Axel: A Heterogeneous Cluster with FPGAs and GPUs

Kuen Hung Tsoi and Wayne Luk
Department of Computing
Imperial College London, UK
{khtsoi,wl}@doc.ic.ac.uk

ABSTRACT

This paper describes a heterogeneous computer cluster called Axel. Axel contains a collection of nodes; each node can include multiple types of accelerators such as FPGAs (Field Programmable Gate Arrays) and GPUs (Graphics Processing Units). A Map-Reduce framework for the Axel cluster is presented which exploits spatial and temporal locality through different types of processing elements and communication channels. The Axel system enables the first demonstration of FPGAs, GPUs and CPUs running collaboratively for N-body simulation. Performance improvement from 4.4 times to 22.7 times has been achieved using our approach, which shows that the Axel system can combine the benefits of the specialization of FPGA, the parallelism of GPU, and the scalability of computer clusters.

Categories and Subject Descriptors

C.5.0 [**Computer Systems Organization**]: COMPUTER SYSTEM IMPLEMENTATIONGeneral

General Terms

Design

Keywords

Heterogeneous Cluster, FPGA

1. INTRODUCTION

Despite the continuously increasing clock frequency and silicon size of modern CPU designs, the performance of a single processor system is greatly constrained by the memory wall, the power wall and the ILP (instruction level parallelism) wall [6]. A common solution is to increase the number of processors in a system to achieve the high computational requirement for applications such as physical simulation, multimedia content creation and financial modeling. This method is proved practical by the dominance of computer clusters in the TOP500 supercomputer list.

Building a cluster of computers is a common technique to realize the parallel computing model. Hundreds to thousands of computing nodes are connected together in networks to form a cluster system. Using commodity CPU based computers as nodes has the advantages of low unit price, flexible configuration, rich peripherals, familiar programming environments and easy upgrade path.

The computing capacity within a node is a critical factor for performance improvement besides communication efficiency. There are two ways to increase node performance: the use of dedicated hardware accelerators and multi-core, multi-threaded parallelism. A number of high-performance computers are already adopting this approach [18].

A heterogeneous computer cluster is more efficient than a homogeneous one since some kinds of processing units have better performance than others for certain computation tasks, and tightly coupled hardware accelerators inside a node can reduce communication requirements by making use of data locality. Overall system performance can be improved by allowing the heterogeneous cores to work collaboratively on different parts of an application. The challenges here are communication efficiency, workload balancing and application portability. We address these issues by providing dedicated inter-FPGA communication channels, a hardware abstraction model for resource estimation, and a development flow for such heterogeneous computer clusters. The major contributions of this work include:

- A heterogeneous computer cluster called Axel. Axel contains a collection of nodes; each node can include multiple types of accelerators such as FPGAs and GPUs.
- A Map-Reduce framework for the Axel cluster which exploits spatial and temporal locality through different types of processing elements and communication channels.
- The first demonstration of FPGAs, GPUs and CPUs running collaboratively for N-body simulation, showing that the Axel system can combine the benefits of the specialization of FPGA, the parallelism of GPU and the scalability of computer clusters.

The rest of this paper is organized as follows: Section 2 reviews related work on heterogeneous clustering. Section 3 presents the hardware and software architecture of the Axel system. Section 4 introduces the Map-Reduce framework for heterogeneous clusters. Section 5 describes the details of the N-body simulation example which illustrates our approach. Section 6 contains experimental results and performance evaluation. Finally Section 7 draws conclusions.

2. RELATED WORK

In 2004, the Cray XD1 computer [17] hosts 12 Opteron CPUs and 6 Xilinx Virtex-II FPGA devices on a single motherboard through a HyperTransport based communication network and achieving 58 GFLOPS. In 2009, SRC releases the SRC-7 MAPstation [16] which integrates CPU with up to six Altera Startix-II FPGA devices in a single host through the DDR DIMM slot interface on the motherboard. These examples show that hardware co-processors such as FPGA devices can offer significant improvement to many applications [18]. However these examples lack the framework for developing distributed applications on clusters with multiple nodes.

There are also systems where FPGA devices are used as the only computing elements forming the cluster. The Berkeley Emulation Engine 2 (BEE2) [2] developed in 2004 has five Xilinx Virtex-II Pro 70 FPGAs hosted on a single motherboard. A 64-bit ring in a star topology connects the four computational FPGAs and a central control FPGA. Computationally intensive tasks run on the outer ring while the control FPGA runs Linux Operating System and manages off-board I/Os. In 2007, the Maxwell project [13] demonstrates a complete cluster with 64 Virtex-4 FPGA devices arranged in a 2-D torus network through Infiniband connections. An object based Parallel Toolkit (PTK) has been developed for the Maxwell hardware. It provides abstract hardware description and data structure to enable resource and task management in a FPGA centric cluster environment. Also in 2007, the Reconfigurable Computing Cluster (RCC) platform is developed in University of North Carolina [14]. The prototype machine, *Spirit*, contains 64 Xilinx ML410 boards connected through SATA expansion module and cables. A modified Message Passing Interface (MPI) library [9] as well as the Linux Operating System are running on the embedded PowerPC processor in the Virtex-4 FPGA. Reconfigurable accelerator cores are built as peripherals attached to PowerPC through system bus. In 2008, the Parton tool [4] was introduced to produce high performance implementations for the MAX2 hardware platform. The MAX2 card has 2 Xilinx Virtex-5 LX330T FPGA and 24GB DDR2 on-board memory. Users can identify the hot spots in a CPU based implementation by profiling and static code analysis. The Parton tool provides a modeling and transformation framework for optimization and performance estimation. A geophysical modeling application achieves 374 times speedup over CPU based implementation.

The above examples provide integrated programming and runtime environment in FPGA-centric clusters. However, not all applications can be accelerated effectively using FPGAs. The high clock rate and large numbers of floating point units in GPUs and DSPs make them good candidates for hardware accelerators. The first generation of TSUBAME (Tokyo-tech Supercompyuter and Ubiquitously Accessible Mass-storage Environment) system, built in April 2006 [3], contains dedicated vector processors as hardware accelerators. The upgrade of TSUBAME in late 2008 adds 170 nVidia Tesla C1070 cards to the system. The measured performance of the new version ramps from 56.43 TFlops to 77.48 TFlops.

The combination of CPU, GPU/DSP and FPGA in cluster nodes has been reported recently. In 2009, NCSA in UIUC publishes the first prototype and performance evaluation of the Quadro Plex (QP) Cluster [7]. The prototype QP system has two AMD Opteron CPUs, four nVidia G80GL GPUs and one Xilinx Virtex-4 LX100 FPGA in each node. With 16 nodes, the theoretical peak performance is 23 TFlops (single precision), where the GPUs contribute 96% of them. Various applications including molecular dynamics (NAMD), weather modeling (WRF), cosmology data analysis (TPACF) have been implemented on the QP system. However, no results involve distributing workloads to both GPU and FPGA and running them concurrently at this stage. A runtime system, Phoenix, is under development for QP to integrate various tools including CUDA [11], DIMEtalk [10] and MPI.

There is also research on the general architecture of heterogeneous system [18]. In this study, the attributes of uniform node nonuniform systems (UNNSs) and nonuniform node uniform systems (NNUSs) are compared and several benchmark applications are used to evaluate these platforms.

3. THE AXEL SYSTEM

In this work, we propose a heterogeneous cluster called Axel, which is intended to enable experiments in reconfigurable cluster computing. In the Axel system, various processing elements (PEs) are integrated in a single compute node through a local system bus, and the nodes are connected together by various global networking topologies. Different software tools are used together to implement the distributed versions of applications across the heterogeneous PEs in the cluster. In the following, we describe the architecture, hardware, and software aspects of Axel, highlighting the major design decisions.

3.1 Architecture

There are two methods to group together the elements of a heterogeneous cluster: either as Uniform Node Nonuniform System (UNNS) or as Nonuniform Node Uniform Systems (NNUS). As shown in Figure 1, we generalize the description of these two approaches of grouping PEs in [18] to include both FPGA and GPU technologies in each node.

In the UNNS approach, each node has a single type of PE such that the installation and management of hardware accelerators (e.g. FPGA and GPU) are simpler in a segment of the cluster. Also, the ratio between these accelerators and the CPU can be easily adjusted to optimize for the needs of a specific application. The major drawbacks of the UNNS aproach are the communication overhead between PEs and requirement of special hosting boards/links for non-CPU PEs. The dedicated Hi-Bar switch in the SRC-7 MapStation and the RC100 Blade module in the SGI RASC server are both examples of such special hardware, which offers high performance but can be costly to develop. Applications with data communication between CPU and accelerators may have to go through the slow and high-latency global network. To address this overhead, a dedicated ASIC, the TIO chip, is developed for the SGI RASC system.

Our Axel cluster adopts the NNUS approach in which heterogeneous PEs are hosted in a single node. All nodes are uniformly created with the same abilities of hosting and communicating with PEs mainly through the system bus. There are several advantages of this approach. First, as long as the number of accelerators is limited, the communication between CPU and accelerators can be supported by

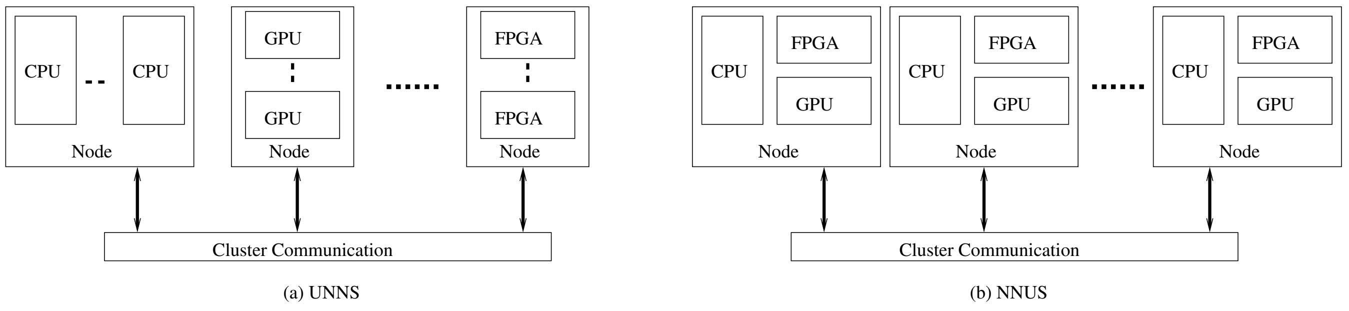


Figure 1: Approaches of heterogeneous accelerator grouping: (a) UNNS and (b) NNUS.

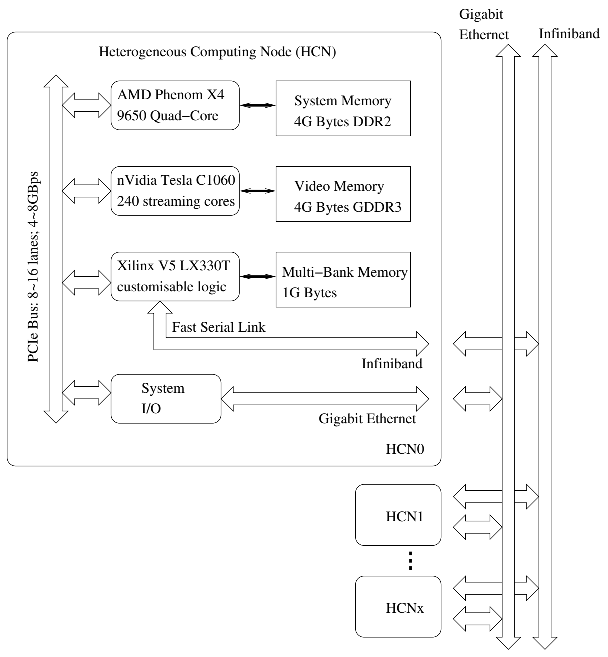

Gigabit Ethernet
Infiniband
Heterogeneous Computing Node (HCN)
AMD Phenom X4 9650 Quad−Core
System Memory 4G Bytes DDR2
nVidia Tesla C1060 240 streaming cores
Video Memory 4G Bytes GDDR3
Xilinx V5 LX330T customisable logic
Multi−Bank Memory 1G Bytes
Fast Serial Link
Infiniband
System I/O
Gigabit Ethernet
HCN0
PCIe Bus: 8~16 lanes; 4~8GBps
HCN1
HCNx

Figure 2: Axel Hardware Architecture.

the CPU system bus, such as the PCIe bus in Axel which often provides adequate performance for many applications. Second, the serial links in our FPGA accelerator provide additional communication channels between nodes. They effectively increase the total communication bandwidth between cluster nodes. Third, the single program multiple data (SPMD) programming paradigm can be easily mapped to the NNUS architecture. Since all the nodes are uniform at system level, a single copy of the user application can be easily scaled to the multi-node cluster. Finally, the NNUS approach provides a flexible solution by freeing users from specialized vendor-specific system hardware for connecting non-CPU PEs in nodes.

3.2 Hardware

Figure 2 provides an overview of the Axel cluster. Our first prototype contains 16 nodes. As shown in the figure, there are three different types of PEs in a single node: an AMD Phenom Quad-Core CPU, an nVidia Tesla C1060 card [12], and a Xilinx Virtex-5 LX330 FPGA hosted on an ADM-XRC-5T2 card [1]. Each node is a full scale server in 4U size with local disk storage. All components in the Axel cluster are commodity parts. Although only GPU and FPGA are currently used as accelerators, it is possible to include other forms of hardware accelerators in a similar system architecture.

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

Another characteristic of the Axel nodes is the large amount of distributed memory associated with each PE. 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 Axel.

3.3 Software

Each node in Axel runs its own copy of Linux Operating System. For an application, there is no single executable image across the cluster. The application is partitioned into specific computations on subset of data in the form of tasks and built by a collection of tools. Each of these tasks is wrapped into a self-contained executable image for the Operating System. Multiple mechanisms are used to support communications between these tasks based on their execution targets. Each node keeps a local copy of data as well as these task images in disk storage for efficient loading in runtime.

The tasks targeting CPU execution are built using the standard GCC compiler (version 4.2.3). This software part also includes some of the communication between tasks. The communication between tasks across different nodes is based on the OpenMPI framework through the Gigabit Ethernet. The communication between tasks across different PEs in the same node is based on the shared memory Inter Process Communication (IPC) framework above the device driver level.

The CUDA SDK 2.2 from nVidia provides a C-like programming environment for building tasks targeting the Tesla platform. The compiled executable accesses the Tesla card through the CUDA driver. Xilinx ISE 10.1 tools are used for FPGA development. The FPGA devices are configured and controlled through the Alpha Data ADMXRC driver APIs.

To provide a fair and efficient workload distribution scheme and to enable a deterministic benchmarking environment,

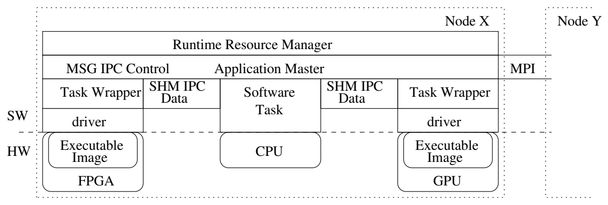


Figure 3: Axel Software Architecture.

the Torque resource manager and the Maui scheduler are installed in the head node, `axelgw`. All nodes from `axel01` to `axel16` have the Torque client activated after booting. User can send their jobs to Torque from `axelgw` and the workload will be automatically distributed to idle or specified nodes.

While the Torque/Maui combination is useful for CPU based resource management, the GPU and FPGA have different attributes which are not captured by standard tools. For example, unlike CPUs, both GPUs and FPGAs are not time sharing processing elements as CPU, and the internal states cannot be preserved during context switching.

We develop a resource management (RM) system to check the condition of GPUs and FPGAs in the cluster. To make this system work, each Axel program needs to allocate the hardware resources needed by calling the runtime API. This RM process communicates with the tasks running on the same node using the message queue IPC framework. The resources will be released by calling the *free* API explicitly in the user application or by the resource manager checking the existence of the process to which the resources are allocated. Figure 3 shows the architecture of the system software running on Axel. The head node collects information from the RM in the cluster and prepares a script to be submitted to the Torque job queue. In this way, special resources such as GPUs and FPGAs are managed properly and efficiently. The isolation between worker processes and the control/communication process, and the introduction of a wrapper layer, enable us to extend the system to include different types of accelerators.

4. DEVELOPMENT FRAMEWORK

One main objective in this work is to improve the accessibility and productivity of application builders targeting the heterogeneous cluster. Hardware Abstraction Model (HAM), a way of systematically characterising resources in heterogeneous systems, has been developed for Axel users to partition computations and estimate performance. Also a development flow is described to specify the methodology of porting and distributing applications for systems similar to Axel.

4.1 Hardware Abstraction Model

The HAM captures the available resources in each PE in three major parts: computation, local memory and communication. This description is stored as a hierarchical configuration file which is distributed and saved locally in each node. It is used to reflect the current status of the system at runtime to help programmers plan better in early stage of development and to enable estimation of possible performance gain in different load distribution schemes.

Each PE in Axel has its own entry in the HAM file regardless of the node it resides in. The major parameters in the HAM description are summarized below.

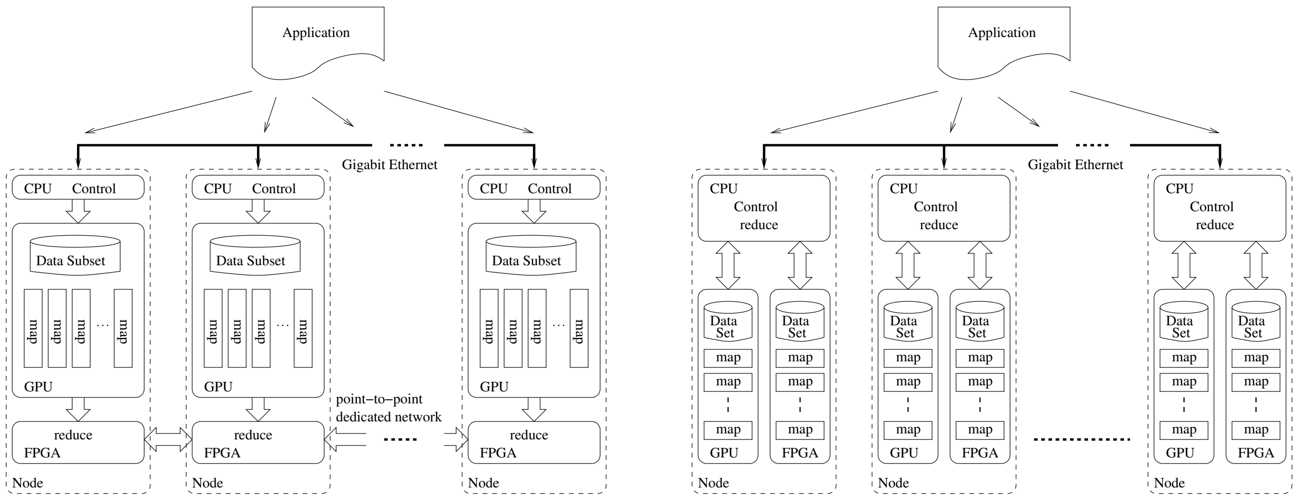
```
Cluster {
  Computing Node {
    Node ID {}
    Processing Element [1 .. PE_Count] {

      Core [1 .. Core_Count] {
        Integer Units { ... }
        Single Precision FPU { ... }
        Double Precision FPU { ... }
        Bit Manipulation Units { ... }
        Internal Memory { ... }
        ... }

      Data Endpoint [1 .. DE_Count] {
        Peer [1 .. Peer_Count] { PE : DE }
        Latency { ... }
        Bandwidth { ... }
        Type { Shared Bus | Peer-to-Peer }
        ... }

      Local Memory {
        Size { ... }
        Latency { ... }
        Bandwidth { ... }
        Channels { ... }
        ... }
} } }
```

In the HAM, the computation resources of a PE are modeled as a collection of cores each of which has its own computation capability. The resources inside a core are categorized into different functional blocks such as integer and floating point units. Besides min/max availability count, the normalized unit cost, in percentage format, of each kind of block is also given. As the same function can be realised by different kinds of blocks according to the user design in the FPGA, this normalized cost ensures that the modeled design will be mapped within the total available resources.

Unlike many other hardware and performance modeling systems, the communication capability in Axel is bound to the individual PE in the hierarchy structure instead of being at parallel level of PE as channels. In our approach, data are sent and received through data endpoints (DE) each associated with a list of PE:DE pairs indicating the connected parts.

The HAM file facilitates porting applications to different systems by providing a standard way of representing available resources. For example, the HAM information can be used to estimate the resource utilization of an N-by-N matrix multiplication. This process requires N^3 multiplications and $(N-1)^3$ additions. When targeting the C1060 GPU platform, programmers may use the sub-matrix multiplication algorithm as in the CUDA SDK to load data to shared memory before computation. From the `Internal Memory` entry of the HAM file, there are 16KB of shared memory per multiprocessor in the GPU. Since we need three buffers for the two input and one output matrices, the maximum number of threads that can work together on a sub-matrix in the application is $\lfloor\sqrt{(16\times 1024)/12}\rfloor^2 = 36^2 = 1296$; we divide the memory size by 12 since 3 buffers are used and 4 bytes are required for a single precision floating point value.

When targeting the FPGA, all distributed logic resources in a FPGA device are considered as a single core. We estimate the floating point capability of the FPGA device using the resource consumption reported by Xilinx CoreGen tools. The Xilinx Virtex-5 LX330T device is able to accommodate



(a) FPGA-based Reduce (b) CPU-based Reduce

Figure 4: Possible Map-Reduce schemes for Axel.

300 multipliers or 460 adders for single precision floating point computation at 333MHz. We assume that both the multipliers and adders use the LUT/FF resources in FPGA, so we normalize their cost to 1/300 and 1/460 respectively. Thus we estimate the FPGA resource utilization to be x multipliers and y adders where $(x/300) + (y/460) \leq 1$ and $x/y = N/(N-1)$.

Special attributes which are not applicable to all PEs are omitted in the HAM. For example, it does not include asymmetric read/write bandwidth, interrupt and latency of floating point units. We take the maximum generic attribute set as the abstract hardware representation since this model is applied to heterogeneous PEs with diverse capabilities and features. For the Axel system, this granularity is sufficient to provide relative performance estimation at an early stage of application development. Also, adding a new PE type requires changes in parameter values instead of HAM file structure. Thus the runtime system will continue functioning without modification and recompilation.

4.2 Map-Reduce Framework

The Map-Reduce framework is commonly used in distributed computing platforms due to its scalability [15]. The Map-Reduce approach accepts a list of key-value pairs and processes the values in the *Map* function. Since the computation inside the *Map* remains the same for different data, it is possible to partition the input data set and process the smaller working sets in multiple instances of the *Map* function across different nodes in a cluster. The output of the *Map* function is usually a new set of key-value pairs which will be directed to the *Reduce* function for final result generation.

Given no dependency between the working data sets, the *Map* function can be executed in parallel. Logically, the *Reduce* function is applied to every intermediate key-value pair. For better communication efficiency, the partially reduced results are generated in each node and collected for further reduction. A library interface similar to the one in [5] is used to facilitate the Map-Reduce framework in Axel. The major improvement in this work is that we consider several schemes for collaborative computation between different types of accelerators, such as FPGAs and GPUs, in heterogeneous computer clusters.

The effectiveness and scalability of the Map-Reduce framework on homogeneous clusters are based on the spatial locality of the input data set. In a heterogeneous environment, temporal locality of the application can also be exploited for Map-Reduce. Since we now have multiple PEs with diverse computing abilities and connection topologies, it is possible to assign different computation and communication tasks to different PEs based on the computation/communication patterns of these tasks. Figure 4 shows two diagrams of possible workload distribution. Figure 4(a) is a general approach for applications which utilize the specialties of the PEs. The high speed low latency GTP communication ports in the FPGA accelerator are utilized for passing data for *Reduce* in this configuration. In the current implementation, we use another Map-Reduce scheme, as shown in Figure 4(b), such that the GPU and FPGA work on the *Map* part in parallel and the *Reduce* part is performed by the CPU using the Gigabit Ethernet as communication channel.

In both approaches in Figure 4, spatial locality is captured in distributing data to parallel *Map* processes which read from and write to their local data storages. Temporal locality is captured in the FPGA-based *Reduce* processes as the data stream from the CPU to the GPU and from the GPU to the FPGA using separate channels.

4.3 Development Flow

A development flow has been introduced to utilize the hardware accelerators in Axel. For example, to port an application from a conventional CPU cluster to Axel under the Map-Reduce framework, one should:

1. Partition the original data set into small subsets which are processed in different nodes. The partition process should maximize parallelism by minimizing data dependency between data subsets. This can be achieved by static analysis of the Data Flow Graph (DFG) of the original application.

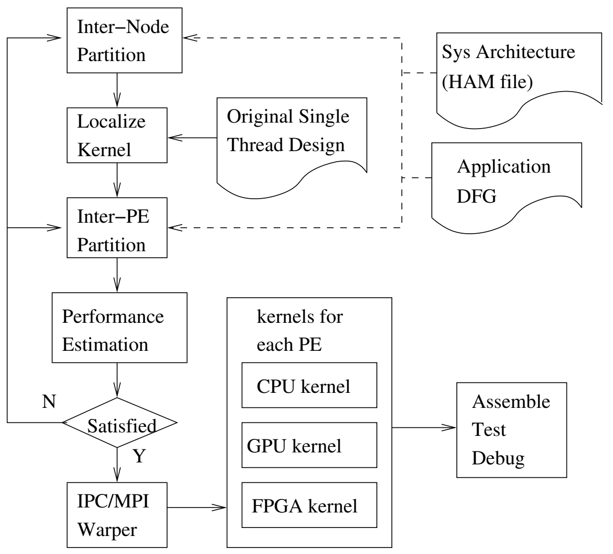


Figure 5: Axel application development flow. Note that the HAM file is independent of the specific application.

2. Rewrite the application kernel in Map-Reduce framework which operates on the local data subset.
3. Partition the Map-Reduce kernel into different tasks targeting FPGA or GPU. This process can be achieved by analyzing and clustering the operations in the kernel source code and matching them to the attributes of the targeted PE. Also, information from later steps can be used to guide the process for improved results.
4. Extract the computation requirements of each partitioned tasks and the communication requirements between them. Apply the HAM described in Section 4.1 to estimate the performance of current configuration. If performance constraints are not met, go back to data and/or task partition steps for fine tuning.
5. Wrap the partitioned tasks using IPC interface for intra-node communication. Create a top level application for data initialization and MPI communication. Create the application configuration file for the Axel runtime system which loads and manage the collection of tasks in the application.
6. Program the tasks targeting GPU and FPGA using appropriate languages and tools, such as CUDA for GPU and VHDL for FPGA. Provide the actual implementation of the GPU and FPGA tasks by using the CUDA compiler and ISE tools.

Figure 5 summarises this development flow of transforming an original single threaded CPU application into several executables running on heterogeneous PEs in multiple nodes.

5. APPLICATIONS

To demonstrate the proposed development flow and scalability of our heterogeneous cluster, the N-Body simulation process [8] is implemented on Axel utilizing all three types of PEs in the system.

5.1 N-body Simulation

The N-body system is a simulation process to model the interaction between N particles under gravitational forces in space. The operations in N-body force computation are shown in Algorithm 1. Here p is the array storing the (x, y, z) coordinate position, the (vx, vy, vz) velocity vector and the mass of the particles m, and a is the array storing the acceleration vectors on the particle in a time step. The constant EPS is a damping factor to prevent excessive force between two very close particles. The computations in Algorithm 1 are repeated for each time step in the simulation process.

Algorithm 1 N-body force computation.

```
for $i = 1$ to $N$ do
  for $j = 1$ to $N$ do
    $r.x = p[i].x - p[j].x$
    $r.y = p[i].y - p[j].y$
    $r.z = p[i].z - p[j].z$
    $d = (r.x^2 + r.y^2 + r.z^2 + EPS)^{-1.5}$
    $s = p[j].m * d$
    $a[i].x += r.x * s$
    $a[i].y += r.y * s$
    $a[i].z += r.z * s$
  end for
end for
for $k = 1$ to $N$ do
  $p[k].x = p[k].x + p[k].vx$
  $p[k].y = p[k].y + p[k].vy$
  $p[k].z = p[k].z + p[k].vz$
  $p[k].vx = p[k].vx + a[k].x$
  $p[k].vy = p[k].vy + a[k].y$
  $p[k].vz = p[k].vz + a[k].z$
end for
```

First, we implement a CPU based N-body simulation program as reference design. We identify the hot spot in this program to be the inner *j-loop* where the distances from one particle to all the other particles are computed.

Following the development flow introduced in Section 4.3 and the knowledge of the configuration of each node, we partition and distribute the *i-loop* evenly to all the nodes. Then we modify the reference design to enable distribution through the map-reduce API and MPI library. We also prepare the shared memory ID for intra-node communication in later steps. In the third step, we partition the inner *j-loop* for FPGA and GPU. Here, the original computation code is replaced by the IPC communication code. In the fourth step, we estimate the performance of computation on each PE by extracting the operations and memory references in the reference design. Such information is then mapped to the hardware resources and bandwidth in the hardware abstraction model. We then change the partition size between FPGA and GPU to balance the run time of the two kernels. In the fifth step, we create the wrapper process for both GPU and FPGA which includes the IPC communication, the device initialization and host-device data movement. The complete application will then be ready for testing and debugging on a multi-node heterogeneous cluster.

5.2 N-body Simulation in Axel Framework

The *i-loop* in Algorithm 1 becomes the *Map* part and the *k-loop* which iterates through all the resulting acceleration vectors from the *Map* function becomes the *Reduce* part. To minimize communication overhead, we reserve the CPU for the *Reduce* function and distribute the *Map* function to both FPGA and GPU in each node as shown in Figure 4(b).

Following the approach described in Section 4.2, partitioning the *i-loop* into the *Map* functions exploits the spatial locality in data parallelism. Although CPU is used for the *Reduce* process, we can still exploit temporal locality in the deeply pipelined FPGA implementation. Data flow through a sequence of operators along the data path and are stored in each pipeline stage.

All data including input, output and internal computation are in 32-bit IEEE-754 single precision format. To compute the acceleration for a particle, all position and mass information must be read. Thus 16 bytes ($4\ bytes \times 4$) of data are read for each iteration of the inner loop in the above algorithm. In the inner loop, 17 floating point operations are performed including 3 subtractions, 3 additions, 6 multiplications, 1 square root, 1 reciprocal and 3 accumulations. So the computation to data ratio in Floating point Operation Per Byte (flopb) is:

$$R_{cd} = 17/16 = 1.0625 flopb \tag{1}$$

Both inner and outer loop need to iterate over all particles. Thus there are in total $N \times (N-1) \times 17$ operations in each simulation step. Applying the HAM method to this situation, the C1060 GPU platform with 480 single precision floating point ADD and MULT units needs:

$$T_{cg} = N \times (N-1) \times 17/480/(1296MHz) \tag{2}$$

to compute one simulation step. Since the intermediate data can be stored in the stream processors' register file and the P_j data is shared by all threads in the GPU, overhead of memory read/write is ignored. For each particle, we send 5 single precision floating point values to the GPU and receive 3 floating point values from it. So the data communication time for the GPU design is:

$$T_{tg} = N \times (5+3) \times 32bit/(8Gbps). \tag{3}$$

The total time for the GPU to complete one time step of N-body simulation is $T_{cg} + T_{tg}$. When targeting FPGAs, we consider adders and multipliers as different resources instead of multi-function ALU as in the GPU design. The time for FPGA computation is:

$$T_{cf} = N \times (N-1) \times max(8/x, 9/y)/(333MHz) \tag{4}$$

where x and y are the numbers of multipliers and adders in the FPGA, $(x/300) + (y/460) \leq 1$ and $x/y = 8/9$. Here we treat the power function as multiplication as in the GPU case. So the data transmission time between CPU and FPGA is:

$$T_{tf} = N \times (5 \times 32bit/(1.36Gbps) + 3 \times 32bit/(1.84Gbps)). \tag{5}$$

The total estimated time for FPGA design is $T_{cf} + T_{tf}$. The above calculations provide a theoretical upper bound of the performance of each type of accelerator. Also, they assume only the fine grained logic is used in the FPGA implementation. For implementations involving other FPGA resources such as hardwired multiplier blocks, more complex models will be required.

6. RESULTS

The experimental results of the N-body simulation are presented in this section. We implement and evaluate the designs under various configuration of hardware platforms including standalone accelerator, heterogeneous combination of accelerators and multi-node cluster.

Previous work [5] focuses on providing a single source multiple backend automation and adopts HyperStream library for FPGA implementation. This simplifies the design entry process but results in low performance in accelerator kernels. In contrast, we emphasize the performance of the final designs on the Axel system, and the scalability of applications across the network. To achieve this, we isolate the accelerator kernel coding process and use the native tools for targeting PE such as VHDL for FPGA devices.

6.1 Input and Output

The original input source of this example is the Dubinksi 1995 data set from the N-body Data Archive web site. There are 81920 particles in the data set. To simplify and speedup benchmark programs, the ASCII data file is reformatted in to binary format. The binary data file contains a sequence of 32-bit floating point numbers following the appearance order of the original Dubinski data file. The simulation data file for the FPGA implementation contains hexadecimal numbers in ASCII format.

The output file has the same format as the input file which contains the updated status of particles after the simulation.

6.2 Single thread CPU Implementation

The reference implementation is a single thread C program targeting Intel x86 architecture. It has been compiled using GCC with the `-O3 -mfpmath=sse` optimization flags and tested on Linux. The reference implementation measures and displays the performance information using the *gettimeofday* function. The reference design reports $T_{comp} = 99.3s$ and $T_{total} = 99.5s$ for the simulation of 81920 particles in a single time step. The *compute time*, T_{comp}, is measured after all required data are ready for the PE (e.g. the CPU in this case) in the main memory and before the results are written back to file system. The *total time*, T_{total}, includes the file I/O time and device setup time in the cases of GPU and FPGA.

6.3 Multi-threaded CPU Implementation

OpenMP is used as the framework to parallelize the computation for a multi-core system. The parallel FOR `#pragma` directive is used to parallelize the main loop for computing the acceleration of all particles in the current time step. The particle data, marked as shared variables, are partitioned evenly according to the loop index. There is no data dependency between the acceleration computations of different particles. The results are written to the corresponding location in the acceleration array, which are also marked as shared. No write conflicts exist in this stage.

The update of position by current velocity and the update of velocity by computed acceleration are performed outside the OpenMP parallel construct. As in the single thread version, the program is compiled with the `-O3` and SSE flags.

In the Axel cluster, we run the OpenMP version of N-body simulation on a dual-CPU platform with 4 cores on each CPU. The program is instructed to run in 4 parallel threads such that the side effects of system loading are minimized.

The performance is measured in the same way as the single thread reference design and it reports $T_{comp} = 29.1s$ and $T_{total} = 29.3s$.

6.4 GPU Implementation

An implementation targeting many-core GPUs is created for the N-body simulation. The computation of acceleration of all particles in the current time step is off loaded to the GPU while the CPU updates the position and velocity.

Information of the particles needed for acceleration computation is copied to GPU memory at the beginning of each simulation time step. These include the mass and position in the format of (m, x, y, z). The acceleration vectors (ax, ay, az) are read back from the GPU memory after the kernel is finished.

In the kernel, we dedicate one thread per particle. Inside the thread, it loops through all other particles to generate the acceleration value and writes it to the GPU memory. The computation in the loop is the same as the source code of the CPU reference design. No special optimization is applied to the kernel and the data are stored in GPU main memory with the same layout as in CPU main memory.

The performance of this design is not as good as the highly optimized version of N-body simulation in the CUDA SDK. Here, we want to provide a fair comparison of the processing capability by restricting the same type and amount of computations targeting different processors. So the measured results are $T_{comp} = 9.26s$ and $T_{total} = 9.53s$.

For the same set of input vectors, the results of the GPU program are different from those of the CPU counterpart. This is due to the implementation deviations of floating point operators in each platform.

6.5 FPGA Implementation

This implementation requires that there are four independent memory banks connected to the FPGA each with 32-bit width data bus. The host program will put the values of m, x, y and z in separate memory banks so that they can be loaded to the FPGA in the same memory access cycle. The host program also sends auxiliary information to the FPGA core including the number of particles in memory N, the starting and ending index of i in the outer loop.

A deeply pipelined data path is created for the acceleration computation. All floating point operations are performed by macros generated from Xilinx Core Generator. The latency of the pipeline is 132 clock cycles and it can operate up to 400MHz according to the core specification.

Our FPGA implementation works as follows. First, the (x, y, z) values of P_i are read from memory and stored in internal registers. Then the <m, (x, y, z)> values of all particles are read as the P_j inputs. The P_j data are fed to the pipeline one per clock cycle when the data from memory read operation are valid. After all P_j are read, the internally stored values in the pipeline are drained. Extra clock cycles (56 in the example) are needed to perform the final reduction and output the internal values of the accumulator before starting the next iteration in the outer loop for a new P_i.

Further optimizations can be done by overlapping the draining and reduction phase of the current iteration and the feeding phase of the next iteration. However, this requires extra control logic resources and affects achievable maximum working frequency. We consider the current time overhead (192 clock cycles) is negligible since the number of particles, N, is sufficiently large. This time overhead is less than 0.1% when 81920 particles are processed as in the example.

Table 1: FPGA utilization: single core at 333MHz.

Resources	Utilization	%
Registers	32,483	15%
LUTs	27,674	13%
DSP48Es	18	9%

The final acceleration vectors (ax, ay, az) are stored in a FIFO where the host program can read through register interface of the FPGA (uint32_t fpgaReg[USER_REG]). The index of current P_i is also output such that the host program knows how many data should be read by comparing with its local index. The host program will update the position and velocity vectors after all acceleration vectors are read. A polling scheme with $3\mu s$ interval is used for minimizing both host CPU utilization and FPGA idle time by batch processing the FIFO data. The host-FPGA communication time overlaps with the FPGA computation time thus no further time overhead is introduced.

The theoretical performance of the N-body core without I/O constraints at 400MHz is:

$$T_t = (81920 + 192) * 81920 * 2.5ns = 16.82s. \qquad (6)$$

Note that the performance estimation from Equation 4 is not applicable since the above implementation contains multiplier blocks.

For 333MHz DDRII-SDRAM in the target platform, 171MBps DMA write speed and 230MBps DMA read speed, the expected performance of the FPGA implementation is:

$$\begin{aligned} T_e &= (81920 + 192) * 81920 * 3ns + \\ &\quad 2293760 bytes/171MBps \\ &= 20.18 + 0.13 = 20.21s. \qquad (7) \end{aligned}$$

The measured performance is $T_{comp} = 46.6s$ and $T_{total} = 48.9s$. The resource utilization of FPGA is listed in Table 1.

The difference between expected and measured performance is due to the memory read overhead. We store all particle data in external DDR2 memory with an effective $4\times$ 32-bit width interface. In estimating the expected performance, we assume that all data will be ready when needed. This is an uninterrupted continuous read operation for the external DDR2 memory with four 32-bit words per clock cycle. Due to the nature of the DDR2 memory and the implementation of the memory controller (command buffer), this assumption is not valid in real world operation.

For the same set of input vectors, the results of the FPGA program are different from those of the CPU. Two factors contribute to this deviation and both are related to the pipelined accumulator. First, the sequential CPU program accumulates the acceleration vectors once they are computed from P_i and P_j. In the FPGA design, these vectors are first accumulated into 12 different partial sums matching the pipeline stages in the floating point adder. These 12 partial sums are then added together in the final stage. The difference between accumulating order introduces the differences in final results. This factor is confirmed and analyzed by a

Table 2: FPGA utilization: 10 cores at 266.67MHz.

Resources	Utilization	%
Registers	117,393	56%
LUTs	92,219	44%
DSP48Es	180	93%

special CPU process which recreates the 12 partial sums as in the FPGA version. The second factor is the stall stage created by the external memory interface. During the clock cycles when the data from external DDR2 memory are not yet ready, the N-Body pipeline should be stalled. Using a single shared enable signal to stall the pipeline is not realistic since the fanout will grow over 10,0000 and the timing constraints will not be met. A delay pipe and input selection of the accumulators are used instead to avoid a global enable signal. This scheme changes the summing order of the acceleration vectors when invalid data are skipped in the accumulator input. Thus the final result will be different as previously discussed. This factor is non-deterministically affected by the state of the external DDR2 memory chip. It is analyzed by inserting invalid stages in RTL simulation.

Since we use only 3% of logic resources and 9% of the DSP blocks in the target FPGA for the pipelined data path, we can easily duplicate 10 copies of the pipelined data path in the FPGA core. This effectively unrolls the outer loop by a factor of 10. With several negligible extra memory read operations for the additional P_i values, the speed up with using multiple cores is linear as all pipelined cores share the same P_j values read from memory. Overheads also include the extra control logic for sequencing the P_i access and longer traces for connecting memory ports to multiple pipelined data path. Thus the maximum working frequency of the multi-core design will be lower than that of the single core counterpart.

At 266.67MHz, the measured performance of this 10-core FPGA design is $T_{comp} = 5.62s$ and $T_{total} = 8.43s$. This is 17.7 times faster than the single thread CPU implementation. Table 2 shows the FPGA resource utilization.

The FPGA and GPU designs in previous work [5] are estimated to take 216 seconds and 18 seconds respectively to complete a simulation step. They are around 25 times and 2 times slower than our 10-core FPGA design.

6.6 Collaborative Execution: One Node

To further improve the speed of N-body simulation, we implement a heterogeneous solution in which both FPGA and GPU work together to compute the acceleration vectors in Algorithm 1. The host programs of both GPU and FPGA are modified to accept specific range parameters of the portion of particles need to be processed. The two programs use a shared memory model to read the position and velocity vectors and write the acceleration vectors back in the corresponding offset in the array. A master thread will write the updated results to the output file after both FPGA and GPU have finished their assigned particles.

The FPGA program and the GPU program may not complete at the same time. Thus the master thread will have to wait for the slowest PE to finish. It is easy to configure the master thread to start working on the completed particles once the PE has finished their work. A better and

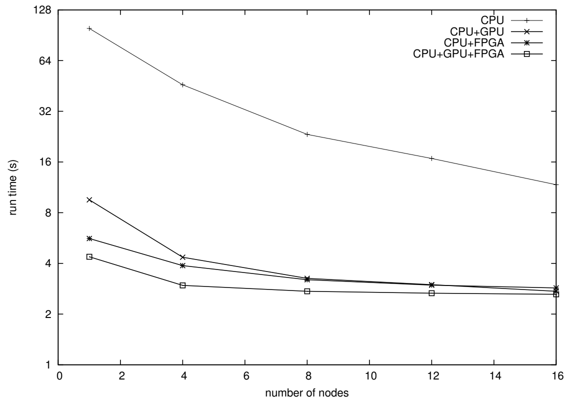


Figure 6: Performance of multi-node collaborative execution of N-body simulation.

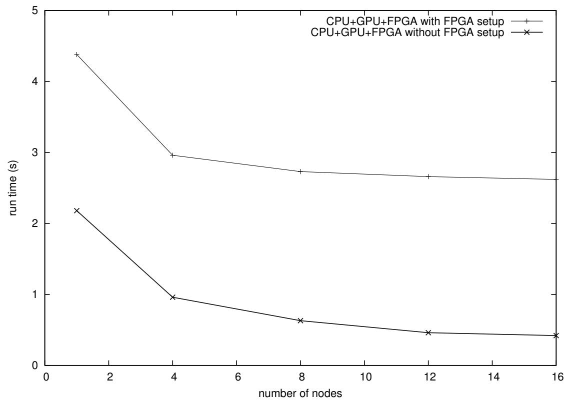


Figure 7: Performance of multi-node collaborative execution of N-body simulation.

simpler solution is to balance the work load of each PE such that they can finish processing the assigned particles at the same time. By combining the HAM in Section 4.1 and the measured results in the standalone FPGA/GPU implementations, we can assign 2/3 of the workload to FPGA and the rest 1/3 to GPU. This load balanced heterogeneous implementation can process the 81920 particles in less than 4 seconds compute time.

The overhead in this solution includes the synchronization between different processes since both the FPGA and GPU programs need to send messages reflecting their status to the master program.

6.7 Collaborative Execution: Multi-Node

The final scheme of speeding up the N-body simulation is to use all the resources available across the network. In this cluster implementation, the heterogeneous scheme in Section 6.6 is reused. The capability of communication to other nodes through MPI is added. The current communication channel is Gigabit Ethernet.

Each instance of the program runs on a single node exclusively. The master program reads the data and identifies the range of particles need to be processed in this node according to the MPI *Rank ID* and *group Size*. Since the configuration of each node is the same in the current system, the workload is distributed to the compute nodes evenly. After the local range of particles have been processed by both FPGA and GPU, the master thread calls the `MPI_Alltoall` function to distribute the local results to and receive results from all other nodes. In this example, only the acceleration vectors need to be distributed. The final results are written to a file by the program with MPI rank 0. Figure 6 shows the performance of the heterogeneous version and the CPU versions with just FPGAs or GPUs running on different number of nodes. It can be seen that for one node, the collaborative implementation is 22.7 times faster than the CPU only version; for 16 nodes, the improvement is 4.4 times.

For this result, we find that the major overhead in the heterogeneous cluster implementation is the FPGA setup time which takes around 2 seconds. This includes the time to download configuration files, initialising and locking multiple clocks and configuring the DDR memory interface. To have a better understanding of the actual performance scaling with number of nodes, we modify the experiment to skip the FPGA initialization process, assuming that they are preloaded with correct configuration in advance; the resulting performance is shown in Figure 7.

In general, it is impossible to preload the FPGA in advance since the nodes are dynamically allocated according to the current cluster status. This overhead can become insignificant if multiple time steps are considered in the simulation. In both figures, the performance does not linearly scale with the number of nodes used. This is due to the communication overhead between nodes. The `Alltoall` function call has N^2 bandwidth requirements. In other applications without the need for the all-to-all communication pattern, the performance should scale more linearly.

7. CONCLUSION

Future cluster technology and cloud computing require high performance at an affordable cost. We have described an approach which makes use of multiple types of off-the-shelf hardware accelerators in a scalable and flexible system. The combination of such systems with an appropriate programming model such as Map-Reduce results in a promising platform for cost-effective high-performance computing.

In this work, we perform experiments on our Axel cluster utilizing various kinds of processing elements including FPGA, GPU and CPU. The proposed framework has been shown to map the N-body simulation process to the underlying hardware efficiently. From the results of the experiments, we achieve significant speed up using a heterogeneous combination of PEs across the cluster network for a practical simulation process.

The 10-core FPGA design is the fastest implementation among the three standalone designs. It also achieves better energy efficiency than GPU and CPU. The largest drawback of FPGA design is the difficulty of implementation. In contrast to the seconds of compile time for CUDA and the less than 1 day development time of the GPU program, the 5 hours place and route time and over a month development time of the FPGA seems unattractive.

Current and future work involves finding ways to speed up the FPGA design and implementation time, possibly by parallelizing the implementation process so that a heterogeneous cluster can be used in optimising its own executables. Techniques for automating performance estimation and code generation for multiple accelerators for various applications will also be investigated.

8. 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.

9. REFERENCES

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

[2] Chen Chang et al. BEE2: a high-end reconfigurable computing system. *Design & Test of Computers, IEEE*, 22(2):114–125, March-April 2005.

[3] T. Endo and S. Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS'08*, pages 1–10, 2008.

[4] M. Flynn, R. Dimond, O. Mencer, and O. Pell. Finding speedup in parallel processors. In *Proc. Int. Symp. on Parallel and Distributed Computing ISPDC'08*, pages 3–7, 2008.

[5] J.H.C. Yeung et al. Map-reduce as a programming model for custom computing machines. In *16th International Symposium on Field-Programmable Custom Computing Machines, 2008. FCCM'08*, pages 149–159, April 2008.

[6] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

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

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

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

[10] Nallatech. *DIMEtalk V3.0 Application Development Environment*, 2009.

[11] nVidia. *nVidia CUDA Programming Guide v2.1*, 2008.

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

[13] R. Baxter et al. Maxwell - a 64 FPGA supercomputer. In *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 287–294, 2007.

[14] R. Sass et al. Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing. In *15th International Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM'07*, pages 127–140, April 2007.

[15] L. Ralf. Google's MapReduce programming model – revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

[16] SRC Computers, LLC. *SRC-7 MAPstation*, 2009.

[17] D. Strenski. The cray XD1 computer and its reconfigurable architecture. Technical report, Cray Inc., July 2005.

[18] T. El-Ghazawi et al. The promise of high-performance reconfigurable computing. In *Computer*, pages 69–76, February 2008.