A Coarse-grained Stream Architecture for Cryo-electron Microscopy Images 3D Reconstruction

Wendi Wang[†‡§], Bo Duan[†‡], Wen Tang[†‡], Chunming Zhang[†], Guangming Tan[†¶], Peiheng Zhang[†], Ninghui Sun[†¶]

†High Performance Computer Research Center, Institute of Computing Technology, CAS
¶State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
‡Graduate University of Chinese Academy of Sciences
{wangwendi, duanbo, tangwen, zhangchunming, zph}@ncic.ac.cn, {tgm, snh}@ict.ac.cn

ABSTRACT

The wide acceptance and the data deluge in the bioinformatics and medical imaging processing require more efficient and application-specific systems to be built. Due to the recent advances in FPGAs technologies, there has been a resurgence in research aimed at the design of special-purpose accelerators for standard computer architectures. In this paper, we exploit this trend towards FPGA-based accelerator design and provide a proof-of-concept and comprehensive case study on FPGA-based accelerator design for a single-particle 3D reconstruction application in single-precision floating-point format. The proposed stream architecture is built by first offloading computing-intensive software kernels to dedicated hardware modules, which emphasizes the importance of optimizing computing dominated data access patterns. Then configurable computing streams are constructed by arranging the hardware modules and bypass channels to form a linear deep pipeline. The efficiency of the proposed stream architecture is justified by the reported 2.54 times speedup over a 4-cores CPU. In terms of power efficiency, our FPGA-based accelerator introduces a 7.33 and 3.4 times improvement over a 4-cores CPU and an up-to-date GPU device, respectively.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Signal processing systems

General Terms

Design, Performance

Keywords

Cryo-electron microscopy, FFT, FPGA, stream processing, memory access patterns

[§]Corresponding Author: Wendi WANG, No.6 KeXueYuan South Road, ZhongGuanCun, Beijing, P. R. China, 100190.

FPGA'12, February 22–24, 2012, Monterey, California, USA.

1. INTRODUCTION

Due to the limiting factors of the memory wall and the power wall, in recent years, the performance gains from general-purpose CPUs are diminishing. Using of alternative devices rather than general-purpose CPUs to accelerate computing-intensive applications has gained renewed interests. By introducing specialized co-processors, configurable data path and customized memory system with respect to applications' characteristics, application-specific hardware design provides a promising way to tweak performance within a given silicon budget.

However, limited by the device capacity and the high implementation cost, the typical uses of FPGAs are usually restricted to either simple and small applications, such as RSA and FIR, or key kernels of applications, such as the molecular dynamics [1]. Recently, due to the advances in modern FPGA technologies and the emergence of fast floating-point and elementary function libraries [2], there has been a resurgence in research aimed at accelerator design that leverages FPGAs to accelerate large-scale scientific applications. We exploit this trend towards FPGA-based accelerator design and propose a data path configurable stream architecture that focuses on separated design strategies for computing and memory flows. In particular, we introduce our work on accelerating a large-scale scientific application, called *EMAN* [3], which is an open source software package for single-particle 3D reconstruction from Cryo-electron microscopy images, on our customized FPGA accelerator card.

EMAN is composed of hundreds of time-consuming kernels showing diverse computing features. How to integrate kernel implementations under a unified framework determines the extent to which the application can be accelerated on FPGA. One the other side, the overall execution time of applications is often dominated by the efficiency of their data access patterns [4]. One of the main complexities for application-specific memory system design is how to support the kernel dominated data access patterns. An alternative solution to this problem is to take into account how to share data across kernels. However, the integration of multiple kernels with various functions under a unified framework and providing an efficient data exchange mechanism among them are both proven to be challenges. To address these concerns, we introduce a hybrid memory controller, which is characterized by its support for explicit pattern-based data accesses. The memory system can be viewed as both a

framework for creating data access patterns and a runtime system that assists the construction of data flows.

Our stream architecture is designed by first offloading computing-intensive software kernels to dedicated hardware modules. Then configurable computing streams can be constructed by arranging the hardware modules and bypass channels sequentially. Various software functions can be implemented by a single computing stream by activating different hardware modules. For complex work, the computing flow needs to be executed and emulated in a multiple step procedure, in which the data path of each step differs. This work provides a proof-of-concept and comprehensive case study on FPGA-based accelerator design for a large-scale scientific application, which emphasizes the importance of optimizing data access behaviors. The contribution of this work is three-fold:

- We propose a coarse-grained stream architecture for accelerating a real complex application. The stream architecture is designed based on the key observations of classification of various kernels. Built upon typical computing and data access modules, the modularized design method improves coarse-grained modular reuse.
- We apply software-hardware co-design approach to build an efficient map between EMAN algorithm and FPGA architecture. Especially, we develop a novel hybrid memory controller featuring the ability to do computing dominated and pattern-based data accesses.
- Our customized accelerator is implemented as a FPGA accelerator card. The efficiency and feasibility of the proposed architecture are justified by the reported 2.54 times speedup over a 4-cores CPU. Comparing to a GPU-CUDA based version, our accelerator improves power cost by 3.4 times.

The rest of this paper is organized as follows. The background of single-particle 3D reconstruction and *EMAN* are introduced in Section II, followed by introduction to system design in Section III. Section IV explains the software and hardware co-design flow. The experimental results are discussed in Section V, whereas Section VI lists related work. Finally, we conclude our work in Section VII.

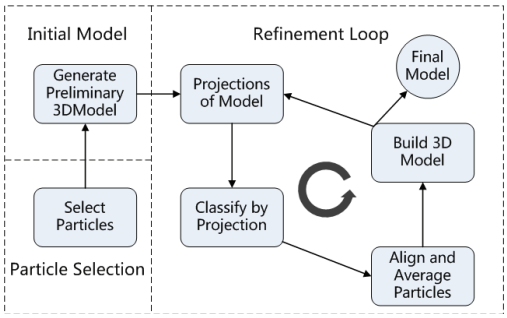


Figure 1: Flowchart of the Cryo-EM image analysis

2. SINGLE-PARTICLE 3D RECONSTRUCTION AND EMAN

EMAN is a software package designed to handle nearly all aspects of the single-particle 3D reconstruction, such as particle selection, particle alignment, 3D model projection/reconstruction and etc. The single-precision floating-point format is used to store input data and intermediate result. In single-particle 3D reconstruction for Cryo-EM images, the algorithm first generates a preliminary 3D model based on amount of particles (images), which are selected from scanned raw micrographs. The preliminary model is used as the starting point for the refinement loop. A refinement procedure for the final 3D image is a model-based iteration. True convergence is achieved when the model remains unchanged for several successive iterations. The refinement loop outlined in the right half of Figure 1.

Algorithm 1: Reconstruction Algorithm

```
1  while model T not converged do
2      generating M projections of T;
       // particle classification
3      foreach i ∈ N particles in images do
4          for j ∈ M projections do
               // rotationally and translationally aligns
               // each particles to projected reference
5              RTFAlign(i, j);
       // class averaging
6      foreach particles i in class j do
7          RTFAlign(i, average_j);
       // generate an initial class average
8      InitialAve();
9      foreach particles i do
10         RTFAlign(i, average_initial);
       // remove particles with less similarity
11     Remove();
12     Build3D(); // build 3D model
```

Algorithm 2: RTFAlign Algorithm

```
   // step 1. rotational alignment
1  MCF (i); // autocorrelation function
2  MCF (i'); // flip's autocorrelation function
3  unwrap (); // rectangular coordination to polar one
4  CCFX (); // cross correlation function on x-dimension
5  Rotate (); // rotates with the best rotation angle
   // step 2. translational alignment
6  CCF (); // cross correlation function with reference
7  Translate (); // translates image with the maximal CCF
   // step 3. score and select the most similarity one
8  dot (); // scores the rotational&translational alignment
```

Algorithm 1 illustrates the pseudo-code of the refinement algorithm. The iteration of refinement is a sequential process because current refinement depends on the model generated in previous iteration. However, abundant of parallelism is observed within each refinement procedure. Since the operations of classification and averaging are similar, the same parallelization can be applied. For simplicity of presentation, we only present the analysis of classification (line 3-5). Note that a common procedure of particle classification is rotational and translational alignment (RTFAlign in Algorithm 2), which occupies over 95% of total execution time. Therefore, in this paper we focus the discussion on FPGA acceleration of the particle alignment only.

Even accounts for only a small fraction of *EMAN*, particle alignment still contains hundreds of kernels and tens of parameters. However, based on partial evaluation and runtime profiling, we can classify the 23 identified kernels into

Table 1: Kernel categorization.

Name	Category	Description
MCF	Computing	Autocorrelation
CCFX	Computing	Cross correlation on x-dimension
CCF	Computing	Cross correlation with reference
Unwrap	Computing	Rectangular coordination to polar one
Rotate	Computing	Rotates with the best rotation angle
Translate	Computing	Translates image with the maximal CCF
DOT	Computing	Scores the rotational&translational alignment
Clip & Zero Padding	Memory	Change the size of images by clipping and padding
Rot180	Memory	Rotate image by 180°
hFlip	Memory	Flip images horizontally
Shift	Memory	Translate image by given 2D offset
Bit Reversal & Transposition	Memory	Used in FFT kernels
Matrix Transposition	Memory	Used in the row-column 2D FFT kernels
RTFAlign	Stream	Align images using RTAlign and hFlip
RTAlign	Stream	Align images using Rotate, Translate, CCFX, CCF and DOT
MakeRFP	Stream	Calculate the rotating footprint using MCF and Unwrap

3 categories: computing kernels, memory kernels and flow kernels. For simplicity of presentation, as given in Table 1, we combine some computing kernels into one kernel, thus reduce the number of computing kernels to 7.

3. MODULE-BASED SYSTEM DESIGN

A key observation underlying our stream architecture is that the kernels of *EMAN* can be broadly classified into 3 categories: computing, memory and stream.

The design of our system is geared towards architecture support that assists the implementation of each kernel category. The kernels in *EMAN* will be mapped to dedicated hardware modules on FPGA. Computing modules are used to wrap arithmetic operations, whereas memory modules are used to implement data accesses as well as related address calculation. The stream module, which incarnates concrete program functions, is composed of several computing and memory modules. In following sections, the stream architecture will be discussed in detail, which covers the topic of how to extract and classify target kernels for acceleration from *EMAN*, how to implement each kind of hardware module and how to map a complex computing flows to a stream module in hardware.

3.1 Kernel classification

There are hundreds of kernels in *EMAN*, along with computing-intensive kernels, there also exist kernels that merely containing either trivial data accesses or kernel invocations. Table 1 summarizes the data-parallel kernels in *EMAN* and the workload descriptions.

- **Memory**: Memory kernels are of limited use in their own right, by which we mean that they must be combined with computing kernels to deliver on practical functions. However, the efficiency of memory access can exert profound impacts on overall system performance. For example, FFT is the only kernel that uses the bit reversal memory access pattern, which gives poor data locality on CPU. The data access patterns, which are extracted by analyzing the memory address sequences of computing kernels, will be implemented with a unified data flow module (DFM). By introducing the DFMs, it becomes possible for computing modules to do pattern-based data accesses. The topic of how to extract data access patterns is covered in Section 3.3, whereas Section 4.3 introduces the DFM in detail.

- **Computing**: With respect to loop boundaries, computing kernels are manually composed of one or more loop statements. Instead of implementing them as heterogeneous and inflexible modules that directly work on raw memory controller interface, two additional steps are used to facilitate the implementation of computing kernels: 1) Data access patterns are expressed in the form of either fix-step counters (regular patterns) or mapping tables between iteration indices and requested memory addresses (irregular patterns); 2) All data access patterns will be implemented collectively with the DFM introduced previously. Section 3.4 discusses various issues related to kernel implantation.

- **Stream**: Stream kernels provide the specification of assembling computing and memory kernels to form large and complex computing streams. The computing flow of *EMAN* is relatively simple and regular, which makes it possible to construct computing streams as linear pipelines. Section 3.4.3 gives a concrete example of implementing the *MakeRFP* stream kernel.

3.2 System Architecture

Before we introduce the hardware implantation in detail, let us first take a glimpse of the system architecture. Figure 2 gives a broad view of our stream architecture, on which the aforementioned 3 kernel categories will be implemented. The core of the system is a hybrid memory controller, whereby on-chip and off-chip memory devices are managed with a single virtual address space. Different memory devices can be explicitly accessed by specifying exclusive memory addresses. A unique feature of the hybrid memory controller is the inclusion of 2 dedicated data flow modules (DFMs), which can be used to map various data access patterns extracted from computing kernels. Intuitively, the overhead of data transfer can be hidden by overlapping them with the computing. In order to achieve this overlap, we integrate a data pre-fetch unit in the memory controller, which can be

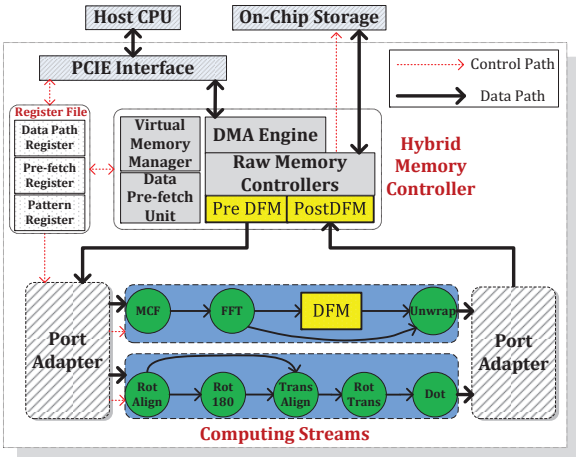


Figure 2: System architecture overview.

configured to do forward as well as backward pre-fetching at the granularity of image line. Section 4.2 gives more details about the hybrid memory controller.

The configurable computing streams are constructed by arranging the hardware modules and bypass channels to form a linear deep pipeline. Figure 4 illustrates the proposed computing stream to accelerate the RTAlign kernel. Along with two DFMs that lie in the memory controller, DFM can also be placed in the computing stream. For example, in Figure 4, there are two data re-order modules (used for data flow permutations) in the 2D FFT kernel. Multiple computing streams can coexist with each other, the port adapters are used to switch the data path between memory controller and computing streams.

Our system is configurable in two aspects by means of writing corresponding control registers: 1) The data path of the computing stream can be controlled to bypass some stages. As a result, a computing stream can be used to implement various program functions; 2) By configuring the DFMs and the data per-fetch unit, the memory controller can be controlled to do pattern-based data access.

3.3 Separating computing flow from data flow

In practice, most applications, including *EMAN*, are rarely developed to take the aforementioned 3 kernel types into consideration. It is the responsibility of the compilers and the cache system to optimize the computing and data accesses. Therefore, in most applications, it is a common phenomenon that computing and data accesses are entangled with each other.

The separation between computing flow and data flow manifests itself across two dimensions: 1) The performance of applications is deeply influenced by the ability to overlap computing with data accesses [4], by separating data flow from the computing flow, some advanced data pre-fetching and reordering strategies can be utilized; 2) A lot of data accesses of an application are highly structured, for example, data accesses within loops are often subscripted by loop indices, which results in either sequential or stride data access patterns. The cost to implement computing kernels can be greatly reduced by insulating memory read/write operations for separate consideration. In this way, computing kernels can be abstracted as black boxes with FIFO data in and FIFO results out. However, at the first place, extracting data access patterns from a large computing flow is still a challenge. The good news is that, the kernel partitioning strategy (in Section 4.1.2) leads to dividing the program in loop statements. Therefore, the analysis of data access patterns can be simplified and confined within loop statements.

We developed a LLVM loop pass to facilitate the analysis process. Data blocks (images) will be clustered into arrays with the data layout optimization (in Section 4.1.3), which makes it possible to use memory addresses to identify data accesses to specific data types. By recording the memory addresses issued in loop statements, regular and irregular data access patterns can be respectively expressed as fixed-step counters and lookup tables that map iteration indices to the memory addresses in loop statement. Therefore, the memory addresses will be calculated statically, and computing kernels can dispense with the overhead of data accesses and get a simplified view of the computing flow. Computing kernels, which contain stateless loops (such as vector addition) only, can be implemented as simple arithmetic modules that operates on a continues data flow. However, it is still need to maintain registers and loop counters for stateful loops with loop-carried dependency (vector summation) and some initialization logics (image filters). In our system, data accesses and related memory address calculations will be offloaded to the data flow modules (DFMs) in memory controller, which pre-fetch, reorder and push data to each computing kernel in its expected order. Computing stream construction can be simplified as a process of instantiating computing kernels and selecting corresponding data access patterns supported by DFMs in memory controller.

Figure 3: Interface of hardware computing modules.

3.4 Kernel implementation

Due to the complexity of *EMAN*, as explained in previous sections, it is unpractical for us to implement the entire computing flow as a single unit. On the other hand, considering the high design and debugging cost of using FPGAs, we argue that it is not a cost effective way to implement some kernels of *EMAN* on FPGAs, such as 3D-FFTs, heuristic particle selection and etc. Therefore, the performance of the FPGA-based accelerator will be dictated to a large extent by the ability to evaluate, extract and offload the most beneficial parts of the application. Runtime profiling indicates that, the *RTAlign* kernel that occupies over 95% of total execution time is the foundation, on which the process of particle classification (*Classesbymra*) and alignment (*Classalign2*) are built. In following part, we limit our discussion to FPGA acceleration of the *RTAlign* kernel only.

In follow sections, three representative kernels, mixed radix 2D FFT, matrix rotation by 180°and the RTAlign stream kernel will be used to demonstrate how to implement each kind of software kernel in our system.

3.4.1 *Computing kernel*

The computing kernel implementation is built upon a one-

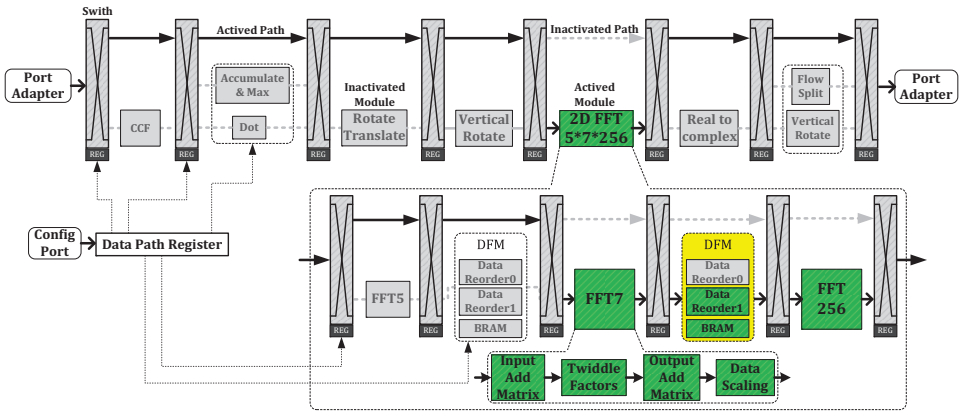


Figure 4: Computing stream for accelerating the *RTAlign* kernel.

to-one correspondence between the loop statements and the computing module (circle node) in Figure 2. In order to reduce design cost and increase modular reuse, the computing modules are implemented and wrapped in a unified interface. Illustrated in Figure 3, the unified interface is composed of standard FIFO signals, flow control signals and other signals used for kernel configuration. The advantage of partitioning and implementing the computing flow as separated kernels with common interface is that it gives us many coarse-grained building blocks that can be flexibly composed to form complex computing streams.

Based on the unified module interface, we developed an automatic HDL core generator framework with Matlab, which can be used to translate given data flow graph (DFGs) of loop statements (generated by GCC in the DOT file format) to HDL implementations. First, operators of the DFG are scheduled to different pipeline stages with respect to data dependency. Next, data flow permutations between stages will be extracted and expressed as sparse matrix, in which the number of rows and columns corresponds to the number of input and output variables of each pipeline stage, respectively. Finally, the pipelined DFG will be used to generate the Verilog HDL descriptions. The strength of the proposed core generator is justified by the ability to generate prime-radix FFT cores in our previous work [13]. There are plenty of existing work related to FFT design on FPGAs, however the issues of prime-radix FFT, 2D FFT and real FFT are rarely discussed. Therefore, it is needed to implement the FFT kernel by ourselves.

The data path of the computing stream can be configured by configuring the bypass switches in Figure 4. Therefore, different factorizations of a FFT process will be mapped to different paths flow through the computing stream. For example, the lower part of Figure 4 gives the structure of a 8960-points FFT pipeline, which is degraded to execute 1792-points FFT in current configuration. In current work, partial reconfiguration of individual hardware modules is not yet considered. With trivial overhead of data path switching, modules can be shared across configuration. We argue that with proper organization of the computing flow, the frequency of reconfiguration can be minimized or avoided.

3.4.2 Memory kernel

In order to improve alignment sensitivity, along with the reference particle, input images will be compared with up to 3 mirror images in *EMAN*. In CPU implementation, these mirror images are generated by time-consuming memory copies, which introduce misses in data cache and exert severe impact on application performance. With the support of our memory controller, the Rot180 kernel in Table 1 can be implemented in following steps: 1) Backward pre-fetch image line-by-line; 2) Buffer an entire line; and 3) Output the buffered line in reverse order. In this way, only one copy of reference image is stored in memory, other mirror images can be generated on-the-fly.

Table 2: Computing steps of the *MakeRFP* kernel.

#	Function	Activated Modules	Memory Controller[1]
1	1D FFT	2D FFT	I: interleave/ O: column write/ deinterleave
2	1D FFT	2D FFT/ Post-Vertical Rotate	O: column write/ deinterleave
6	Filter/MCF	CCF	I: interleave/2-Op.[2] O: column write
4	1D IFFT	2D FFT/ Pre-Vertical Rotate	O: column write
5	1D IFFT	R2C	I: interleave
6	1D IFFT	2D FFT/Flow Split	
7	Unwrap	RotateTranslate	I: clip/random access

[1]I for input, O for output memory access patterns.
[2]Reading two operands from two separated addresses.

3.4.3 Stream kernel

Figure 4 illustrates the computing stream built to accelerate the *RTAlign* kernel. The computing and memory modules are separated by runtime configurable bypass switches. The data flow modules (DFM) in the 2D FFT kernel are used to support data flow permutations between FFT stages. Along with the control signals for function selection (for example, the module for dot production can also be configured to do accumulation), signals for switch configuration are wrapped in runtime configurable registers. Different functions can be fulfilled by activating the required modules while bypassing the others.

If a computing flow is complex and cannot be implemented with a single data path configuration, the function can be emulated by means of activating the computing stream multiple times with a different data path each time. For ex-

ample, it is needed to configure the stream pipeline 6 times to implement the *MakeRFP* kernel. Table 2 gives the function of each step, the activated modules and the involved data access patterns in detail. The 2D FFT/IFFT process is based on the row-column algorithm, which requires invoking the 1D FFT module twice.

4. HARDWARE/SOFTWARE CO-DESIGN

In order to achieve the best performance, the original computing flow of *EMAN* needs to be redesigned to be aware of the architecture features of FPGAs. We resort to a software and hardware co-design flow, which addresses the problem from both the software side (Section 4.1) and the hardware side (Section 4.2), to deliver on our performance goals.

4.1 Application redesigning

The computing flow of *EMAN* can be redesigned as a phased computing stream, which coincides with the fine-grained configurability of the FPGA and enables the possibilities to build coarse-grained deep pipeline for achieving high spatial parallelism. However, *EMAN* is largely written with C++ language, which heavily relies on templates, objects and pointers. These language features are great for high-level application design; however, they are in conflict with the semantics and syntax of Hardware Description Language (HDL), on which the FPGA designs are built. Therefore, it is needed to introduce computing flow modification that removes all high-level language features that got in the way of FPGA acceleration. On the other hand, we rely on loop fusion/fission and data layout optimizations to reduce the number of kernel stages and improve data locality, respectively.

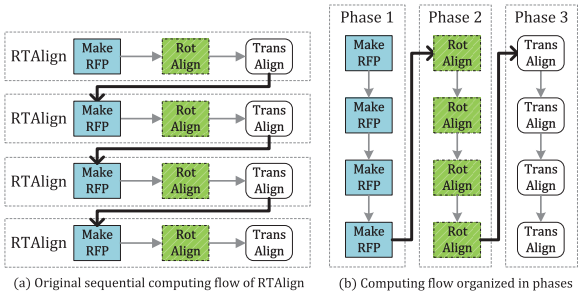


Figure 5: Modification on computing flow. The modified computing flow is organized in phases.

4.1.1 Computing flow optimization

The optimization of the computing flow consists of 3 steps: 1) Code rewriting that removes high-level language features and unreachable code (different computing paths); 2) Explicitly memory allocation that determines the input and output addresses for kernel invocations; 3) With respect to function boundaries, the original computing flow will be divided into phases. Among the motivations for computing flow partitioning, here are 2 that particularly stand out. First, a small FPGA can be used to simulate a large computing flow that otherwise would be too big to fit in. Only a part of a large computing flow needs to be presented on FPGA each time, the area consumption can be reduced considerably. Second, for a periodic computing flow (loop statements), the overhead of pipeline fill/drain as well as kernel invocation can be reduced. For example, as illustrated in Figure 5(a), the computing flow of *RTAlign* consists of 3 functions: *MakeRFP*, *RotAlign* and *TransAlign*, which will be invoked periodically and sequentially for each image. By applying the computing flow partitioning, Figure 5(b) shows that it is possible to invoke each kernel only once. As a result, if each of the 3 kernels needs a separated configuration, the number of FPGA configuration can be reduced from 12 to 3. The importance of doing computing flow partitioning will be further justified in following section.

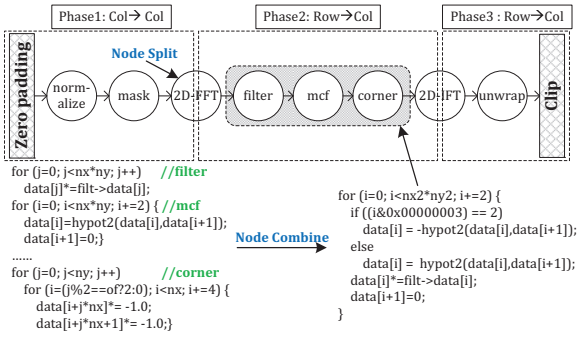


Figure 6: Coarse-grained DFG of *makeRFP* and the illustration of node split and node combine.

4.1.2 Kernel partitioning

By applying the modifications in previous section, the computing flow will be divided into separated phases that contain one or more loop statements. Each computing phase will be transformed to a data flow graph (DFG) at the granularity of loop statements, and finally mapped to a computing module in Figure 2. Illustrated in Figure 6, in order to reduce the number of stages of computing phase, we apply the well-accepted optimization of loop fusion and fission, which corresponds to node splitting and node combining of the computing flow, respectively.

Limited by the size of on-chip memory, it is important to take into account of fine-grained partitioning strategies within computing kernels. For example, we applied the row-column algorithm to implement 2D FFT/IFFT process, in which row-oriented 1D FFTs are executed before moving to column-oriented 1D FFTs. Therefore, it is need to cache the entire image on-chip (for an image of 512×512 32 bit pixels, it corresponds to 1 MB storage) to implement a fully pipelined computing module, which easily exceeds the on-chip memory limitation of the FPGA and renders this approach impractical. Motivated from the fact that images are processed in batch, the overhead of splitting the 2D FFT/IFFT nodes can be amortized. Therefore, as illustrated in Figure 6, the computing flow of *MakeRFP* can be further partitioned into 3 phases. On the other hand, it is beneficial to combine adjacent loop statements to get more compact computing nodes. For example, each circle in Figure 6 represents a loop statement (nested loop is allowed) in source code. The 3 loop statements in phase 2 can be merged as a single loop with a trivial branch.

4.1.3 Data layout optimization

To facilitate the analysis of data access patterns, as illustrated in Figure 7, we propose an optimization to improve data layout and structure in *EMAN*. Along with some housekeeping parameters, the time domain data, the frequency

domain data and the rotational footprint (RFP calculated on-the-fly by *MakeRFP*), are uniformly wrapped in the same *EMData* object. In original *EMAN*, different data blocks are scattered in memory space. Considering the phased computing flow introduced previously, it easier to justify spending an optimization step that clusters data by their types. By collecting separated data with the same type into continuous arrays, it becomes straightforward to map data blocks to fixed addresses in FPGA address space.

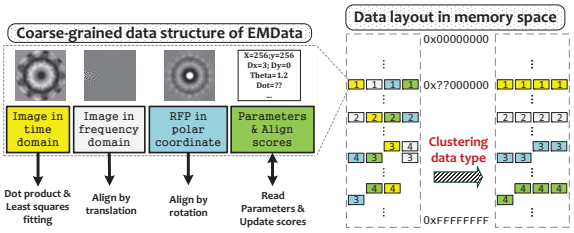


Figure 7: Data structure and layout of EMAN.

4.2 Memory system design

For applications, such as *EMAN*, that can be expressed at the granularity of loop statements and are data-intensive, investigating the possibilities of pattern-based data accesses is of paramount importance. The performance of the FPGA-based accelerator is largely dominated by the ability to support various data access patterns. The problem is compounded by the fact that data access patterns are usually entangled with the computing flow (address calculation based on loop indices). Our response to these concerns is a hybrid memory controller, which is designed to supported pattern-based data accesses. The memory controller is both a framework for creating modularized data access functionalities as well as a dynamic runtime that assist the construction of the data path.

The memory controller used in our system is a hybrid one, in which off-chip DRAM, SDRAM and parts of the on-chip BRAMs are managed with a single virtual address space. From the viewpoint of high-level users, data accesses on each kind of storage can be controlled explicitly by invoking write/read operations at the corresponding addresses in memory space. The configuration of the virtual memory space is controlled by software and can be changed online. In a typical memory space configuration, the lower address (512 KB) are allocated to on-chip BRAM followed by off-chip SDRAM (16 MB) and DRAM (2 GB). The on-chip BRAMs are used as a scratchpad memory to store small intermediate data (scalar and vector) between consecutive pipeline stages. In addition, the BRAMs can also be used to build configurable data flow modules (DFM), which will be used to handle data flow permutations, such as the (de-)interleave operations in Table 1. The SDRAM is further divided into two parts, which are used respectively for data pre-fetching and intermediate results buffering. The final results will be offloaded to the off-chip DRAM and transferred back to host CPU via DMA through PCIe. The data bandwidth of the off-chip DRAM and SDRM is 3.2 GB and 6.4 GB, while the system bottleneck lies in the PCIe interface, which is only 2 GB.

4.3 Configurable data flow module

We rely on a data flow module, which is built upon on-chip BRAMs, address generator, configurable counter and flow control logic, to support pattern-based data accesses. As illustrated in Figure 8, the 4 BRAMs are organized into 2 groups and configured to work as a Ping-Pong buffer. When one group is buffering data in current phase, the other group will be used to provide data that are buffered in previous phase. The data rate between input and output of the computing stream may become unmatched; therefore, it is necessary to provide a backward flow control mechanism, which is implemented by monitoring the back-pressure signals of the computing stream.

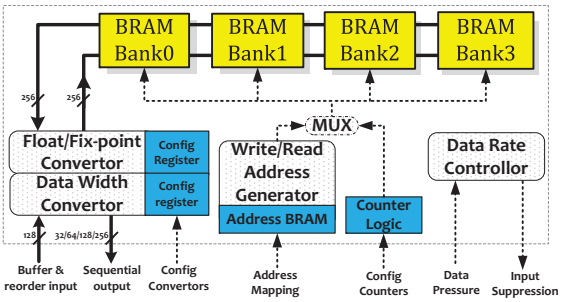


Figure 8: Data flow module with configurable address generation, data format conversion, configurable counter and flow control logic.

The extracted data access patterns are used to configure the DFMs as well as the data pre-fetching unit in memory controller. In particular, the DFMs can be controlled by varying address generation logic, data width, port number and data format converter. For example, in Figure 8, data access patterns in form of fixed-step counters (regular data access patterns) will be implemented with the counter logic by means of setting the counter range and step. In contrast, the address mapping arrays (irregular data access patterns) will be used to load the address BRAM in the write/read address generator. Various data access patterns can be achieved by configuring the address BRAM with different address sequences.

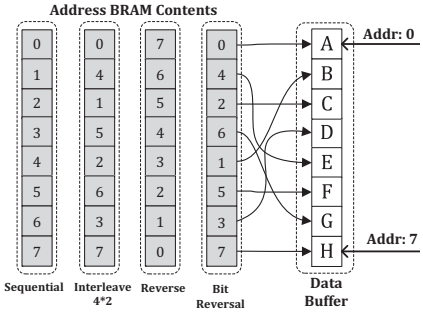


Figure 9: Address mapping in DFM. The contents are calculated off-line and used to load the address BRAM.

Take the computing steps to implement the *MakeRFP* kernel in Table 1 as an example, the DFMs in memory controller can be used to carry out various data access patterns. The process of interleaving two image lines with a length N can be achieved as follows: 1) Configuring the data pre-fetch unit in memory controller to read two lines of image each time; 2) Storing the pre-fecthed data in a BRAM group in DFM; 3) As illustrated in Figure 9, the address BRAM in

the write/read address generator, which is loaded with the interleave $N \times 2$ (N is the width of an image line) content in advance, is used to generate addresses to read the data buffer. In this way, two image lines can be interleaved by reading the data buffer with respect to the content in the address BRAM. Along with the 4 address contents illustrated in Figure 9, new data access patterns can be generated on-the-fly. The primary drawback of using the indirect address mapping is that it introduces extra on-chip storage overhead. However, for this problem, the length of the address sequence is relative short, which makes the cost of additional storage affordable. For example, the longest address sequence in our system comes from the 8960 FFT module, which requires an on-chip storage as large as 15.312 KB ($\lceil \log(8960) \rceil \times 8960$).

The ability to do column-oriented writing is the major factor that contributes to the performance speedup of our system. However, the data path of the DRAM (128 bit) and SDRAM (256 bit) are both wider than current computing data path (64 bit). The DFM is used, on one side, to mitigate the data path differences by buffering 2 (4) columns before offload data to DRAM (SDRAM). On the other hand, the DFM is used to generate the column-oriented writing addresses, which can be easily mapped to the counter logic with modulo N (pixel number of an image) operation.

5. EXPERIMENT RESULTS

5.1 System and Experiment Setup

The proposed stream architecture is implemented and evaluated on our customized FPGA accelerator card. Along with various off-chip memory storage, the accelerator card, which communicates with the host CPU via PCIe gen1 interface, is composed of two FPGA chips. The computing stream is implemented on the computing node (Xilinx XC5VLX330-1), while functions related to configuration and data transferring are offloaded to the control node (Xilinx XC5VLX70T). The performance and area of the accelerator design on computing node are evaluated with Xilinx ISE 11.4 [18].

The CPU-based *EMAN* (single-threaded) is evaluated on a 4-cores 2.27GHz Intel Xeon E5520 and compiled with the latest Intel compiler. The sequential EMAM program is parallelized to 4 threads with the OpenMP. With our best effort, *EMAN* is also optimized on two GPU devices: the GeForce 8800 (G80) and the Fermi Tesla C2050 (GTX480). The G80 consists of 16 streaming multiprocessors (SMs), which contain eight streaming processors (SPs) each. The SPs within a SM run at 1.35 GHz and share a 16 KB on-chip memory, whereas all SMs share the 1 GB global off-chip memory. The GTX480 has 448 cores organized into 14 SMs, which can concurrently execute multiple warp blocks. Each SM in GTX480 has a set of registers and a 64 KB local storage, which can be configured as 16 KB L1-cache and 48 KB shared memory. All SMs share a unified 768 KB L2-cache and the 3 GB global memory.

Full system power is measured with the Fluke Norma 4000 Power Analyzer [19]. Similar as the method proposed in [16], we only consider the dynamic power, which is measured as difference between active and idle power.

5.2 Kernel performance

Figure 10 compares the execution time of the kernels listed in Table 1. Measured in terms of speedup over single-threaded CPU, the speedup of kernel implementation on FPGA varies from 2 times (Max) to 12 times ($CCFX$). We observe that kernels can achieve different speedup even with a similar computing flow. For example, the *Rotate* and $Unwrap$ kernel in Table 1 exhibit a similar computing flow. The coordinate transformation of $Unwrap$, however, spans only one half of an image, which contributes to the reported lower speedup when compared with *Rotate*.

The performance of GPU can be greatly influenced by the data parallelism degree. For example, due to the inefficiency of parallelism degree, the CCFX kernel that executes only 1D FFTs achieves a low speedup on GPUs when compared with the MCF and CCF kernels that contain extensive 2D FFTs. Due to the advantages of frequency and bandwidth of the GDDR memory, except for 2 kernels ($CCFX$ and $Translate$), the performance of using GPU is clearly better than FPGA. On the other hand, the GDDR memory on GPUs, which are optimized for sequential data accesses, incurs a high performance penalty for irregular data access patterns in kernels such as $Translate$. The performance of GTX480 is always better than the obsolete G80. The speedup of using OpenMP to parallelize EMAN on CPU ranges from 1.6 times ($Translate$) to 3.5 times (MCF). With the increase in image size, the speedup of using OpenMP, FPGA or GPU will get improved marginally because of the increased data-level parallelism.

5.3 Application speedup

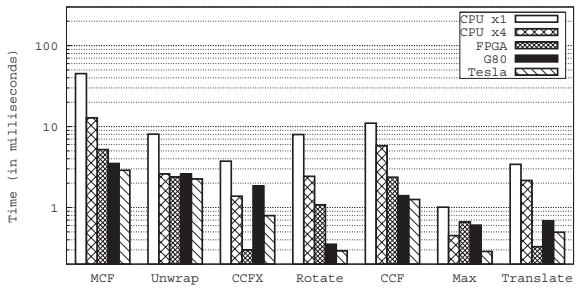
Table 3 shows the total execution time of a single iteration step of the 3D reconstruction process. The FPGA outperforms the 4-cores CPU by 2.54 times, while the GTX480 further outperforms the FPGA by 3.76 times. Workload in *EMAN* increases quadratically with increasing image size. However, the arithmetic intensity remains unchanged; therefore the increased workload cannot be utilized to introduce further performance improvement. For FPGA-based designs, the total execution time increases about 3 times when image size changed from 256^2 to 512^2, the improved speedup over CPU is largely attributed to the performance degradation on CPU. In contrast, large image can easily saturate the 448 cores in GTX480, which accounts for the $2x$ speedup. We consider that the FPGA and GPUs are more beneficial for Cryo-EM 3D reconstruction on large images.

Table 3: Execution time (milliseconds) and speedup (normalized to single-threaded CPU).

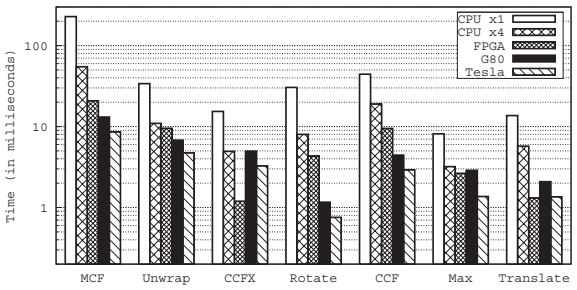
	Time-256	Speedup	Time-512	Speedup
CPU×1	80.86	-	377.21	-
CPU×4	30.86	2.62	124.9	3.02
G80	11.39	7.1	38.89	9.7
GTX480	5.82	13.9	13.10	28.8
FPGA	12.31	6.6	49.26	7.7

5.4 Power consumption

Table 4 lists the measured static, dynamic and working power of each device, where the static power is measured as idle power and the dynamic power is averaged across the entire execution. The working power, which is calculated as the difference between dynamic and static power, is used in following power analysis. It has long been noticed that the



(a) Execution time (the image size is 256×256)



(b) Execution time (the image size is 512×512)

Figure 10: Comparison of the kernel execution time on CPU, FPGA, G80 and GTX480.

FPGA can be used to build standalone devices, which are efficient than the PCIe-based accelerator card considered in this paper. When measuring the power of system configuration with either FPGA or GPU card, in order to get an accurate evaluation, we subtract the power consumption in the host PC. And, for simplicity, we assume that the host PC remains idle during the execution of the accelerator cards. The power cost, which is computed as

$$Power\ Cost = Working\ Power \times Execution\ Time, \quad (1)$$

is defined as the energy consumed by useful computing work and measured in millijoule. The multi-threaded EMAN introduces a 3.02 times speedup at the cost of 27% increase in working power; however, with reduced total execution time, the power cost of using 4 threads is only 48% of single-threaded EMAN. The result shows that our accelerator card outperforms the CPU and GPU by 7.33 times and 3.4 times.

Moreover, it is worth to note that the GTX480 chip is built with 40nm process, while the Virtex5 FPGA used in this paper is built with the less efficient 65nm process. We argue that the performance and power cost of our FPGA-based accelerator can be further improved by upgrading to the Virtex6 and Virtex7 families.

Table 4: Power consumption analysis

Device	Static	Dyn.	Working	Power (millijoule)
GTX480	53W	147W	94W	547mJ
FPGA	7W	20W	13W	160mJ
CPU×1	89W	119W	30W	2426mJ
CPU×4	89W	127W	38W	1173mJ

Table 5: FPGA Resource Consumption

	DSP48Es	LUT-FFs	BRAMs	Freq.
MakeRFP	106(55%)	56.5K(27%)	140(48%)	180MHz
RTAlign	140(72%)	55.5K(23%)	133(46%)	180MHz

5.5 Area and frequency

Two of the most time-consuming stream kernels of *EMAN* are implemented as separated computing streams: *MakeRFP* and *RTAlign*. The area consumption and frequency of each stream are summarized in Table 5. In practice, the two streams share a lot of common blocks and can be combined to form a more compact stream. However, the two computing streams will be executed on separated FPGA accelerator card, which renders such kind of combination of little interests for now.

6. RELATED WORK

Using of 2D images to predicate and reconstruct a 3D model is a well-accepted method applied in disciplines of medical imaging, bioinformatics, multimedia and etc. In [12], the authors introduce and discuss the design of the software architecture for a 3D reconstruction system in medical imaging. The importance of using different hardware platforms to accelerate corresponding parts of the algorithm is noticed, however, no concrete accelerator design considerations are included in their work. Compared with the Cryo-electron microcopy 3D reconstruction, the computed tomography (CT) reconstruction is a much popular topic in literature. Due to the advances in the GPU technologies, accelerating the CT reconstruction on GPUs is an active research topic. In [14], the authors introduced a real-time 3D reconstruction system for x-ray CT on the GeForce 8800GTX. The results prove that GPU is very suitable for CT reconstruction. A FPGA-based accelerator is also introduced in [15], in which the authors noted that the memory accesses is a crucial point for CT reconstruction on FPGAs.

Our previous work on accelerating the *EMAN* on heterogeneous cluster described the possibilities of optimizing kernels on the G80 architecture [7]. The optimized kernels are adaptive to the new Fermi architecture [8]. The focus of this work is to exploit the power of the FPGA for further performance improvement. To our best knowledge, this is the first work on FPGA-based accelerator design for the Cryo-electron microcopy 3D reconstruction. The stream architecture on FPGAs has been widely studied in the literature [5][6]. The architecture proposed in this paper is similar to the work presented in [6], in which specialized stream units are provided to handle stream operations, the control of data stream operations can be achieved by adjusting the stream descriptor.

The background of this work is to build a heterogeneous cluster, called Chaolong-1, which contains X86 CPUs, GodsonT CPUs, GPUs and FPGAs. *EMAN* analyzed in this paper is one of the key applications in this system. Heterogeneous systems with co-processors are nothing new to the HPC community. The CUBE [10] is a massively-parallel FPGA-based cluster that consists up to 512 FPGAs. The Axel [11], which is built with the Xilinx Virtex-5 FPGAs and nVidia C1060 GPUs, demonstrates a collaborative environment for heterogeneous accelerators. The Convey HC-1^{ex} [17] contains 4 Xilinx Virtex6 LX760 FPGAs, which are connected to the host CPU thought a FSB-based interface. The memory system of the Convey HC-1^{ex} is optimized for handling concurrent random memory accesses, which ren-

ders it perfect candidate to accelerate graph-based applications.The Cray XK6 supercomputer [20] combines Cray's Gemini interconnect, AMD Opteron processors and NVIDIA Tesla 20-Series GPUs to create a heterogeneous cluster that can be upgraded to unleash more than 50 PFlops.

7. CONCLUSION

In this paper, we introduce a coarse-grained stream architecture, in which the computing data path is configurable at runtime. To facilitate the decompiling of the computing flow and the data flow, we resort to a dedicated data flow module (DFM) to provide the ability to do pattern-based data accesses. Complex functions can be emulated by configuring the data path of the computing stream multiple times, whereas both data operations as well as related address calculations are offloaded to the DFMs.

The stream architecture is evaluated by accelerating a large-scale scientific application, called *EMAN* [3], which is an open source software package for single-particle 3D reconstruction from Cryo-electron microscopy images, on our customized FPGA accelerator card. Our FPGA-based accelerator design is compared with CPU and GPU solutions. Measured in raw performance, the FPGA-based design outperforms the 4-cores CPU by 2.54 times. When compared with our previous GPU-based designs, the FPGA-based design is about $3 \sim 4$ times slower. However, we argue that it is still beneficial to use the FPGA when taking the $7 \sim 8$ times power improvement into consideration.

The data bandwidths of the PCIe interface (2GB) and off-chip memory devices ($3.2 \sim 6.4$GB) are the limiting factors that prevent our system from reaching higher performance. There are several ways to overcome this limitation: 1) upgrading current gen1 PCIe interface; 2) developing in-socket accelerator with wider memory bandwidth. The second option, which has been pioneered by the Convey HC-1^{ex} [17] system, points out our future research direction.

The overarching goal for our research is to design a heterogeneous cluster that is built with X86 CPUs as well as heterogeneous accelerators, such as GPUs and FPGAs. Although dedicated optimizations can be used to improve the performance of individual device, in order to make the heterogeneous devices working synergistically and unleash the highest aggregated computing power, a collaborative environment is still needed, in which each type of device can be used to accelerate the most appropriate work. Our previous work on job partitioning between CPUs and GPUs [8] can be considered as a primitive attempt, as an extension, we expect to include FPGAs into the job partitioning process.

8. ACKNOWLEDGMENT

This work is supported by Chinese Academy of Sciences (No.KGCX1-YW-13), 973 Program of China (NO.2012CB316502), 863 Program of China (NO.2009AA01A129), and National Natural Science Foundation of China (NO.60803030, NO.60633040, NO.60925009, NO.60921002).

9. REFERENCES

[1] Scrofano, R.; Gokhale, M.; Trouw, F.; Prasanna, V.K.; , "Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers," Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on , vol., no., pp.23-34, 24-26 April 2006.

[2] Florent de Dinechin, Cristian Klein, and Bogdan Pasca, "Generating high-performance custom floating-point pipelines," In Field Programmable Logic and Applications, IEEE, August 2009.

[3] S. J. Ludtke, P. R. Baldwin, et.al. "Eman: Semiautomated software for high-resolution single-particle reconstructions", *Journal of Structural Biology*, 128(1):82-97, 1999.

[4] Jaydeep Marathe and Frank Mueller. 2008. "PFetch: software prefetching exploiting temporal predictability of memory access streams", In Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA '08). ACM, 1-8.

[5] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient realization of streaming applications on FPGAs," International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ACM, pp. 41-50, 2008.

[6] Nikolaos B., Sek M. Chai, Malcolm D., Dan L., Abelardo L., "Proteus: An Architectural Synthsis Toll Based on The Stream Programming Paradigm", Field Programmable Logic and Applications, 2009.

[7] G. Tan, Z. Guo, et.al. "Single-particle 3d reconstruction from cryo-electron microscopy images on GPU", In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 380–389, New York, NY, USA, 2009. ACM.

[8] L. Li, X. Li, G. Tan, M. Chen, and P. Zhang. "Experience of parallelizing cryo-EM 3D reconstruction on a CPU-GPU heterogeneous system", In Proceedings of the 20th International Symposium on High performance distributed computing (HPDC '11). ACM, 195-204, 2011.

[9] D. DeRosier and A. Klug, "A reconstruction of 3-dimensional structure from electron micrographs", *Nature*, 217:130–134, 1968.

[10] Mencer, O., Tsoi, K.H., Craimer, S., Todman, T., Luk, W., Wong, M.Y., Leong, P.H.W.: "CUBE: A 512-FPGA CLUSTER", In: Proc. IEEE Southern Programmable Logic Conference (SPL 2009) (April 2009)

[11] 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 115ÍC124, 2010.

[12] Holger Scherl, Stefan Hoppe, Markus Kowarschik, and Joachim Hornegger. "Design and implementation of the software architecture for a 3-D reconstruction system in medical imaging", In Proceedings of the 30th international conference on Software engineering (ICSE'08). ACM, 661-668, 2008.

[13] Duan B., Wendi W., et al. "Floating-point Mixed-radix FFT Core Generation for FPGA and Comparison with GPU and CPU", The 2011 International Conference on Field-Programmable Technology, Dec, 2011.

[14] F. Xu and K. Mueller, "Real-time 3D computed tomographic reconstruction using commodity graphics hardware", Physics in Medicine and Biology 52(12) (2007), 3405-3419.

[15] S. Coric, M. Leeser, E. Miller, and M. Trepanier. "Parallel-beam backprojection an FPGA implementation optimized for medical imaging", In Proc. ACM Int. Symp. Field-Programmable Gate Arrays (FPGAąŕ02), pages 217-226, February 2002.

[16] Brahim Betkaoui, David B Thomas, Wayne Luk, "Comparing Performance and Energy Efficiency of FPGAs and GPUs for High Productivity Computing," The 2010 International Conference on Field-Programmable Technology (FPT'10), 8-11 Dec. 2010.

[17] Brewer, T.M.; , "Instruction Set Innovations for the Convey HC-1 Computer," Micro, IEEE , vol.30, no.2, pp.70-79, March-April 2010.

[18] Xilinx, http://www.xilinx.com.

[19] Fluke, http://www.fluke.com.

[20] http://www.cray.com/Products/XK6/KX6.aspx.