></worker>
    <worker module="1" ofst="0" size="3"></worker>
    <worker module="2" ofst="3" size="2"></worker>
  </group>

  <group id="1" ofst="5" size="5">
    <worker module="0" ofst="0" size="0"></worker>
    <worker module="1" ofst="5" size="2"></worker>
    <worker module="2" ofst="7" size="3"></worker>
    <output id="0"></output>
  </group>
```

```
</application>
```

The first section of the example ACF indicates three available modules in the application. Key attributes, such as the path to executable in the file system and the target PE types, are also included. Each module is assigned a unique identity. The second and third section in the ACF create two groups of workers for the application. There are three workers mapped to the three modules in each group. Like the module, each group is given a unique identity. The `ofst` and `size` attributes indicate the starting offset and the size of the data assigned to each group. The data assignment includes key information for an array data structure, while the actual interpretation of these two attributes depends on the implementation of user modules. The data are further divided and assigned to the workers in the group using similar work attributes. For example, the second worker in group 1 is an instance of module 1 which will process two data entries starting from the fifth entry. If the assigned data size is 0, the workers, which are usually taking the control role, will not process any data. The last element in group 1 requests the M_{ctrl} in the group to send the finished data to the M_{ctrl} in group 0.

Users can create application specific attributes in the ACF and retrieve the values later in the M_{ctrl} using the provided APIs. Also, the interpretation of the predefined attributes is controlled by user program. Some structures, such as the `output` element, can even be bypassed in applications if a specific scheme has been captured in the program code. The ACF gives developers significant freedom to configure the execution of an application in the cluster.

4.3 Execution Models

The life cycle of a distributed application is presented here. After all necessary modules are created by their specific building tools and the ACF is created to configure the execution scheme, the application is ready to be launched.

A user will launch the M_{ctrl} module through the MPI system. The remote process manager of the MPI system is used to create multiple instances of the M_{ctrl} module in the cluster nodes. There is no indication of which nodes are used in the application, since this information cannot be determined at compile time. The available resources are dynamic in a multi-user multi-application environment. Thus the programming framework can integrate well with third party resource management tools, such as the Torque/Maui system installed in the Axel cluster. There is also no limitation to a one-to-one mapping between groups in software and nodes in hardware, as long as there are no conflict of accelerator requirements.

Once the worker for an M_{ctrl} instance is created, it reads and parses the ACF file. By default, it maps its `RANK` in the MPI system to the group ID. This can be altered by sending the `WHOAMI` request to the RANK0 node. Then it creates other worker instances by loading the corresponding modules into memory. After that, it creates the message queue for all the workers as a control channel. This M_{ctrl} worker may read the data from network shared data storage or receive data from a remote M_{ctrl} worker.

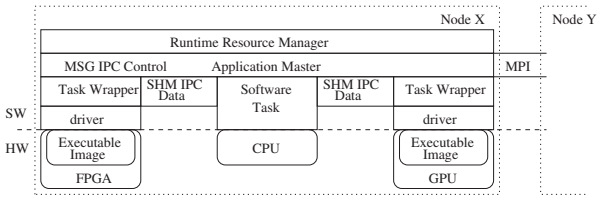

Figure 4: Communications in the heterogeneous programming framework.

To minimize the data transfer overhead, the M_{ctrl} worker will store the data in shared memory which all workers are allowed to access. Application specific data protection and synchronization are implemented in the user code section. The controlling worker then sends a `DATRDY` message to each M_{comp} worker, and tells them the remapped data offset and size in the shared memory.

After receiving the `DATRDY` message, the M_{comp} worker start loading the computation to the targeted hardware accelerator. The processed results from the accelerator are also stored in shared memory. An M_{comp} worker indicates the completion of its work by sending a `DATRDY` message back.

Once the M_{ctrl} worker receives this message, it marks the data segment processed. If all data are marked processed, the M_{ctrl} worker enters the data collection stage. Depending on the application, it may send the data in the shared memory to remote M_{ctrl} workers as instructed by the ACF, or writes them to a file, or iterates to send `DATRDY` to the M_{comp} workers again.

The M_{ctrl} worker can be terminated according to user program code or by sending an `APPEND` message to it. The M_{ctrl} worker will broadcast the `APPEND` message to the group and wait until all other workers have terminated before terminating itself.

The application programmers can instruct the M_{ctrl} workers to periodically send information requests to a predefined node. The returned information is a complete data structure in the ACF format. By doing so, users can dynamically adjust the execution scheme to obtain the most desirable trade-offs fro this application.

5. PROGRAMMING INTERFACE

5.1 Communication Interface

There are four different types of communication APIs in the heterogeneous accelerator cluster as shown in Figure 4. The first type is at the lowest level, involving communications between the M_{comp} workers and the targeting accelerators. The API for this type is provided by the accelerator vendors or the application programmers. The functions include configuring FPGA devices with bitstreams and copying memory contents from host memory to GPU memory.

The second type covers the control messages between workers within a group. Since all workers of the same group reside in the same node, this API is built on top of the message queue POSIX inter process communication (IPC). The 1K byte message packet has the format of a destination worker ID followed by the message type and user defined contents. Besides system predefined

message types such as DATRDY and APPEND, user are free to create custom types. The user defined content area is useful for small and frequently changing data such as system parameters.

The third type covers large amounts of data communication between local workers. Based on the shared memory POSIX IPC, special APIs are needed to map array variable in user applications to the created shared memory blocks. The number of memory copies and data movement overhead is significantly reduced while all the synchronization and protection of the data are not explicit to application programmers.

The last type covers communication between remote workers from different groups. This set of APIs is a warper around the MPI functions calls. The main idea is to encapsulate the MPI related details and provide a group-worker view for global communications. Currently, only point-to-point and all-to-all schemes are supported.

5.2 Design Automation

The rich set of APIs makes the framework very flexible but also requires attention to implementation details. A method that would reduce the need for such details is desirable to further improve developer productivity. Observing that there are similar pattern in the applications structures, we decide to provide a template-based code generation tool. There are code patterns which are required in all applications, such as the parsing and analyzing of the ACF and the termination of workers. These processes are relatively static from application to application. Thus we put them into the static section which will be emitted directly from the template codes.

There are also code patterns which depend on parameters of the application. These parameters are usually defined before the implementation of the modules. For example, the structure and size of the main data variables in an application are usually known in early stages. Thus we can allow users to enter the reference, type, size and read/write patterns of these variables. Code segments such as creating the shared memory and filling in the contents can easily be generated according to user information. Another parameterizable code pattern involves global communications. There are common communication schemes as suggested by the MPI function calls. We allow users to select from these common schemes and related them to data variables. Code segments of sending and receiving these data can then be generated automatically.

The control logic and the computations are the core elements of an application. These control patterns are often different between applications. However the development framework imposes limitations on how the control mechanism is implemented. Since the framework forces an event-driven flow based on message passing, the main loop in a module usually consists of the process of idle listening and message parsing. The code generation tool creates this basic main loop structure as a skeleton or place holder, allowing users to provide application specific control later.

Despite its simple mechanism, this template-based code generator can produce a complete application based on a few user inputs. This will reduce development time and help users focus on the accelerator kernel and control logic optimizations.

5.3 Scheduler Extension

A straight forward implementation using the framework will turn the application to static load distribution, since the subsets of data assigned to each worker and its associated accelerators are explicitly specified in the ACF.

Assuming that parallelism is achieved by distributing and processing small subsets of data concurrently, the performance gain depends on the sequence of workers that take the longest time. A static schedule may not be efficient since workers finished earlier cannot help. This problem of unbalanced load cannot easily be addressed at design time when the ACF is created. The estimated performance of the accelerated kernels is usually not sufficiently accurate and the runtime performance may also be affected by input data or parameters. Also, in a cluster system, new computing resources may be available due to the termination of other applications. A static schedule cannot take advantage of new resources.

To enable a dynamic scheduler to adapt to the actual kernel performance and run-time conditions, special arrangement in the distributed application must be made. First, the work load distribution must not depend on the ACF, and should be acquired by the M_{ctrl} at run time. This can be supported by assigning 0 to all size attributes in the ACF. Second, there must be a central facility to coordinate the load distribution process. Every time a group finishes the assigned work, the M_{ctrl} in the group will send the DATRDY message to the scheduler. Depending on the recorded performance of this group, the scheduler assigns a new batch of data for the group to process. Various scheduling schemes can be applied to achieve different optimization goals. A second level of scheduling process is performed by every M_{ctrl} worker to balance the load of the local M_{comp} workers. As discussed before, the data structure in ACF can be used to alter the data size assigned to each group.

The challenge here is that the ACF data are sent independently from the actual assigned data. To eliminate the synchronization requirement between scheduler control and data transfer, the M_{ctrl} workers are forced to accept data through global communication APIs only when the control and data are from the same sources. Using these methods, a dynamically scheduled Monte Carlo simulation is implemented and various scheduling schemes are evaluated.

6. IMPLEMENTATION RESULTS

Three example applications are implemented on the Axel cluster using the proposed programming framework. These three applications have different computation requirements and communication patterns which exercise various aspects of the cluster. All FPGA acceleration kernels are developed using VHDL and implemented using the Xilinx ISE 11.5 tools. The GPU kernels are compiled using CUDA 2.2 tools. The accelerator results are compared with an AMD Phenom Quad-Core CPU. All floating point computations are performed in IEEE-754 single precision format.

6.1 N-Body Simulation

N-body simulation is a process to model the interaction between N particles under gravitational forces in space [7]. In this example, we implemented a second order 3D N-body simulation application on the Axel cluster. To compute the acceleration vectors of moving particles, the distances between them and every rest particles are computed.

There is no data dependency within a simulation iteration. So the FPGA design can be deeply pipelined and high throughput is achieved. The particles are arranged into small groups and assigned to different M_{comp} workers including the GPU and FPGA kernels. Each worker computes the acceleration vectors of the assigned particles and broadcasts the results. The final updates of positions and velocities are performed by a CPU-centric worker. Since all workers need to access the most update positions of all the particles for the current simulation, the all-to-all broadcast global communication increases the overhead.

The workload is distributed to GPU and FPGA in an asymmetric way, since the measured performance of FPGA is higher than that of the GPU. This schedule is static and produced after each kernel is fully tested.

6.2 Pub/Sub Matching

The publication/subscribe model is for applications including web services and financial monitoring [4]. The model enables subscribers to set matching rules to filter interested events. A two step algorithm [5] is implemented with FPGA acceleration kernels and tested on the Axel cluster. There are currently no GPU version of this application.

The input events are independent and distributed to different M_{comp} workers for parallelization. The outputs contain lists of triggered subscribers which are also independent. There are only fixed-point number comparison operations in the algorithm. The performance is limited by the FPGA off-chip memory bandwidth and the synchronization process between the two steps.

The workload is evenly distributed to FPGA accelerators. Since there is no correlation between the matching processes of different events and no final results need reduction, the design scales linearly with the increase in nodes.

6.3 Monte Carlo Simulation

Monte Carlo simulation is a useful tool to solve complex problems [12, 13]. We have developed an option pricing application using Monte Carlo simulation. The major computation is to simulate the paths of option price movement in a mathematical model. This application includes both FPGA and GPU acceleration kernels and supports dynamic scheduling using the method described in section 5.3.

The inputs to the system are a set of random variables. Each of these variables contributes to a step in the pricing path. The put and call payoffs of all the simulated paths are collected to compute the final expected value. There is no data dependency between simulated paths.

Since the application produces random variables to initialize the design, there is no external input for both the FPGA and GPU kernels. The random sequences are generated within the acceleration kernels. The workload is distributed by assigning different sets of paths to the M_{comp} workers. The workers partially reduce the results of the assigned paths and send them to a single data sink where the final expected values are computed.

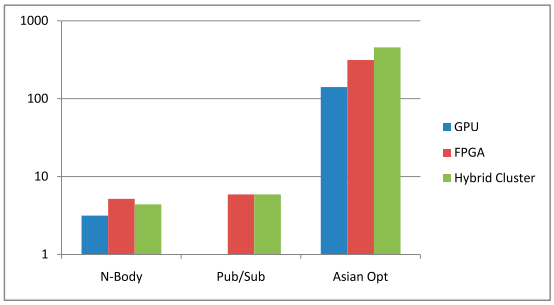

Figure 5: Performance results.

6.4 Results

Figure 5 shows the achieved speedup of the hardware accelerators based on the proposed programming framework. The results of a single GPU and a single FPGA are compared with that of a four-thread software version running on the Quad-core CPU. The hybrid cluster result is a comparison of a 16-node heterogeneous accelerator cluster to a 16-node CPU only cluster. Since there is no GPU implementation of the Pub/Sub application, the cluster version of Pub/Sub utilizes the FPGA accelerators only.

7. CONCLUSION

This paper presents a framework for improving productivity and efficiency of application development in a cluster environment targeting heterogeneous hardware accelerators. The modular structure for modules and configuration files enables rapid development and easy extension. Current and future work includes improving run-time optimization such as dynamic load balancing, and supporting a wide range of applications and hardware platforms.

Acknowledgments

The support of Imperial College London Research Excellence Award, UK Engineering and Physical Sciences Research Council, Alpha Data, nVidia and Xilinx is gratefully acknowledged.

8. REFERENCES

[1] Alpha-Data Parallel System Ltd. *ADM-XRC-5T2 User Manual*, 2008.

[2] R. Baxter and et al. The FPGA high-performance computing alliance parallel toolkit. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 301–310, 2007.

[3] T. Endo and S. Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2008.

[4] P. T. Eugster and et al. The many faces of publish/subscribe. *ACM Computer Survey*, 35(2):114–131, 2003.

[5] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, 2001.

[6] M. Showerman et al. QP: A heterogeneous multi-acceleator cluster. In *10th LCI International Conference on High-Performance Clustered Computing*, Boulder, Colorado, March 2009.

[7] J. Makino. The GRAPE project. *Computing in Science & Engineering*, 8(1):30–40, Jan.-Feb. 2006.

[8] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard Version 2.1*, 2008.

[9] nVidia Co. *Tesla C1060 Computing Processor Board*, Sep. 2008.

[10] A. Pant, H. Jafri, and V. Kindratenko. Phoenix: A runtime environment for high performance computing on chip multiprocessors. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 119 –126, 2009.

[11] R. Baxter et al. Maxwell - a 64 FPGA supercomputer. In *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems*, pages 287–294, 2007.

[12] X. Tian and K. Benkrid. High performance quasi-monte carlofinancial simulation: FPGA vs. GPP vs. GPU. *ACM Transaction on Reconfigurable Technology and Systems*, to appear, 2010.

[13] A. H. T. Tse, D. B. Thomas, and W. Luk. Accelerating quadrature methods for option valuation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[14] K. H. Tsoi and W. Luk. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 115–124, 2010.